A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources

ZHIHENG ZHONG and RAJKUMAR BUYYA, University of Melbourne

Containers, as a lightweight application virtualization technology, have recently gained immense popularity in mainstream cluster management systems like Google Borg and Kubernetes. Prevalently adopted by these systems for task deployments of diverse workloads such as big data, web services, and IoT, they support agile application deployment, environmental consistency, OS distribution portability, application-centric management, and resource isolation. Although most of these systems are mature with advanced features, their optimization strategies are still tailored to the assumption of a static cluster. Elastic compute resources would enable heterogeneous resource management strategies in response to the dynamic business volume for various types of workloads. Hence, we propose a heterogeneous task allocation strategy for cost-efficient container orchestration through resource utilization optimization and elastic instance pricing with three main features. The first one is to support heterogeneous job configurations to optimize the initial placement of containers into existing resources by task packing. The second one is cluster size adjustment to meet the changing workload through autoscaling algorithms. The third one is a rescheduling mechanism to shut down underutilized VM instances for cost saving and reallocate the relevant jobs without losing task progress. We evaluate our approach in terms of cost and performance on the Australian National Cloud Infrastructure (Nectar). Our experiments demonstrate that the proposed strategy could reduce the overall cost by 23% to 32% for different types of cloud workload patterns when compared to the default Kubernetes framework.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Theory of computation** → **Scheduling algorithms**; • **General and reference** → Performance; • **Computing methodologies** → Model development and analysis;

Additional Key Words and Phrases: Cluster management, container orchestration, resource heterogeneity, cost efficiency

ACM Reference format:
Zhiheng Zhong and Rajkumar Buyya. 2020. A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources. *ACM Trans. Internet Technol.* 20, 2, Article 15 (April 2020), 24 pages.
https://doi.org/10.1145/3378447

This work was supported by the China Scholarship Council and the Australia Research Council Discovery Project.
Authors' address: Z. Zhong and R. Buyya, University of Melbourne, Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information System, Parkville Campus, Melbourne, Victoria, 3010, Australia; emails: zhiheng@student.unimelb.edu.au, rbuyya@unimelb.edu.au.

1533-5399/2020/04-ART15 $15.00
https://doi.org/10.1145/3378447

1 INTRODUCTION

Containers, as a lightweight application virtualization technology, provide a logical packing mechanism for application abstraction that packages software and dependencies together. Compared with virtual machines that support resource virtualization at hardware level, containers provide a virtual runtime environment based on a single operating system (OS) kernel and emulate an OS. Containers support resource sharing across multiple users and tasks concurrently rather than booting an entire OS for each application. This allows agile application deployment, environmental consistency, OS distribution portability, application-centric management, and resource isolation for container-based applications. These features have led to the continuously rising popularity and adoption of this technology in mainstream cloud computing platforms, whereas most of these state-of-the-art container orchestration systems are already mature with advanced features, such as CoreOS Fleet [1], Borg [2], and Kubernetes [3].

However, these existing systems remain with limited quality-of-service (QoS) management policies that mainly focus on infrastructure-level metrics, whereas specific application-level QoS requirements are usually ignored [4]. For instance, long-running web applications usually need to satisfy a condition where the number of requests processed per time unit must reach a certain level, whereas batch jobs are configured with a deadline as a time constraint. In addition, other factors including resource utilization, fault tolerance, and energy efficiency should be considered. Although the aforementioned systems address these issues to a certain degree, further research is needed to explore more sophisticated container-based application management strategies supporting heterogeneous QoS requirements.

Unlike traditional application management and organization-specific clusters, where schedulers are usually designed to manage a static pool of homogeneous machines with the same price and characteristics for specific workloads, heterogeneous compute clusters are more likely to be composed of various machine classes to satisfy the highly dynamic resource demands requested by different types of workloads [38]. For instance, the machine configurations in a Borg cluster could differ in multiple dimensions: sizes (CPU, RAM, disk, network), processor type, performance, and capabilities [2]. Besides resource requirements (CPU, memory, disk, etc.), Google workloads usually specify task placement constraints of particular classes of machines where they could be deployed to acquire machines with specific configurations or accelerators like GPUs. Such preferences or constraints complicate the decision-making process of task scheduling in heterogeneous clusters, as neither machines nor workloads could be treated equally. The existing scheduling approaches that only target on specific workloads or environments become broken with poor performance [22]. Given such a high degree of heterogeneity and variability in workload demand and cluster configuration, task allocation strategies become significantly complicated for reducing tasking scheduling delay and improving resource utilization [39].

Scheduling is a fundamental technique in container orchestration systems. Considering the heterogeneity of these systems and changes in the execution environment, it is a new challenge to present dedicated scheduling strategies for each group of resources in complex cloud computing environments [5]. However, the scheduling strategies in most of the existing frameworks are still tailored to the assumption of a static cluster of resources, with bin-packing algorithms commonly used to schedule containers on a fixed-sized cluster. This has become a bottleneck in terms of compute resource elasticity and service heterogeneities. No matter how optimal the initial task arrangement is, it will degrade as the resource utilization and workload change over time. For example, a cluster could easily become overloaded when handling an unexpected bursty load of tasks or underutilized when experiencing a drastic workload downgrade. Therefore, on-demand cluster size adjustment is an essential complementary method for better cost-efficient cluster management.

In light of this, supporting heterogeneous task configurations of resources such as VM types and sizes would be another possible direction to satisfy dedicated QoS requirements and cost efficiency with elastic pricing models [6]. For example, long-running web applications should be deployed on steady instances leased with a lower price per time unit for long-term cost saving, whereas batch jobs could be placed on unreliable rebated instances or compute-optimized nodes, depending on the QoS requirement. However, considering different VM sizes and prices during autoscaling in response to the workload fluctuation could produce more pricing models to minimize the cost compared to rigid homogeneous scaling strategies. Another possible optimization approach for container orchestration is resource utilization improvement through rescheduling and instance deprovisioning. For instance, when a compute node continuously remains in underutilized status, its tasks could be migrated to other nodes through task rescheduling so that it could be shut down to avoid resource wasting and reduce cost.

To address these issues in container orchestration without assuming a fixed-size pool of homogeneous compute resources like the traditional clusters, the dynamicity and elasticity in workload demand, resource management, and instance billing have become the key concerns [22]. To minimize the overall costs under the context of a heterogeneous cluster, it is required for the orchestration system to make proper decisions during scheduling and scaling in terms of managing task allocation, on-demand instance acquisition, application queuing delay, and resource utilization. Therefore, this article proposes a heterogeneous task allocation strategy (HTAS) for cost-efficient container orchestration through resource utilization optimization and elastic instance pricing. Our work makes the following three key contributions:

(1) We demonstrate how heterogeneous task allocation could be used to optimize the initial placement of containers by task packing.
(2) We identify an elastic instance pricing model for cluster size adjustment in response to the workload fluctuation at runtime through multiple autoscaling algorithms.
(3) We describe a rescheduling mechanism that adopts the container checkpointing technique to enable VM cleaning of underutilized instances for the purpose of cost saving without losing task progress.

The rest of the article is organized as follows. Section 2 introduces the original Kubernetes framework, followed by other related scheduling algorithms. Section 3 formalizes the problem definitions. Section 4 describes our system architecture and key components, whereas algorithm implementations are given in Section 5. In Section 6, we present the experiment environment and evaluation results of the proposed approach in terms of cost and performance. Finally, our conclusion and future directions are discussed in Section 7.

2 BACKGROUND AND RELATED WORK

This section briefly presents a review of the default Kubernetes framework and related resource management algorithms for orchestration of container-based applications.

2.1 Kubernetes

Inspired by its internal container-oriented cluster management system Borg [2], Google developed Kubernetes (K8s), an open source container orchestration system for automated task deployment and cluster resource management with high flexibility and scalability [7]. Based on a container-oriented resource allocation policy, it automates the orchestration process of computation resources, storage system, and networking for various types of workloads, such as stateless, stateful, and data-processing workloads [8].

2.1.1 Key Components. The key components within a K8s cloud cluster referenced in this work are described as follows:

(1) The master node manages the cluster's control panel and handles global events for the cluster, such as scheduling, node provisioning, and deprovisioning. Designed to scale more instances horizontally, its API server exposes the K8s API like a front-end of the control panel. The default kube scheduler also sits within the master node, which monitors the new tasks and manages their resource allocation.
(2) Controlled by the master node, a worker node could be a VM instance or physical machine, maintaining the running applications and the K8s runtime environment.
(3) A kubelet is the primary "node agent" running on each node, which monitors and manages all of the containers created by K8s.
(4) As the basic element in K8s, a pod represents a running process at the lowest level within the cluster. A pod could be configured with a set of specifications, such as processing power requirement (CPU, memory, disk, etc.), execution policy, and application image.

2.1.2 Scheduling. The efficient initial placement of a pod is a key concern in terms of resource utilization and system performance [35]. Every new pod created in a K8s cluster should be added to a pending queue by default, waiting for deployments. The default K8s scheduler continuously picks pods from the queue and assigns them to nodes with enough requested resources. For each pending pod, a two-phase selection process is designed to determine the best-fit node within the cluster:

(1) *Filtering*: This is the first decision made by the scheduler to pick out the nodes that satisfy all requirements of the pod and ignore the rest in the next phase.
(2) *Ranking*: As there is usually more than one node selected during the filtering phase, these nodes are then scored and ranked by certain priority methods. Currently, the least requested priority (LRP) algorithm is referenced as the default priority method during ranking. The node with the lowest resource utilization is most preferred for pod allocation under this priority algorithm, where CPU and memory are evaluated with equal values. Note that this priority function has the effect of spreading pods across the nodes for load balance and application protection against possible resource shortage. Nevertheless, this is only a baseline approach for preventing resource overloading. For complex task co-location and resource overbooking models addressed in Guo et al. [13] and Liu and Yu [16], the current Kubernetes scheduling mechanism cannot efficiently manage the risk of resource overloading when multiple applications on the same compute nodes are competing for the same type of resources, such as memory.

2.1.3 Autoscaling and Migration. The K8s Cluster Autoscaler (CA) is a stand-alone program that periodically adjusts the size of a K8s cluster in response to workload fluctuation. CA checks for any pending pods at a configurable frequency. A pod is defined as unschedulable when the K8s scheduler is unable to find a suitable node that can accommodate the pod. Hence, node provisioning could be triggered when there are pods that cannot be scheduled on any of the current nodes due to insufficient resources. It is assumed that the cluster is running on top of a homogeneous node group, where all machines have identical capacity and the same set of assigned labels. Therefore, increasing the size of a node group will create a new machine that is similar to those already in the cluster. Although the primary function of CA is to apply extra resources for the deployment of pending pods, only homogeneous scaling up with one instance at a time is supported by CA, which limits the compute resource elasticity and service heterogeneities. For instance, if the workload volume significantly rises over the scale of the instance size of the current node group, it might take several autoscaling cycles to boot up enough machines for the pending pods,

Table 1. Comparison of Related Platforms

Platform	Container Technology	Workload	Rescheduling	Cluster Infrastructure	Cluster Elasticity
Borg [2]	Linux cgroups based	All	✓	Physical	Static
Kubernetes [3]	Docker, rkt, CRI, OCI	All	✓	Virtualized, physical	Elastic, manual and autoscaling
Swarm [9]	Docker	Long-running jobs	✓	Virtualized, physical	Elastic, manual scaling
Mesos [10]	Docker, Mesos containers	All		Virtualized, physical	Elastic, manual scaling
Marathon [11]	Docker, Mesos containers	Long-running jobs	✓	Virtualized, physical	Elastic, manual scaling
Aurora [12]	Docker, Mesos containers	Long-running and cron jobs	✓	Virtualized, physical	Elastic, manual scaling
OpenShift [40]	Docker, Gear	All	✓	Virtualized, physical	Elastic, manual and autoscaling

which leads to long-term cluster overloading and severe QoS degradation. However, the cost of a set of relatively small instances could be potentially higher than an equivalent large machine. For compute instances continuously staying in underutilized status, CA also manages the node deprovisioning process after migrating all of their existing pods somewhere else. As K8s does not support container checkpointing or live migration, the current migration logic would evict and redeploy the target pod by force, which would directly cause QoS degradation. Furthermore, this limited deprovisioning feature cannot efficiently prevent resource wasting from overprovisioning or dramatic workload downgrade.

As summarized in Table 1, most of these mainstream platforms support advanced orchestration features for specific types of workloads with regard to resource demand profiling and estimation. However, their monitoring systems mainly focus on resource utilization metrics at the infrastructure level, such as CPU and memory usage, whereas QoS metrics at the application level are usually ignored. In addition, there are no existing methodologies in terms of workload modeling for container-based applications [35]. There is a potential demand for utilizing machine learning approaches for analysis and prediction of the workload characterizations and arrival patterns in a representative way to address complex task co-location scenarios [13]. A task rescheduling mechanism is employed by most of these systems as an approach for load balance, cost saving, and energy efficiency. However, their current rescheduling techniques, including forced task eviction, checkpointing, and live migration, would still cause a certain level of QoS degradation. How to accurately estimate and control the costs of rescheduling algorithms remains an open research challenge. Among these platforms, Kubernetes and OpenShift [40] support cluster-level autoscaling to resize the cluster when task scheduling fails due to insufficient resources. Nevertheless, it is only allowed to scale up one homogeneous instance per scaling cycle in their autoscaling strategies, which limits the scalability in handling highly dynamic workloads. Therefore, cluster-level autoscaling with heterogeneous resources is another possible enhancement regarding cost efficiency and QoS management.

2.2 Resource Management and Scheduling Algorithms for Container Clouds

Container-based cloud applications and systems with sophisticated resource management algorithms have been researched in various experience studies, aside from the production platforms mentioned earlier. Table 2 summarizes the comparison of related works focused on resource man-

Table 2. Comparison of Related Algorithmic Works and Our Work

Work	Objective	Platform	Workload	Cluster Elasticity	Cluster Infrastructure	Resource Heterogeneity
Guo et al. [13]	Resource utilization improvement	Fuxi [48], Sigma [49]	Long-running and batch jobs	Static	Physical	
Zhang et al. [14]	Resource utilization optimization	Docker	Network intensive	Static	Physical	✓
Kaewkasi and Chuenmuneewong [15]	Load Balance	Docker	Long-running service	Static	Physical	✓
Liu and Yu [16]	Resource utilization improvement	Fuxi, Sigma	Long-running and batch jobs	Static	Physical	
Guerrero et al. [17]	Resource allocation and elasticity optimization	Kubernetes	Simulation	Static	Physical	✓
Kehrer and Blochinger [18]	Task deployment automation	Mesos	All	Static	Virtualized	
Xu et al. [19]	Energy saving	Swarm	Long-running service	Elastic	Physical	
Xu and Buyya [43]	Energy saving	Swarm	Long-running service	Static	Physical	
Xu et al. [44]	Response time reduction	Docker	Simulation	Static	Physical	
Yin et al. [45]	Task delay reduction	Docker	Simulation	Static	Physical	
Taherizadeh and Stankovski [20]	QoS and resource elasticity assurance	Docker and CoreOS	Simulation	Static	Virtualized	
Paščinsk et al. [36]	QoS optimization	Kubernetes	Network intensive	Elastic	Virtualized	✓
Kochovski et al. [37]	QoS and NFR assurance	SWITCH [47]	Network intensive	Elastic	Virtualized	✓
Stratus [21]	Cost saving	Kubernetes	Batch jobs	Elastic	Virtualized	✓
Rodriguez and Buyya [22]	Cost saving	Kubernetes	Long-running and batch jobs	Elastic	Virtualized	
HTAS	Cost saving	Kubernetes	Long-running and batch jobs	Elastic	Virtualized	✓

agement and scheduling algorithms with our proposed approach (HTAS). HTAS is comprehensive and supports resource heterogeneity for automated deployment of long-running and batch jobs.

Private cluster management. Private clusters usually have a static pool of resources with whatever configurations. The current state-of-the-art works [15, 17, 18, 20, 44, 45] have tried to optimize task allocation within the fixed set of instances from various perspectives. However, such solutions would fail to manage costs and QoS requirements efficiently under the context of an elastic cluster where its size and composition could be adjusted on demand. The resource rental costs and task queuing delay could change in response to any cluster adjustment.

QoS-aware orchestration. Given the polyglot nature of container-based applications where different applications are implemented in various languages and data ecosystems, it significantly raises the complexity of tracking and managing QoS requirements for individual applications [35]. Kochovski et al. [37] proposed a formal QoS assurance method (FoQoSAM) based on stochastic Markov models. It manages to make the optimal task deployment decision by ranking the available cloud resources according to the QoS monitoring data and user-provided metrics. However, it is common to evaluate the QoS metrics under a task-per-instance assumption in such studies, ignoring the dynamically changing environment where any kinds of task packing or workload co-location could change the cost efficiency and overall system performance.

Workload characterization. The diversity of workloads is becoming an essential characteristic in modern data centers [50]. Through workload characterization of long-running services (diurnal pattern) and batch jobs (nocturnal pattern) in the Alibaba data center [13, 16], Fuxi [48] manages the task overcommitment process by assigning more batch jobs to the co-located cluster at midnight when long-running services stay inactive. Although this approach achieves better resource utilization at the infrastructure level, there remains a potential resource mismatch caused by unpredicted workload spikes, which leads to frequent task evictions and rescheduling. Balancing the tradeoff between resource utilization and QoS degradation caused by rescheduling through accurate workload characterization remains a major concern in task co-location.

Application-specific scheduling. Considering the unique nature and characteristics of different application workloads, there is a rich history of developing application-specific scheduling approaches. Within SDDCs [46], Paščinsk et al. [36] developed a Kubernetes-based Global Cluster Manager specialized in geographic orchestration of network-intensive workloads. It supports autonomic task arrangement by picking the best-fit geographic instance according to application-specific QoS models. By utilizing service similarity matching and time-series nearest neighbor regression to predict future resource need, Zhang et al. [14] present a container-based novel video surveillance system supporting dynamic resource utilization optimization and QoS awareness. Stratus [21] is a container-based cluster scheduler designed for batch job scheduling on public IaaS platforms. To achieve high cost efficiency and cluster utilization, it follows an aggressive scheduling mechanism by scheduling jobs as tightly as possible and shutting down underutilized instances. A representative heterogeneous workload in modern data centers commonly contains both batch jobs and long-running services [2, 13, 16]. To build a robust solution for cost-efficient orchestration of such workloads, our previous study [22] designed a customized scheduler on top of the Kubernetes platform by extending the existing rescheduling feature for better arrangements of task co-location. Nevertheless, it fails to solve the bottleneck of resource elasticity and QoS degradation caused by frequent task evictions. To address these problems, this work is a natural extension and improvement of it, and we include it in the evaluations.

Energy profiling and energy awareness. Xu et al. [19] and Xu and Buyya [43] managed to apply a brownout mechanism in container-based data centers through resource overbooking, autoscaling, and power-saving techniques, which dynamically deactivates optional containers or physical machines for energy consumption control in response to workload changes. Such research draws a parallel to our study, which aims to minimize cloud resource rental costs through efficient task packing and resource utilization optimization. Provisioning/deprovisioning of a VM instance from the public cloud is equivalent to activating/deactivating a physical machine.

3 PROBLEM FORMALIZATION

This section enumerates our assumptions for container-based applications and cloud resources, followed by the problem definition.

3.1 Assumptions

We assume a container orchestration framework deployed on top of an IaaS cloud environment as a service provider, which has unlimited access to VM instances for on-demand resource provisioning and deprovisioning. Cloud applications could be submitted to the service provider by multiple users simultaneously and charged by the configurations of requested resources (CPU, memory, storage, etc.) and billing periods. Our motivation is to optimize this resource management process with the lowest amount of costs. Hence, our assumptions are listed next.

3.1.1 Workload Models. We consider workload models consisting of two types of containerized tasks:

(1) Long-running services that require high availability, stability, and low latency, such as web-based applications and databases.
(2) Batch jobs that have a limited lifetime from a few seconds to a few days with looser performance requirements. These jobs could tolerate being stopped and migrated to another compute node.

These are the two most typical workloads used by most existing cluster management systems, such as Borg and Alibaba. We assume that all tasks specify the amount of requested resources with the best-effort QoS requirement, including CPU and memory. To ensure successful task deployments, tasks will only be allocated to nodes with at least their required amount of resources available. Since resource heterogeneity, elasticity, and task migration are the key concerns in this work, workload characterization is limited to long-running services and batch jobs without further considerations on detailed workload behaviors such as resource consumption patterns or task dependencies. Application modeling for complex scenarios, such as resource co-location, overbooking, and time-based workload prediction discussed in other works [13, 16, 24, 38], is not considered within the scope of this work.

As Kubernetes does not support live pod migration or replicating new pods from checkpoint images, we employ a task migration approach based on Checkpoint/Restore in Userspace (CRIU) [29] to simulate the migration process with the following assumptions:

(1) Tasks could be migrated without progress loss.
(2) There are no hard scheduling constraints defined in job configurations
(3) Both long-running services and batch jobs are configured as single-container pods, as the "one-container-per-pod" model is the most common Kubernetes use case. Multicontainer pods composed of modular containers with inner task dependencies, such as Sidecar Containers and Ambassador Containers [41], are beyond the scope of our study.
(4) Each task is stand-alone without dependencies to others. To simplify the overall task structure, workflows with directed acyclic graphs of tasks are not considered [42].

3.1.2 Resource Configurations. Kubernetes is a cloud-native platform that not only manages container orchestration but also utilizes cloud provider capabilities, such as resource elasticity, autoscaling, IP allocation, security, monitoring, load balancing, and multiregion deployment. Hence, the public cloud is the ideal fit for Kubernetes. It supports a virtual cluster to be resized over time on an on-demand basis with heterogeneous VMs of various dimensions, such as resource capacities (CPU, RAM, etc.), processor type, and performance. Furthermore, costs could be evaluated explicitly based on the rental prices of the underlying instance types. Although hosting a Kubernetes cluster on top of physical servers could save platform virtualization cost for VM provisioning with a potentially higher performance ratio and shorter response time, the management and maintenance costs are excessively high and time consuming. Each physical server must be configured and maintained manually in terms of node initialization, failure recovery, IP allocation, volume management, and load balancing [3]. Autoscaling and multiregion deployment will not be plausible. Therefore, we only consider virtual Kubernetes clusters composed of heterogeneous VMs in our resource configuration assumptions:

(1) The IaaS cloud environment provides heterogeneous VMs with various configurations (CPU, memory, storage, etc.) and prices. Following the prevalent pay-as-you-go billing standard at mainstream cloud providers, VM instances are charged by the minute,

assuming any partial usage rounded up to the closest minute. The expense of an instance is evaluated per minute with reference to the Microsoft Azure B-series instance type without impacting the significance of the results.

(2) VM instances could be provisioned or deprovisioned at any time. On the one hand, it may take a few minutes to boot up a new instance and merge it into the current cluster. Such a time gap is called an *instance acquisition lag*. On the other hand, it only needs a few seconds to shut down a running instance.

(3) During the evaluation process of autoscaling, each VM type is associated with a cost-efficiency score defined by its price p and normalized used constraining resource *nc*:

$$\text{Score} = \frac{nc}{p}.$$

This figure is referenced to find the instance with the lowest cost per resource used [21]. The normalized used constraining resource is calculated by computing the utilization for each resource type within the constraining scope, including CPU usage *cu* and memory usage *mu*:

$$nc = w1 \times cu + w2 \times mu.$$

To balance CPU and memory utilization during resource allocation, they are evaluated equally [34] with their weights, namely *w1* and *w2*, both set as 0.5.

3.2 Problem Definition

We adopt the cloud application model derived from Mao and Humphrey [25]. The performance of each VM is defined as a vector with reference to the predicted execution time of each task S_i, the cost per billing period c_v, and the evaluated acquisition lag lag_v:

$$VM_v = \left\{ [t_{S_i}]_v,\ c_v,\ lag_v \right\}.$$

Hence, the cost of each running task is $t_{S_i} \times c_v$. Based on this assumption, our orchestration strategy consists of three parts:

(1) *Scheduling*: The scheduling algorithm decides the best-fit resource allocation to compute node VMv for task S_i at time point t:

$$Schedule_t = \{S_i \rightarrow VMv\}.$$

(2) *Autoscaling*: It is decided by the autoscaling mechanism how many new instances N_v of each VM type VM_v should be provisioned at time point t in response to the workload fluctuation within the cloud cluster:

$$Scaling_t = \{VM_v,\ N_v\}.$$

(3) *Rescheduling*: At time point t, the rescheduling algorithm picks out each VM instance VM_v of which the resource utilization ratio u_i remains below a threshold for a certain amount of time, under circumstances where there is enough space on the other nodes to deploy the tasks S_v currently running on it. Furthermore, this instance is shut down after all of its tasks are migrated to other nodes:

$$Rescheduling_t = \{VM_v,\ S_v\}.$$

Therefore, our primary goal is to develop a solution with these three parts, which minimizes the overall cost C:

$$Min\,(C) = Min\left(\sum_v c_v N_v\right).$$

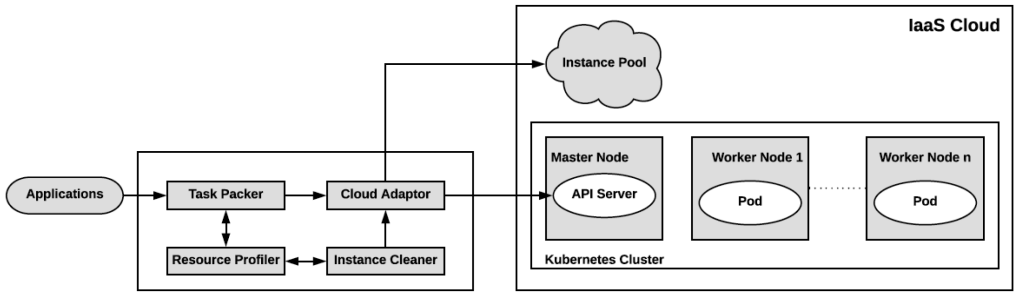

Fig. 1. System architecture.

4 SYSTEM ARCHITECTURE

As is depicted in Figure 1, our approach is built as a customized scheduling component on top of the K8s platform. Worker nodes are divided into two node groups with heterogeneous configurations, including a batch node group with short-living VMs that only handle batch jobs and a long-running node group with steady, cost-saving instances that only accept long-running services. Each node group is bound with different scheduling and autoscaling algorithms. Its task processing lifecycle contains the following steps:

(1) Each new task is submitted to the application queue with a metadata file specifying its name, type, container image, and amount of requested resources.
(2) Task Packer periodically consumes the application queue and classifies the pending jobs by their types and estimated runtime. It decides when and where tasks are scheduled.
(3) Task Packer acquires the latest VM instance list from Resource Profiler and generates job-instance name bindings for further deployments. If there are not enough resources in the cluster, extra instances will be acquired from the IaaS cloud through autoscaling.
(4) The cloud adaptor reads the name bindings sent from Task Packer and accordingly manages job deployments in the K8s cluster.
(5) At the end of each scheduling cycle, if the application queue stays empty without any new jobs submitted for a configurable period (e.g., 5 minutes), the VM cleaner will shut down those underutilized VMs in the batch node group after migrating the existing jobs on them to other available VMs.

4.1 Resource Profiler

Resource Profiler keeps a list of all available instances in the K8s cluster with their types, capacities, latest statuses, and available resources, among others. For a large-scale container orchestration system, it is computationally expensive to retrieve the list of available instances directly from the API server in every scheduling cycle. Therefore, it is necessary to keep a snapshot of the latest cluster status to reduce response time. Based on this assumption, Resource Profiler is built on top of the Cassandra database [28], with all VM instance information stored as time-series data in the data structure shown in Table 3.

The cluster status stored in Cassandra will be updated by Resource Profiler under the following scenarios:

(1) New task deployment
(2) Existing task termination
(3) Rescheduling
(4) VM provisioning
(5) VM deprovisioning.

Table 3. Data Structure in Cassandra

Column Name	Data Type
event_time	timestamp
instance_name	text
instance_type	text
runtime	int
ramcapacity	decimal
cpucapacity	decimal
ramavailable	decimal
cpuavailable	decimal

4.2 Cloud Adaptor

Implemented using K8s API and OpenStack API, Cloud Adaptor plays the role of a broker between the scheduling component and the K8s cluster with the following functionalities:

(1) Task deployment
(2) VM instance acquisition from the IaaS cloud into the K8s cluster
(3) VM instance deprovisioning
(4) Container migration.

Since K8s does not presently support live pod migration or container checkpointing, we implement a workaround solution based on the migration of a single container to preserve task progress with minimal service disruption. The CRIU project provides the functionality of checkpointing/restoring Docker-based containers under the Linux environment [23]. By adopting CRIU, a container could be live captured into a transmittable format, whereas its memory state could be transferred and stored in the image file through memory dumping [29]. Hence, the container coupled to a pod could be moved from the source pod to the destination pod without losing task progress. In case of any unexpected failures during the migration process, we choose the "leave-running" mode for memory dumping, which checkpoints the container and leaves it running afterward. As internal data transmission within the K8s cluster is quite limited and communications between nodes are not allowed at the orchestration level, container checkpoint images are manually transported between nodes through Linux SCP (Secure Copy Protocol) within our design. The overall migration process of a single-container pod consists of the following steps:

(1) Checkpoint the container belonging to the pod at the source node.
(2) Copy the container image to the destination node through SCP.
(3) Receive data until the checkpoint image arrives.
(4) Restore the container from the image.
(5) Once container migration is complete, eliminate the pod at the source node.
(6) If data transmission fails, it will be retried within a configurable number of times (three times by default). In case of any irrecoverable situations (e.g., destination node crashes), the migration process would be terminated with the original pod left running at the source node and all relevant changes rolled back.

As the manually migrated container in the destination node is not backed by any controller objects, it is not controlled by kubelet. Its execution status and resource allocation will be monitored and managed by Resource Profiler. Therefore, this approach is only feasible for emulating workload migrations across nodes.

4.3 Task Packer

Task Packer is a customized K8s scheduler that decides when and where a pod should be deployed. Its primary objective is to achieve high cost efficiency and quick deployments. It supports heterogeneous configurations for two main types of workloads: (1) long-running services that should never go down, such as web applications and databases, and (2) batch jobs that could take from a few seconds to a few days to finish. The core procedures in the task packing mechanism are described as follows:

(1) *Task bundling*: Jobs are periodically consumed from the application queue and classified into separate task groups, according to their job types and estimated runtime. In other words, batch jobs and long-running services are packed into two separate task groups. Furthermore, batch jobs are sorted by their runtime in descending order. The primary motivation of task bunding is tight task allocation by assigning tasks with the same type and similar runtime to the same compute node. Hence, these tasks would complete around the same time so that the node could be terminated for cost saving if the workload falls down.
(2) *Task scheduling*: With the current node groups retrieved from Resource Profiler in each scheduling cycle, each task group is mapped to its corresponding node group, according to its job type. Then job-instance name bindings are generated by scheduling algorithms in terms of task allocation.
(3) *Autoscaling*: If some tasks cannot be scheduled to any existing nodes due to resource shortage, autoscaling algorithms will be triggered to launch more instances as needed. Since there are multiple node groups within the cluster, Task Packer would determine which node group should be scaled up, based on the types and volumes of the pending tasks left after the scheduling phase.

4.4 Instance Cleaner

Instance Cleaner is an extension of the K8s CA deprovisioning feature. Within the scope of our work, only batch jobs are assumed to be tolerant of being frozen and migrated. However, long-running jobs are not considered feasible for task migrations with regard to the possible risk of impacting service stability (e.g., stateful web applications currently handling requests). By the end of each scheduling cycle when no batch jobs remain in the application queue, Instance Cleaner retrieves all underutilized instances from Resource Profiler in ascending order and starts the rescheduling process from the worst-utilized node. Since it is always more time consuming for provisioning than deprovisioning, Instance Cleaner should avoid frequent deprovisioning, which may lead to severe QoS degradation and high task delay.

5 ORCHESTRATION ALGORITHMS

Our primary goal is cost saving with heterogeneous resource configurations under the Kubernetes-based cloud environment in three ways: (1) by supporting heterogeneous task configuration to optimize the initial placement of containers, (2) by resizing the cluster through autoscaling in response to the workload fluctuation at runtime, and (3) by rescheduling and deprovisioning of underutilized VM instances.

5.1 Scheduling

For long-running services, the scheduling process could be regarded as an online version of the two-dimensional bin packing problem. Each pod is associated with two properties: CPU and memory requests. Our goal is to allocate these pods with the least number of VMs. As one

of the prevalent approximation algorithms to solve this scenario, the best fit decreasing (BFD) bin packing algorithm [26] would arrange each pod to the most utilized instance with enough requested resources. Assuming CPU as a compressible resource, memory is evaluated at a higher priority than CPU when ranking the available instances in BFD for each pod. Executing the BFD algorithm in Resource Profiler with n available instances has time complexity of $O(\log n)$.

However, batch jobs are scheduled starting from the longest task by an extended time-bin BFD algorithm. Each batch node is associated with a runtime field that represents the remaining runtime of the longest batch job deployed on the node. In each scheduling cycle, instances in the batch node group are divided into multiple time bins defined by their runtimes rt and scaling cycle sc:

$$bin = \frac{rt}{sc}.$$

The reason behind this is twofold: (1) grouping batch jobs with similar runtime and (2) resource utilization prediction within the next scaling cycle. Each bin_i contains tasks with remaining runtimes falling within its corresponding time interval $[i \times sc,\ (i+1) \times sc)$. At the initial stage, nodes from the same bin as the batch jobs are scanned through the BFD algorithm. If no node is eligible within its original bin, instances in progressively greater bins will be considered. After examination of all of the greater bins, the lesser bins could also be checked in descending order until a suitable instance is found. If no scheduling plan can be produced after all of the evaluations mentioned earlier, autoscaling algorithms will be invoked accordingly. In a batch node group of n bins where bins j to k are iterated in this function, the total time overhead T is

$$T = \sum_{i=j}^{k} t_i.$$

In the worst case, where $j = 0$ and $k = n - 1$, all bins are checked before finding a suitable node. When $j = k$ as the original time bin of the task, only one bin is examined as the best case.

ALGORITHM 1: BFD

Input: Pending pod p and node group ng
Output: Schedule plan $S = \{p \rightarrow node\}$

1. **select** *node*, min(*node*.availableRAM, *node*.availableCPU) from *ng*
2. **where** node.availableRAM >= *p*.memory && *node*.availableCPU >= *p*.CPU
3. $S = \{p \rightarrow node\}$
4. **If**(p.runtime > *node*.runtime) **then**
5. *node*.runtime = p.runtime
6. end **if**
7. return S

ALGORITHM 2: Time-bin BFD

Input: Pending pod p, batch node group ng, and scaling cycle sc
Output: Schedule plan $S = \{p \rightarrow node\}$

1. int bin = p.runtime/sc
2. **for**(int i = *bin*; i <= *ng.maxBin*; i++)
3. $S = \text{BFD}(p, ng_i)$
4. **If**(S != null) **then**
5. **return** S
6. **end if**

7. **end for**
8. **for**(int *i* = *bin; i* >= *ng.minBin*; *i*–)
9. *S* = BFD(*p*, ng_i)
10. **If**(*S* != null) **then**
11. **return** *S*
12. **end if**
13. **end for**
14. **return** null

5.2 Autoscaling

For a long-running node group, a greedy autoscaling (GA) algorithm is implemented to find a combination of heterogeneous VMs with the highest cost-efficiency scores in terms of the requested resource volume by iterating through the available instance flavor list multiple times. As for the batch node group, the total resource capacity of the current batch nodes sitting in the 0-bin is checked first, with their runtimes less than the scaling cycle. Since these resources can be released before any instance acquisitions, their capacity directly affects the autoscaling decision. Autoscaling will be considered unnecessary if the capacity could satisfy the need for all pending jobs. Otherwise, an enhanced GA algorithm is invoked with the 0-bin capacity deducted from the requested resource volume.

For scaled instance v, the time complexity of the scoring process within f flavors is $O(f)$. The node provisioning time T_v is the sum of scoring time T_s and instance acquisition lag L:

$$T_v = T_s + L.$$

Assuming that n instances are concurrently scaled up in response to p pending pods, the overall time overhead T_a of autoscaling is

$$T_a = \max_{i \in n} (T_i) .$$

ALGORITHM 3: GA

Input: Pending pods p and available VM instance flavors f
Output: VM instance combination vc

1. *tcpu* ← total CPU request sum(*p.CPU*)
2. *tram* ← total RAM request sum(*p.memory*)
3. **while**(*tcpu* > 0 || *tram* > 0)
4. f_i ← flavor with the highest cost-efficiency score max($score_f$)
5. tcpu = tcpu - f_i.CPU
6. tram = tram - f_i.RAM
7. vc.append(f_i)
8. **end while**
9. **return** *vc*

ALGORITHM 4: Batch node group autoscaling

Input: Pending pods *p*, batch node group *np*, and available VM instance flavors *f*
Output: VM instance combination *vc*

1. *tcpu* ← total CPU request sum(*p.CPU*)
2. *tram* ← total RAM request sum(*p.memory*)
3. **for** *node* in np_0

```
4.        tcpu = tcpu - node.CPUCapacity
5.        tram = tram - node.RAMCapacity
6.  end for
7.  if(tcpu <= 0 && tram <= 0) then
8.        return null
9.  end if
10. while(tcpu > 0 || tram > 0)
11.       f_i ← flavor with the highest cost-efficiency score max(score_f)
12.       tcpu = tcpu - f_i.CPU
13.       tram = tram - f_i.RAM
14.       vc.append(f_i)
15. end while
16. return vc
```

5.3 Rescheduling

If the resource utilization ratio of a batch node continuously stays under a configurable threshold (50% by default), its running batch jobs will be migrated by a rescheduling algorithm. If there is not enough space in the cluster to redeploy all containers in the pods, task migration will not be triggered. Otherwise, the underutilized node will be gracefully shut down for cost saving after rescheduling completes without losing task progress. For rescheduled pod m, the migration time T_m is decided by the container image size C, available bandwidth B, and task termination time K at the source node:

$$T_m = \frac{C}{B} + K.$$

Hence, the time overhead T_r of the whole rescheduling process includes the maximum migration time in all of the underlying pods p and a constant cost D for node shutdown:

$$T_r = \max_{i \in p} (T_i) + D.$$

ALGORITHM 5: Rescheduling

Input: Underutilized **batch** node n, its running pods p, and its corresponding node group ng

```
1.  an ← ng - n, other available nodes in ng
2.  for each p_i in p
3.        S_i = schedule(p_i, an)
4.        if(S_i is null)
5.              break;
6.        end if
7.  end for
8.  if(all S_i is not null) then
9.        deploy migrations defined by all S_i and shut down n
10. else
11.       rescheduling is not triggered due to resource shortage
12. end if
```

6 PERFORMANCE EVALUATION

To compare the cost and performance of the proposed approach (HTAS) with related algorithmic works, including the default K8s framework, COCA [22], and Stratus [21], we implemented our

Table 4. Task Configurations

Type	Name	Task	Memory Requests	CPU Requests
Batch job	batch_small	sleep 1–4 minutes	0.3 GiB	100 m
	batch_med	sleep 4–8 minutes	0.6 GiB	200 m
	batch_large	sleep 9–12 minutes	0.9 GiB	300 m
Long-running service	nginx	nginx server [30]	0.4 GiB	100 m

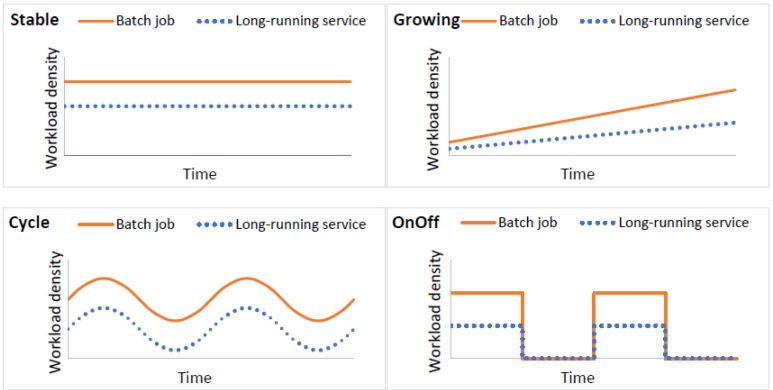

Fig. 2. Cloud workload patterns.

algorithms and carried out the empirical evaluation by deploying experiments on the Australian National Cloud Infrastructure (Nectar) [27].

6.1 Workload

We evaluate our approach through different cloud workload patterns by utilizing two types of synthetic applications, including batch jobs and long-running services. As depicted in Table 4, the workload models from Rodriguez and Buyya [22] are adopted in our experiments.

As shown in Figure 2, four representative workload scenarios are included in our experiments:

(1) A stable workload pattern represents situations where the workload density does not have any obvious fluctuations.
(2) A growing workload pattern may emulate a scenario where an application suddenly becomes popular and attracts a considerable number of visits, holding a high accelerating rate.
(3) A cycle/bursting workload could simulate online shopping websites where the network traffic experiences periodical changes and reaches a peak during certain time ranges.
(4) An on-and-off workload pattern depicts applications periodically having a short active stage, such as map-reduce jobs executed on a daily or weekly basis.

6.2 Testbed

Four Kubernetes-based virtual clusters are configured on Nectar Cloud to evaluate the proposed container orchestration algorithms under the workloads aforementioned. There are two types of nodes in each cluster:

(1) *Master node*: The master node manages the cluster's control panel and global state. The default K8S master is initialized with the original K8s scheduler and CA as mentioned in

Table 5. Initial Node Configurations

Node Type	VM Flavor	VCPU	RAM (GiB)	OS
Master	m3.small	2	4	Ubuntu 17.01
Worker	m1.medium	2	8	Ubuntu 17.01

Table 6. System Parameters

Parameter	Value
Scheduling cycle	20 seconds
Autoscaling cycle	5 minutes
The upper threshold of resource utilization ratio for underutilized nodes	50%
Maximum number of times for retrying container migration	3

Table 7. VM Pricing Details

Nectar Instance Flavor	VCPU	RAM (GiB)	Price ($/hour)
t3.xsmall	1	1	0.0198
m3.xsmall	1	2	0.0344
m3.small	2	4	0.0686
m1.medium	2	8	0.1371
m1.large	4	16	0.2746
m1.xlarge	8	32	0.5479

Section 2. COCA master adopts the default K8s CA with its customized best-fit scheduler and rescheduler that follows an aggressive compaction strategy through task eviction. Stratus master is configured with its customized LRP scheduler and cluster scaler based on HotSpot [51]. The extended scheduling components of our work described in Section 4 are deployed in HTAS master.

(2) *Worker node*: The worker node maintains the K8s runtime environment and applications assigned by master. Each worker node is initialized similarly with the default K8s kubelet.

Each cluster consists of one master node and two worker nodes, with reference to the node configurations described in Table 5 and other system parameters depicted in Table 6. To avoid significant costs of cross-region task migrations, we limit VM instance acquisitions to the *melbourne-qh2* region where the bandwidth per VM is 1,200 Mbps. Since Nectar Cloud provides free cloud service for researchers, the costs of resource usage are calculated according to the prevalent pay-as-you-go billing standard with reference to Microsoft Azure B-series heterogeneous instance types detailed in Table 7. The VMs selected are of different types and capabilities, making it a heterogeneous computing environment.

6.3 Results

Each workload is repeated for 10 iterations to calculate the average values of significant figures for verification of the results with higher validity. Hence, all test results shown are mean values over 10 runs with the error bars representing the maximum and minimum. As shown in Figures 3 through 8, each solution is primarily evaluated based on total cost, average task scheduling time, resource utilization, number of running worker nodes, rescheduling frequency, and average time overhead of task migration. We define scheduling duration as the time gap from the moment when

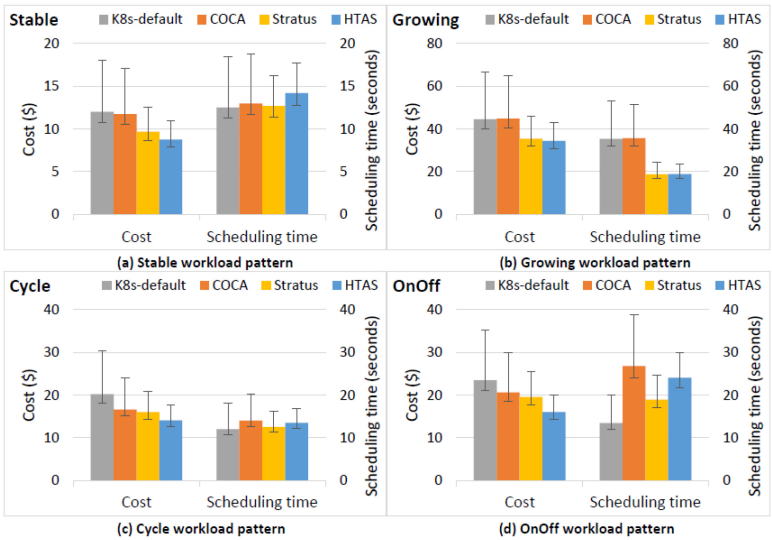

Fig. 3. Total cost and average task scheduling time.

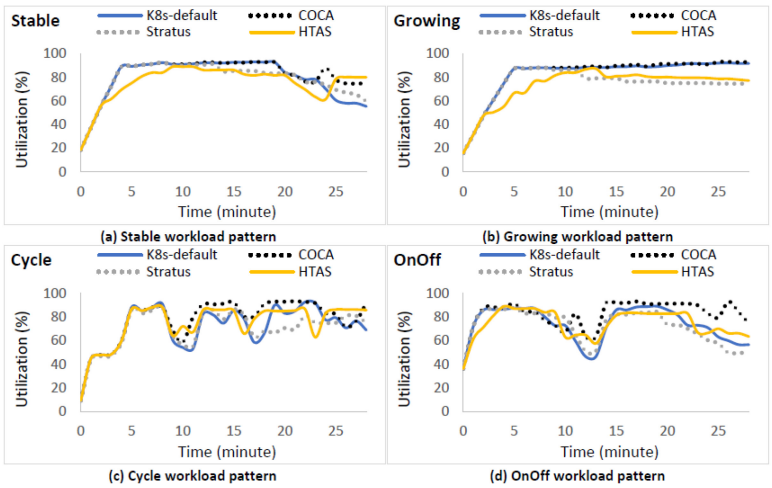

Fig. 4. Resource utilization.

a job is submitted to the time point when it is deployed in running status. The total cost in Figure 3 is estimated based on the accumulation of the resource usage by each VM instance when the same combination of batch jobs and long-running services are successfully deployed and completed.

Stable workload pattern. In this scenario, the workload density stays at a steady level. HTAS outperforms the default K8s by cost reduction of 27% through tight task packing and cost-efficient autoscaling, as K8s follows a loose packing manner based on the LRP algorithm and rigid homogeneous autoscaling strategy. As observed in Figure 4(a) and Figure 5(a), all clusters experience resource overprovisioning after the first three rounds of autoscaling and their resource utilizations start to drop. HTAS, COCA, and Stratus employ different task rescheduling strategies to clean underutilized nodes and improve utilization, whereas Figure 6(a) and Figure 7(a) depict their

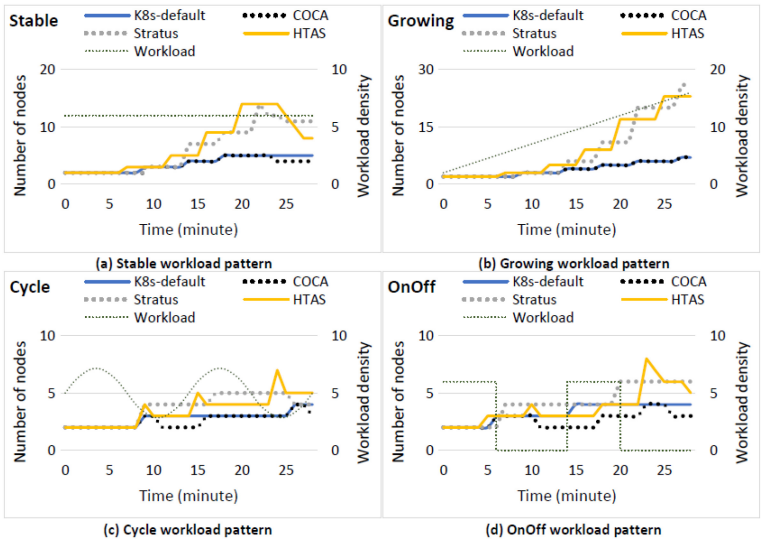

Fig. 5. Number of active worker nodes in response to the workload over time.

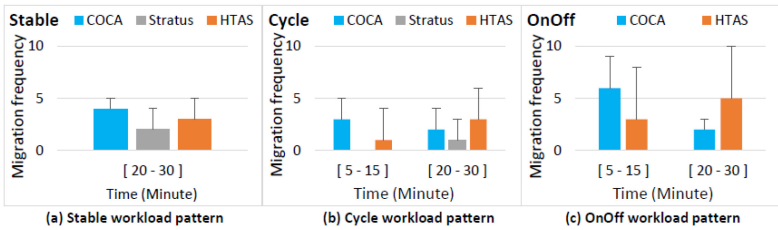

Fig. 6. Task migration frequency.

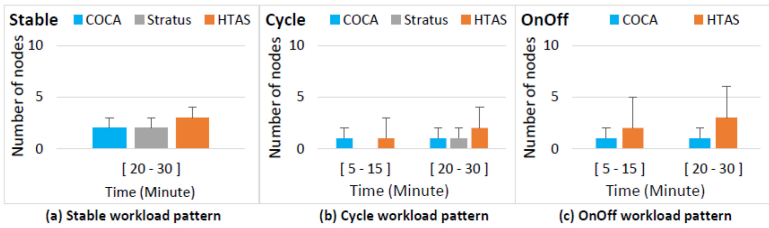

Fig. 7. Node deprovisioning caused by rescheduling.

task rescheduling and node deprovisioning rates. Compared to the others, HTAS enjoys the highest node deprovisioning rate after rescheduling and improves resource utilization more efficiently. Specialized for orchestration of batch jobs, Stratus's runtime bin LRP algorithm could co-locate both long-running services and batch jobs to the same instance. Since long-running services are not feasible for migration, this leads to a lower rescheduling rate. As COCA applies direct task eviction and relocation during rescheduling, it causes severe task progress loss. Therefore, HTAS outperforms Stratus and COCA by cost saving of 10% and 25%, respectively. As a tradeoff, HTAS has the highest task scheduling time.

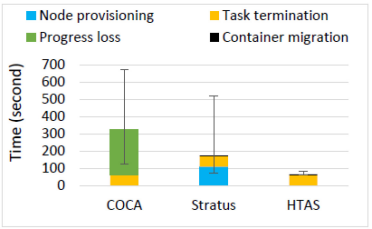

Fig. 8. Average time overhead of task migration.

Growing workload pattern. Here, the workload intensity keeps increasing over time. As shown in Figure 4(b) and Figure 5(b), both COCA and the default K8s continuously suffer from cluster overloading with their resource utilization ratios close to 100%, due to their limited autoscaling mechanisms (only scaling up a homogeneous instance per scaling cycle). Consequently, most jobs remain in pending status with poor QoS performance and long response time. However, this leads to higher costs, as the expenses of resource usage from all active long-running jobs keep accumulating simultaneously. By contrast, HTAS supports an on-demand autoscaling strategy with an elastic pricing model for cost-efficiency evaluation. Hence, HTAS outperforms COCA and the default K8s by cost reduction of 24% and 23%, respectively, and additionally, its average task scheduling time is also 47% and 46% shorter than them, respectively. As Stratus also supports dynamic autoscaling of heterogeneous instances to meet the rising workload in a cost-efficient manner, its average task scheduling time, resource utilization, and number of active instances are quite close to HTAS. Task rescheduling is not triggered under this scenario, as the workload density keeps rising and overall resource utilization remains high.

Cycle workload pattern. The cycle workload pattern experiences periodical changes and reaches a peak/bottom in each cycle. As shown in Figure 4(c) and Figure 5(c), resource utilization of the default K8s starts to fall after the workload reaches the peak in each cycle. By utilizing task rescheduling to deal with the periodical workload decrease and avoid resource wasting, HTAS reduces the cloud bill of the default K8s by 30%. Although COCA's rescheduling algorithm could improve resource utilization and prevent overprovisioning to a certain degree, the task progress loss caused by its task eviction approach remains a performance bottleneck in terms of QoS assurance and migration costs as observed in Figure 8. Hence, HTAS outperforms COCA by cost reduction of 15%. As presented in Figure 6(b) and Figure 7(b), Stratus enjoys the lowest rescheduling rate due to its task co-location strategy, which leads to poor overall utilization. Compared to Stratus, HTAS also reduces costs by 12%.

On-and-off workload pattern. Similar to the cycle pattern, the workload density periodically experiences a dramatic decrease after each active period. In light of the same reasons aforementioned, HTAS outperforms the default K8s, COCA, and Stratus by cost reduction by 32%, 22%, and 18%, respectively. Since Stratus manages the bursty workload during each active period through task co-location in a loose packing manner, no task rescheduling is observed in the Stratus cluster under this scenario.

Figure 8 shows the average time overhead of different task migration approaches. The migration duration in HTAS consists of container migration and task termination at the source node. Because of the small task sizes in our experiments, container migration usually finishes under 10 seconds, which is only a small portion of the overall cost. If large tasks and cross-region migrations are considered, more sophisticated migration strategy is needed. By contrast, Stratus includes node provisioning time as an extra cost. It may apply for a smaller/cheaper instance as a replacement

of an underutilized node where the running jobs cannot be moved elsewhere during rescheduling. However, this could easily lead to frequent scaling and overprovisioning if those jobs complete before any instance acquisition. Moreover, the underutilized node is tagged as unschedulable before it is terminated, which could also lead to resource wasting during the time gap of instance acquisition lag and container migrations. Considering that batch jobs are usually short lived [13], scaling extra instances for migration is rather impractical. As for COCA, it enjoys the highest time overhead of task migration. Since it directly evicts and redeploys a task during migration, task progress loss becomes an inevitable cost that depends on the nature of the underlying task. The longer the task runs before migration, the more the overall system performance will suffer from resource wasting and QoS degradation.

Overall, HTAS outperforms the other solutions by cost reduction through tight task packing, cost-efficient autoscaling, and resource-saving rescheduling. Except for the growing workload pattern, the average task scheduling time of HTAS is higher than that of the other solutions, which could be regarded as an acceptable tradeoff for cost-efficiency.

7 CONCLUSION AND FUTURE DIRECTIONS

In light of elastic compute resources, we propose HTAS in response to the dynamic business volume for four types of representative cloud workloads. Each application model is coupled with a specific priority method and node group to make task scheduling and scaling decisions in a cost-efficient manner through resource utilization optimization and elastic resource pricing. Furthermore, our rescheduling mechanism prevents resource wasting and overprovisioning while minimizing QoS degradation through container checkpointing. It is demonstrated through our evaluation results that our proposed approaches could reduce the overall cost by 23% to 32% for different types of cloud workload patterns when compared to the default Kubernetes framework.

Only the heterogeneous configuration of VM sizes is supported at this stage, due to the limitation of our test environment. VM types should also be considered for different task requirements. For instance, the performance of a task may fluctuate drastically on different instance types. A task with a strict time constraint would prefer a compute-optimized machine (high expense per time unit) instead of a standard machine, on account of the significant execution time reduction. If the task processing time could be estimated on different types of VMs, more pricing models could be provided to minimize the overall cost under the QoS requirement. Furthermore, QoS management is another improvement worth investigating. Within the scope of our work, we assume all jobs with the best-effort QoS requirement. However, jobs are usually associated with specific service-level agreements, such as application throughput requirement and execution time constraint. A more sophisticated approach is needed to support heterogeneous service-level-agreement-aware resource management.

In this work, we assume that container migrations across nodes through CRIU would not lose task progress or cause severe QoS degradation. Although it appears to be an ideal solution for an unbalanced cluster load, it is limited by various factors. For instance, during the initial stage of container checkpointing, container image creation would lead to a certain level of performance and QoS degradation. However, this could be unacceptable for applications with rigid requirements regarding performance and stability, such as a real-time data monitoring system under high workload density. Therefore, it is only feasible for services that could tolerate a short period of downtime without impacting their task progress. Another important factor is network latency, which decides whether the migration could be successfully finished within the given time constraint. However, the data volume and status of an application could affect the migration process. Image generation and data transmission across nodes could be compute expensive and time consuming for large applications with fast-changing data [31].

Autoscaling is an extremely time-sensitive process by satisfying the need of dynamically changing workload through instance acquisition [25]. As one of the key factors in autoscaling, the instance acquisition lag could impact the overall resource utilization more than the task execution time. If the instance acquisition lag is unexpectedly high, the autoscaling decisions might not be optimal, which could directly lead to underprovisioning or overprovisioning cases. Therefore, our next goal is to reduce and precisely estimate the instance acquisition lag. Another possible future direction would be cloud workload prediction. Considering instance acquisition lag and workload prediction, the autoscaling decision could be further optimized in response to the workload changes at the exact time point of instance acquisitions. Hence, various resource demand prediction models could be applied to improve the accuracy of dynamic task volume estimates (e.g., [32, 33]).

ACKNOWLEDGMENT

We thank Shashikant Ilager for his help in improving the quality of the article.

REFERENCES

[1] R. Mocevicius. 2015. *CoreOS Essentials*. Packt Publishing Ltd.
[2] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. 2015. Large scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*. 18.
[3] K. Hightower, B. Burns, and J. Beda. 2017. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media.
[4] M. A. Rodriguez and R. Buyya. 2019. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience* 49, 5 (2019), 698–719.
[5] H. D. Karatza. 2004. Scheduling in distributed systems. In *Performance Tools and Applications to Networked Systems*. Lecture Notes in Computer Science, Vol. 2965. Springer, 336–356.
[6] G. Copil, D. Moldovan, H. Truong, and S. Dustdar. 2016. rSYBL: A framework for specifying and controlling cloud services elasticity. *ACM Transactions on Internet Technology* 16, 3 (2016), 18.
[7] D. Bernstei. 2014. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.
[8] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui. 2016. Modelling performance and resource management in Kubernetes. In *Proceedings of the 9th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'16)*. 257–262.
[9] N. Naik. 2016. Building a virtual system of systems using Docker swarm in multiple clouds. In *Proceedings of the 2nd IEEE International Symposium on Systems Engineering (ISSE'16)*. 1–3.
[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. 295–308.
[11] GitHub. 2019. Marathon. Retrieved March 22, 2020 from https://mesosphere.github.io/marathon.
[12] R. DelValle, G. Rattihalli, A. Beltre, M. Govindaraju, and M. J. Lewis. 2016. Exploring the design space for optimizations with Apache Aurora and Mesos. In *Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD'16)*. 537–544.
[13] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. 2019. Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces. In *Proceedings of the ACM International Symposium on Quality of Service (IWQoS'19)*. 39.
[14] H. Zhang, H. Ma, G. Fu, X. Yang, Z. Jiang, and Y. Gao. 2016. Container based video surveillance cloud service with fine-grained resource provisioning. In *Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD'16)*. 758–765.
[15] C. Kaewkasi and K. Chuenmuneewong. 2017. Improvement of container scheduling for Docker using ant colony optimization. In *Proceedings of the 9th International Conference on Knowledge and Smart Technology (KST'17)*. 254–259.
[16] Q. Liu and Z. Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba Trace. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18)*. ACM, New York, NY, 347–360.
[17] C. Guerrero, I. Lera, and C. Juiz. 2018. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing* 16, 1 (2018), 113–135.

[18] S. Kehrer and W. Blochinger. 2018. TOSCA-based container orchestration on Mesos. *Computer Science—Research and Development* 33, 3–4 (2018), 305–316.
[19] M. Xu, A. Toosi, and R. Buyya. 2019. iBrownout: An integrated approach for managing energy and brownout in container-based clouds. *IEEE Transactions on Sustainable Computing* 4, 1 (2019), 53–66.
[20] S. Taherizadeh and V. Stankovski. 2018. Dynamic multi-level autoscaling rules for containerized applications. *Computer Journal* 62, 2 (2018), 174–197.
[21] A. Chung, J. W. Park, and G. R. Ganger. 2018. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*. 121–134.
[22] M. A. Rodriguez and R. Buyya. 2018. Containers orchestration with cost-efficient autoscaling in cloud computing environments. arXiv:1812.00300.
[23] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan. 2018. A holistic evaluation of Docker containers for interfering microservices. In *Proceedings of the 2018 IEEE International Conference on Services Computing*. 33–40.
[24] J. Son, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya. 2017. SLA-aware and energy-efficient dynamic overbooking in SDN-based cloud data centers. *IEEE Transactions on Sustainable Computing* 2, 2 (2017), 76–89.
[25] M. Mao and M. Humphrey. 2011. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*. 1–12.
[26] J. Kang and S. Park. 2003. Algorithms for the variable sized bin packing problem. *European Journal of Operational Research* 147, 2 (2003), 365–372.
[27] Nectar. Home Page. Retrieved March 22, 2020 from https://nectar.org.au/.
[28] Lakshman and P. Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[29] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti. 2016. Migrating LinuX containers using CRIU. In *High Performance Computing*. Lecture Notes in Computer Science, Vol. 9945. Springer, 674–684.
[30] Nedelcu, Clément. 2010. *Nginx HTTP Server: Adopt Nginx for Your Web Applications to Make the Most of Your Infrastructure and Serve Pages Faster Than Ever*. Packt Publishing Ltd.
[31] M. Chen, W. Li, G. Fortino, Y. Hao, L. Hu, and I. Humar. 2019. A dynamic service migration mechanism in edge cognitive computing. *ACM Transactions on Internet Technology* 19, 2 (2019) 30.
[32] Z. Gong, X. Gu, and J. Wilkes. 2010. PRESS: PRedictive elastic resource scaling for cloud systems. In *Proceedings of 2010 International Conference on Network and Service Management*. 9–16.
[33] Khan, X. Yan, S. Tao, and N. Anerousis. 2012. Workload characterization and prediction in the cloud: A multiple time series approach. In *Proceedings of the 2012 IEEE Network Operations and Management Symposium*. 1287–1294.
[34] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui. 2016. Adaptive application scheduling under interference in Kubernetes. In *Proceedings of the 9th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'16)*. 426–427.
[35] C. T. Joseph and K. Chandrasekaran. 2019. Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. *Software: Practice and Experience* 49, 10 (2019), 1448–1484.
[39] U. Paščinsk, J. Trnkoczy, V. Stankovski, M. Cigale, and S. Gec. 2018. QoS-aware orchestration of network intensive software utilities within software defined data centres. *Journal of Grid Computing* 16, 1 (2018), 85–112.
[37] P. Kochovski, P. D. Drobintsev, and V. Stankovski. 2019. Formal quality of service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method. *Information and Software Technology* 109, 2 (2019), 14–25.
[38] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*. 7.
[39] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. 2011. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 3.
[40] C. Pahl and B. Lee. 2015. Containers and clusters for edge cloud architectures—A technology review. In *Proceedings of the 3rd IEEE International Conference on Future Internet of Things and Cloud*. 379–386.
[41] B. Burns and D. Oppenheimer. 2016. Design patterns for container-based distributed systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*. 2016.
[42] J. Yu and R. Buyya. 2005. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD Record* 34, 3 (2005) 44–49.
[43] M. Xu and R. Buyya. 2019. BrownoutCon: A software system based on brownout and containers for energy-efficient cloud computing. *Journal of Systems and Software* 155, 5 (2019), 91–103.
[44] X. Xu, H. Yu, and X. Pei. 2014. A novel resource scheduling approach in container based clouds. In *Proceedings of the 17th IEEE International Conference on Computational Science and Engineering*. 257–264.
[45] L. Yin, J. Luo, and H. Luo. 2018. Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing. *IEEE Transactions on Industrial Informatics* 14, 10 (2018), 4712–4721.

[46] R. Buyya, R. N. Calheiros, J. Son, A. V. Dastjerdi, and Y. Yoon. 2014. Software-defined cloud computing: Architectural elements and open challenges. In *Proceedings of the 3rd IEEE International Conference on Advances in Computing, Communications, and informatics (ICACCI'14)*. 1–12.

[47] Z. Zhao, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, and C. de Laat. 2015. A software workbench for interactive, time critical and highly self-adaptive cloud applications (SWITCH). In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 1181–1184.

[48] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. 2014. Fuxi: A fault-tolerant resource management and job scheduling system at Internet scale. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1393–1404.

[49] L. Qi. 2019. Maximizing CPU Resource Utilization on Alibaba's Servers. Retrieved March 22, 2020 from https://102.alibaba.com/detail/?id=61.

[50] C. Delimitrou, D. Sanchez, and C. Kozyrakis. 2015. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing*. 97–110.

[51] S. Shastri and D. Irwin. 2017. HotSpot: Automated server hopping in cloud spot markets. In *Proceedings of the 8th ACM Symposium on Cloud Computing*. 493–505.

Received August 2019; revised November 2019; accepted January 2020