IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 21, 2020, accepted October 15, 2020, date of publication October 22, 2020, date of current version November 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3033019

Refining Microservices Placement Employing Workload Profiling Over Multiple Kubernetes Clusters

JUNGSU HAN[1], YUJIN HONG[1], AND JONGWON KIM[2], (Senior Member, IEEE)

[1]School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology (GIST), Gwangju 61005, South Korea
[2]AI Graduate School, Gwangju Institute of Science and Technology (GIST), Gwangju 61005, South Korea

Corresponding author: Jongwon Kim (jongwon@nm.gist.ac.kr)

This work was supported in part by the Vehicles AI Convergence Research and Development Program through the National IT Industry Promotion Agency of Korea (NIPA) funded by the Ministry of Science and ICT (MSIT) under Grant S1605-20-1002, and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korean Government, MSIT, Artificial Intelligence Graduate School Program (GIST), under Grant 2019-0-01842.

ABSTRACT As cloud-native computing is becoming the de-facto paradigm in the cloud field, Microservices Architecture has attracted attention from industries and researchers for agility and efficiency. Moreover, with the popularity of the IoT in the context of edge computing, cloud-native applications that utilize geographically-distributed multiple resources are emerging. In line with this trend, there is an increasing demand for microservices placement that selectively use optimal resources. However, optimal microservices placement is a significant challenge because microservices are dynamic and complex, depending on diversified workloads. Besides, generalizing workloads' characteristics consisting of complex microservices is realistically challenging. Thus, microservices deployment with mathematically structured algorithms based on simulation is less practical. As an alternative, a microservices placement framework is required that can reflect the characteristics of workloads derived from empirical profiling. Therefore, in this research work, we propose a refinement framework for profiling-based microservices placement to identify and respond to workload characteristics in a practical way. To achieve this goal, we perform profiling experiments with selected workloads to derive delicate resource requirements. Then, we perform microservices placement with a greedy-based heuristic algorithm that considers application performance by using resource requirements derived from the profiled results. Finally, we verify the proposed concept by comparing the experimental results that use our work and those that don't.

INDEX TERMS Cloud-native computing, microservices placement, workload profiling, container orchestration, resource monitoring.

I. INTRODUCTION

Cloud computing has been unanimously accepted as the technological solution for different problems in various sectors including businesses, healthcare, factory, farm [1]. With the growth of cloud technologies, cloud-native computing is rising as the de-facto paradigm with Microservices Architecture (MSA)-based service composition for agility and efficiency [2]. Microservices are defined as small-sized functions that may be deployed and scaled independently of each other, and they may employ different middleware stacks for their implementation [3]. In particular, legacy services based on monolithic architecture are being migrated to MSA to adapt to technological changes and reduce time-to-market. These applications, also known as cloud-native applications still harness the benefits of the cloud-native paradigm [4], [5]. For cloud-native applications, the adoption of containerization is gradually increasing since it has benefits in terms of flexibility and performance rather than virtual machines [6], [7]. Containers that are inherently agile offer the most feasible option to run microservices in cloud computing [8].

Early adopters mainly deployed cloud-native applications in the form of containers on a single cloud architecture [9]. However, with the popularity of the IoT in the context of edge computing, cloud-native applications that utilize geographically-distributed multiple resources have recently

The associate editor coordinating the review of this manuscript and approving it for publication was Chin-Feng Lai.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

emerged. Several container orchestration engines have been released to take advantage of the resources of multiple clouds and clusters. However, current container management tools do not consider optimization policies, remaining room for improvement [10]. In other words, there are many hidden challenges in optimal microservices placement over multiple resources.

Many studies have been devised to solve the optimal resources management problem in cloud computing. Resources management in cloud-native computing should satisfy acquiring resource requirements in microservices to ensure the seamless running of cloud-native applications. Determining the right amount of resources to allocate to incoming microservices is an intricate task that involves many challenges [11]. However, most researchers deal with deriving a placement algorithm through mathematical modeling based on simulations to tackle these challenges. These works often overlook obtaining precise requirements for microservice resources. We should come up with a component that grasps the fine-grained resource requirements depending on workloads by taking into account the intricate microservices.

Therefore, this research proposes a framework that addresses microservices placement for considering application performance based on empirical profiling in a practical way. The specific contributions of this article are summarized below.

- First, we devise a refinement framework that empirically refines microservices placement based on workload profiling. The framework provides a practical lifecycle range from cloud-native workloads profiling to microservices placement. For workload profiling, we first design resource variation monitoring to acquire microservices resource consumption. Based on the monitored data, the framework profiles cloud-native applications responding to workloads repeatedly. Subsequently, profiled data is utilized to do the appropriate microservices placement empirically.
- Second, to make concrete our proposed concept, we implement a profiling system in accordance with the framework. Then, we experiment with workload profiling on three different types of cloud-native applications. Resource consumption is measured repeatedly in three selected applications depending on the workload. The cycle of automatic deployment keeps track of resource status changes by dividing the start and completion points according to the primary resource utilization by paying attention to the resources used empirically through testing deployment. After that, we analyze profiled results to obtain fine-grained resource requirements depending on workload characteristics.
- Third, we do microservices placement with the refinement framework on multiple Kubernetes (K8s) clusters by utilizing profiling results. We apply a greedy-based heuristic algorithm that reacts to workload characteristics observed in empirical profiling for the microservices placement. Also, we configure a testbed that consists of three different Kubernetes clusters for the placement experiments. We then verify the proposed concept by deploying microservices on the testbed by following the framework's placement policies. The experiments show that improve the quality of service compared to not adopting our work.

The rest of this article is organized as follows. In Section 2, we briefly summarize related work. We then explain the proposed framework in Section 3. Section 4 discusses a profiling system and the profiling results using the proposed framework by deploying three cloud-native applications responding to the workload. In Section 5, we perform microservices placement with the proposed framework based on the analysis of workloads profiling and discuss experiments results. Finally, we conclude this article in Section 6.

II. RELATED WORK

Resource placement in cloud computing is one of the most important research issues. These researches have been studied in subdivisions according to the form of resources, the optimization value, and the target environment. In the early days of cloud computing growth, most of the works focus on resource management for virtual machines. Tordsson *et al.* [12] proposed a novel cloud brokering approach that optimizes virtual machines' placement across multiple clouds for cost and performance using linear programming. Li *et al.* [13] proposed a Layered Progressive resource allocation algorithm for multi-tenant cloud data centers based on the Multiple Knapsack Problem (LP-MKP). Heilig *et al.* [14] address cloud resource management in multiple clouds that is a recent optimization problem aimed at reducing the monetary cost and the execution time of consumer applications using a genetic algorithm. Legillon *et al.* [15] proposed a new realistic model with genetic algorithms to tackle the problem, placing services into heterogeneous VMs from different IaaS providers.

Meanwhile, as cloud computing moves towards cloud-native computing with MSA technology adoption, researchers have addressed resource management for containers [16]–[18]. Guerrero *et al.* [19] proposed a genetic algorithm for the allocation of containers running microservices in multiple clouds. They considered the objectives of cloud service cost, network latency, and service reliability. Filip *et al.* [20] proposed a new model for scheduling microservices placement across heterogeneous cloud-edge environments. Wan *et al.* [21] proposed an efficient communication framework and a suboptimal algorithm to determine the data center's container. Wen *et al.* [22] proposed a new dependable microservices orchestration framework with GA-Par that adopts a genetic algorithm to perform microservices composition whilst reducing the discrepancy between user security requirements and actual service provision. Buyya *et al.* proposed a framework for the cost-efficient orchestration of containers in the Cloud environment [23]. They also stressed the need for more research work to optimize deployments at run time, especially through

TABLE 1. Comparison of existing works on resource placement in cloud computing.

Research Work	Solution Approach	Resource Unit	Opimization Value	Infrastructure
[12]	Heuristic (LP)	Virtual machine	Cost, Performance	Multiple clouds
[13]	Heuristic (LP-MKP)	Virtual machine	Network distance, resource fragmentation	Multiple clouds
[14]	Heuristic (Genetic)	Virtual machine	Cost, Execution time	Multiple clouds
[15]	Heuristic (Genetic)	Virtual machine	Monetary cost	Multiple clouds
[19]	Heuristic (NSGA-II)	**Container**	Cost, Network latency, Reliability	Multiple clouds
[22]	Heuristic (Genetic)	**Container**	Security	Multiple clouds
[24]	Heuristic (NSGA-II)	**Container**	Cost, Deployment time	Multiple clouds
[25]	Heuristic (LP)	**Container**	Cost, Deployment time	Single cloud
[26]	Heuristic (Greedy)	**Container**	Energy	Single cloud
[27]	Heuristic (RR+MST)	**Container**	Performance	**Kubernetes cluster**
[29]	**Profiling** + Heuristic (LP)	Virtual machine	Tail latency	Single cloud
Our work	**Profiling** + Heuristic (Greedy)	**Container**	Performance	**Kubernetes cluster**

containers' initial deployment and the migration, rebalancing, or auto-scaling of clusters. Hoenisch *et al.* [24] addressed four-fold auto-scaling by formulating the scaling decision as a multi-object optimization problem. In this work, four dimensions of scaling were considered: VMs and containers can be adjusted horizontally (changes in the number of instances) and vertically (changes in the computational resources available to instances). Nardelli *et al.* [25] provided a general formulation of the elastic provisioning of containers on virtual machines as an integer linear programming problem, which takes explicitly into account the heterogeneity of container requirements and virtual machine resources. Piraghaj *et al.* proposed a framework that consolidates containers on virtual machines to improve the energy efficiency of servers [26]. Joseph *et al.* proposed a novel, robust heuristic approach called IntMA to deploy the microservices in an interaction-aware manner with the aid of the interaction information obtained from the interaction graph [27]. This work's remarkable thing is that the target environment was transited from multiple clouds to the Kubernetes cluster. Kubernetes, which is used as a de-facto standard in cloud-native computing, can provide a consistent resource pool over multiple clouds [2]. Therefore, developers entirely focus on deploying their applications without worrying about underlay clouds.

The aforementioned researches address container placement in more complex situations than virtual machine placement to deal with resource management in cloud-native computing. However, these studies often overlook a precise understanding of resource requirements in a practical way. As a result, optimization placement with the only well-modeled algorithm has difficulty applying them to real problems. Some researchers try to use a profiling method to apply their resource management system in real-world scenarios. Basit Qureshi proposed a power-aware framework for efficient placement of application workloads in the data center based on profiling [28]. Ye *et al.* [29] proposed a novel profiling-based consolidation of multi-tier interactive workloads from a new perspective of user-perceived tail latency. This work is the most similar to our approach but with significant differences. Kejiang focused on virtual machine placement on physical servers. On the other hand, our approach considers microservices placement rather than virtual machines to cope with the cloud-native paradigm shift. Also, we decide to select the target environment of deployment to a more flexible Kubernetes cluster instead of multiple clouds. We provide a summary of the comparison between the relevant research works and our study in Table 1.

III. REFINEMENT FRAMEWORK FOR PROFILING-BASED MICROSERVICES PLACEMENT

This section first introduces requirements to satisfy microservices placement that reflects the workloads derived from empirical profiling. Then, we present a refinement framework based on empirical profiling and describe the framework's overall workflows.

A. REQUIREMENTS

The following requirements are discussed to satisfy the cloud-native application deployment for effectively guaranteeing performance on cloud-native infrastructure.

- **R1. Microservices Monitoring:** Cloud-native computing has changed the way we deploy our applications into the containers complying with microservices architecture since containerization enables developers to make lightweight isolation that can easily and quickly deploy their codes to realize services [2]. Basically, most monitoring solutions perform resource-layer visibility, mainly focused on physical/virtual machines. However, resource-layer visibility that covers containers is becoming more important since cloud-native applications are carried out in the form of containers. In particular, cloud-native applications consist of multiple microservices that interact with each other, so it should comprehensively monitor and understand not only computing resources but also network and storage.
- **R2. Empirical Workload Profiling:** Cloud-native applications based on MSA are quite challenging to simplify resource requirements because many microservices interact with each other dynamically. Therefore, it is essential to accurately analyze cloud-native applications' resources requirements based on empirical profiling depending on diversified workloads of cloud-native

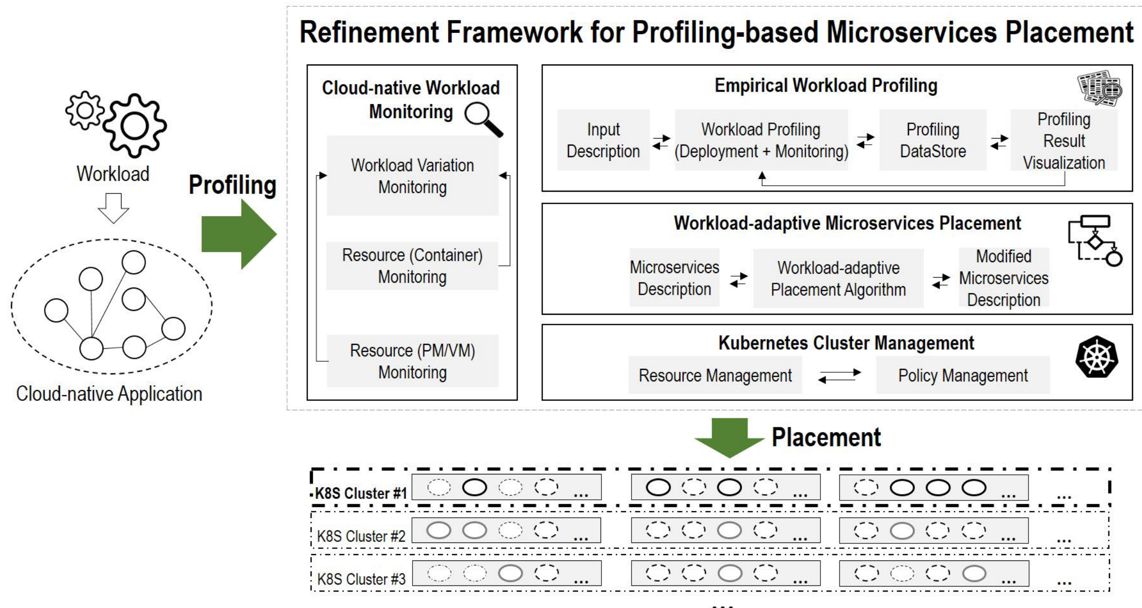


FIGURE 1. Overall design of the refinement framework.

applications. Thus, for workload profiling, analysis results based on various workload parameters should be presented repeatedly in conjunction with microservices monitoring for specific cloud-native applications. In addition, this process has to help you understand delicate resource requirements based on the workloads through statistical data generated from profiling.

- **R3. Workload-adaptive Microservices Placement:** Research in resource placement to maximize cloud-native applications' performance is challenging because microservices are dynamic and complex with diversified workloads. A heuristic algorithm derived in assuming resource requirements depending on workload's characteristics in advance tends to be challenging to apply in the practice area. Thus, Workload-adaptive microservices placement techniques are required that can be reacted according to dynamic resource requirements in conjunction with empirical workload profiling. The microservices placement should be adaptive in a manner that is not fixed to a particular situation in running applications and is dynamically responded by various workloads.

B. PROPOSED REFINEMENT FRAMEWORK

Based on the above requirements, we suggest a refinement framework for profiling-based microservices placement, as depicted in Figure 1. The proposed framework consists of four main components: i) Cloud-native workload monitoring, ii) Empirical workload profiling, iii) Workload-adaptive microservices placement, iv) Kubernetes cluster management. The framework performs user requirements that include their cloud-native application with workload parameters through inter-communication between main components for efficient microservices placement in multiple Kubernetes clusters.

To satisfy the R1, we carefully design the cloud-native workload monitoring component. The component's key feature is to provide container-based resource variation monitoring corresponding to workloads' dynamic characteristics. Most of the resource monitoring tools are focus on main resources metrics such as CPU, memory, disk, network, and so on. However, we design workload monitoring based on resources that we have identified comprehensively by separating the physical and container resources.

To handle the R2, we suggest empirical workload profiling based on workload monitoring. The workload profiling provides statistical results based on the resources metrics that are repeatedly placed in accordance with the user's application responding to workload.

For the R3, we design workload-adaptive microservices placement. We use a greedy-based heuristic algorithm for microservices placement based on workload profiling. We also devise Kubernetes cluster management to identify the cluster's current residual resource capacities and manage labeling information belonging to the cluster to help with actual microservices placement.

The main components of the framework are described in detail below.

i) Cloud-native workload monitoring: It is a monitoring component focusing on microservices deployed in the container. It comprehensively collects and visualizes computing resources such as CPU, memory, and storage and network metrics by the microservices unit. Physical resource

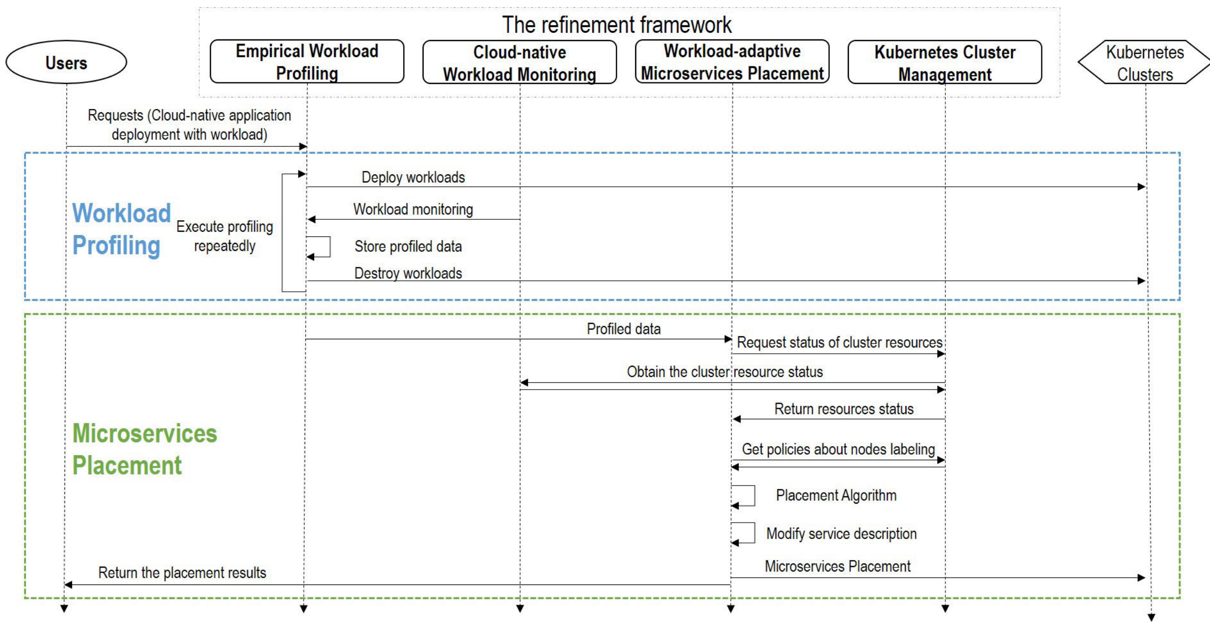


FIGURE 2. Overall workflow of the propose framework.

monitoring and container monitoring are combined with being designed to perform resource variation monitoring for microservices for the component. Data collected from the monitoring is stored in a K8s persistent volume to be utilized in the profiling component.

ii) Empirical workload profiling: It consists of functionalities that perform profiling repeatedly, depending on the user's description that contains a cloud-native application with workload parameters. After repeatedly deploying microservices in response to workload parameters, the workload monitoring data is collected. It stores profiling information for the workload on the profiling datastore. We can analyzes resource consumption characteristics since the collected profiling results are statistically visible to the user.

iii) Workload-adaptive microservices placement: It is microservices deployment over multiple Kubernetes clusters depending on the workload profiling by using our proposed algorithm. We use the Greedy method, one of the heuristic algorithms, to respond to workloads profiling. Note that our framework is designed to apply various heuristic algorithms. When the user submits a Kubernetes description for the desired cloud-native application, it performs the heuristic algorithm based on the profiling results. After that, it deploys to the Kubernetes clusters via location-labeled modified Kubernetes description to prevent applying the Kubernetes' default placement policy. Note that this work deals with the problem of selecting a single cluster in the multiple Kubernetes clusters and placing microservices into it. We do not address cloud-native application deployment beyond multiple Kubernetes clusters since it requires advanced techniques such as service mesh and cluster federations.

iv) Kubernetes cluster management: It consists of two modules to support microservices deployment in Kubernetes clusters. The resource management module checks the current idle resources for our Kubernetes clusters. It uses the cloud-native workload monitoring components to identify resource situations and provide resource information with the workload-adaptive microservices placement component at the deployment stage. The policy management labels all of the nodes that make up the Kubernetes clusters in policies. These labels help the workload-adaptive microservices placement component deploy microservices to desired nodes in the Kubernetes clusters.

C. OPERATION WORKFLOW OF THE PROPOSED FRAMEWORK

The proposed framework works with the operation workflow, as shown in Fig. 2. Initially, cloud-native workload monitoring is configured on multiple Kubernetes clusters. It regularly collects and records resource status for nodes and containers that belong to the Kubernetes clusters. Once initialization is done, users can send their requests to the proposed framework. It contains a cloud-native application description in YAML with workload parameters.

When a user's requests are received, the framework's first stage is to perform workload profiling. The framework deploys the target application into the Kubernetes clusters. Simultaneously, the workload profiling component works together with the monitoring components to measure and store the resource usages in microservices units. The framework automatically performs the microservices deployment/de-allocation at a certain number of times.

The profiled data generated by repeated microservices deployment are visible in graph form for workload analysis.

The second stage in the framework is microservices placement in Kubernetes clusters. The placement stage used in the framework requires the following conditions: i) the results of workload profiling, ii) The Kubernetes cluster information with node labeling policies. The first stage's profiling data quantifies the microservice's resource requirements according to the workload characteristics. For acquiring resource status in the Kubernetes cluster, the microservices placement component requests the current resource situation per node to the monitoring component. Node labeling is obtained through the Kubernetes cluster management. After preparing all conditions, it performs algorithms within the framework to determine the microservices placement. Finally, the framework deploys microservices with a modified cloud-native application description following the algorithm's result. Detailed experiments of profiling and placement with the proposed framework are covered in the next sections.

IV. EMPIRICAL WORKLOAD PROFILING EXPERIMENTS

In this section, we prepare three types of cloud-native applications to perform workload profiling. Then, we configure an experimental environment and finally analyze experimental results based on the profiling data.

A. CLOUD-NATIVE APPLICATIONS RESPONDING TO WORKLOAD

The number of cloud-native applications is explosively emerging with the growth of cloud, IoT, and AI. Building infrastructure management that handles cloud-native applications by categorizing them is an excellent way to cover various cloud-native computing applications. However, the typification of cloud-native applications is difficult due to the diversity and complexity of the applications. Instead, we prepare the following three types of cloud-native applications by paying attention to the infrastructure across IoT-Edge/Core clouds, as depicted in Fig. 3.

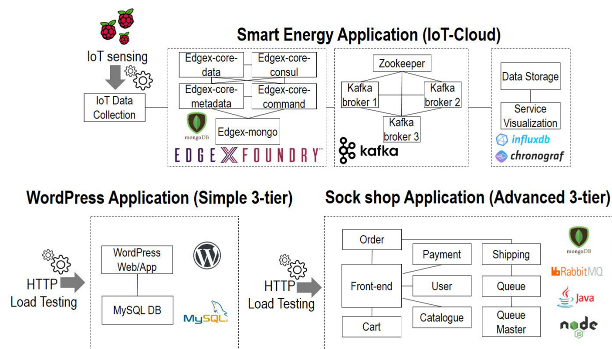


FIGURE 3. Three types of cloud-native applications.

i) Smart energy application: Smart energy service uses Raspberry Pi 2 to collect temperature, humidity, and power consumption of the server room and the server's system temperature. The data coming from IoT is processed with Edge Foundry open-source framework for data standardization. The Kafka messaging system transmits standardized data to persistent storage. It also has visualization and monitoring on a dashboard via a web browser [31]. All of the functions are developed in the form of a Docker container. The detailed implementation is written in our previous work [30]. Basically, the smart energy application consists mainly of data transmitted from IoT. Instead of using real IoT devices, we make a workload with mockup data in RESTful requests that send sensing data to the application.

ii) WordPress application: It is a famous application example with a Web-App-DB 3-tier structure. The application consists of Web/App functions and database function with a persistent volume. For the persistent volume in Kubernetes, we additionally configure Ceph storage to use RBD provisioner [32]. We use a RESTful HTTP load test tool called locust to make workload [33].

iii) Sock shop application: Sock shop is a reference cloud-native application developed by Weaveworks from the start with current microservice and cloud-native best practices in mind. It is a mock on-line shopping website for purchasing socks. The application consists of multiple microservices written in Go, Java, and Node.js, which also make use of supporting functions such as RabbitMQ, Mongo, and Nginx [34]. The microservices communicate using REST over HTTP. We also use the locust to create workload into the applications.

B. EXPERIMENTAL ENVIRONMENT

For the workload profiling experiments, we first configure cloud-native workload monitoring on multiple Kubernetes clusters, as depicted in Fig. 4. We have built three different Kubernetes clusters to realize diversified cloud-native applications with various scenario cases. The first cluster has one master node and four worker nodes. Each node utilizes the Intel Xeon E5-2650v3 CPU with 20 cores. The memory of each node utilizes Samsung DDR4 256GB RAM. In addition, the GPU is mounted with four NVIDIA Titan volta, one for each node. Finally, the Disk is equipped with an Intel P4600 2TB NVMe SSD and 120G SATA SSD. The second cluster consists of one master node and one worker node. The CPU utilizes the Intel Xeon Scalable 5118 model, with 24 cores. For memory, Samsung DDR4 256GB RAM is used. It also comes with two Intel 512G SATA SSDs. The GPU mounts three NVIDIA Titan volta on one node. The third cluster also consists of one master node and one worker node. The CPU utilizes the Intel Xeon Scalable 5118 model with 24 cores. The memory utilized the same Samsung DDR4 256GB RAM as the other clusters. Four Samsung 1.6 TB NVMe SSD and two Intel 512G SATA SSD is installed on the worker node. The worker node has six NVIDIA Tesla T4 GPU computing. We separate the hardware wiring to physically isolate the data plane for the high-performance networking on all of the nodes. The data plane is connected via Mellanox 100G smart NIC with

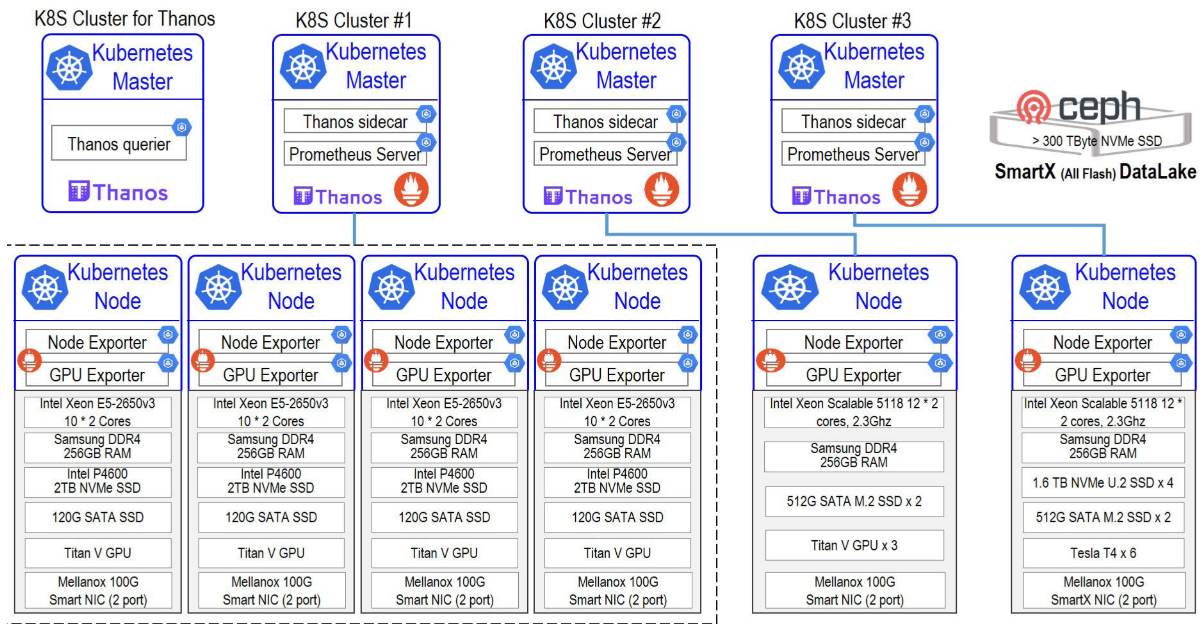


FIGURE 4. Experimental testbed on multiple Kubernetes clusters.

RoCE functionality. For the RBD-based persistent volume in Kubernetes clusters, we have built more than 300 TByte Samsung NVME SSD with Ceph storage [35].

We use the Prometheus open-source monitoring system that monitors multiple Kubernetes clusters for figuring out the resource condition. It is an excellent tool for understanding resource status for cloud-native computing. We also leverage Thanos for integrated monitoring management from multiple Prometheus tools. On each node of the clusters, node and Nvidia-SMI exporters are installed to collect basic monitoring metrics through Prometheus.

Based on the experimental testbed with the cloud-native workload monitoring, we implement the workload profiling to cover three types of cloud-native applications, as shown in Fig. 5.

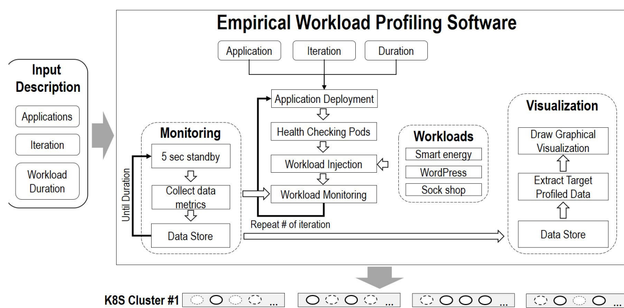


FIGURE 5. The implemented empirical workload profiling software.

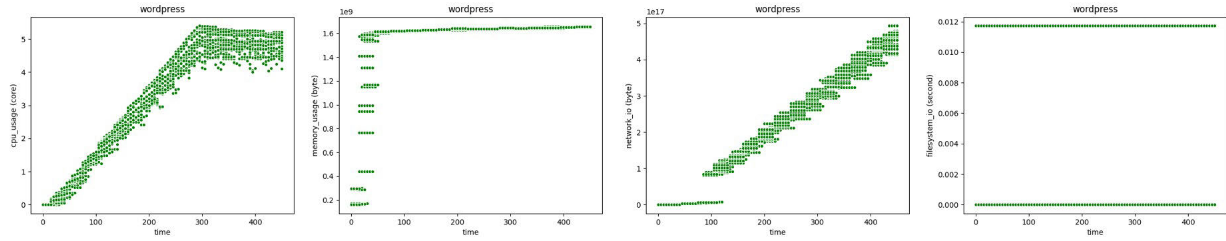
We design input parameters as application type, the number of iteration for deployment, and duration time for application deployment. When the input parameter comes in the profiling software, it deploys the target application into Kubernetes cluster #1. Note that we limit the profiling environment to the single Kubernetes cluster because we do not focus on the availability of resources in the clusters but workload resources variation. After the application deployment, the software checks the health status of Kubernetes pods to determine whether the application is performing normally. And then, the software injects workloads depending on the application type. We develop three types of workloads in a shell script, leveraging benchmarking tools. Once the workload is injected, it collects the resource status of all microservices that constitute the application. It also stores the monitored data separately in the profiling datastore.

Although the monitoring component collects various resource metrics, we only focus on CPU, Memory, Network, and Disk IO metrics in the profiling stage. In the case of the GPU metric, the vanilla Kubernetes environment doesn't support GPU virtualization. Thus, we do not focus on the GPU metric since considering the GPU metrics as a deployment factor makes microservices placement too simplistic.

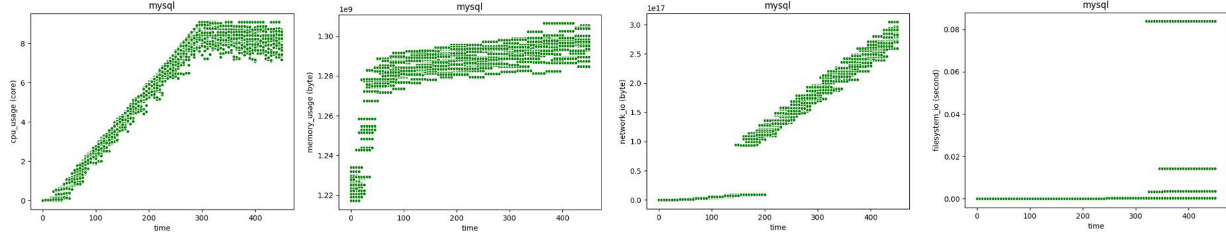
If the elapsed time of application deployment equals the duration parameter, it destroys the application. The above processes are repeated automatically until satisfying the number of the iteration parameter. At the end of the profiling process, it visualizes stored profiling data as a graph in the form of a scatter plot.

C. EMPIRICAL PROFILING EXPERIMENTS

Before starting workload profiling, we configure a workload using the Locust tool for three types of cloud-native applications. We implement python scripts that send 1,000 RESTful HTTP requests per second for 450 seconds. In the smart energy application, we send RESTful post requests with mock-up style sensor data for temperature. For other

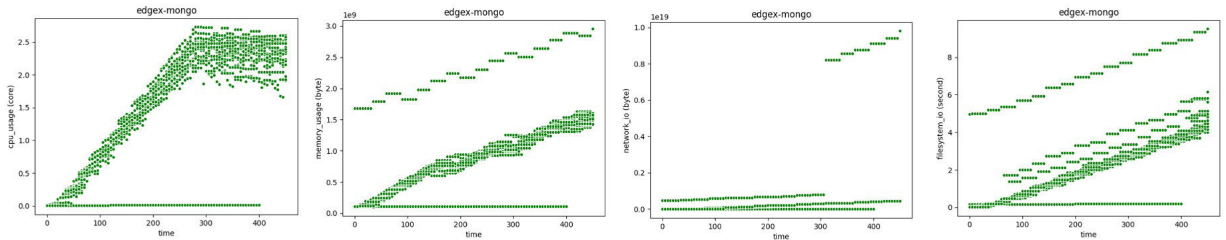


(a) Profiling for the wordpress microservice (CPU, memory, network, and disk).

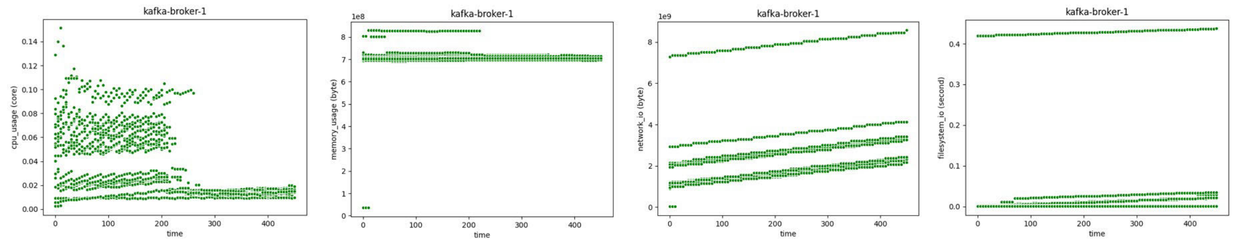


(b) Profiling for the mysql microservice (CPU, memory, network, and disk).

FIGURE 6. Empirical workload profiling for wordpress application.



(a) Profiling for the edgex-mongo microservice (CPU, memory, network, and disk).



(b) Profiling for the kafka-broker-1 microserivce (CPU, memory, network, and disk).

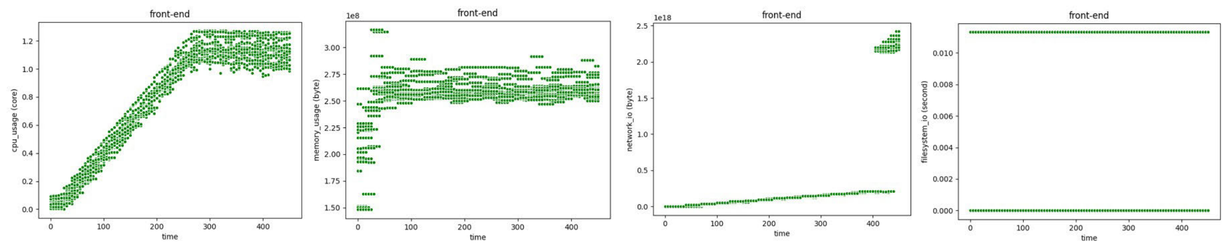
FIGURE 7. Empirical workload profiling for a part of smart energy application.

applications, we send RESTful get requests to front-end websites. The profiling is executed by setting the number of iterations 1, 5, 10, and 20, respectively.

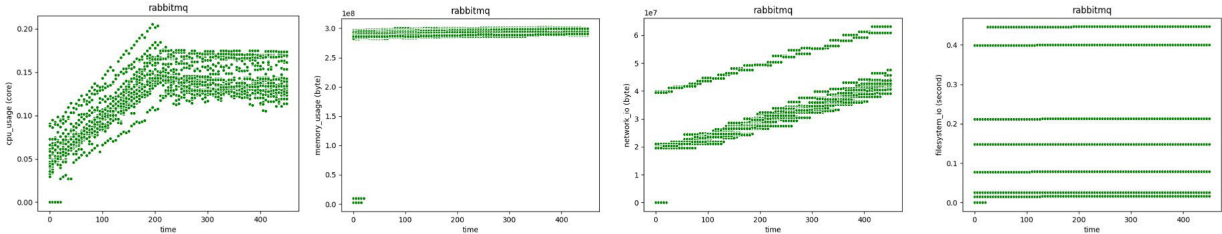
Fig. 6, 7, and 8 show the scatter plot visualization regarding the profiling results with twenty iterations in functions of a part of three type applications with CPU, memory, network, and disk IO metrics.

In the WordPress application case, CPU and memory metrics of all functions are converging at some values a certain period of time. We also see that the number of sending/ receiving networks is continually increasing because it is asked for network requests by the workload. Fig. 6(a) and 6(b) show that disk io is consistent because the workload does not send data into the application.

Fig. 7(a) and 7(b) show workload profiling results regarding some functions that make up the smart energy application. In the CPU metric case, the edgex-mongo microservice is converging at some values for a certain period of time. On the other hand, the kafka-broker-1 microservice goes back to stable values after CPU usage soared at a certain time. We observed that it is stable after some computing has been performed. In the case of the memory metric, two microservices seem to be converging on certain values except for a few unstable values. The network also metric increases steadily, similar to the previous application example, except for a few unstable values. For the disk IO, only the edgex-mong microservice continually increases because the workload continually sends data to the application.

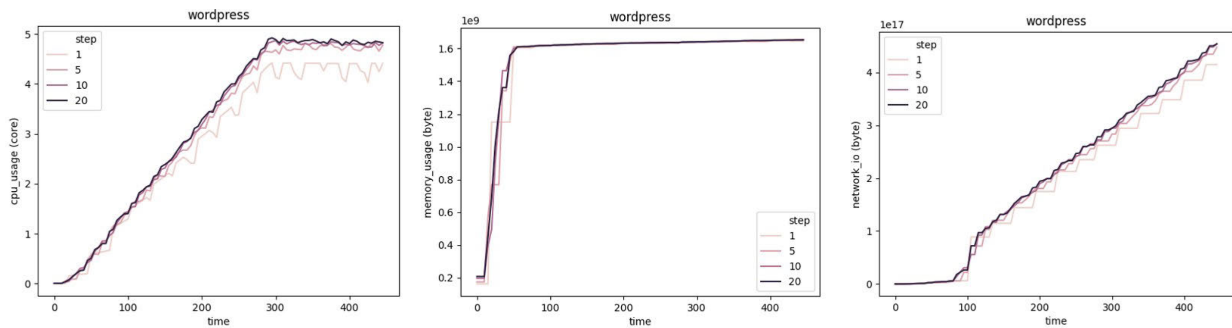


(a) Profiling for the front-end microservice (CPU, memory, network, and disk).

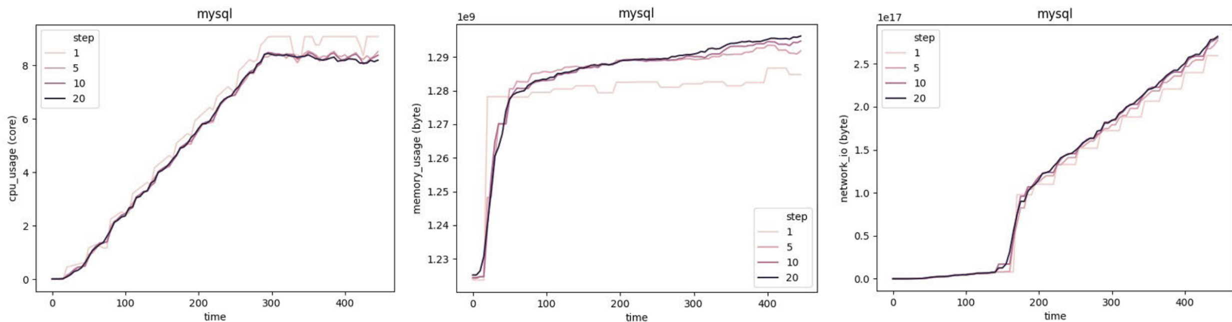


(b) Profiling for the rabbitmq microserivce (CPU, memory, network, and disk).

FIGURE 8. **Empirical workload profiling for a part of sock shop application.**



(a) The comparison of results in the wordpress microservice (CPU, memory, network).



(b) The comparison of results in the mysql microservice (CPU, memory, network).

FIGURE 9. **The comparison of results according to the number of profiling iterations for WordPress application.**

Fig. 8(a) and 8(b), which are part of the sock shop application, also show the graph is similar to the above examples. The CPU and memory utilization in two microservices are converging to the constant value after a certain period. The amount of sending/receiving networking is also steadily increasing, excluding some unstable values. Disk IO remains at the constant value because the workload does not send/writing data to the application.

After we analyze profiling results for three types of application, we decide to exclude the disk IO metric as the placement consideration because it's hard to find noticeable disk IO features in most of the microservices. In addition, for the issues regarding data processing that the cloud-native environment deals with storage and computing separately, in our example, improving performance in Ceph storage itself seems much better. Therefore we decide to select CPU, memory, and network having apparent profiling features as microservices placement considerations.

Fig. 9 shows the comparison of results according to the number of profiling iterations for the WordPress application.

For measuring similarity, many measurements in the same time period are calculated with the mean values. Even if profiling is performed for one time, we can effectively identify the applications' resource requirements depending on the workload, as shown in both Fig. 9(a) and Fig. 9(b). The resource requirements derived from performing profiling with five iterations seem to be more than 95% consistent with the profiling results of performing profiling with twenty iterations. Therefore, performing profiling once can also be effective, but we recommend doing workload profiling with five or more iterations to ensure a high-reliability level.

Table 2 presents the resource requirements of three types of applications derived from empirical workload profiling with twenty iterations. The CPU and memory metrics reflect the converging values of the profiling results. However, in the case of the network metric, it is not suitable as a resource requirement because it shows the total amount of send/receive transmission in the network. Thus, the network values are converted to network transmission per second. These results described in Table 2 are used in the microservices placement stage.

TABLE 2. Resource requirements in three types of applications derived from empirical profiling.

Application	Function	CPU	Mem	Interaction
Wordpress	wordpress	4.95	1,526.65	8.27e5
	mysql	8.37	1,239.21	4.13e5
Smart energy	api-server	0.01	57.48	3.83e-3
	chronograf	0.05	44.82	7.01e-4
	edgex-core-command	0.45	572.20	6.81e-3
	edgex-core-consul	0.01	31.47	20.69
	edgex-core-data	11.53	953.67	2.06e6
	edgex-core-metadata	0.53	667.57	2.89e-1
	edgex-mongo	2.53	1,430.51	2.07e6
	edgex-support-logging	0.68	585.55	0.24
	influxdb	0.01	31.47	7.03e-4
	kafka-broker-1	0.02	667.57	1.69e-2
	kafka-broker-2	0.03	686.64	1.61e-2
	kafka-broker-3	0.04	678.06	1.47e-2
	kafka-zookeeper	0.01	191.68	0.29
Sock shop	carts	1.04	839.23	2.48e-3
	carts-db	0.01	133.51	5.58e-3
	catalogue	0.01	10.25	1.69e-5
	catalogue-db	0.02	810.62	1.24e-5
	front-end	1.2	262.26	2.27e5
	orders	0.89	831.60	1.65e-3
	orders-db	0.01	133.51	1.03e-3
	payment	0.01	9.29	5.07e-9
	queue-master	0.42	2,765.65	8.27e-5
	rabbitmq	0.16	286.10	8.25e-5
	shipping	0.75	823.02	5.07e-9
	user	0.01	11.82	2.58e-3
	user-db	0.01	77.24	4.13e-4

† CPU = the number of cores, Mem = Mbyte, Interaction = Gbyte/sec

V. MICROSERVICES PLACEMENT ENHANCEMENT WITH THE FRAMEWORK

In this section, we apply a heuristic algorithm for microservices placement to validate our proposed framework. We first design an optimization model for our environment and introduce a greedy-based heuristic algorithm. Then we do experimental evaluation using the placement algorithm based on empirical profiling data. We finally discuss our research idea.

A. GREEDY-BASED HEURISTIC ALGORITHM FOR MICROSERVICES PLACEMENT

To validate our proposed framework design, we need experiments for microservices placement by applying a suitable algorithm. Many kinds of research have proposed great algorithms for microservices placement considering application performance. Since our work's key idea is not a placement algorithm but a profiling-based framework for microservices deployment, any algorithms can be adapted to our framework. Thus, we introduce simple heuristic algorithms using the greedy method for the verification of our work.

Before applying the heuristic algorithm within the framework, we describe a system model as follows. We consider multiple K8s clusters C with identification n, $C = \{c_1, c_2, .., c_n\}$. Our model assumes that all nodes in the same K8s cluster have the same computing processing power. To distinguish CPU performance per the K8s cluster, let $c_n^{cpu_per}$ denote the value of CPU performance per core on nth K8s cluster.

All of the K8s clusters have their physical/virtual hosts. Let H_n denote a collection of all physical hosts in the K8s cluster n. Let $h_{z,n}$ denote an individual host in Kubernetes cluster n. Each host is characterized by the tuple <cpu, mem> denoting the number of processing cores and memory units on the physical host, respectively. Every host is capable of running microservices in the form of containers. For the placement process, only the remaining resource capacity should be considered. We calculate the residual resource capacities denoted as $h_{z,n}^{cpu_res}$ and $h_{z,n}^{mem_res}$, respectively on each node using Eq. (1).

$$h_{z,n}^{u_res} = h_{z,n}^{u}(1 - h_{z,n}^{u_util});\ u \in \{cpu, mem\} \tag{1}$$

Cloud-native applications consist of multiple microservices. The microservices are run on containers hosted in the nodes. Let M be a set of microservices in the cloud-native application. Let m_i be ith microservice in M. Each microservice requires resources with CPU and memory denoted as $m_i^{cpu_req}$ $m_i^{mem_req}$, respectively. We also denote m_i^{int} as interactions rate by the ith microservice. Table 3 presents described notations in our system model.

Our problem space can be formulated as a (0/1) Quadratic Programming Problem with linear constraints. The binary decision variables are defined in Eq. (2), (3).

$$x_{c_n} = \begin{cases} 1, & \text{if } c_n \text{ is selected as cluster for placement} \\ 0, & otherwise \end{cases} \tag{2}$$

$$y_{m_i h_{z,n}} = \begin{cases} 1, & \text{if } m_i \text{ is allocated on node } h_{z,n} \\ 0, & otherwise \end{cases} \tag{3}$$

MSA-based cloud applications are network sensitive because different functions interact with each other. The simplest optimal microservices placement is that microservices having frequent interactions are placed on the same node or next to each other as much as possible [36], [37]. To apply this methodology, we also define a parameter, node interaction

TABLE 3. Notations used in the microservices placement model.

Notation	Description
C	Set of K8s clusters
c_n	The K8s cluster with identification n
$c_n^{cpu_per}$	CPU processing performance per core on nth K8s cluster c_n
H_n	Set of all physical hosts in the K8s cluster n
$h_{z,n}$	zth physical host in the K8s cluster n, $\forall_{z=1}^{N_n} h_{z,n} \in H_n$
$h_{z,n}^{cpu}$	The number of processing cores on zth physical host $h_{z,n}$
$h_{z,n}^{cpu_util}$	CPU utilization of zth physical host $h_{z,n}$
$h_{z,n}^{cpu_res}$	Residual processing cores on zth physical host $h_{z,n}$
$h_{z,n}^{mem}$	Amount of memory on zth physical host h_z
$h_{z,n}^{mem_util}$	Memory utilization of zth physical host $h_{z,n}$
$h_{z,n}^{mem_res}$	Residual memory capacity on zth physical host $h_{z,n}$
M	Set of microservices in the cloud-native application
m_i	ith microservice, $\forall_{i=1}^{M} m_i \in M$
$m_i^{cpu_req}$	The number of processing cores requested by ith microservice
$m_i^{mem_req}$	Amount of memory requested by ith microservice
m_i^{int}	Interactions rate by ith microservice

denoted as $h_{z,n}^{int}$ in Eq. (4).

$$h_{z,n}^{int} = \left(\sum_{i=1}^{|M|} m_i^{int} * y_{m_i h_{z,n}}\right) / \sum_{i=1}^{|M|} y_{m_i h_{z,n}} \tag{4}$$

The objective of the problem is to minimize node interactions as follows.

$$Minimize \sum_{z=1}^{|H_n|} \sum_{n=1}^{|C|} \sum_{i=1}^{|M|} h_{z,n}^{int} * x_{c_n} * y_{m_i h_{z,n}}$$

subject to the following constraints:

$$\forall h_{z,n} \in H_n, \sum_{i=1}^{|M|} m_i^{cpu_req} * y_{m_i h_{z,n}} \leq h_{z,n}^{cpu_res} \tag{5}$$

$$\forall h_{z,n} \in H_n, \sum_{i=1}^{|M|} m_i^{mem_req} * y_{m_i h_{z,n}} \leq h_{z,n}^{mem_res} \tag{6}$$

$$\forall m_i \in M, \sum_{n=1}^{|C|} \sum_{z=1}^{|H_n|} y_{m_i h_{z,n}} = 1 \tag{7}$$

$$\forall c_n \in C, \sum_{n=1}^{|C|} x_{c_n} = 1 \tag{8}$$

The Eqs. (5) and (6) are the constraints for the processing and memory requirements. The total amount of resources requested by all microservices placed on the host must not exceed the host's residual resource capacities. The Eq. (7) ensures that each microservice is placed on one specific host. The Eq. (8) also ensures that only one K8s cluster is selected for microservices placement.

The described system model can be calculated in the case of small problem sizes. However, with an increase in problem size, the calculation time grows exponentially, making it infeasible to correctly derive the solution. Thus, we consider a greedy-based heuristic algorithm to be more computationally feasible for large problem spaces than classical optimization techniques. The detailed procedure of the proposed greedy algorithm is given as Algorithm 1.

Algorithm 1 Greedy-Based Heuristic Algorithm

```
Input: C, H_n, M
Output: Allocated list, c_k
1:  Initialize k, x, y as 1 // Set index
2:  Sort C in non-increasing order of c_n^{cpu_per}
3:  Create a list, Allocated with size as |M| and initialize with
    zeros
4:  while cluster has not been selected do
5:     if k > |C| then
6:        Microservices placement failed, exit
7:     else if (The total of resource requirements of CPU,
       Mem in M) < (The total of residual resources of CPU,
       Mem in all hosts on c_k cluster) then
8:        select K8s cluster k
9:     else
10:       k = k + 1
11:    end if
12: end while
13: Sort M in non-increasing order of m_i^{int}
14: Sort H_k in non-increasing order of h_{y,k}^{cpu_res} //cpu-intensive

15: while all microservices have not been placed do
16:    while microservice x have not been placed do
17:       if y > |H_k| then
18:          Allocated = [0] //list initialization again
19:          x = 1, y = 1 //index initialization again
20:          placement failed, goto Line 4
21:       else if m_x^{cpu_req} < h_{y,k}^{cpu_res} and m_x^{mem_req} < h_{y,k}^{mem_res}
          then
22:          Assign Allocated[m_x] = y
23:          x = x + 1
24:       else
25:          y = y + 1
26:       end if
27:    end while
28: end while
```

In the proposed algorithm's initial step, we set three variables to track the index in clusters, nodes, and microservices. The elements in the K8s clusters are then sorted in non-increasing order of CPU performance. A list, *Allocated*, is initialized with zeros. *Allocated* plays a role in keeping track of the node allocated for each microservice.

Lines 4-12 select one of the clusters available in the set of K8s clusters. Since all microservices should be placed on the same cluster in our case, we first select one cluster. This process roughly selects clusters in CPU performance order that can handle the total amount of resource capacities in all microservices. Note that this process may do not guarantee the deployment of all microservices in the selected cluster.

Once the cluster is selected, the set of microservices in M are sorted in non-increasing order with m_i^{int}. The hosts in the K8s cluster k are also sorted in non-increasing order of $h_{z,k}^{cpu_res}$. Lines to 15-28 place microservices on the available node in order of high interaction value in the microservice. The algorithm attempts to place as many microservices as possible on the same node as long as the node's resource capacity is allowed. If even one microservice fails to be placed on the node during the above process, it goes to line 4 for the available cluster selection.

On completion of the algorithm, the microservices allocation can be obtained from the *Allocated* list. The index of the list denotes the microservice identifier. The value of the list is the identifier of the node in the K8s cluster c_n.

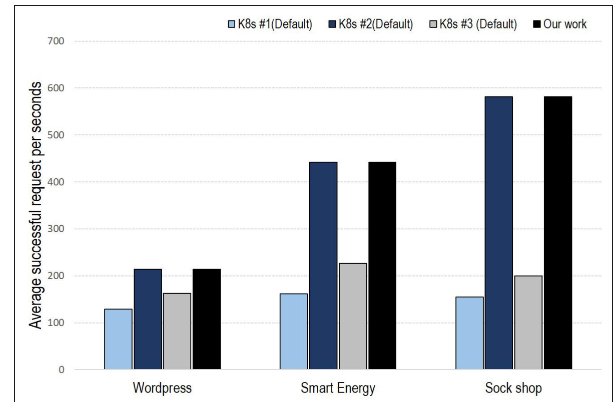
B. EXPERIMENTAL EVALUATION FOR MICROSERVICES PLACEMENT

Based on the profiling data in three application types with twenty iterations described as Table 2, we apply the proposed algorithm to decide microservices placement. We deploy microservices by changing the YAML description to match the *Allocated* list resulting from the algorithm. We also inject the same workload that sends 1,000 RESTful HTTP requests per second for measurement of transaction rate. However, in the case of measuring the response time in the application, we inject a workload that sends 200 requests per second because response time would increase exponentially when numerous requests come in.

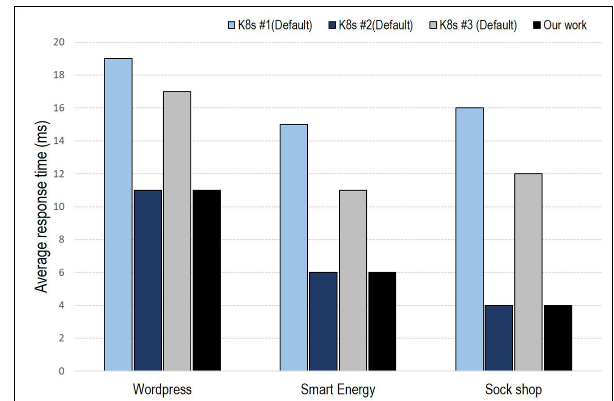
Fig. 10 shows the performance comparison graph regarding the average transaction rate and response time. Since default K8s has no option to select an appropriate cluster during multiple K8s clusters, we deploy the applications with the workload on each cluster, respectively, and measure the performance values for comparison to our proposed placement strategy.

Both Fig. 10(a) and Fig. 10(b) show the performance results that apply our deployment strategy are mostly similar to those placed in the K8s cluster #2. Because the algorithm gets results in the simple deployment strategy in selecting the node of K8s cluster #2 since the CPU performance of the K8s cluster #2 is most powerful, and the cluster only consists of a single worker node. In the case of the K8s cluster #3, CPU performance is the same as the K8s cluster #2, but performance results seem to be less due to many tasks loaded on the K8s cluster #3. To clearly compare our proposed methods in more complex situations, we experiment again limited to K8s cluster #1 that has multiple worker nodes.

Fig. 11 shows the experimental results on the performance comparison between our work and the default K8s placement policy on K8s cluster #1. Fig. 11(a) and Fig. 11(b) show the performance that applies our proposed work results in a little better than the default policy in K8s. These results seem to come out because our placement strategy tries to increase network performance by placing as many microservices with a high interaction value on the same node.



(a) Average transaction rate comparison across defaults and our work.



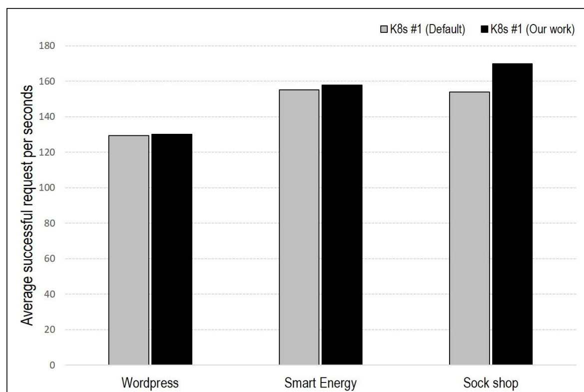
(b) Average response time comparison across defaults and our work.

FIGURE 10. Performance comparison across defaults (K8 cluster #1, #2, and #3, respectively) and our work.

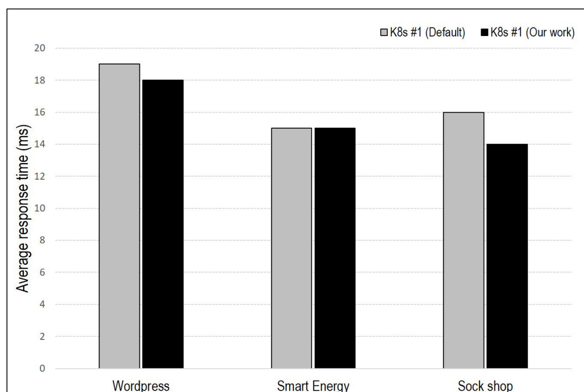
C. DISCUSSION

The resource requirements of the cloud-native application vary widely depending on the workload. So, we experiment with profiling regarding three types of cloud-native applications by selecting the workload. Based on the profiling results, we deploy the target applications by applying the proposed greedy-based algorithm. As we can see from the performance results, microservices placement based on empirical workload profiling is effective in the real environment. Profiling for one time is still effective, but performing profiling with many time iterations is easy to derive accurate resource requirements based on the workload, as depicted in Fig. 9. We have tentatively defined the recommended number of profiling iterations as five times in the previous section. However, we need to carefully examine those points by doing experiments that deploy the cloud-native application with various workloads.

The proposed framework focuses on the performance aspects of cloud-native applications. However, the monitoring component in the framework collects various resources metrics, so we can easily extend the metrics that can be considered in the algorithm to consider other aspects of the applications. Besides, cost-efficiency can be considered by adding the nodes' cost metrics that make up the K8s clusters.



(a) Average transaction rate comparison between default and our work.



(b) Average response time comparison between default and our work.

FIGURE 11. Performance comparison between default and our work on K8s cluster #1.

In the case of GPU matrices, we exclude it from our work scope. However, if the GPU virtualization can be applied in the K8s clusters, it may be possible to expand the GPU metrics' placement strategy.

Our work's main contribution lies in the framework that addresses microservices placement based on empirical workload profiling. Many kinds of researches on microservices placement are good for placement optimization regarding a particular aspect but omit the process of carefully deriving resource requirements based on workloads. Thus, if we leverage such a great algorithm in our framework, we can perform microservices deployment in more practical situations.

VI. CONCLUSION

In this article, we propose a refinement framework for microservices placement based on empirical workload profiling. To verify the proposed concept, we design the framework based on requirements. We perform profiling experiments with the proposed framework by selecting three types of applications with the workload. We also deploy the three types of applications on the experimental testbed by applying the greedy-based placement algorithm based on the profiling results. We verify that the application performance using our framework is much better than the results without our work.

Although we verify the proposed framework on our on-premise K8s clusters, this research has the potential to be leveraged for various public clouds such as Amazon AWS and Google GCP. In the future, we plan to expand and verify our proposed framework to the public clouds by considering both application performance and cost-efficiency.

REFERENCES

[1] B. Shojaiemehr, A. M. Rahmani, and N. N. Qader, "Automated negotiation for ensuring composite service requirements in cloud computing," *J. Syst. Archit.*, vol. 99, Oct. 2019, Art. no. 101632.

[2] J. Han, S. Park, and J. Kim, "Dynamic OverCloud: Realizing microservices-based IoT-cloud service composition over multiple clouds," *Electronics*, vol. 9, no. 6, p. 969, Jun. 2020.

[3] W. Hasselbring, "Microservices for scalability: Keynote talk abstract," in *Proc. 7th ACM/SPEC Int. Conf. Perform. Eng.*, Delft, The Netherlands, Mar. 2016, pp. 133–134.

[4] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: An experience report," in *Proc. Eur. Conf. Service-Oriented Cloud Comput.*, Taormina, Italy, Sep. 2015, pp. 201–215.

[5] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 16–21, Sep. 2017.

[6] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proc. 2nd ACM SIGOPS EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 275–287.

[7] J. P. Martin, A. Kandasamy, and K. Chandrasekaran, "Exploring the support for high performance applications in the container runtime environment," *Hum.-Centric Comput. Inf. Sci.*, vol. 8, no. 1, pp. 1–15, Dec. 2018.

[8] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[9] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure DevOps," in *Proc. IEEE Int. Conf. Cloud Eng. (ICE)*, Apr. 2016, pp. 202–211.

[10] E. Casalicchio, "Autonomic orchestration of containers: Problem definition and research challenges," in *Proc. 10th EAI Int. Conf. Perform. Eval. Methodol. Tools*, 2017, pp. 1–14.

[11] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 81–88, Sep. 2016.

[12] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Gener. Comput. Syst.*, vol. 28, no. 2, pp. 358–367, Feb. 2012.

[13] J. Li, D. Li, Y. Ye, and X. Lu, "Efficient multi-tenant virtual machine allocation in cloud data centers," *Tsinghua Sci. Technol.*, vol. 20, no. 1, pp. 81–89, Feb. 2015.

[14] L. Heilig, E. Lalla-Ruiz, and S. Voß, "A cloud brokerage approach for solving the resource management problem in multi-cloud environments," *Comput. Ind. Eng.*, vol. 95, pp. 16–26, May 2016.

[15] F. Legillon, N. Melab, D. Renard, and E.-G. Talbi, "Cost minimization of service deployment in a multi-cloud environment," in *Proc. IEEE Congr. Evol. Comput.*, Jun. 2013, pp. 2580–2587.

[16] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner, "Container-based architecture for flexible industrial control applications," *J. Syst. Archit.*, vol. 84, pp. 28–36, Mar. 2018.

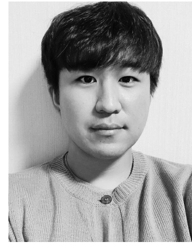
[17] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Apr. 2017, pp. 21–30.

[18] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Proc. IEEE 9th Int. Conf. Service-Oriented Comput. Appl. (SOCA)*, Nov. 2016, pp. 44–51.

[19] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: A case study in multi-cloud microservices-based applications," *J. Supercomput.*, vol. 74, no. 7, pp. 2956–2983, Jul. 2018.

[20] I.-D. Filip, F. Pop, C. Serbanescu, and C. Choi, "Microservices scheduling model over heterogeneous cloud-edge environments as support for IoT applications," *IEEE Internet Things J.*, vol. 5, no. 4, pp. 2672–2681, Aug. 2018.

[21] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, ''Application deployment using microservice and docker containers: Framework and optimization,'' *J. Netw. Comput. Appl.*, vol. 119, pp. 97–109, Oct. 2018.
[22] Z. Wen, T. Lin, R. Yang, S. Ji, R. Ranjan, A. Romanovsky, C. Lin, and J. Xu, ''GA-par: Dependable microservice orchestration framework for geo-distributed clouds,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 1, pp. 129–143, Jan. 2020.
[23] R. Buyya, M. A. Rodriguez, A. N. Toosi, and J. Park, ''Cost-efficient orchestration of containers in clouds: A vision, architectural elements, and future directions,'' 2018, *arXiv:1807.03578*. [Online]. Available: http://arxiv.org/abs/1807.03578
[24] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete, ''Four-fold auto-scaling on a contemporary deployment platform using docker containers,'' in *Proc. Int. Conf. Service-Oriented Comput.*, Berlin, Germany, 2015, pp. 316–323.
[25] M. Nardelli, C. Hochreiner, and S. Schulte, ''Elastic provisioning of virtual machines for container deployment,'' in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. Companion (ICPE Companion)*, 2017, pp. 5–10.
[26] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, ''A framework and algorithm for energy efficient container consolidation in cloud data centers,'' in *Proc. IEEE Int. Conf. Data Sci. Data Intensive Syst.*, Dec. 2015, pp. 368–375.
[27] C. T. Joseph and K. Chandrasekaran, ''IntMA: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments,'' *J. Syst. Archit.*, vol. 111, Dec. 2020, Art. no. 101785.
[28] B. Qureshi, ''Profile-based power-aware workflow scheduling framework for energy-efficient data centers,'' *Future Gener. Comput. Syst.*, vol. 94, pp. 453–467, May 2019.
[29] K. Ye, H. Shen, Y. Wang, and C. Xu, ''Multi-tier workload consolidations in the cloud: Profiling, modeling and optimization,'' *IEEE Trans. Cloud Comput.*, early access, Feb. 24, 2020, doi: 10.1109/TCC.2020.2975788.
[30] S. Lee, J. Han, J. Kwon, and J. Kim, ''Relocatable service composition based on microservice architecture for cloud-native IoT-cloud services,'' in *Proc. Asia–Pacific Adv. Netw.*, 2019, pp. 23–27.
[31] S. Kim and J. Kim, ''Designing smart energy IoT-cloud services for mini-scale data centers,'' in *Proc. KICS Winter Conf.*, Jeongseon, South Korea, Jan. 2017, pp. 18–21.
[32] J. Kwon, N. L. Kim, and J. Kim, ''Design and evaluation of cloud-native-based SmartX AI computing cluster supporting AI-enabled services,'' *KIISE Trans. Comput. Practices*, vol. 25, no. 12, pp. 571–584, Dec. 2019.
[33] *Locust*. Accessed: Sep. 14, 2020. [Online]. Available: https://locust.io/
[34] *Sock Shop*. Accessed: Sep. 14, 2020. [Online]. Available: https://github.com/microservices-demo/microservices-demo/
[35] J. Han, J. S. Shin, J. Kwon, and J. Kim, ''Cloud-native SmartX intelligence cluster for AI-inspired HPC/HPDA workloads,'' in *Proc. ACM/IEEE Supercomput. Conf.*, 2019, pp. 1–9.
[36] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, ''Geo-distributed efficient deployment of containers with kubernetes,'' *Comput. Commun.*, vol. 159, pp. 161–174, Jun. 2020.
[37] Y. Hu, C. de Laat, and Z. Zhao, ''Optimizing service placement for microservice architecture in clouds,'' *Appl. Sci.*, vol. 9, no. 21, p. 4663, Nov. 2019.



JUNGSU HAN received the B.S. degree in computer science engineering from Inha University, Incheon, South Korea, in 2014, and the M.S. degree from the School of Information and Communication, Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea, in 2016. He is currently pursuing the Ph.D. degree with the School of Electrical Engineering and Computing Science, GIST. His research interest includes cloud-native infrastructure orchestration with multitenants/clusters/sites operation.

YUJIN HONG graduated from the Chungbuk Science High School, in 2016, and a Backend Engineer Intern at Naver Webtoon, in 2020. She is currently pursuing the B.S. degree with the School of Electrical Engineering and Computing Science, Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea. Her research interest includes cloud-native infrastructure with high availability and fault tolerance.

JONGWON KIM (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in control and instrumentation engineering from Seoul National University, Seoul, South Korea, in 1987, 1989, and 1994, respectively.

From 1994 to 2001, he was a Faculty Member with Kongju National University, Gongju, South Korea, and the University of Southern California, Los Angeles, CA, USA. In 2001, he joined the Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea, where he is currently a Full Professor. Since 2008, he has been leading the GIST Super Computing and Collaboration Environment Technology Center as the Director. He is also leading the Networked Computing Systems Laboratory, where he is involved in dynamic and resource-aware composition of media-centric service employing programmable/virtualized computing/networking resources. Since 2020, he has been the Dean of the GIST AI Graduate School and the Chief Director of the GIST Institute for Artificial Intelligence. His recent research interest includes agile and visible p+v+c function-leveraged composition of the SmartX IoT-cloud services employing programmable/sliced/hyper-converged (computing/storage/networking) resources.