RunWild: Resource Management System with Generalized Modeling for Microservices on Cloud

Sunyanan Choochotkaew
IBM Research
Tokyo, Japan
sunyanan.choochotkaew1@ibm.com

Tatsuhiro Chiba
IBM Research
Tokyo, Japan
chiba@jp.ibm.com

Scott Trent
IBM Research
Tokyo, Japan
trent@jp.ibm.com

Marcelo Amaral
IBM Research
Tokyo, Japan
marcelo.amaral1@ibm.com

2021 IEEE 14th International Conference on Cloud Computing (CLOUD) | 978-1-6654-0060-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/CLOUD53861.2021.00079

***Abstract*—Microservice architecture competes with the traditional monolithic design by offering benefits of agility, flexibility, reusability resilience, and ease of use. Nevertheless, due to the increase in internal communication complexity, care must be taken for resource-usage scaling in harmony with placement scheduling, and request balancing to prevent cascading performance degradation across microservices. We prototype *RunWild*, a resource management system that controls all mechanisms in the microservice-deployment process covering scaling, scheduling, and balancing to optimize for desirable performance on the dynamic cloud driven by an automatic, united, and consistent deployment plan. In this paper, we also highlight the significance of co-location aware metrics on predicting the resource usage and computing the deployment plan. We conducted experiments with an actual cluster on the IBM Cloud platform. RunWild reduced the 90th percentile response time by 11% and increased average throughput by 10% with more than 30% lower resource usage for widely used autoscaling benchmarks on Kubernetes clusters.**

***Index Terms*—microservices, cloud, autoscaling, scheduling algorithm, workload modeling**

I. Introduction

Microservices leverage message-passing protocols to uncouple service functions in an application and enable many advantages over the conventional monolithic approach including the ability to independently optimize the technology stack and scale hot functions. In a modern cloud, each microservice is packaged as a container running independently and seamlessly in a cluster managed by an orchestration system such as Kubernetes [1] or Docker Swarm [2]. Such orchestrators typically provide autoscaling mechanisms to reactively respond to changes in the workload CPU and memory usage (accordingly to a threshold) by increasing or decreasing the number of running instances of the containers, technically known as *horizontal scaling*, or modifying the containers' resource request, technically known as *vertical scaling*. The generally accepted practice is to denote horizontal scaling using the X-axis and vertical scaling in the Y-axis. Additionally, in this paper, we further append another dimension that is the network capacity proportion divided among duplicated microservice replicas on the Z-axis.

According to López and Spillner [3], incremental improvement in actual performance stops at a certain point in the scaling of both resource quota and number of replicas. This observation matches preliminary experiments we conducted on the widely-known horizontal pod autoscaler, *HPA* [4]. In

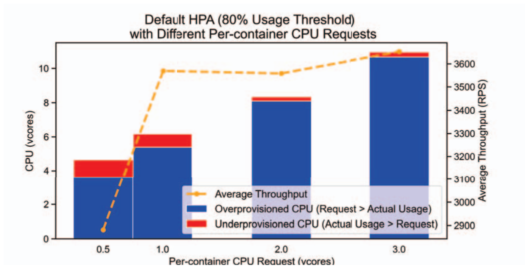

Fig. 1. Reactive horizontal autoscaler

our experiment, we varied the per-container resource allocation (quota) of the benchmark application. The results in Fig. 1 show that more resource quota causes resource waste; however, this does not always yield the same growth in performance with HPA. In other words, overprovisioning is not efficient after some point as explained by the law of diminishing returns, whereas underprovisioning can degrade performance or even cause fatal service failures. Kubernetes also provides a vertical pod autoscaler, *VPA*, to automatically adjust resource requests for gradual changes. However, Kubernetes's *VPA* and *HPA* are not made to work together. They can interfere with each other and decrease scaling efficiency as there is no mechanism for them to cooperate [5]. In addition to scaling strategy, placement scheduling and load-balancing strategy are also factors that affect performance in a shared infrastructure, especially for microservices deployment. The current load balancer has its own goal to automatically balance the workload among replicated containers. Yet, it does not take into account the nested impact to workload at runtime in the X- and Y- axes. Our motivation is to identify, describe, and manage all factors and dimensions to enable a united deployment solution instead of running mutual-interfering mechanisms.

The deployment problem is a variation of the generalized assignment problem which is composed of two fundamentals: the objective values to minimize (or maximize) and the constraints. The objective value is ideally performance over price [6]. This requires sufficient variety of performance profiling to calculate. Instead, to reduce complexity, some researchers use the number of replicas or amount of resource usage to only represent price [3], [7]. Nevertheless, due to microservice characteristics, there is an indirect metric that can indicate performance improvement by superiority of intrahost over inter-host communication throughput and latency among microservices. The constraint in resource provisioning is that the usage allocated must be less than the capacity. As usage is not trivial to determine, commonly used approaches model the expected usage from the past usage with regression techniques [8]. Nonetheless, this analysis requires the profiles of actual processing and message requests (i.e., workload) running on each deployed node in advance. On top of that, co-located processes on the deployed node can also be a factor that influences the usage prediction results [9].

In this paper, we propose *RunWild*, a resource management system that solves the optimization problem for microservice deployment with the following contributions.

I) Extensive Deployment Framework: The framework we designed is placed on top of the de facto cloud platform standard, Kubernetes [10], to control all deployment mechanisms including those during both allocation and runtime phases.

II) Generalized Usage Modeling Approach: This combines features of service characteristics, node-independent workload, and co-location aware node status to predict resource usage by integrating clustering and regression techniques.

III) Inter-Microservice Objective Metric: A metric called *cohesion* to reflect the advantages of placing highly inter-communicating microservices on the same node in addition to a resource usage factor in the objective function that we formulated for the allocation problem.

IV) Runtime Workload Partitioning through Service Mesh Utilization: The traffic routing and control ability of a service mesh operator to partition the workload to comply with the computed allocation is leveraged.

For evaluation, we used *BLUEPERF* [11], an open-source project to analyze the performance of microservice deployment, and deployed it on a Red Hat OpenShift platform [12]. We compared RunWild with the HPA, and the bin packing solution without considering inter-microservice communication. With 24% lower resource, the results indicate that RunWild can reduce response time by 25% and increase throughput by 37% compared with HPA. Even if we provision 10% more vertical resource with HPA, RunWild will still outperform by 11% in throughput and 10% in response time. The inter-microservice communication metric showed a 26% improvement in throughput and a 11% reduction in response time.

The rest part of this paper is organized as follows. We first clarify the problem statements regarding the microservice-deployment process in Section II and explain our considerations regarding recent state-of-the-art studies in Section III. We then give an overview of RunWild and our key innovations in Section IV. Next, we discuss experiments we conducted with an actual cluster on the IBM Cloud platform to evaluate RunWild and present the experimental results in Section V and discuss the current limitations of RunWild and future work in Section VI. Finally, we conclude the paper in Section VII.

II. Problem Statements

We consider the viewpoint of service deployment on the cloud with three dimensions. The first dimension is duplication of the service, e.g. the number of replicas. The second dimension is resource allocation inside the worker instance (node) for each duplicated replica. The third dimension is network capacity controlled for each duplicated replica. These three dimensions must be well-aligned towards the same goal to achieve a desirable deployment solution. Particularly, this paper provides solutions for the following problem statements.

Statement *P-I*: Scaling decisions including horizontal duplication, resource allocation, placement scheduling, and load balancing are interdependent. Various combinations can have an unanticipated impact on performance. Instead of treating these decisions as disjoint control knobs, they should be considered together to provide desirable performance as partially affirmed in [13] on horizontal and vertical scaling mix.

Statement *P-II*: In addition to the nature of services, resource usage as well as quality of service can be sensitive to the co-location such as resource occupancy of the deployed node. In [9], Nathuji et al. contrasted performance interference and varied CPU usage with performance when running a different number of threads on the same node. According to their study, it is difficult to eliminate interference and achieve perfect performance isolation due to complexity and desirable optimization on resource utilization. Interference should be taken into account to realize accurate resource modeling.

Statement *P-III*: Unlike monolithic services, the internal communication between microservices inside the cluster also affects overall service performance such as response time and throughput. Microservice-affinity consideration can improve performance as shown in [14]. Accordingly, the objective values for optimization in the allocation problem should include the benefit of reduced communication across deployed nodes.

Statement *P-IV*: The ability to control the network load to each specific duplicated replica is missing in current resource management systems. Although service mesh technology provides the ability to control network traffic via a sidecar [15], there is no mechanism for coordination.

III. Related Work

The basic approach to deal with microservices on the cloud is to scale-out (or in) when the monitored metric is over (or under) a threshold. The widely-used threshold is resource usage. Nevertheless, message-queue metrics have also been suggested as an alternative to avoid fluctuations from external factors [16], [17]. The scaling decision can be made by a central auto-scaler or by multiple decentralized agents such as *Gru* in [18].

Instead of relying on a determined threshold, López and Spillner formulated an optimization problem in terms of replica combination in a microservice application for elastic horizontal scaling to find the point of no return according to their findings on the performance boundary of horizontal scaling [3]. However, in a large-scale deployment, it could be challenging to gather performance and cost results for all required scaling combinations from the actual experiments. Correspondingly, Gias et al. observed beneficial cases for cooperating between horizontal and vertical scaling in [6]. Based on the queuing-demand model, they proposed *ATOM*, a system to find an optimal scaling decision in terms of replica and CPU shares. Similarly, Sedaghat et al. formulated a price-performance relationship to estimate performance from the number and size of virtual machines [7]. They proposed an optimizing system, called Re-packing, to reconfigure the services on the cloud using pre-knowledge of expected request capacity. Instead of pre-modeling the usage, Alipour and Liu introduced an online workload pattern learning mechanism to forecast CPU usage using a combination of classification and linear regression modeling [8]. Rzadca et al. exploited the power of machine learning with multiple configurations running in parallel to achieve the best performing workload model [19]. This autoscaling system, called *Autopilot*, reduced the gap between the resource quota limit and actual resource usage from past performance. *Autopilot* supports both vertical and horizontal scaling; still, scaling decisions are executed independently. Meanwhile, Rossi et al. propose a solution using reinforcement learning to control horizontal and vertical decisions together with configurable goals including performance penalty, adaptation cost, and resource cost [13].

In addition to analyzing past profiles, Nathuji et al. proposed that variation in resource usage and performance is due to interference between co-located processes sharing the same host [9]. They proposed a control framework named Q-Cloud that tunes resource allocation to handle the performance interference effect. The interference effect is predicted by the multi-input multi-output linear model. With the same motivation, Shekhar et al. classified the performance interference level of co-located workloads and then applied Gaussian Processes to analyze the performance impact of vertical scaling [20].

For scheduling or placement, *ATOM* balances the maximization of the weighted number of completed transactions per unit time with the minimization of total allocated CPU capacity relying on a layered queuing network of microservices [6]. To achieve the goal, a full profile of each transaction between microservices is required to parameterize the layered queuing network and determine the utilization demands of the application. Sampaio et al. instead introduce the placement remapping approach called *REMaP* that spotlighted potential improvement with placement strategy by using only the amount of messages exchanged between microservices, which is a deployment-independent metric [14]. However, this system does not make room for a dependent microservice by offloading an independent microservice to another node. For implementation, they do not cover how the workload is partitioned and distributed among the replicated containers.

To summarize, the relationship of the above studies to the problem statements introduced in Section II is presented in Table I. Regardless of metrics or decision makers in [3], [16]–[18], threshold-based approaches do not cover any of problems stated above. Re-packing [7], ATOM [6], online workload pattern learning [8], and reinforcement learning based solution [13] partially cover problem statement *P-I*. Still, they mainly focus on the combination of horizontal and vertical scaling but do not integrate a mechanism to control placement scheduling and load-balancing decisions. For problem statement *P-II*, Q-Clouds [9] and interference-aware vertical elasticity in [20] analyze the performance interference effect from co-located processes sharing the same host. For problem statement *P-III*, *ATOM* [6] and *REMaP* [14] also take the intercommunication of microservice into consideration. However, there are issues left regarding the practicality of performance profiling and limitation on post-making decision. For problem statement *P-IV*, none of the related studies defined or implemented partitioning for fractional workload assignment.

TABLE I
COMPARISON OF RELATED WORK

System	**Problem Statements**			
	P-I	***P-II***	***P-III***	***P-IV***
Elastic Horizontal Scaling [3]	×	×	×	×
Overload Control for Scaling [16]	×	–	×	×
Queue Metrics for Scaling [17]	×	–	×	×
Gru [18]	×	×	×	×
AutoPilot [19]	×	×	×	×
Re-packing [7]	✓	×	×	×
ATOM [6]	✓	×	✓	×
Online Workload Pattern Learning [8]	✓	×	×	×
Reinforcement Learning [13]	✓	×	×	×
Q-Clouds [9]	×	✓	×	×
Interference-aware Vertical Elasticity [20]	×	✓	×	×
REMaP [14]	×	×	✓	×
RunWild (Our proposal)	✓	✓	✓	✓

✓ considered, × not considered, – out of scope

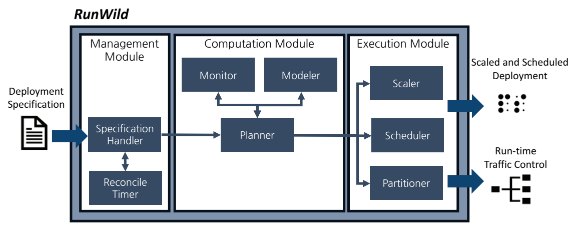

Fig. 2. Designed architecture

IV. RunWild

RunWild provides a deployment solution for microservices on a cloud platform that fully takes into account all steps in the deployment process. The deployment process starts by determining the number of replicas, the amount of resource to be reserved, which node to put each replica on, and how to partition the workload. Whereas most resource management systems run each of these processes independently, RunWild was developed to synchronize these decisions together to increase the degree of optimization on deployment.

RunWild follows an autonomic monitor-analyze-plan-execute cycle [21]. In the monitor and analyze phase, we prepare a resource-usage model to predict the resource usage. The microservices are clustered into a predefined number of clusters based on their standardized workload features. For each type of cluster, we exploit automated AI technology to get an optimized regression model to predict resource usage considering the amount of message requests and interference-aware factor that is node occupied status (corresponding to problem statement *P-II*). In the planning phase, we find an optimized solution for the resource-allocation problem. In particular, we introduce the integration of the cohesion metric, an inter-microservice communication, as input to the ladder-step bin packing resource-allocation problem, and compute a deployment plan to apply to all mechanisms including horizontal duplication, resource allocation, placement scheduling, and load balancing (corresponding to problem statements *P-I* and *P-III*). For the execution phase, we implement RunWild on top of the de facto cloud platform standard. We design the extensive deployment framework with a new operator to dynamically execute the scaling and scheduling decision from the planning phase. We leverage runtime traffic control on the service mesh to dynamically partition the desired workload with customized labeling (corresponding to problem statement *P-IV*).

Accordingly, we designed the RunWild architecture as shown in Fig. 2. There are three modules of management, computation, and execution. The management module first handles the deployment specification from users. The computation module computes a deployment solution for every periodic window. The execution module operates complying with the computed solution. Explanations of each component are as follows.

- **Specification Handler** manages the inputted deployment specifications and automates the planning computation.
- **Reconcile Timer** counts and triggers the deployment automation process for each inputted specification.
- **Monitor** monitors the resource usage and workload on each replica. We consider workload to be the amount of message-passing requests.
- **Modeler** models and predicts resource usage.
- **Planner** computes the deployment plan, which consists of the number of replicas, node placement, resources to be reserved, and workload weight for partitioning.
- **Scaler** updates the replica controller to achieve the planned replicas with planned resource reservations.
- **Scheduler** assigns the containers to the planned node.
- **Partitioner** configures traffic control mechanism to partition the planned workload.

A. Resource Usage Model

Modeling resource usage involves two steps: (i) clustering arbitrary services into groups using machine learning, and (ii) generating a best-fit regression model for each group and each type of resource. The first step aims at reducing the modeling overhead. The second step aims at finding the model that accurately predict resource consumption behavior of each microservice. Without loss of generality, RunWild focuses on the two critical resource types of CPU and memory.

1) Microservice Clustering: Microservices have a high variety of resource usage behavior. A larger number of clusters can increase modeling granularity; at the same time, it increases model-updating overhead. Accordingly, we apply the machine-learning technique of clustering to reduce the number of prediction models with acceptable clustering performance. We use k-means clustering [22], which is simple but efficient to cluster microservices based on usage and request features. Specifically, the features used are CPU usage, memory usage, and the amount of message requests rated as minimum, average, maximum, as well as 90, 95, and 99 percentiles.

To determine the number of k-means clusters, we formulate the total clustering score (κ) to reflect the clustering performance using the distance from cluster centers (normalized sum of squared errors: $\widehat{SSE}$), normalized imbalance of training set ($\widehat{\theta}$), and normalized minimum size of training set ($\widehat{\sigma}$), as shown below, where w_0, w_1, and w_2 are important weight for each factor respectively. The default value is equal to 1; but, they can be tuned in a future study.

$$\kappa = \frac{w_0(1-\widehat{SSE}) + w_1(1-\widehat{\theta}) + w_2(1-\widehat{\sigma})}{w_0 + w_1 + w_2} \tag{1}$$

2) Prediction Model: Each type of microservice can have a different behavior of resource usage. For instance, the CPU usage of one microservice can be accurately predicted by an optimum selection of Random Forest technique while that of another microservice can be better explained by adding randomization in Extra Tree. To choose the best-fit model, the target function is root mean square error if the explained variance is above the threshold which is set to 0.5. Otherwise, the model with maximum explained variance will be picked. Particularly, we use AutoAI to obtain a best-fit model for each microservice cluster and each type of resource. AutoAI is a tool that fully automates model generation and identification of the best-fit modeling techniques for the provided data collection. This tool evaluates multiple combinations of algorithm selection, feature engineering, and hyperparameter optimization [23]. Feature engineering is used to extract new features from the input, and hyperparameter optimization is used to tune learning parameters. Regarding features, in addition to using workload-dependent metrics such as the amount of message requests, we also use node status metrics including current CPU and memory utilization.

B. Deployment Planning

With a prerequisite of resource usage model from the previous section, we define an objective function under the estimated resource constraint for the deployment-planning problem and present solutions as described below.

Problem Definition: Let a deployment request contain a set of target microservices ($T : [t]$) and their dependency matrix ($W_{T \times T}$) to a set of worker nodes (N) in the cluster where a set of services (S) is currently running. It can optionally provide a normal behavior of resource usage (R_0) and service load (L_0) for the initial state. If the normal behavior is not provided, the system default will be used. The deployment request can be new to the cluster or can be called by the periodic timer for scaling and scheduling.

Presume the monitoring module and modeling modules provide the information described below.

- **Monitoring module** gives (i) past resource usage $R_{(n,s)}$ and (ii) expected service load $L_{(n,s)}$ generated by a replica for service $s \in S$ deployed in each node $n \in N$.

Algorithm 1 Find Plan B, P, U

```
Input: $T, W, R_0, L_0$
$S, R, L \leftarrow Monitor$
init $L_{map}, G_{map} = \phi$
STEP 1:
for each $t \in T$ do
  if $t \notin S$ then
    add $t \mapsto L_0$ to $L_{map}$
    add $t \mapsto Cluster(R_0, L_0)$ to $G_{map}$
  else
    add $t \mapsto L_{(n,t)}$ to $L_{map}$
    add $t \mapsto Cluster(R_{(n,t)}, L_{(n,t)})$ to $G_{map}$
    $R \leftarrow R - R_{(n,t)}$
  end if
end for
STEP 2:
init $B, P, U = \phi$
$X \leftarrow [(T, L, R, B, P, U)]$
$X_{next} \leftarrow X$
$A_0 \leftarrow Capacity_n - \sum_{(n,t)\in B} R_{(n,t)}$
for $i$ from 1 to $|N|$ do
  for $x \in X$ do
    $A \leftarrow A_0 - x.U$
    for $F \in Algorithms$ do
      $x' \leftarrow F(x, i, A, M_{G_{map}(t)}, L_{map}, G_{map})$
      if $|X_{next}| < k$ then
        add $x'$ to $X_{next}$ and sort $X_{next}$ asc
      else
        for $\gamma \in X_{next}$ do
          if $BETTER(x', \gamma, W)$ then
            replace $\gamma$ with $x'$
            break
          end if
        end for
      end if
    end for
  end for
  $X \leftarrow X_{next}$
end for
$\gamma \leftarrow BEST(X, W)$
$B, P, U \leftarrow \gamma.B, \gamma.P, \gamma.U$
```

- **Modeling module** predicts non-profile resource usage of the target microservice t on the deploying-target node n using (i) workload characteristic Group g, (ii) expected service load l, and (iii) the node n availability. Group g is determined by the clustering function explained in Section IV-A1. Load l is estimated by the standard time-series forecasting model *ARIMA* [24] from the past load collected by the monitoring module or the default characteristic of the classified group. The node n availability is derived by deduction of node capacity and monitored resource used by other services in the considering node. The predicted resource usage is denoted by $R'_{(n,t,l)} = M_g(l, n); t \in g$. $M_g(l, n)$ is the prediction model explained in Section IV-A2.

The three-dimensional deployment decision is denoted by a set of binding of a duplicated replica of target microservice t to node n, $B : [b = (n, t)]$. Each binding item b refers to what proportion of service load is partitioned (P_b) and how much resource is required for a specific binding item (U_b). We set two objective values: usage $\mathbb{U}$, and microservices cohesion $\mathbb{W}$. The microservices cohesion metric is the summation of partitioned workloads weighted with a pair dependency.

$$\mathbb{U} = \sum_{(n,t)\in B} U_{(n,t)} = R'_{(n,t,\lambda)}; \lambda = P_{(n,t)} \times L_{(n,t)} \tag{2}$$

$$\mathbb{W} = \sum_{n \in N} \sum_{\substack{(i,j)| \\ (n,i)\in B,(n,j)\in B, i\neq j}} P_{(n,i)} P_{(n,j)} W_{i\times j} \tag{3}$$

Suppose parameters α and β are the significance over normalized values of the resource usages and that of microservice dependency. When each node n has an allocatable $Capacity_n$, the goal is to decide on allocation and the corresponding partitioned workload and resource usage request such that

$$\begin{gathered} \min_{B,P,U} \alpha \cdot \hat{\mathbb{U}} + \beta \cdot (1 - \hat{\mathbb{W}}) \\ \text{subject to} \\ \forall_n \sum_{t \in T|(n,t)\in B} U_{(n,t)} + \sum_{t' \in S-T} R_{(n,t')} \leq Capacity_n \end{gathered} \tag{4}$$

The above objective function is an additive scoring model. However, any MCDA [25] model including the multiplicative model and comparative scale is applicable.

To solve this problem, instead of searching for an optimal value from all possibilities, which has a computational complexity of $O(|N||T|^{|N|})$, we introduce a heuristic algorithm with a complexity of $O(|N|^2|T|)$ (simplified in Algorithm 1). Furthermore, we present two methods for dynamic update: absolute and deferring. The allocation update performs every periodic window solving the above problem definition (for example, every 10 minutes). However, while the former does not take into account the previous decision, the latter starts with reduced migration to suppress overhead then scales down the number of replicas if there is enough room.

Absolute Update:
The absolute solution has two main steps. The first step is to take a snapshot of recent resource usage (R) and service loads (L) of deploying microservices from the monitoring module. Then, we find two mapping sets for each microservice to group index and to expected service load, G_{map} and L_{map}, respectively. G_{map} can be determined from the features inputted to the clustering model, as explained in IV-A1, denoted by $Cluster(R, L)$. If it is new to the worker cluster, $t \notin S$, the initial values of its group index will be applied. Otherwise, the monitored value will be applied.

The second step is to find an optimized allocation decision. The loop starts by varying the replica number of all demanded microservices from 1 to the maximum value (the total number of nodes). For each iteration, we obtain the top-k solution state, X. The state is composed of the remaining demanded microservices T, corresponding load L, occupied usage R, and latest decision from the previous replica limit, (B, P, U). Multiple techniques can be applied to perform optimization for each candidate $x \in X$. To reduce the computation time used to search for an optimal solution, we use a greedy-algorithm function F to compute the next state x' from the state x under the replica limit i and the resource constraint defined in Equation 4. $Capacity_n$ is the remaining resource A and $U_{(n,t)}$ is the resource usage predicted by the model $M_{G_{map}(t)}$, L_{map} and G_{map}. The remaining resource (node availability) considers not only the actual usage but also the reservation. The next top-k set X_{next} is determined by $BETTER$ function applied to results from all greedy algorithms. $BETTER$ function returns whether the objective value formulated in Equation 4 of the comparing binding state x' is higher than that of the considering binding state γ in the next top-k set. At the end, $BEST$ function is used to return the best binding state γ which obtains the minimum objective value among the latest top-k candidates. W is the microservice cohesion metric required for computing the objective value. The candidate functions we applied are a combination of applying demand values of large/small/random and availability values of large/small/random. As this is a fractional knapsack problem, we use a greedy approach to divide the load proportion by the maximum value of the resource usage that is still under capacity constraint.

Deferring Update:
This solution adds a step between Steps 1 and 2 of the absolute solution. The objectives are to defer re-allocation overhead and to reduce the waste replica. In particular, we sort the currently-deployed replicas and migrate them from workload minimum order to the deployed node with maximum workload until it breaks the node-capacity constraints. The deployed containers that are no longer subject to the constraints will be considered new demands. The solution is a three-way merger of the solution from Step 2 in Algorithm 1 with the new target set and that from the intervening step.

C. Implementation of RunWild

Corresponding to the designed architecture in Fig. 2, RunWild implements the Operator SDK framework provided by Red Hat [26] with a new custom resource called *µPlan* as shown in Fig. 3. Once clients submit their manifests to deploy, the specification handler will modify attributes to generate the *µPlan* automatically (Steps 1-3). *µPlan* contains deployment specifications and the latest deployment plan. The communication dependency between services is a service graph traced by the *Istio* service mesh. This *µPlan* activates the operator to perform the remaining components in computation and execution modules (Step 4).

The monitor of the computation module uses the range query API and summarizes the collected statistics from the *Prometheus* monitoring system [27] (Steps 5-6). The modeler component uses the prediction models, as explained in Section V-A (Steps 7-8). Based on information from the monitor

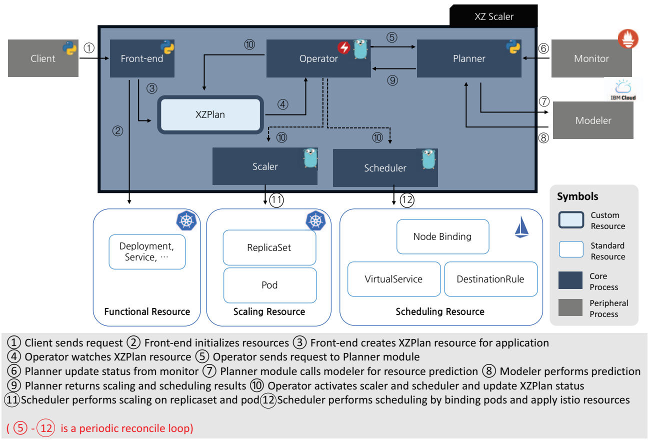
XZ Scaler
Client
Front-end
Operator
Planner
Monitor
IBM Cloud
Modeler
XZPlan
Scaler
Scheduler
①
②
③
④
⑤
⑥
⑦
⑧
⑨
⑩
⑪
⑫
Deployment, Service, ···
Functional Resource
ReplicaSet
Pod
Scaling Resource
Node Binding
VirtualService
DestinationRule
Scheduling Resource
Symbols
Custom Resource
Standard Resource
Core Process
Peripheral Process
① Client sends request ② Front-end initializes resources ③ Front-end creates XZPlan resource for application
④ Operator watches XZPlan resource ⑤ Operator sends request to Planner module
⑥ Planner update status from monitor ⑦ Planner module calls modeler for resource prediction ⑧ Modeler performs prediction
⑨ Planner returns scaling and scheduling results ⑩ Operator activates scaler and scheduler and update XZPlan status
⑪ Scheduler performs scaling on replicaset and pod ⑫ Scheduler performs scheduling by binding pods and apply istio resources
(⑤ - ⑫ is a periodic reconcile loop)

Fig. 3. RunWild Prototype Architecture

and modeler, we developed a planner component to process the deployment solution, as explained in Section IV-B (Step 9). The predicted usage is rounded up to megabytes for memory and hundreds of milli-cores for CPU to ignore small changes. The planner component is triggered by a creation event from *µPlan* and a reconcile timeout from the management module.

All components of the execution module are triggered by the solution returned by the planner component in the computation module (Step 10). The scaler calls the patching API to update the replica in service deployment (Step 11). We performed scheduling and partitioning together by implementing a scheduler API to schedule pending containers with scheduler name configuration (Step 12). For workload partitioning, we leveraged the traffic routing control of the *Istio* service mesh [15]. There are three steps to achieve this. The first step involves labeling each container with its assigned node. The second step involves creating corresponding routing destination subsets of service and rules for each node-assigning label. The third step involves creating a virtual service with calculated network load for each corresponding subset.

V. Experiments and Results

We conducted two experiments. The first was on our clustering and modeling approach which is for the usage prediction model corresponding to problem statement *P-II*. The second experiment involved evaluating the system performance corresponding to problem statements *P-I*, *P-III*, and *P-IV*.

The test bed was a cluster with 8 OpenShift 4.3.27 worker nodes running Red Hat Enterprise Linux AMD64 Kernel 3.10.0 managed by IBM Cloud. Each node was configured with 4 vCPUs and 16 GiB of RAM. The cluster was operated with an *Istio* service mesh and the *Prometheus* monitoring system.

A. Microservice Clustering and Resource Usage Modeling

We collected data for 3 days on the cluster with 70 microservices, 600 replicated containers, and 38,000 data points to classify microservices into groups. To determine the best number of clusters, we applied k-means clustering on the collected data and computed the score according to Eq. 1 in

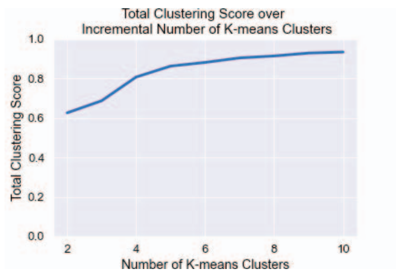

Fig. 4. Scoring result used to determine the number of clusters

Section IV-A1. The scores are plotted in Fig. 4. The result showed that using 4 clusters had 17% better clustering score (κ) than using 3 clusters, and using more than 4 clusters only showed less than 7% improvement. Even if a larger number of clusters performs better, there is a cascading overhead cost of modeling, predicting, and planning for the deployment process. Accordingly, the number of clusters is set to 4 as a trade off between clustering performance improvement and incremental overhead caused by the number of clusters. The center points of each clustered group can be found in Table II. Microservices classified into Group 0 tended to have a smaller number of requests but used more resources compared with those classified into Group 1. While microservices in Group 3 were CPU-intensive, those in Group 4 were memory-intensive.

The usage-prediction model is automatically generated by the *AutoAI* engine available on the IBM Cloud [23]. As described in IV-A2, the input features include collected message requests and the node's resource utilization. For our preliminary models, the alternative algorithms are linear, decision tree, extra trees, random forest, and ridge. The optimized metric is root mean squared error (RMSE). The details of the selected algorithm and engineering technique for each cluster group indexed from 0 to 3 are summarized in Table III. The RMSE represents a cross-validated prediction error. The explained variance (R^2) is also used for considering the discrepancy between the model and actual data. A high explained variance indicates a strong association, which implies a good model.

In Fig. 6, we further show the correlation matrix of all collected data to underline the strong correlation between the node-occupied status defined by utilization percentage and container resource usage. Despite of considering the worst-performing models: Group 0 for CPU and Group 3 for memory, the models with adding the node-occupied status feature can reduce prediction error by about 30% and improve explained variance to the borderline compared to these models with the conventional workload feature of message requests Fig. 5. These results affirm the importance of problem statement ***P-II***.

B. Evaluation of RunWild

To evaluate RunWild, we adopted and modified a simplified microservice version of *Acme Air*, a fictional airline benchmark application [28], as simply pictured in Fig. 7. This benchmark is originally provided in the *BLUEPERF* repository [11]. The databases in this application are centralized. For applications with a distributed database, the data fragmenting decision as

TABLE II
CLUSTERING RESULTS

Cluster	Average	P90	P95	P99	[Min, Max]
Normalized Message Request					
Group 0	0.001	0.001	0.001	0.001	[0.001, 0.002]
Group 1	0.005	0.010	0.023	0.038	**[0.002, 0.070]**
Group 2	**0.011**	0.015	0.017	0.018	[0.009, 0.019]
Group 3	0.003	0.004	0.005	0.007	[0.003, 0.007]
Normalized Node CPU Usage					
Group 0	0.040	0.047	0.049	0.052	[0.035, 0.055]
Group 1	0.003	0.003	0.003	0.003	[0.002, 0.004]
Group 2	**0.350**	0.442	0.461	0.482	[0.284, 0.488]
Group 3	0.052	0.067	0.096	0.112	[0.044, 0.114]
Normalized Node Memory Usage					
Group 0	0.168	0.173	0.174	0.176	[0.158, 0.177]
Group 1	0.018	0.019	0.019	0.020	[0.016, 0.020]
Group 2	0.237	0.253	0.256	0.259	[0.214, 0.260]
Group 3	**0.631**	0.656	0.658	0.662	[0.550, 0.663]

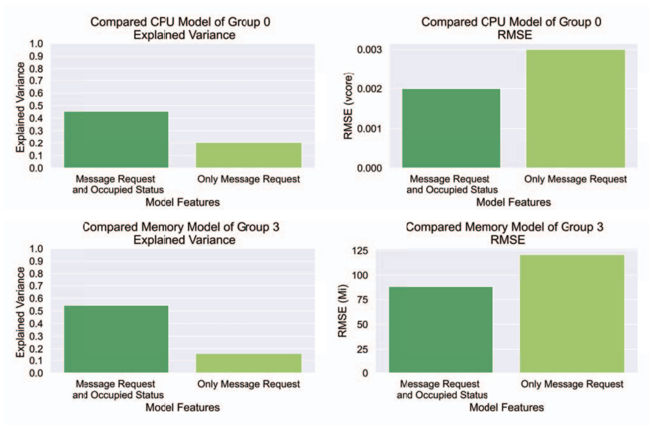

Fig. 5. Performance comparison between modeling with our clustering and modeling approach (dark green) and modeling using only workload feature of message requests (light green)

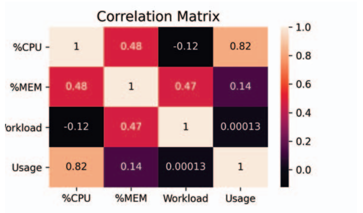

Fig. 6. Correlation matrix between usage of workload feature of message requests and candidate node-occupied status (%resource utilization)

well as the number of database servers can be calculated by various logic concerning semantic characteristics such as query type which is out of the paper scope. For simplicity, we treat the database replication as a fixed number. The experiment simulates 1000 users submitting workloads for 300 seconds. The pre-defined number of microservice groups is four. The reconcile period for the operator is 10 minutes. The system parameters for each service are listed in Table IV.

We compared RunWild to the following two resource management systems to evaluate deployment performance: (i) Kubernetes HPA with default scaling threshold (i.e., 80%), and

TABLE III
MODELING RESULT

Model	***Regressor***	***FE***	***HP***	***RMSE***	R^2
CPU-Usage Model					
Group 0	Random Forest	✓	×	0.002 vcore	**0.455**
Group 1	Random Forest	×	✓	0.037 vcore	0.655
Group 2	Extra Trees	×	✓	0.054 vcore	0.829
Group 3	Extra Trees	✓	✓	0.185 vcore	0.903
Memory-Usage Model					
Group 0	Extra Trees	✓	×	12 Mi	0.598
Group 1	Random Forest	✓	×	48 Mi	0.713
Group 2	Extra Trees	✓	×	100 Mi	0.861
Group 3	Random Forest	×	✓	88 Mi	**0.544**

FE: Feature engineering; HP: Hyperparameter; R^2: Explained variance

TABLE IV
PARAMETER SETTING

Service	**Parameter**	**Value**
Client	R_0	not provided
	L_0	not provided
Driver	users	1000
	duration	300 seconds
	ramp up	300 seconds
	delay	120 seconds
	time slot (snapshot)	30 seconds
Monitor	moving step	10 minutes
	look back	1 hour
Modeler	cluster number	4
Planner	default group	0
	safe margin on allocation	10% of Capacity
	safe margin on prediction	model Mean Absolute Error
	α	1
	β	1 if $\lvert\widehat{B}\rvert$ equals
	k	2
Operator	Reconcile Period	10 minutes

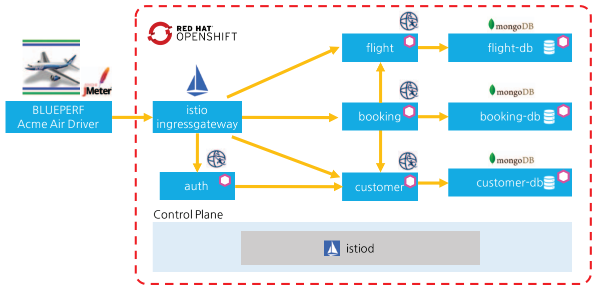

Fig. 7. Benchmark deployment

(ii) baseline without scaling. We set the per-container CPU request parameter that enables HPA to 0.5 virtual cores (vcores) (labeled "Default HPA-0.5") and 1 vcore (labeled "Default HPA-1") as representatives of vertical and horizontal scaling. The metrics for performance evaluation are response time and throughput reported by summary report from the remote JMeter driver. Throughput is the total number of responses to user requests per second. The request covers a flow starting from logging-in, flight querying, customer updating, booking, and ending with logging-out defined by BLUEPERF [11]. To reduce the interference of unrelated but coexisting containers, we ran the experiment three times and calculated the 90th percentile of response time and the average throughput for each system.

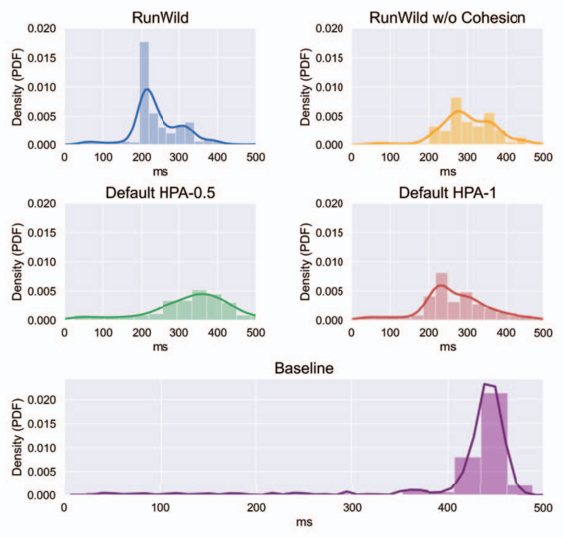

Fig. 8. Comparison of response time metric

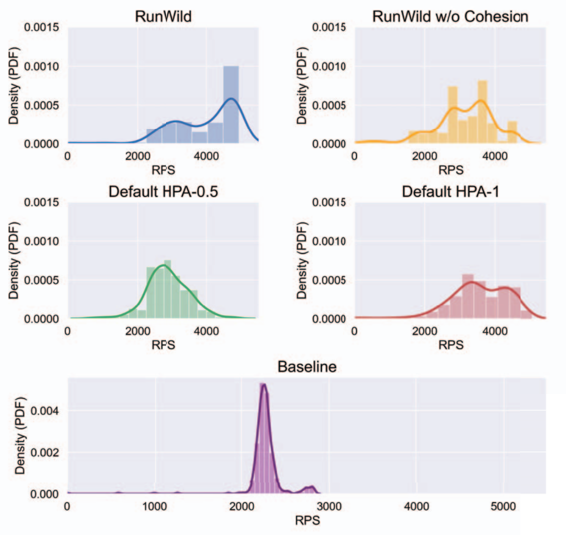

Fig. 9. Comparison of throughput metric

The fundamental Kubernetes implementation also restarts the pod for patching of vertical scaling which can take more than 40 seconds for a large container (e.g., the container *flight*). To reduce such an overhead, we round the number to avoid small changes in the vertical scale.

Whereas the 90th percentile of baseline response time was 453 ms, RunWild responded 1.4 times faster, i.e., it responded in 328 ms. When the cohesion metric was not used, the 90th percentile of response time increased to 364 ms. The HPA took 435 ms to respond with a 0.5-vcore request. The compared distribution of results is shown in Fig. 8. For throughput, RunWild processed up to 3,937 RPS when using the cohesion metric and 3,125 RPS when not using. In the same trend, the baseline and HPA with a 0.5-vcores request could process only 57% and 73.2%, respectively, of RunWild results. As more resource requests allow a container to be allocated to the node with higher resource availability, the HPA reduced the 90th-percentile response time to 368 ms and increased average throughput to 3,569 RPS when the resource request increased to 1 vcore. The distribution results are presented in Fig. 9. Nevertheless, RunWild with the cohesion metric yet showed an 11 and 10% performance improvement in response time and throughput, respectively, compared to the HPA with 1-vcore request with 35% lower CPU used or reserved as presented in Fig. 12. These results indicate that RunWild outperformed the default HPA. Furthermore, the performance of RunWild improved using the cohesion metric which is corresponding to the problem statement ***P-III***. Notably, RunWild without the cohesion metric may have minimum resource usage; however, this resulted decreased performance. On the contrary, vertically increasing the resource quota per instance for HPA might result in higher performance than RunWild both with and without cohesion; however, it must be traded off with over-provisioning resource waste and unavailability.

To look into performance behavior, we plot the response time in Fig. 10 and throughput in Fig. 11. We observed that RunWild improved in performance after it determined which predefined group a service fit in based on the features defined in Section IV-A1. We also noticed a difference in the early state when using the default HPA. This can be explained as the default HPA performs scaling immediately after the usage passes the threshold, whereas RunWild performs in a periodic time. We observed highly fluctuating and undetermined dynamic behavior of the default HPA due to threshold-based scaling. Even if the dynamic behavior can be reduced by applying a threshold policy, such as deferred downscaling, slow decay, deferred small changes, and limiting growth, as explained in [19], the parameter tuning for these policies is not straightforward and cannot guarantee success.

To emphasize the significance of aligning network load control with the scaling decision as stated in problem statement ***P-IV***, we selected and plotted the usage of the *flight* service since it was of one most scaled services deployed by the default HPA in Fig. 13. According to computation with the default HPA, all containers are expected to have a similar amount of usage as the number of replicas is simply computed by the ceiling of the usage summation divided by the targeted per-container usage. However, without well-aligned workload partitioning at runtime, the actual usage on some containers is more than the expected request, while some containers are reserving resources but not using them. The above observations and findings support our motivation to enable a united extensive deployment on a cloud platform that covers not only horizontal duplication and resource allocation but also placement scheduling and load balancing as stated in problem statement ***P-I***.

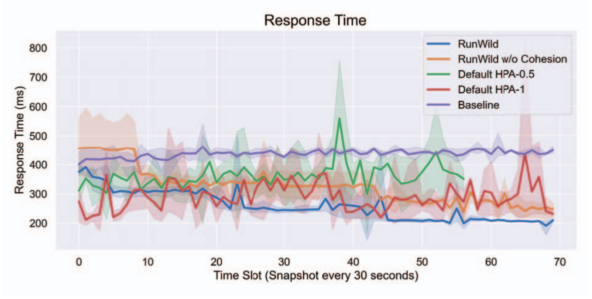

Fig. 10. Response time at runtime

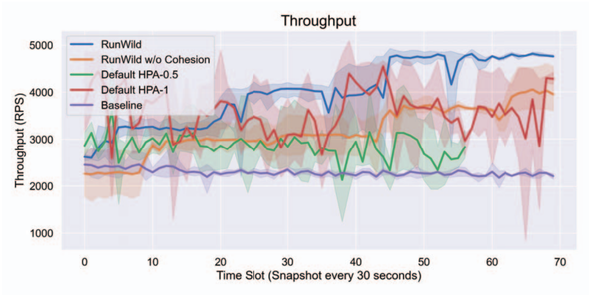

Fig. 11. Throughput at runtime

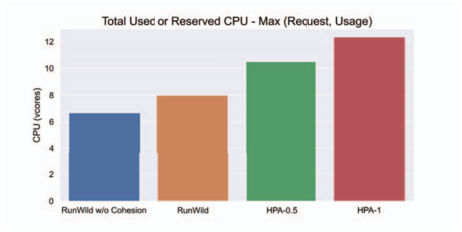

Fig. 12. Comparison of used or reserved CPU

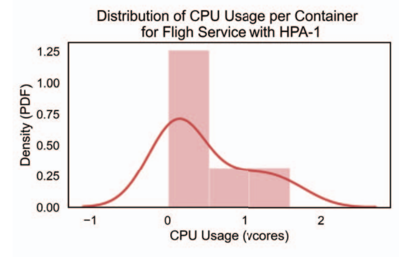

Fig. 13. Distribution of CPU usage per container for the service *flight* deployed using HPA-1 (HPA with 1-vcore CPU request)

VI. Limitations and Future Work

In the RunWild system, we have not yet fully covered the following three remaining issues. The first is the limitation on the operating mechanism. Even if the change in resource requests can be much more frequent than duplication, the Kubernetes API for patching deployment disrupts performance when processing vertical scaling, which is nearly as bad as that for horizontal duplication. RunWild's rounding approach avoids small changes. However, this disruptive approach can be improved by implementing a checkpoint for container migration to the cluster as proposed by Rattihalli et al. [29]. Furthermore, the replica set controller does not support unequal resource requests or limits. Accordingly, we set the resource request to the minimum computed value of all replicas. Still, the value to compute resource allocation is predicted from our usage-modeling approach rather than this minimum value.

The second issue is with the modeling scheme. Even though our current cluster and prediction models were obtained from offline profiling to avoid negative impact on the experiment, modeling can be upgraded to an online manner without impacting the contribution of this paper. One promising online learning scheme was proposed by Alipour and Liu [8].

Finally, the third point is the scaling of a distributed database. As data fragmenting is out of our scope, we did not consider the impact of scaling on the database. Fragmentation logic, as described in [30], is required to support a distributed database.

VII. Conclusion

This paper provides solutions to four original problems in the state-of-the-art research on scaling, placement scheduling, and load-balancing microservices in cloud platforms. Correspondingly, we proposed *RunWild*, a resource management system for solving these problems. The two main contributions are (i) a usage-modeling approach and (ii) full autoscaling and schedule planning problem definition and solution. Our resource usage modeling approach combines a node occupied status metric and historical node-independent workload. Furthermore, a machine-learning data-clustering technique is applied to reduce the modeling overhead with a method of determining the number of data clusters to be used. To solve the planning problem, we introduced the cohesion metric as an objective value in addition to the commonly found number of replicas. We extended a greedy algorithm with a lightweight overhead mitigation logic. We conducted experiments using an actual cluster to compare RunWild with the widely used autoscaling, HPA, provided in the Kubernetes platform. The results indicate that response time decreased by 25% and throughput increased by 37%. We also showed additional significant improvement when adding the cohesion metric into the objective function by comparing with RunWild without the cohesion metric. For the usage-prediction model, the modeling evaluation demonstrated that the status metric we introduced results in higher explained value and lower error. Furthermore, the results of reservation overuse were further summarized and discussed by comparing a deployment managed by RunWild to default deployed services.

References

[1] "What is Kubernetes?" 2020. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[2] F. Soppelsa and C. Kaewkasi, *Native Docker Clustering with Swarm*. Packt Publishing Ltd, 2016.

[3] M. R. López and J. Spillner, "Towards Quantifiable Boundaries for Elastic Horizontal Scaling of Microservices," in *International Conference on Utility and Cloud Computing*, 2017, p. 35–40.

[4] "Horizontal Pod Autoscaler." [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale

[5] "Vertical Pod Autoscaling," 2020. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler

[6] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *International Conference on Distributed Computing Systems*, 2019, pp. 1994–2004.

[7] M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth, "A Virtual Machine Re-Packing Approach to the Horizontal vs. Vertical Elasticity Trade-off for Cloud Autoscaling," in *Cloud and Autonomic Computing Conference*, 2013, pp. 1–10.

[8] H. Alipour and Y. Liu, "Online machine learning for cloud resource provisioning of microservice backend systems," in *International Conference on Big Data*, 2017, pp. 2433–2441.

[9] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-Clouds: Managing performance interference effects for QoS-aware clouds," in *European Conference on Computer Systems*, 2010, p. 237–250.

[10] E. Brewer, "Kubernetes and the new cloud," in *Proceedings of the 2018 International Conference on Management of Data*. Association for Computing Machinery, 2018, p. 1.

[11] "BLUEPERF." [Online]. Available: https://github.com/blueperf/Overview

[12] "OpenShift." [Online]. Available: https://www.openshift.com

[13] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning," in *International Conference on Cloud Computing*, 2019, pp. 329–338.

[14] A. R. Sampaio, J. Rubin, I. Beschastnikh, and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," *Journal of Internet Services and Applications*, vol. 10, no. 1, pp. 1–30, 2019.

[15] L. Calcote, *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. O'Reilly Media, 2019.

[16] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload Control for Scaling WeChat Microservices," in *Symposium on Cloud Computing*, 2018, pp. 149–161.

[17] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating Performance Metrics for Scaling Microservices in Cloud IoT-Environments," in *International Conference on Performance Engineering*, 2018, pp. 157–167.

[18] L. Florio and E. Di Nitto, "Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures," in *International Conference on Autonomic Computing*, 2016, pp. 357–362.

[19] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at Google," in *European Conference on Computer Systems*, 2020.

[20] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications," in *International Conference on Cloud Computing*, 2018, pp. 82–89.

[21] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[22] G. A. Wilkin and X. Huang, "K-Means Clustering Algorithms: Implementation and Comparison," in *International Multi-Symposiums on Computer and Computational Sciences*, 2007, pp. 133–136.

[23] D. Wang, P. Ram, D. K. I. Weidele, S. Liu, M. Muller, J. D. Weisz, A. Valente, A. Chaudhary, D. Torres, H. Samulowitz, and L. Amini, "AutoAI: Automating the End-to-End AI Lifecycle with Humans-in-the-Loop," in *International Conference on Intelligent User Interfaces Companion*, 2020, p. 77–78.

[24] X. Wang and Y. Liu, "ARIMA time series application to employment forecasting," in *International Conference on Computer Science & Education*, 2009, pp. 1124–1127.

[25] J. S. Dodgson, M. Spackman, A. Pearman, and L. D. Phillips, "Multi-criteria analysis: a manual," pp. 19–29, 2009.

[26] "Operator SDK." [Online]. Available: https://sdk.operatorframework.io

[27] Prometheus, "What is Prometheus?" [Online]. Available: https://prometheus.io/docs/introduction/overview

[28] "Acme Air." [Online]. Available: https://github.com/acmeair/acmeair

[29] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes," in *International Conference on Cloud Computing*, 2019, pp. 33–40.

[30] H. I. Abdalla and A. A. Amer, "Dynamic horizontal fragmentation, replication and allocation model in DDBSs," in *International Conference on Information Technology and e-Services*, 2012, pp. 1–7.