DeepScaling: Microservices AutoScaling for Stable CPU Utilization in Large Scale Cloud Systems

Ziliang Wang
Chongqing University
Ant Group
Chongqing, China
wangziliang@cqu.edu.cn

Shiyi Zhu
Ant Group
Hangzhou, China
zhushiyi.zsy@antgroup.com

Jianguo Li*
Ant Group
Hangzhou, China
lijg.zero@antgroup.com

Wei Jiang
Ant Group
Hangzhou, China
shouzhi.jw@antgroup.com

K. K. Ramakrishnan
University of California, Riverside
Riverside, CA, USA
kk@cs.ucr.edu

Yangfei Zheng
Ant Group
Hangzhou, China
yangfei.zyf@antgroup.com

Meng Yan
Chongqing University
Chongqing, China
mengy@cqu.edu.cn

Xiaohong Zhang*
Chongqing University
Chongqing, China
xhongz@cqu.edu.cn

Alex X. Liu
Ant Group
Hangzhou, China
alexliu@antgroup.com

ABSTRACT

Cloud service providers conservatively provision excessive resources to ensure service level objectives (SLOs) are met. They often set lower CPU utilization targets to ensure service quality is not degraded, even when the workload varies significantly. Not only does this potentially waste resources, but it can also consume excessive power in large-scale cloud deployments. This paper aims to minimize resource costs while ensuring SLO requirements are met in a dynamically varying, large-scale production microservice environment. We propose DeepScaling, which introduces **three innovative components** to adaptively refine the target CPU utilization to a level that is maintained at a stable value to meet SLO constraints while using minimum resources. *First*, DeepScaling forecasts the workload for each service using a Spatio-temporal Graph Neural Network. *Second*, DeepScaling estimates the CPU utilization by mapping the workload intensity to an estimated CPU utilization with a Deep Neural Network, while taking into account multiple factors in the cloud environment (e.g., periodic tasks and traffic). *Third*, DeepScaling generates an autoscaling policy for each service based on an improved Deep Q Network (DQN). The adaptive autoscaling policy updates the target CPU utilization to be a maximum, stable value, while ensuring SLOs is not violated. We compare DeepScaling with state-of-the-art autoscaling approaches in the large-scale production cloud environment of the Ant Group. It shows that DeepScaling outperforms other approaches both in terms of maintaining stable service performance, and saving resources, by a significant margin. The deployment of DeepScaling in Ant Group's real production environment with 135 microservices saves the provisioning of over **30,000 CPU cores per day**, on average.

1 INTRODUCTION

Large scale cloud systems that provide a range of services (such as latency-sensitive payment services) have significant challenges in balancing a number of concerns: ensuring user quality of experience with a timely, responsive service [36], ensuring appropriately provisioned resources without significant over-provisioning (thus avoiding resource wastage) [24, 37], and adapting to continual, yet potentially significant variations in workloads. Several studies have shown that servers with chronically low CPU utilization waste power resources [33, 46, 47]. By reducing resources for low-load services, re-allocating those saved servers to high-load services can effectively improve the performance of those high-load services. Alternatively, we can temporarily shut servers down to save a large amount of power. To this end, cloud service providers seek to utilize an automatic scaling system to make the CPU utilization of the systems being provisioned to the desired target level and ensure their services' Service Level Objectives (SLOs) are met.

Prior work on cloud autoscaling can be categorized as *rule-based schemes* [5, 6, 17] or *learning-based schemes* [7, 21, 30, 34, 36, 37]. *Rule-based autoscaling schemes* use static upper and lower thresholds for certain system performance metrics (such as CPU and memory utilization) or application metrics (such as request arrival rate) so that resources can be scaled

*Corresponding author.

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11…$15.00
https://doi.org/10.1145/3542929.3563469

up when the metrics go above an upper threshold and scaled down when the metrics go below a lower threshold. The key limitation of rule-based autoscaling schemes is that they require significant domain knowledge from experts to set thresholds appropriately. Further, setting thresholds for each microservice could be labor-intensive as there are more than thousands of microservices for large-scale industrial systems. As different microservices may use a different amount of resources (such as CPU and memory), the autoscaling thresholds have to be set differently. *Learning-based autoscaling schemes* can effectively reduce the load on humans, but current researches rarely consider resource wastage (due to service workload variations) and SLO assurances together [36]. Production cloud environments observe considerable diurnal variation in their workloads. The online payment system of Ant Group requires tens of thousands of containers to meet peak demand during certain periods of the day while requiring a few thousand containers during the rest of the day. The required number of containers can be even lower in the late evening. Administrators tend to over-provision service resources to meet SLOs during peak demand, leading to a chronic state of low utilization of various resources (CPU/memory) most of the time.

Our paper addresses the above problems with DeepScaling, an autoscaling framework for microservices. It seeks to adapt to the workload by estimating resource requirements at a fine granularity and maintaining resource utilizations consistently at a target level to meet SLOs while reducing over-provisioning. DeepScaling consists of two basic monitors (a service monitor and an SLO monitor) and a target utilization level controller, along with **three innovative components**: a workload forecaster, a CPU utilization estimator, and a scaling decision-maker.

DeepScaling proposes a new autoscaling goal aimed at maximizing and maintaining resource utilization at a stable value while meeting SLO assurances, starting from an initial coarse target level. The initial coarse target is set empirically based on historical usage data. DeepScaling then adaptively refines the target level to increase the utilization of resources until the SLO is just violated. This is performed in a tight control loop, with an SLO monitor and target level controller for each service. DeepScaling generates autoscaling policies with the aforementioned three innovative components. Therefore, DeepScaling achieves its goal of each service running at a stable resource utilization around the refined target level, even when the workload for the service varies greatly.

There are three major challenges for autoscaling to maintain CPU utilization around a stable target level. *First*, it is challenging to forecast the workload of different services accurately. Many existing works only consider simpler regression techniques to forecast the workload [1, 22]. However, a cloud-native microservice architecture introduces complex interactions, including those between microservices, which existing works do not effectively capture. Such interactions impact how accurately we can forecast workloads. To address this issue, we propose to use Spatio-temporal graph neural networks (STGNN) for workload forecasting, modeling the interactions with the service call-graph and the inherited multi-variant relationships among the workload metrics (including the representation of multiple RPC workloads as well as other workloads). By capturing such relationships, our workload forecasting is much more accurate in providing additional time for the pod to be initiated proactively while scaling-up resources.

Second, it is challenging to accurately characterize a service's workload so that it can reflect CPU utilization properly. Forecasting the CPU utilization for a microservice directly based on its historical CPU data results in significant forecasting errors (usually $\geq$ 30% in our trials) due to the fact that it is difficult to handle the high variability of the instantaneous CPU utilization when multiplexing several services together, as well as the influence of background tasks on a server. Existing autoscaling approaches typically use the number of remote procedure calls (RPCs) to characterize service workload and establish the mapping from RPC to CPU utilization [16, 18, 39, 40]. However, characterizing only the RPCs is insufficient, since different requests invoke different aspects of the service to execute tasks. Some services are compute-dominant while others are I/O-dominant. To address this issue, we collected multi-dimensional workload metrics for each service, including RPC requests, file I/O, DB access, message requests, HTTP requests, as well as specific auxiliary features (instance count, service ID, timestamp) to comprehensively characterize the service's workload for accurate estimation of the CPU utilization. We design a deep probabilistic network based model as the CPU utilization estimator, to accurately estimate CPU consumption even under special conditions, such as with periodic tasks and internal system events.

Third, it is challenging to decide on the precise resources required for each service based on the estimated CPU utilization, which of course is the ultimate goal of autoscaling. The relationship between the required service pods and CPU utilization is usually complex and non-linear, especially when the service resources are at critical utilization levels. To address this issue, we propose an improved DQN model with a dedicated loss function based on the target level, enabling us to find the optimal resource requirement quickly. As each service is associated with a unique ID in the DQN, this allows modeling multiple services with a single shared model.

As can be seen, DeepScaling uses a three-step approach, which consists of a DQN model to predict the required resources quickly, an STGNN model for workload forecasting

and the DNN algorithm for accurate CPU utilization estimation. This not only helps in proactively provisioning resources to accommodate the delays involved for the actual task of instantiating the required pods, but also helps us overcome the challenges of providing a single, truly comprehensive end-to-end (E2E) solution since there are not enough samples we can obtain of SLO violations to build such a single E2E model.

Our work is performed in the context of a large-scale production cloud service system from the Ant Group. The system consists of 3000+ microservices running on over 1 million virtual containers, while the workload typically exceeds 1 million user access requests per minute. The Ant payment service requires 7×24 availability, and the SLO in terms of the success rate for accesses every second is required to be higher than 99.9995%. We primarily focus on autoscaling to ensure that this large-scale system meets its stringent SLOs. In addition to the challenge of large-scale and stringent SLOs, different microservices are heterogeneous with significantly different workload intensities (request rate per second). Finally, even the same microservice is observed to have large variability in the workload during each day, as well as have seasonal variations.

Our work makes the following major contributions:

- We propose DeepScaling, a deep neural network based framework for autoscaling of large-scale cloud systems. DeepScaling achieves maximum resource savings by maintaining CPU utilization at a stable target without loss in the quality of service.
- We propose a spatio-temporal graph neural network that forecasts the workload for each service accurately by learning the relationship between different workload metrics as well as among services, with the use of service call-graphs.
- We propose a deep neural network as the CPU utilization estimator for different services, and a reinforcement learning model for generating the autoscaling policy. Thus, the two models generate the optimal autoscaling policy for services with different workloads, by collaboratively computing the metrics of multi-dimensional workloads and specific auxiliary features.
- We conducted extensive experiments on the production cloud service, demonstrating the significantly improved performance of DeepScaling. The production system supports the world-leading online payment services from Ant Group. DeepScaling improves the time we operate at a stable CPU utilization by 24.6% per day, and saves 14.0% more resources compared to the state-of-the-art autoscaling approach. This further confirms the practical applicability and scalability of DeepScaling. A deployment of DeepScaling on 135 production microservices at the Ant Group for over six months showed that it saves the provisioning of over 30,000 CPU cores per day on average (measured in CPU core hours, 30K * 24 CPU core hours are saved each day), compared to not using DeepScaling.

2 BACKGROUND AND RELATED WORK

Many studies confirm low utilization in cloud systems that are observed when service resources are manually or rule-based provisioning. Sun et al. [42] analyze the usage of a YARN cluster at Alibaba, and report that resource utilization is less than 55% about 80% of the time. Lu et al. [27] analyzes an Alibaba trace, showing that for most instances, the peak resource utilization is 80% or less. Autoscaling microservices for cloud systems has been studied extensively in the past few years. There are several surveys [9, 12, 19, 26]. Our work, similar to a number of other works, focuses on cluster-level resource management with horizontal pod autoscaling (HPA) for deployed applications. As noted earlier, existing autoscaling efforts can be classified broadly into rule-based versus learning-based approaches.

Rule-based Autoscaling Approaches. Most traditional autoscaling approaches are rule-based reactive approaches [5, 6, 17], which scale the service resources based on specific events. For these approaches, the key goal is to find a proper threshold value that triggers the scaling mechanism, such as a fixed CPU utilization threshold or an average response time (RT). The rule-based autoscaling approach requires empirical experience, which is difficult to scale.

Learning-Based Autoscaling Approaches. Many existing learning-based autoscaling approaches [1, 14, 25, 28, 34–37, 45] focus on avoiding service anomalies. Abnormal states include service response time (RT) exceeding a threshold, the CPU utilization being too high, or having too many out-of-memory events [15, 20, 37]. There are two typical anomaly-state driven autoscaling schemes: FIRM [36] and Autopilot [37]. FIRM uses machine learning techniques to detect service performance anomalies (e.g., the RT of a microservice is abnormally long) so that when such anomalies do happen, we can scale up by adding more pods or computing resources in general. The key limitation of FIRM is that autoscaling occurs only after performance anomalies happen. This has the potential to have the service anomaly lasting for an extended period of time. Autoscaling at a large-scale may take minutes for cold start of pods, and at least a few seconds for warm starts. Another limitation of FIRM is that it does not provide a policy to scale down when more resources are allocated to a microservice than necessary. Autopilot takes the time-series of the CPU utilization of a microservice as input, uses a simple heuristic mechanism to output the target CPU utilization, and then uses this target CPU utilization to calculate the number of pods required as a linear function. Pods are added or reduced, with the goal of minimizing the

risk that a microservice suffers from anomalies. Based on our implementation and experiments with Autopilot, despite its simplicity, Autopilot still suffers from relatively poor system stability because the estimation of the number of pod required is frequently inaccurate. This is due to two facts: (1) the relationship between CPU utilization and computing resources is usually non-linear, while Autopilot stays with linear assumption; (2) the relationship differs for different microservices due to the different demands they have for computing and I/O resources.

In addition, a large number of workload driven autoscaling approaches have been proposed. For example, Abdullah [1] uses regression trees to model the relationship between the number of pods and RT, and then generates the recommended number of pods to avoid the overtime of service RT. GRAF [34] leverages the current state of the workload, microservices trace data for modeling microservices, and their invocation structure using graph neural networks (GNN). GRAF focuses on predicting the tail latency. It seeks to proactively optimize the total CPU resources available for each microservice while satisfying the latency SLO. These approaches usually do not take the distribution of the transition states (of RT) into account, which are far from precisely modeling. The CPU utilization has different properties with RT. It will change along with the change of the service's workload. Therefore, a simple idea is to establish a performance model to characterize the service CPU utilization or memory utilization changes [14, 16, 23, 38, 41, 45]. For instance, Jiang et al. [11] proposed benchmarking the individual performance profile of each pod instance and forecasting the workload. Gevros et al. [14] proposed an efficient discrete-time model based on a weighted max-min fair resource allocations for each single resource, shared among a number of users with a heterogeneous Additive Increase Multiplicative Decrease (AIMD) controller. Yu et al. [45] proposed an approach to automatically identify the services that need scaling and scale them to meet the service level agreement (SLA), with an optimal cost for the system. However, we observe that these methods emphasize maintaining the SLO first, without significant savings of service resources.

DeepScaling is inspired by the learning-based autoscaling approach proposed in [1] and [35]. The main difference between DeepScaling and these approaches are two folds. First, DeepScaling introduces multiple metrics for accurate workload characterization instead of just the RPC metric as in [1, 35]. Second and importantly, DeepScaling aims to maintain the SLO and stabilize the CPU utilization for each service close to a maximum utilization target.

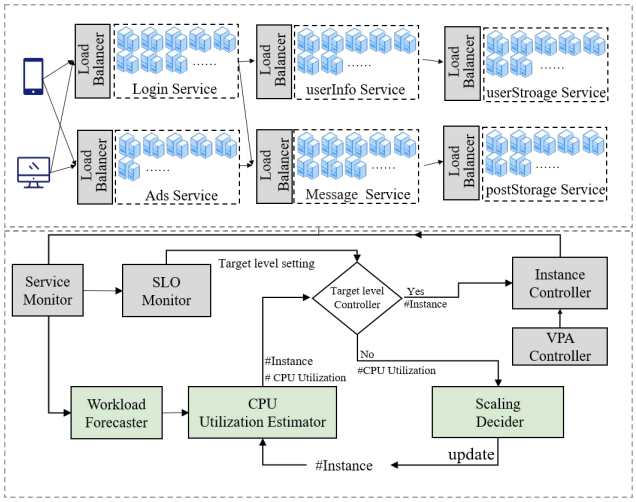


Figure 1: System architecture of DeepScaling.

3 DEEPSCALING OVERVIEW

Both our experimental and deployed cloud environments use virtualization for hosting and managing services [2]. We conduct experiments on a system in which each CPU server supports multiple virtual containers or pods, with each pod supporting one microservice. The workflow of DeepScaling is illustrated in Figure 1 and Algorithm 1.

3.1 Modules of DeepScaling

In this section, we describe the overall system architecture of DeepScaling and its core modules. As shown in Figure 1, DeepScaling uses the following modules.

Load Balancer: The Load Balancer for each service enforces a policy to equitably distribute requests to the pods provisioned for the corresponding service. A service is deployed and auto-scaled in the same data center zone, which generally has the same hardware configuration. The Load Balancer[1] distributes requests so that each instance of a service, and thus each pod of a service, carries approximately the same workload and has the same CPU utilization.

Service Monitor: The Service Monitor focuses on collecting metrics for all services in real time, including seven workload metrics (§ 4), CPU utilization, information on achieving SLOs, and instance counts. The collected data are aggregated to a minute granularity. All workload metrics data are summed, and the CPU utilization data is averaged across all the instances of each service.

Workload Forecaster: We analyze the call-graph and the main workload metrics for each microservice. Then, a

[1]The load balancer is associated with the service-message gateway implemented by Ant Group. Details can be found at https://github.com/mosn/mosn.

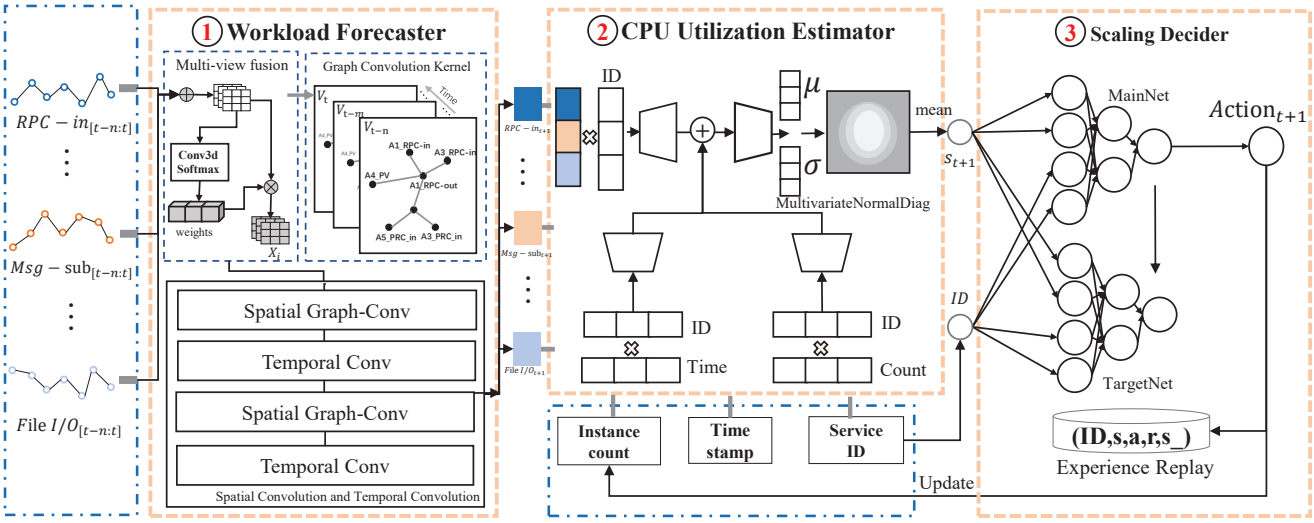


Figure 2: Integration of Forecaster, Estimator, and Scaling Decider. s: CPU utilization s_t at time t; a: action a_t at t; r: reward for a_t; $s\_$: the CPU utilization after the a_t is executed.

spatial-temporal graph neural network (STGNN) is used to forecast the workload metrics for each microservice. The call-graph helps to model the traffic relationship among services. The STGNN thus produces a highly accurate forecast of workload metrics for each time epoch (e.g., 30 min.) The workload metrics thus forecast are fed to the CPU utilization estimation module.

CPU Utilization Estimator: This module estimates CPU consumption based on 7 workload metrics along with 3 specific auxiliary features (service ID, timestamp, and instance count). The nonlinear relationship between these 10 (=7+3) indicators and the CPU utilization is modeled by a deep probabilistic regression network. The input to the network is the metric value from the workload forecaster and the latest number of instances given by the Scaling Decider. The output is the estimated CPU utilization.

Scaling Decider: In this module, a reinforcement learning (RL) model is adopted to generate autoscaling policies. Going into more detail, a model-based DQN model works with the CPU utilization estimator to quickly determine the optimal number of service instances. We design a reward function for rapidly determining the optimal solution. To allow a shared policy generation model acorss multiple services, we include the service identifier to the experience pool of the DQN model.

Target level Controller: The CPU target level (T) is a core parameter value for DeepScaling. The initial value of T is the maximum CPU utilization value seen historically for normal execution of the service in the past. DeepScaling provides a buffer value ($\tau = 5\%$) to allow for possible model errors and other noise. The recommended instance count is obtained when the service is operating at the target level ($T \pm \tau$), while the SLOs are satisfied. Slight load balancing variations, forecasting or estimation errors and

Algorithm 1 DeepScaling Workflow

1: The *Target Level Controller* ① gets the initial value T and service status $S = 1$
2: Initialize *SLO monitor* ③ and instance count I_0, and target level increase $\delta = 5$, the buffer size $\tau = 5$
3: Training Workload Forecaster ④, CPU Utilization Estimator ⑤ and Scaling Decider ⑥
4: **while** True **do**
5: **if** *SLO monitor* ③ detects SLO violation **then**
6: $T = T - \delta$, $S = 0$
7: **end if**
8: **if** $S = 1$ **then**
9: $T = T + \delta$
10: **end if**
11: **if** $S = 0$ **then**
12: $T = T$
13: **end if**
14: ④ forecasts the workload y
15: ⑤ estimates the CPU utilization value $c = f(y, I_0)$
16: **while** $abs(c - T) > \tau$ **do**
17: ⑥ recommends the instance counts $I_c = f_s(c)$
18: ⑤ estimates the CPU utilization value $c = f(y, I_c)$
19: **end while**
20: *Instance Controller* ⑦ recommends the instances I_c.
21: **end while**

non-homogeneous CPU characteristics can also be accommodated by this buffer value for the target utilization. Sharp changes or large estimation errors will trigger SLO violations. This causes a reduction of the target, T, (Line 5-7 and 10-12 in Algorithm 1) until a new stable value is reached.

SLO Monitor: The SLO monitor determines whether a core microservice violates its SLO by monitoring the microservice response time (RT), GC (garbage collection) time, I/O RT, and other metrics. When the SLO monitor detects an abnormal value of a metric for the core microservice, it notifies the target level controller to lower the target level for that service, in addition to changing the controller state.

Instance Controller: The Instance Controller, as part of the overall autoscaling task completes service de-activation (shutting instance(s) down) task in real-time, and the service deployment task by warm-starting the required number of instances (within 15 seconds), or cold starting them (within 5 minutes). The Instance Controller adjusts the service resources of each microservice by increasing or decreasing the number of standard pods.

The Vertical Pod Autoscaler (VPA) Controller: DeepScaling uses standard containers (Kubernetes 4C8G - 4 CPU cores with 8 GB RAM) to realize the horizontal pod autoscaling (HPA) of services. To support specific resource requirement for different kinds of services, we also include the VPA controller for *the global setting of pod configuration* of each service before deployment. That means, each instance of the service shares the same pod configuration during horizontal scaling. For most services, the VPA controller is not activated. For some memory-sensitive services, the administrator invokes the VPA controller to modify the default pod configuration to 4C16G for example. In summary, VPA is usually configured for tuning before deployment, while DeepScaling considers cluster level resource management with autoscaling pod replicas after deployment.

We now elaborate on the core componets of DeepScaling next: the Workload Forecaster in § 4, the CPU Utilization Estimator in § 5, and the Scaling Decider in § 6. The experimental design and overall performance of DeepScaling are presented in § 7.

4 WORKLOAD FORECASTING

In this part, we describe the microservice workload characterization metrics and elaborate on how to forecast the future workload metrics with a STGNN model. The specific structure of STGNN is shown in Figure 2, and we describe the network in detail below.

4.1 Workload Characterization

For each microservice, as shown in Figure 3, we study its invocation relationships in the system of microservices. We

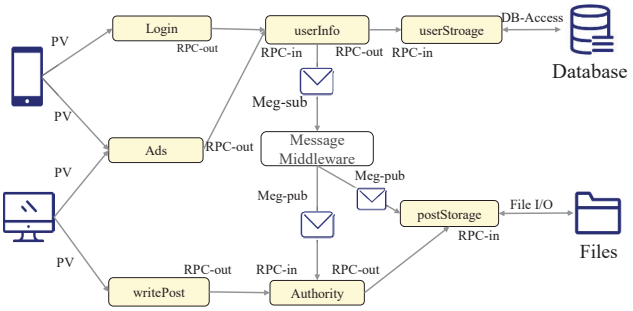


Figure 3: Different workload metrics in a sample microservice system. Each yellow box represents a microservice, with totally 7 microservices for the service.

collected 7 commonly used CPU consumption metrics to characterize each workload. These workload metrics are generated from the active execution of the complete microservice system. In general, the lifecycle of a task starts with a user request and ends with an operation on the data. During this period, each microservice is responsible for different tasks and consumes CPU. The seven workload metrics are:

- **RPC-in** counts the number of incoming RPC requests for this microservice and reflects the CPU consumption for completing a specific logical function. RPC-in is usually the main CPU consuming component.
- **RPC-out** counts the number of RPC requests from each microservice to other microservices in the task chain and is usually in proportional to the RPC-in metric. CPU consumption also grows in proportion to the RPC-out request count.
- **Msg-pub** counts the number of message publications to the middleware message broker.
- **Msg-sub** counts the number of message subscriptions from the middleware message broker. CPU consumption grows linearly with increasing Msg-pub and Msg-sub [4].
- **DB-Access** counts the number of database accesses (query, insert, delete, etc.). An RPC request usually yields database accesses, which is not counted in the RPC-in and RPC-out metrics.
- **Page-View(PV)** counts the number of dynamic/static webpage views resulting from HTTP requests in a time interval.
- **File I/O** counts the number of file read and write operations. Frequent I/O operations also consume a lot of CPU resources [10].

The monitor captures these seven workload metrics for each service every minute, yielding seven corresponded timeseries.

4.2 Forecasting with STGNN

As one of the core components, STGNN accomplishes an accurate forecast of future workloads as a prerequisite for DeepScaling. As mentioned above, there is a clear relationship among various workloads. For example, the higher the number of Page-Views generated by a user, the higher is the number of accesses to the data by the underlying service. To improve the accuracy of forecasting the workloads, we model the relationship among the seven workload metrics through a multi-view fusion layer in the graph neural network (GNN).

Given a workload metric time-series $\mathbf{x}_{i,1:t-1}$ (for instance RPC-in), the basic problem here is to forecast the workload metric at time-stamp t with

$$\mathbf{x}_{i,t} = F(\mathbf{x}_{i,1:t-1}; \mathbf{w}), \tag{1}$$

where $F(\cdot; \mathbf{w})$ is a forecast function such as a deep neural network with parameter $\mathbf{w}$. As there are 7 workload metrics to characterize the microservice, the intuitive way is to forecast each metric separately. However, we find there is an intrinsic relationship between different metrics such as RPC-in and RPC-out. If we forecast them separately, it may produce inaccurate and inconsistent output. Hence, it is better to simultaneously forecast multiple time-series together by taking their relationships into consideration:

$$\{\mathbf{x}_0, \ldots, \mathbf{x}_{I-1}\}_t = F(\{\mathbf{x}_0, \ldots, \mathbf{x}_{I-1}\}_{1:t-1}; \mathbf{w}). \tag{2}$$

In this work, we build such a forecaster with spatial-temporal graph neural networks (STGNN), with several components.

Multi-view Fusion Layer

When we forecast the value of a particular workload such as RPC-in in the next time step, we take the impact of other workload metrics in the same time window into account. We use a fusion layer based on an attention mechanism for importance modeling of different workload metric features. For multiple workload metrics $\{\mathbf{x}_0, \ldots, \mathbf{x}_{I-1}\}$, we consider each metric as a single view, and fuse the vector of each view together through

$$\mathbf{x}_m = \mathbf{x}_0 \oplus \mathbf{x}_1 \oplus \ldots \mathbf{x}_{I-1}, \tag{3}$$

where operator $\oplus$ means vector concatenation, and $\mathbf{x}_m$ is the concatenation of all input views. Let

$$\mathbf{x}'_m = \mathbf{x}_m \odot Softmax(f_a(\mathbf{x}_m; \mathbf{w}_a)), \tag{4}$$

where $\odot$ means element-wise product, $f_a(; \mathbf{w}_a)$ is a sub-neural-network with parameter $\mathbf{w}_a$ which produces a weight for each view, and $Softmax$ is the softmax function which transfers the weight to the range [0,1]. Equation 4 defines the weighted multi-view fusion layer dedicated for our STGNN in DeepScaling.

Graph Convolution Kernel

Next, we further consider the impact of workload metrics between different services. We build a graph structure to model the relationship of workload metrics across services. We organize multiple workload metrics as a graph structure $\mathcal{G}_t = (\mathcal{V}, \mathcal{E}, \mathbf{A})$, where node $\mathcal{V}$ represents different workload metrics and edge $\mathcal{E}$ indicates the relationship between them. For example, as shown in Figure 3, we consider RPC-out of the Login service as one object and RPC-in of service UsreInfo as another object. They act as two separate nodes in the graph and are connected. $\mathbf{A}$ is the weighted adjacency matrix of $\mathcal{G}_t$. Based on the spectral graph theory, we introduce the scaled Laplacian matrix $\tilde{\mathbf{A}}$ as in [3] to represent the relationship graph among different features due to its good mathematical properties, such as it being positively defined, etc.

$$\tilde{\mathbf{A}} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}, \tag{5}$$

where $\mathbf{D}$ is the diagonal matrix with $\mathbf{D}_{ii} = \sum_{ij} \mathbf{A}_{ij}$, and $\mathbf{I}$ is the identity matrix.

Give a spatial slice vector $x_t \in \mathbf{R}^n$ (e.g., a slice of $\mathbf{x}'_m$ at time step t with n workload metrics), a graph convolution is defined as

$$\Theta \circledast_{\mathcal{G}} x_t = \sum_{k=1}^{K} \theta_k T_k(\tilde{\mathbf{A}}) x_t, \tag{6}$$

where $\circledast_{\mathcal{G}}$ is the graph convolution operator, $\theta \in \mathbf{R}^K$ is the vector of coefficients, $\Theta = \{\theta_k\}$, $T_k \in \mathbf{R}^n$ is the k-th order Chebyshev polynomial at the scaled Laplacian. For the detailed explanation, please refer to [44].

Spatial Convolution and Temporal Convolution

The spatial convolution is defined over the graph kernel as

$$x_s = f_s(x_t; \Theta) = \Theta \circledast_{\mathcal{G}} x_t, \tag{7}$$

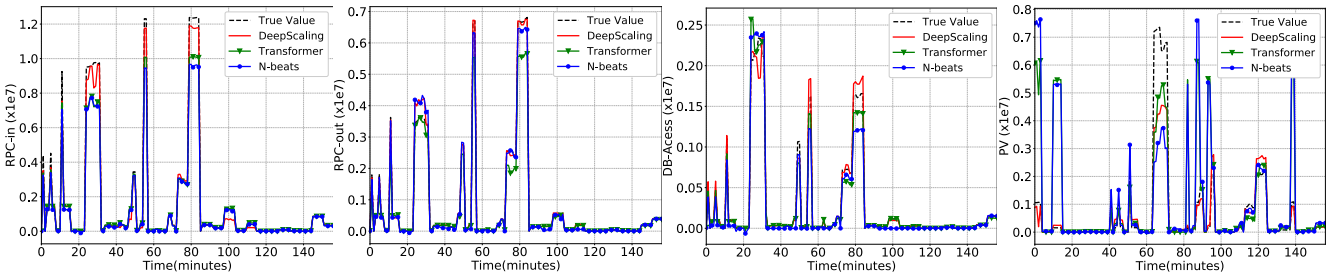
which captures the interactions among time-series at time-stamp t [44].

The temporal convolution is defined in the temporal dimension, where different time-series share the same convolution kernel weight.

$$\mathbf{x}'_i = f_t(\mathbf{x}_i; \mathbf{w}_t) = \mathbf{x}_i \circledast \mathbf{w}_t \tag{8}$$

where $\mathbf{x}_i$ is the i-th time-series, $\mathbf{w}_t \in \mathbf{R}^{k_t}$ is 1-dimensional convolutional kernel with size $k_t \times 1$ shared across all the n time-series ($k_t = 7$ empirically picked in our experiments).

We stack/connect multiple spatial convolutions and temporal convolutions together to capture both the spatial interactions and temporal correlations. We also use the normalization layer after each convolution-block to avoid overfitting. The whole STGNN model (F in Equation 2) is thus a nested function of f_a, f_s, f_t as defined in Equation 4, Equation 7, Equation 8, respectively, which can be trained in an end-to-end way.



(a) RPC-in (b) RPC-out (c) DB-Acess (d) PV

Figure 4: Real workloads forecast using the proposed autoscaling method used in the experimental evaluation

Table 1: Results with different workload forecasting models

Result / Metric / Method	MAE	Gain	RMSE	Gain
N-beats	1.61	35.66%	188.89	36.80%
Transformer	1.39	25.26%	166.95	28.51%
DeepScaling	1.04	-	119.37	-

4.3 Performance of Workload Forecasting

To comprehensively evaluate the workload forecast model, we trained different models including N-beats [31] and Transformer [43]. N-beats is a state-of-the-art forecast model, which is a deep neural network architecture based on backward and forward residual links and a very deep stack of fully-connected layers. Transformer is another popular forecasting model. We describe our dataset in detail in § 7.1. Each workload metric takes values in the range 0 to 10^7. Table 1 shows the normalized mean absolute error (MAE) and root mean squared error (RMSE) [8] of different forecasting models of the workload metrics. Compared to N-beats and Transformer, DeepScaling reduced the MAE by 35.66% and 25.26%. The RMSE also dropped by 36.80% and 28.51% compared to N-beats and Transformer, respectively. These experimental results show that DeepScaling's workload forecast has excellent performance relative to the state-of-the-art forecasting methods.

To demonstrate the forecasting performance of DeepScaling in different situations, we show several test cases in Figure 4. For example, in Figure 4(a), DeepScaling is more effective at forecasting RPC-in bursts over time periods 20-40 and 70-90. As shown in Figure 4(d), both Transformer and N-beats forecast values that far exceed the true values at time points 90-130. This indicates that the predictive performance of these other methods is poorer than that of DeepScaling when there are significant changes to the workload, especially with bursts.

5 CPU UTILIZATION ESTIMATOR

With the workload metrics forecast, we design a DNN model to estimate the CPU utilization. This estimator receives the workload metrics and predicts the CPU utilization of each service with a given number of instances provisioned. Figure 2 shows the network structure of the estimator. We first introduce the feature embedding step and then describe the CPU estimation model.

5.1 Feature Embedding

The input for CPU utilization estimation is the workload metric vector $x_t \in \mathbf{R}^n$ ($n = 7$) at time t. During the test (inference) phase, x_t is the output of the workload forecast from the previous stage. During the training phase, we could use both historical data and forecast data to train the model. Different microservices have different behaviors for each workload. Therefore, when we develop a single general model to estimate CPU utilization for all microservices, it is necessary to introduce certain microservice specific auxiliary features to enhance the general applicability of the shared model. The auxiliary features are:

- **Instance-count**: the number of instances (or pods) for each microservice, ranging from 1 to *#max-instance*;
- **Service-ID**: the unique identifier of each microservice, ranging from 1 to *#max-service*;
- **Time-stamp**: the time-stamp during the day, in minutes, when the workload metrics are collected/forecast, ranging from 1 to 1440 (24hours).

Workload metrics and auxiliary features are combined together as input for accurate CPU estimation. As workload metrics are multi-dimensional vectors of time-series, and auxiliary features are scalars, we use a feature embedding method to realize the combination. The feature embedding consists of 4 steps.

First, we use one-hot encoding to represent these auxiliary features. Specifically, it encodes the features to generate a vector with a specified dimension d (d may be *#max-instance*,

#max-service or 1440), while the vector has all elements being 0 except the i-th element being 1.

Second, we define an embedding function as:

$$u' = f(u; \mathbf{W}_u) = ReLU(\mathbf{W}_u \cdot u + \mathbf{b}), \tag{9}$$

where $\mathbf{W}_u \in \mathbf{R}^{n \times d}$ is the embedding matrix, $\mathbf{b}$ is the bias vector, and $ReLU(\cdot)$ is the activation function. In this way, $u \in \mathbf{R}^d$ is thus projected to $u' \in \mathbf{R}^n$. Given the three auxiliary features u_1, u_2, u_3, with this method, we could produce embedded auxiliary features u'_1, u'_2, u'_3. At this point, we have encoded the three specific features of each service as unique variables.

Third, we model the relationship between auxiliary features (service-ID, instance counts, time-stamp) and the workload metrics in the embedded space. Given the embedded auxiliary feature u'_i (i=1,2,3) and the workload feature x_t, we define the joint embedding layer as

$$z_i = f(u'_i, x_t; \mathbf{W}_i) = ReLU(\mathbf{W}_i \cdot (u'_i \odot x_t)),\ i = 1, 2, 3, \tag{10}$$

where $\mathbf{W}_i$ is the joint embedding matrix, and $\odot$ is the element-wise product operator.

Fourth, we concatenate the three joint embedding results (z_i corresponding to each u_i) together as final representation:

$$Z = z_1 \oplus z_2 \oplus z_3. \tag{11}$$

5.2 Estimation Model

A deep probabilistic neural network is used to estimate the expectation and variance of the CPU utilization distribution based on the feature embedding Z, which consists of three steps.

First, Z is passed through several shared, fully-connected layers to obtain intermediate results Z', to learn the nonlinear relationships between features:

$$Z' = f_z(Z; \mathbf{W}_z). \tag{12}$$

Second, two separate output branches (or heads) are connected over Z' to produce mean and variance

$$\mu = f_\mu(Z'; \mathbf{W}_\mu), \tag{13}$$
$$\sigma = f_\sigma(Z'; \mathbf{W}_\sigma). \tag{14}$$

Third, we estimate CPU utilization with

$$y_t = Mean\{f_M(\mu, \sigma)\}. \tag{15}$$

where f_M is the MultivariateNormalDiag layer in Tensorflow, $Mean\{\cdot\}$ computes the mean value of the distribution.

All the layers described in Equation 9-15 are connected together into one Deep Neural Network as shown in the middle-box of Figure 2, where all the model parameters $\mathbf{W}$ can be trained in an end-to-end way.

Table 2: Results of different CPU estimation methods and relative gain with DeepScaling.

Method	MAE	Gain	RMSE	Gain	Max_{error}
LR	1.44	54.8%	2.26	64.15%	21.06
SVM	1.99	67.3%	2.97	72.72%	20.96
DTR	1.25	48.0%	2.11	61.61%	21.97
DeepScaling	0.65	-	0.81	-	2.69

5.3 Performance of Estimator

We compare our estimator with the regression methods used in learning-based autoscaling methods (Linear Regression (LR) [22], Support Vector Machine (SVM) [30], and Decision Tree Regressor (DTR) [1]). We describe our dataset in detail in § 7.1, which includes workload metrics and CPU utilization information for 58 services in one month, collected at 10-minute intervals. Table 2 shows the results on this off-line dataset of the 58 services in terms of MAE, RMSE, and MAX_{error}, where MAX_{error} is the maximum error between the estimated value and the true value. Our proposed model achieves the best performance across all three evaluation metrics. Specifically, DeepScaling achieves 0.65 MAE, 0.81 RMSE, and 2.69 Max_{error}. For the most critical metric Max_{error}, DeepScaling is far lower than all the other compared methods. These results demonstrate the effectiveness of DeepScaling's CPU estimator.

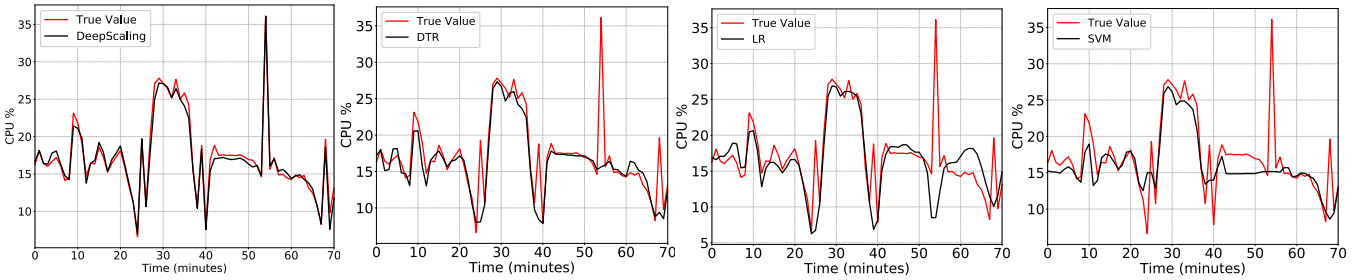
In order to further compare different estimation models, we plot the experimental results of all the estimators on the test set for services A1-A5 in Figure 5. We observe that traditional machine learning methods are not accurate in the complex situations we see, when the workload changes sharply. At time point 52 in Figure 5, the CPU utilization has a sudden increase due to the execution of periodic tasks. LR, SVM and DTR models perform poorly since they are unable to handle complex nonlinear data. DeepScaling efficiently learns the performance variations caused by non-traffic factors and gives a much more accurate CPU utilization estimation.

6 AUTOSCALING DECISION MAKING

6.1 Decision with Reinforcement learning

The function of the decision maker for autoscaling is to find the recommended number of instances for a microservice. It utilizes an RL model as shown in the 3rd box of Figure 2.

Problem Formulation. Given the average CPU utilization estimate for a microservice that is deployed across a large number of pods, we formulate the decision making problem of auto-scaling as one of Reinforcement Learning (RL). We consider the variation of CPU utilization during the running of a microservice as the *environment* and take the CPU utilization estimation model as the environment *observer*. The



(a) DeepScaling (b) DTR (c) LR (d) SVM

Figure 5: Actual CPU vs. Estimated CPU utilization using DeepScaling's method compared to other alternatives

Table 3: DQN Training Parameters

Parameter	Value
Time Steps	500
Minibatch	64
Size of Experience Replay	10^5
Learning Rate(MainNet)	5×10^{-5}
Learning Rate(TargetNet)	5×10^{-5}
Discount Factor	0.9
Exploration Factor	$\varepsilon(0.9)$,$\gamma(0.9)$

RL model will continually autoscale the instance count of pods based on the environment *observer*. The goal of the RL model is to keep the average CPU utilization stable despite variations in the workload. We consider the scaling decision for the microservices as a policy optimization problem with a policy function p that maps from the state space S to the action space A, $p : S \rightarrow A$.

Specifically, we build a shared RL model for all microservices. The input of the RL model is a tuple, consisting of the CPU utilization and the service-ID $S = (CPU, ID)$, and outputs the latest instance count n after performing the recommended action A. Then, we use the CPU utilization estimation model to estimate the new CPU utilization and update the state S_{t+1}. This process loops until the CPU utilization reaches the pre-set target level. The RL model outputs the instance count n as the final result.

State Space. In our model, the environment state includes both the CPU utilization of the service and the service-ID. The service-ID, as described section in § 5.1, is an integer, ranging from 0 to k. The CPU utilization from the monitoring system is also a number ranging from 0 to 100. For simplicity, we round up the CPU utilization to an integer. Combining the service-ID and CPU utilization together, the size of the state space for the whole RL model is thus $k * 100$.

Action Space. Our action space includes three different actions: increased, decreased, and unchanged.

- *Increased*: This action is to increase the instance count, which is realized in two ways: increasing the instance count proportionally (e.g., 5%) or increasing the instance count by a certain number (e.g., 10).
- *Decreased*: This action reduces the instance count. Again, it can be implemented in two ways: reducing the instance count proportionally (e.g., 5%) or reducing the instance count by a certain amount (e.g., 10).
- *Unchanged*: This action keeps the instance count of the service unchanged.

Learning the Optimal Policy. The policy optimization problem ($p : S \rightarrow A$) is solved with a Deep Q-Network (DQN). The pseudo-code of the training algorithm is shown in Algorithm 2. We introduce an experience replay buffer [32], to solve the dependency issue among contextual samples. In practice, we add the service-ID to the experience replay buffer $(ID, s_t, a_t, r_t, s_{t+1})$ so that it can better distinguish samples between different services, where s_t, a_t, r_t are state, action and reward at time-step t, respectively.

Suppose the value function is $Q(s = cpu, a; \Phi)$ with parameter Φ, DQN defines the loss function [29] as:

$$L(\Phi) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \Phi_{i-1}) - Q(s, a; \Phi_i))^2], \quad (16)$$

where state $s = cpu$ is the CPU utilization, a is the action taken by the DQN model, and r is the reward which we described below.

Similar to [32], DQN consists of two networks (MainNet and TargetNet) with the same structure but different parameters as shown in Figure 2. $Q(s, a; \Phi_i)$ represents the MainNet with the latest parameters, which is used to evaluate the value function of actions a. $Q(c', a'; \Phi_{i-1})$ represents the TargetNet, which provides the target value to avoid overfitting. The parameters of TargetNet are the time-delayed version from MainNet. That means we copy the parameters of MainNet to TargetNet at every fixed step.

DeepScaling aims to reach a stable value for the CPU utilization as quickly as possible with the RL model. We

Algorithm 2 DQN Training Algorithm

Initialize replay buffer D to capacity N
Initialize main action-value function Q with random weights Φ
Initialize target action-value function $\hat{Q}$ with weights $\Phi^{-} = 0$
for episode = 1→M do **do**
  for t=1→T do **do**
    With probability ε select a random action a_t
    otherwise select $a_t = argmax_a Q(s, a; \Phi)$
    Execute action a_t in emulator and observe reward r_t and image s_{t+1}
    Store transition $(ID, s_t, a_t, r_t, s_{t+1})$ in D
    Sample random mini-batch transition from D
    Set $y_j = \begin{cases} r_t & stop\ at\ step\ t+1 \\ r_t + \gamma max_{a'} \hat{Q}(s_{t+1}, a'; \Phi^{-}) & otherwise \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_t, a_j; \Phi))^2$with respect to the network parameters Φ
    Every n setps reset $\hat{Q} = Q$
  end for
end for

define the reward function as follows:

$$r = \begin{cases} -\delta - |s_{t+1} + \delta| & |s_{t+1} - \delta| > |s_t - \delta|. \\ \delta - |s_{t+1} - \delta| & otherwise. \end{cases} \quad (17)$$

where δ is a positive value representing the target CPU utilization level of the service. When s_{t+1} is far away from the target level, DQN gets a negative reward. Otherwise, when s_{t+1} is close to the target level, DQN gets a positive reward.

6.2 Implementation Details

We implemented the DQN based scaling decision algorithm in TensorFlow. The DQN network combines MainNet and TargetNet as shown in Figure 2. MainNet contains two hidden fully connected layers with [256,128] hidden units, all using the ReLU activation function. TargetNet has an identical network structure to MainNet. Hyper-parameters of the DQN model are listed in Table 3.

DeepScaling usually performs model updating/refreshing every two weeks in our production environment. Once there is a new microservice introduced between these model refresh events, we need to train the CPU estimation model and decision maker model (2nd and 3rd step of DeepScaling) for that microservice since these two steps require the service-ID information. On the other hand, the workload forecasting model can be reused. Training models for these two steps requires a small amount of data for that microservices which can be obtained at a relatively small cost since the training procedure actually involves fine-tuning based on existing models. During the model update period, expired services are removed, new services are integrated, and the whole model parameters of DeepScaling are refreshed.

For such a complex 3-step DeepScaling system, we use the following ways to debug errors [13], and verify the correctness. The first two steps (workload forecasting and CPU utilization estimation) can be extensively evaluated with collected data, with their accuracy guaranteed through evaluation. Then, we use a simulation environment with real traffic to evaluate the correctness of the scaling-decision maker as well as the overall DeepScaling system.

7 EVALUATION AND DEPLOYMENT

7.1 Experimental Setup

Datasets. For the workload metrics presented above, we categorized 3000 different microservices of the Ant Group according to their workload behaviors. Among them, 38.2% of the microservices are dominated by RPC requests, 26.96% of the microservices are dominated by DB access workload, 17.41% of the microservices are dominated by File I/O, 14.60% of the microservices are dominated by a workload of MSG requests, and 2.80% of the microservice are dominated by Page-Viewing. To validate the wide applicability of DeepScaling, we selected 58 different kinds of real microservices from Ant Group. The main task of these production microservices is to provide a high availablility online payment platform, and the services are usually accessed more than 500 million times everyday. We collected their workload data and CPU utilization data for one month.

All the 58 microservices are used to evaluate our workload forecasting and CPU utilization estimation. For evaluating the whole pipeline of DeepScaling, we selected five microservices which forms a minimal, and full-functional chain. Table 4 illustrates the workload characteristics of these five microservices. A3 is the front end of this service chain with an average of 290,000 visits per minute. The number of instances of these microservices is set by the administrator to 1800 when not using any resource autoscaling methods. It is interesting to see that these five microservices have different workload characteristics. A1 is a database dominated microservice. A2 is a messaging middleware. A3 is a web page microservice with large number of Page Views (§ 4.1). A4 is a RPC-in dominated microservice. A5 is a core file microservice with significant File I/O and msg-sub.

Implementations Environment. We implemented DeepScaling based on TensorFlow 1.13.1, and adopted the Deep Network implementation provided with TensorFlow-probability 0.6.0. All experiments run on a Linux server (Ubuntu 16.04) with Intel(R) Xeon(R) Silver 4214 2.20GHz CPU, 16GB memory and a nVidia V100 GPU.

Table 4: Workload metrics of the Sample Service (Times/minute)

NO.	RPC-in	RPC-out	Msg-pub	Msg-sub	DB-Access	File I/O	PV
A1	6.7×10^5	3.4×10^7	2.7×10^5	3.5×10^5	7.7×10^8	3.7×10^6	1.9×10^3
A2	2.5×10^5	0	1.3×10^7	3.4×10^7	2.0×10^8	7.1×10^5	0
A3	4.8×10^4	6.4×10^6	1.4×10^4	1.9×10^4	6.8×10^5	2.1×10^5	2.9×10^5
A4	1.4×10^6	0	0	1.0×10^5	8.0×10^6	3.2×10^5	0
A5	4.1×10^5	9.3×10^5	0	2.8×10^6	1.1×10^6	1.2×10^6	0

Measurement. We introduce the relative CPU stability rate (RCS) and the relative resource utilization (RRU) metrics to measure the performance of DeepScaling. RCS is defined as

$$RCS = y_t/1440. \tag{18}$$

where y_t is the number of minutes in a day when the service is in a stable range when the CPU utilization fluctuates around the target level, within the range of the buffer value, $\tau = 5\%$ (as in Line-18 of Algorithm 1). RRU is defined as

$$RRU = c/c_r, \tag{19}$$

where c is the instance count for the particular method being evaluated, and c_r is the instance count set by the rule-based autoscaling method.

State-of-the-art Autoscaling Approaches. We compared DeepScaling against three state-of-the-art (SOTA) autoscaling approaches: Autopilot, FIRM and the rule-based approach, in simulation experiments. Note the rule-based approach is also used in our production comparison, where the rules are developed from our experts based on historical experiences. A simple categorization of SOTA autoscaling approaches are listed as below.

(1) Workload-based autoscaling approach (Autopilot): Google's autoscaling approach, named Autopilot [37], builds the optimal resource configuration by seeking the historical time window that matches best with the current window.

(2) SLO-based autoscaling approach (FIRM): FIRM [36] is a RL-based autoscaling approach, which learns to adjust resources based on feedback. It finds services with abnormal response times (RT) through an anomaly detection algorithms and adjusts multiple resources for the service through RL iterations.

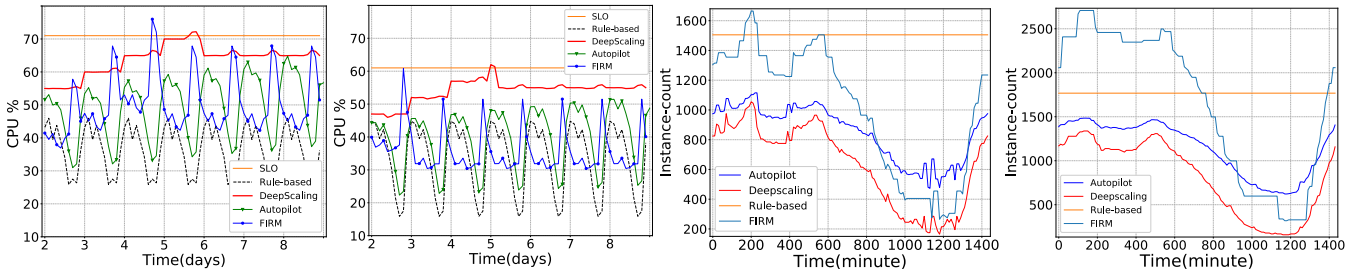
(3) Rule-based autoscaling approach: It dynamically scales the instance number of a microservice based on monitored performance metrics and using the expert's rules.

7.2 Overall Performance of DeepScaling

Before production deployment, we made a comprehensive performance evaluation of DeepScaling against the rule-based approach, Autopilot, and FIRM, for the minimal service-chain A1-A5 in a simulation environment.

Service Stability. We compare DeepScaling with several existing approaches. First is a rule-based approach involving manual scaling. The second is Autopilot [37], proposed by Google, using a history window of 24-time slices. The third is FIRM, using the SVM algorithm to determine whether there are abnormal response time values. FIRM's optimization goal is to minimize the probability of occurrence of such abnormal values. The CPU utilization target levels for service A1 was initially set at 50% and 47% for A2, based on their historical record of maximum values. The target level is increased $\delta = 5\%$ daily by the target level controller until we reach very close to the SLO. We show the results of these two services in Figure 6, while providing the overall summary results for all the five services in Figure 7 (due to space limitations, we don't go through all the other services). Each approach takes a certain amount of time to achieve a stable target CPU utilization. We consider a service as having achieved a stable operating range when the CPU utilization fluctuates around the target level, $\tau = 5\%$. In our experiments, the SLO monitor detects a large number of request timeouts when service A1's CPU utilization exceeds 71%, and a large number of GC (garbage collection) timeouts occur, and when service A2's CPU utilization exceeds 60%.

For service A1, as can be seen in Figure 6(a), the rule-based autoscaling method is unable to accurately estimate the resources required for the service. This results in a dramatic fluctuations of the CPU utilization each cycle. For FIRM, the CPU utilization exceeds 70% on the second day, although the target level was set at 55% at this time. The SLO monitor detected abnormal response time values. Eventually, the Target Level Controller sets the target level of FIRM to 50%, to allow it to run at that value from that point on without too many SLO violations. Autopilot, with a target level 80%, achieved a maximum service CPU utilization of 62% with large fluctuations even after seven days. Meanwhile, DeepScaling reaches close to the SLO desired, setting the final target CPU utilization level at 65%. A similar situation is also observed with service A2, as shown in Figure 6(b). The target level for FIRM was finally set to 42%. In contrast, DeepScaling reaches close to the desired SLO on the third day, with the final target level set at 55%. The target level for Autopilot was set to 77%, but the maximum value of CPU utilization



(a) CPU (Service A1) (b) CPU (Service A2) (c) Instance count (Service A1) (d) Instance count (Service A2)

Figure 6: Effect of different autoscaling approaches for services A1 and A2.

only reached 55%. It is clear that DeepScaling makes services run closer to the desired SLO bound, for both service A1 and service A2, compared to the other alternatives.

Figure 7(a) shows the percentage of time in a day that each autoscaling approach maintains the CPU utilization in the desired target range. An approach that maintains the CPU utilization stable over a longer time period is superior to others as it allows services to operate at resource utilization closer to the limit with a smaller fluctuation range, helping maximize the aggregate multiplexed throughput. DeepScaling achieves and maintains a service's CPU utilization around the target level almost across the entire day (close to 98% for services A3, A4, and A5, and at 95% for A1 and A2). Taking the average over the entire service-chain A1-A5, DeepScaling improves the relative CPU stability rate (RCS) by 61.1%, 40.8%, and 24.6% compared to the rule-based approach, Autopilot and FIRM, respectively.

Right-sizing Resources for Each Service. Figure 6(c,d) shows the recommended instances provided by each method for services A1 and A2 on one day. Further, Figure 7(b) shows the number of instances used for each service by each approach relative to the rule-based autoscaling approach. After seven days of running all the autoscaling methods, we compare DeepScaling with the alternatives in terms of the amount of resources being allocated to serve the submitted workload. The rule-based approach uses domain knowledge expertise to allocate resources. We set the rule-based autoscaling approach to use 100% of instances. It is obvious from the figure that the learning-based approaches allow for a more flexible configuration of service resources. So these approaches have much higher resource-efficiency than the rule-based approach. From the perspective of resource saving (using the RRU metric), DeepScaling saves 49.4%, 20.2% and 14.0% more container resources compared to the rule-based approach, Autopilot and FIRM, respectively.

Effect of using Workload Forecasting. To verify the validity of using workload forecasting, this experiment was set up as follows. The target levels of service A1 and A2 are 37% and 36%, respectively. We compare DeepScaling with and without forecasting workload metrics. For the case without forecasting, we fetch the real-time workload metric from the Service Monitor.

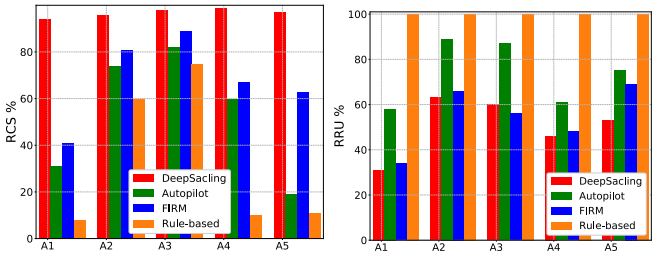
Figure 8 shows the benefit of workload forecasting. Without the forecast, we compute the recommended instances for the current workload in real-time and pass them to the instance controller. The instance controller takes about 5 minutes to start a microservice instance using the cold start method. As can be seen from Figure 8, the CPU utilization varies significantly in the absence of a forecaster. When workload forecaster is available, we can proactively start the right number of containers early and minimize the number of cold starts. When autoscaling with real-time workloads, the instance controller is usually unable to allocate the right service resources in time once the workload changes too quickly.

7.3 Production Deployment

After validating the effectivenes on the service-chain A1-A5, we deploy DeepScaling in our production environment at Ant Group. Currently, DeepScaling has already been adopted in a service subset consisting of **135** microservices, related to payment systems, for daily automatic resource provisioning management. In the more than one year of production use, DeepScaling has saved more than 30,000 CPU cores per day on average, compared to the previous rule-based solutions (measured in CPU core hours saved, we see on average 30K * 24 CPU core hours are saved each day). We expect to extend DeepScaling to support the provisioning of all of the online microservices (3000+) in Ant Group's production payment services in near future.

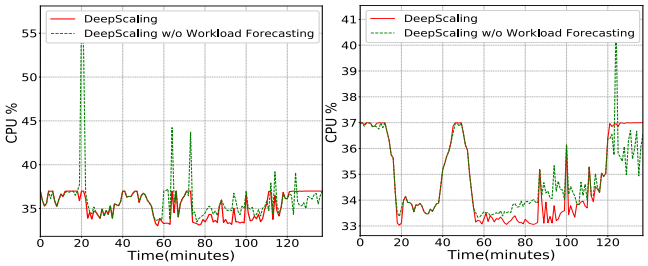
8 CONCLUSIONS

Existing autoscaling approaches for cloud platforms have difficulties in balancing the goal of minimizing resource allocation while avoiding large delays and SLO violations, thus resulting in significant over-provisioning and resource wastage.



(a) Relative CPU stability rate (b) Relative resource utilization

Figure 7: RCS and RRU for different autoscaling approaches.



(a) A1 (b) A2

Figure 8: DeepScaling with and without the Workload Forecasting step.

To overcome this challenge, we developed DeepScaling, a deep learning-based autoscaling approach that emphasizes the goal of stabilizing CPU utilization at a desired target level. DeepScaling consists of three innovative components: workload forecasting, CPU utilization estimation and an autoscaling decision-maker. We characterize the typical workload for production-level microservices using a comprehensive multi-dimensional set of metrics. The workload forecasting models the relationship between microservices and the metrics with Spatio-temporal graph neural networks, so that we can precisely forecast future workloads to avoid excessive latency from services having to suffer from the 'cold start' latency. Our comprehensive workload characterization also helps us achieve accurate CPU utilization estimation, so that the autoscaling decision achieves a stable CPU utilization at the target level. We conducted extensive experiments on a large-scale production cloud environment. DeepScaling achieves a stable CPU utilization for much longer periods (by 24.6%), and saves 14.0% more resources compared to the state-of-the-art autoscaling approach. DeepScaling is deployed in the production environment of Ant Group, and saves considerable resources per day in practice. All these confirm the practical applicability and scalability of DeepScaling.

ACKNOWLEDGEMENT

Xiaohong Zhang and Meng Yan were supported in part by the National Key Research and Development Project (No. 2021YFB1714200), the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538), the Postdoc Foundation of Chongqing (No. 2020LY13) and the research fund from Ant Group.

REFERENCES

[1] Muhammad Abdullah, Waheed Iqbal, Josep Lluis Berral, et al. 2020. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing* (2020).

[2] Muhammad Abdullah, Waheed Iqbal, and Faisal Bukhari. 2018. Containers vs virtual machines for auto-scaling multi-tier applications under dynamically increasing workloads. In *International Conference on Intelligent Technologies and Applications*. 153–167.

[3] Rie Kubota Ando and Tong Zhang. 2007. Learning on graph with Laplacian regularization. In *Advances in neural information processing systems (NIPS)*.

[4] Aleksandar Antonić, Martina Marjanović, Krešimir Pripužić, et al. 2016. A mobile crowd sensing ecosystem enabled by CUPUS: Cloud-based publish/subscribe middleware for the Internet of Things. *Future Generation Computer Systems* 56 (2016), 607–622.

[5] AWS. 2022. AWS auto scaling documentation. Retrieved June, 2022 from https://docs.aws.amazon.com/autoscaling/index.html

[6] Azure. 2022. Azure autoscale. Retrieved June, 2022 from https://azure.microsoft.com/en-us/features/autoscale/

[7] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. 2009. A reinforcement learning approach to online web systems auto-configuration. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[8] Tianfeng Chai and Roland Draxler. 2014. Root mean square error (RMSE) or mean absolute error (MAE)?–Arguments against avoiding RMSE in the literature. *Geoscientific model development* 7, 3 (2014), 1247–1250.

[9] Tao Chen, Rami Bahsoon, and Xin Yao. 2019. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Survey* (2019).

[10] Ludmila Cherkasova and Rob Gardner. 2005. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor.. In *USENIX Annual Technical Conference (USENIX ATC)*.

[11] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. 2011. Resource provisioning of web applications in heterogeneous clouds. In *USENIX conference on Web application development*.

[12] Guilherme Galante, Luis Carlos Erpen De Bona, Antonio Roberto Mury, et al. 2016. An analysis of public clouds elasticity in the execution of scientific applications: a survey. *Journal of Grid Computing* 14, 2 (2016), 193–216.

[13] Yu Gan, Mingyu Liang, Sundar Dev, et al. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 135–151.

[14] Panos Gevros and Jon Crowcroft. 2004. Distributed resource management with heterogeneous linear controls. *Computer Networks* 45 (2004), 835–858.

[15] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, et al. 2012. Optimal autoscaling in a IaaS cloud. In *International conference on Autonomic computing*. 173–178.

[16] Daniel Gmach, Jerry Rolia, L Cherkasova, et al. 2007. Workload analysis and demand prediction of enterprise data center applications. In *IEEE*

International Symposium on Workload Characterization (IISWC).

[17] Google. 2022. Google cloud load balancing and scaling. Retrieved June, 2022 from https://cloud.google.com/compute/docs/load-balancing-and-autoscaling

[18] Kim Hazelwood, Sarah Bird, David Brooks, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629.

[19] Abdul R Hummaida, Norman W Paton, and Rizos Sakellariou. 2016. Adaptation in cloud resource configuration: a survey. *Journal of Cloud Computing* 5, 1 (2016), 1–16.

[20] Waheed Iqbal, Matthew Dailey, and David Carrera. 2009. SLA-driven adaptive resource management for web applications on a heterogeneous compute cloud. In *IEEE International Conference on Cloud Computing*. 243–253.

[21] Waheed Iqbal, Mathew N Dailey, and David Carrera. 2015. Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications. *IEEE Systems Journal* 10, 4 (2015), 1435–1446.

[22] Waheed Iqbal, Matthew N Dailey, David Carrera, et al. 2011. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems* 27 (2011), 871–879.

[23] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, et al. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 783–798.

[24] Michael T Krieger, Oscar Torreno, Oswaldo Trelles, et al. 2017. Building an open source cloud environment with auto-scaling resources for executing bioinformatics and biomedical workflows. *Future Generation Computer Systems* 67 (2017), 329–340.

[25] Bingfeng Liu, Rajkumar Buyya, and Adel Nadjaran Toosi. 2018. A fuzzy-based auto-scaler for web applications in cloud computing environments. In *International Conference on Service-Oriented Computing*.

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

[27] Chengzhi Lu, Kejiang Ye, Guoyao Xu, et al. 2017. Imbalance in the cloud: An analysis on alibaba cluster trace. In *IEEE International Conference on Big Data*. 2884–2892.

[28] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, et al. 2021. Mu: an efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *ACM Symposium on Cloud Computing (SoCC)*. 168–181.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. 2012. Playing atari with deep reinforcement learning. In *Advances in neural information processing systems (NIPS)*.

[30] Rafael Moreno-Vozmediano, Rubén S Montero, Eduardo Huedo, et al. 2019. Efficient resource provisioning for elastic Cloud services based on machine learning techniques. *Journal of Cloud Computing* 8, 1 (2019), 1–18.

[31] Boris N Oreshkin, Dmitri Carpov, Nicolas Chapados, et al. 2020. N-BEATS: Neural basis expansion analysis for interpretable time series forecasting. In *International Conference on Learning Representations (ICLR)*.

[32] Ian Osband, Charles Blundell, Alexander Pritzel, et al. 2016. Deep exploration via bootstrapped DQN. In *Advances in neural information processing systems (NIPS)*.

[33] Sourav Panda, K. K. Ramakrishnan, and Laxmi N Bhuyan. 2021. pMACH: Power and Migration Aware Container scHeduling. In *IEEE International Conference on Network Protocols (ICNP)*. 1–12.

[34] Jinwoo Park, Byungkwon Choi, Chunghan Lee, et al. 2021. GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *International Conference on emerging Networking EXperiments and Technologies*. 154–167.

[35] Issaret Prachitmutita, Wachirawit Aittinonmongkol, Nasoret Pojjanasuksakul, et al. 2018. Auto-scaling microservices on IaaS under SLA with cost-effective framework. In *International Conference on Advanced Computational Intelligence*. 583–588.

[36] Haoran Qiu, Subho S Banerjee, Saurabh Jha, et al. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 805–825.

[37] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, et al. 2020. Autopilot: workload autoscaling at Google. In *European Conference on Computer Systems (EuroSys)*. 1–16.

[38] Upendra Sharma, Prashant Shenoy, Sambit Sahu, et al. 2011. A cost-aware elasticity provisioning system for the cloud. In *International Conference on Distributed Computing Systems (ICDCS)*. 559–570.

[39] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 733–750.

[40] Akshitha Sriraman and Thomas F Wenisch. 2018. μsuite: a benchmark suite for microservices. In *IEEE International Symposium on Workload Characterization (IISWC)*. 1–12.

[41] Sonja Stüdli, M Corless, Richard H Middleton, et al. 2015. On the modified AIMD algorithm for distributed resource management with saturation of each user's share. In *IEEE Conference on Decision and Control (CDC)*. 1631–1636.

[42] Xiaoyang Sun, Chunming Hu, Renyu Yang, et al. 2018. Rose: Cluster resource scheduling via speculative over-subscription. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 949–960.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. 2017. Attention is all you need. In *Advances in neural information processing systems (NeurIPS)*. 5998–6008.

[44] Bing Yu, Haoteng Yin, and Zhanxing Zhu. 2017. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *International Joint conference on Artificial Intelligence (IJCAI)*.

[45] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic scaling for microservices with an online learning approach. In *IEEE International Conference on Web Services (ICWS)*. 68–75.

[46] Liang Zhou, Laxmi N Bhuyan, and K. K. Ramakrishnan. 2019. Goldilocks: Adaptive resource provisioning in containerized data centers. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[47] Liang Zhou, Laxmi N Bhuyan, and K. K. Ramakrishnan. 2020. Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.