Load-Aware Container Orchestration on Kubernetes Clusters

Angelo Marchese[a] and Orazio Tomarchio[b]
Dept. of Electrical Electronic and Computer Engineering, University of Catania, Catania, Italy

Keywords: Microservices Applications, Container Technology, Kubernetes Scheduler, Kubernetes Descheduler, Node Monitoring, Application Monitoring, Cloud Continuum.

Abstract: Microservice Architecture is quickly becoming popular for building extensive applications designed for deployment in dispersed and resource-constrained cloud-to-edge computing settings. Being a cloud-native technology, the real strength of microservices lies in their loosely connected, autonomously deployable, and scalable features, facilitating distributed deployment and flexible integration across powerful cloud data centers to heterogeneous and often constrained edge nodes. Hence, there is a need to devise innovative placement algorithms that leverage these microservice features to enhance application performance. To address these issues, we propose extending Kubernetes with a load-aware orchestration strategy, enhancing its capability to deploy microservice applications within shared clusters characterized by dynamic resource usage patterns. Our approach dynamically orchestrates applications based on runtime resource usage, continuously adjusting their placement. The results, obtained by evaluating a prototype of our system in a testbed environment, show significant advantages over the vanilla Kubernetes scheduler.

1 INTRODUCTION

The orchestration of modern applications, including Internet of Things (IoT), data analytics, video streaming, process control, and augmented reality services, poses a complex challenge (Salaht et al., 2020; Oleghe, 2021; Luo et al., 2021). These applications impose stringent quality of service (QoS) requirements, particularly in terms of scalability, fault tolerance, availability, response time and throughput. To address these requirements the microservices architecture paradigm has become the predominant approach for designing and implementing such applications. This paradigm involves breaking down applications into multiple microservices that interact with each other to fulfill user requests.

Furthermore, Cloud Computing offers a reliable and scalable environment to execute these applications, while the recent adoption of Edge Computing allows executing workloads near the end user (Varghese et al., 2021; Kong et al., 2022). In this context, both Cloud and Edge infrastructure are combined together to form the Cloud-to-Edge continuum, a shared environment for executing distributed microservices-based applications. However the orchestration of such applications in these environments is a complex problem, considering the heterogeneity in the computational resources between Cloud and Edge nodes and that multiple microservices compete to use these resources (Khan et al., 2019; Kayal, 2020; Manaouil and Lebre, 2020; Goudarzi et al., 2022).

Kubernetes[1] is a widely adopted orchestration platform that supports the deployment, scheduling and management of containerized applications (Burns et al., 2016). Today different Kubernetes distributions are maintained by the major Cloud providers, while Edge-oriented distributions like KubeEdge[2] have recently been proposed. However, the default Kubernetes static orchestration policy presents some limitations when dealing with complex microservices-based applications that share the same node cluster. In particular, Kubernetes does not evaluate the runtime resource usage of microservices when scheduling them, thus leading to higher shared resource interference between microservices and then reduced application performance in terms of response time.

To deal with those limitations, in this work, starting from our previous works (Marchese and Tomarchio, 2022b; Marchese and Tomarchio, 2022a), we propose to extend the Kubernetes platform with a

[a] https://orcid.org/0000-0003-2114-3839
[b] https://orcid.org/0000-0003-4653-0480

[1] https://kubernetes.io
[2] https://kubeedge.io

Marchese, A. and Tomarchio, O.
Load-Aware Container Orchestration on Kubernetes Clusters.
DOI: 10.5220/0012738800003711
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)*, pages 92-102
ISBN: 978-989-758-701-6; ISSN: 2184-5042
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

load-aware orchestration strategy to make it suitable for the deployment of microservices applications with dynamic resource usage patterns on shared node clusters. Our approach enhances Kubernetes by implementing a dynamic application orchestration and scheduling strategy able 1) to consider the runtime application resource usage patterns when determining a placement for each application microservice and 2) to continuously tune the application placement based on the ever changing infrastructure and application states. While different aspects can be considered for optimization, like cost, application dependability and availability, the main goal of our approach in this work is to improve the application performance.

The rest of the paper is organized as follows. Section 2 provides some background information about the Kubernetes platform and discusses in more detail some of its limitations that motivate our work. In Section 3 the proposed approach is presented, providing some implementation details of its components, while Section 4 provides results of our prototype evaluation in a testbed environment. Section 5 examines some related works and, finally, Section 6 concludes the work.

2 BACKGROUND AND MOTIVATION

2.1 Kubernetes Scheduler

Kubernetes, a container orchestration platform, automates the lifecycle management of distributed applications deployed on large-scale node clusters (Gannon et al., 2017). A typical Kubernetes cluster comprises a control plane and a set of worker nodes. The control plane encompasses various management services running within one or more master nodes, while the worker nodes serve as the execution environment for containerized application workloads. In Kubernetes, the fundamental deployment units are *Pods*, each containing one or more containers and managed by *Deployments* resources. In a microservices-based application, each Deployment corresponds to a microservice, and the Pods managed by that Deployment represent individual instances of that microservice. Various properties of a Deployment resource are configurable by application architects, with Pod resource requests being one of them. These requests specify the computational resources to reserve for Pods managed by a Deployment when running on worker nodes[3]. For example, Listing 1 illustrates a Deployment with specified CPU and memory Pod resource requests.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        resources:
          requests:
            cpu: 1
            memory: 64Mi
```

Listing 1: Example of Kubernetes Deployment resource.

The *Kube-scheduler*[4] is a control plane component responsible for selecting a cluster node for each Pod, considering both Pod requirements and resource availability on cluster nodes. Every Pod scheduling attempt undergoes a multi-phase process, illustrated in Figure 1, where the sorting, filtering, and scoring phases encompass the primary execution logic. Each phase is implemented by one or more plugins, and these plugins can further implement additional phases. During the sorting phase, an ordering for the Pod scheduling queue is established. In the filtering phase, each plugin executes a filtering function for each cluster node to determine if that node satisfies specific constraints. The output of the filtering phase is a list of candidate nodes that are deemed suitable for running the Pod. In the scoring phase, each plugin executes a scoring function for each candidate node, assigning a score based on specific criteria. The final score for each node is determined by the weighted sum of the individual scores assigned by each scoring plugin. Subsequently, the Pod is assigned to the node with the highest final score. If there are multiple nodes with equal scores, one of them is randomly selected. It's important to note that the Kubernetes scheduler is designed to be extensible. Each scheduling phase serves as an extension point where one or

[3]https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

[4]https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler

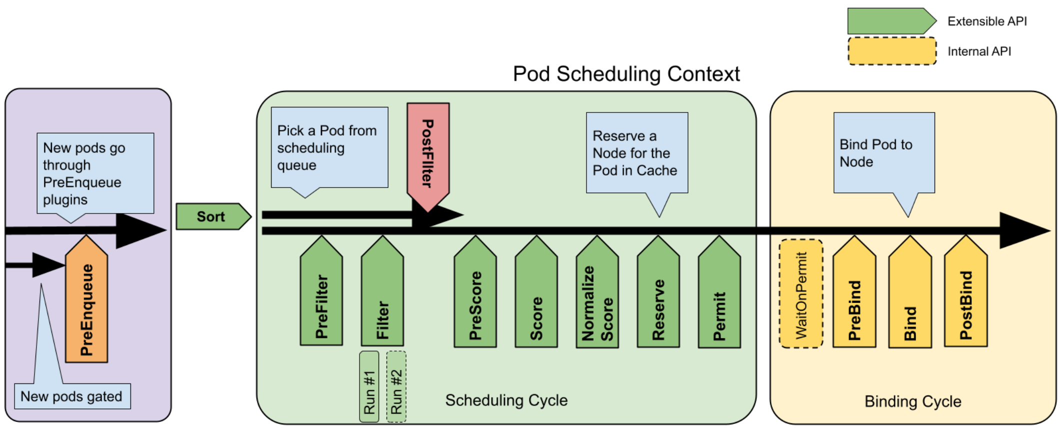

Figure 1: Kubernetes scheduling framework.

more custom plugins can be registered.

Among the default scheduler plugins, the *NodeResourcesFit* plugin handles both the filtering and scoring phases, ensuring that nodes lacking adequate computational resources to satisfy Pod resource requests are filtered out. Additionally, the *NodeResourcesBalancedAllocation* plugin is responsible for the scoring phase. The NodeResourcesFit plugin assigns scores to nodes based on mutually exclusive strategies:

- *LeastAllocated*: it favors nodes with the lowest ratio between the weighted sum of requested resources of Pods running on nodes (including the resource requests of the Pod to be scheduled) and the total amount of allocatable resources on those nodes. The objective is to achieve a balanced resource utilization among cluster nodes.
- *MostAllocated*: it favors nodes with the highest ratio between the weighted sum of requested resources of Pods running on nodes (including the resource requests of the Pod to be scheduled) and the total amount of allocatable resources on those nodes. This strategy aims to enhance resource utilization among cluster nodes while concurrently reducing the number of nodes required to efficiently run the workload.
- *RequestedToCapacityRatio*: It distributes Pods to ensure a specified ratio between the sum of requested resources of Pods running on nodes and the total amount of allocatable resources on those nodes.

The NodeResourcesBalancedAllocation plugin prioritizes nodes that would achieve a more balanced resource usage if the Pod is scheduled there. Despite the flexibility offered by the default NodeResourcesFit and NodeResourcesBalancedAllocation scheduler plugins to define strategies based on different goals, they necessitate knowledge about the resource requirements of each Pod to be scheduled and those already running in the cluster. This information typically comes from resource requests on Pods, which must be specified by application architects before the deployment phase. However, this task is intricate, given that microservices' resource requirements are dynamic parameters strongly dependent on the runtime load and distribution of user requests. Defining Pod resource requirements before the runtime phase can result in inefficient scheduling decisions, subsequently impacting application performance. Overestimating resource requirements for Pods may decrease the resource usage ratio on cluster nodes, leading to increased costs. Conversely, underestimating resource requirements may elevate Pod density on cluster nodes, intensifying shared resource interference and consequently increasing application response times.

2.2 Kubernetes Descheduler

Kubernetes scheduler placement decisions are influenced by the cluster state at the time a new Pod appears for scheduling. Given the dynamic nature of Kubernetes clusters and their evolving state, optimal placement decisions can be enhanced concerning the initial scheduling of Pods. Various reasons may prompt the migration of a Pod from one node to another, including node under-utilization or over-utilization, changes in Pod or node affinity requirements, and events such as node failure or addition.

To achieve this goal, a descheduler component has recently been proposed as a Kubernetes sub-project[5]. This component is responsible for evicting running Pods so that they can be rescheduled onto more suit-

[5]https://github.com/kubernetes-sigs/descheduler

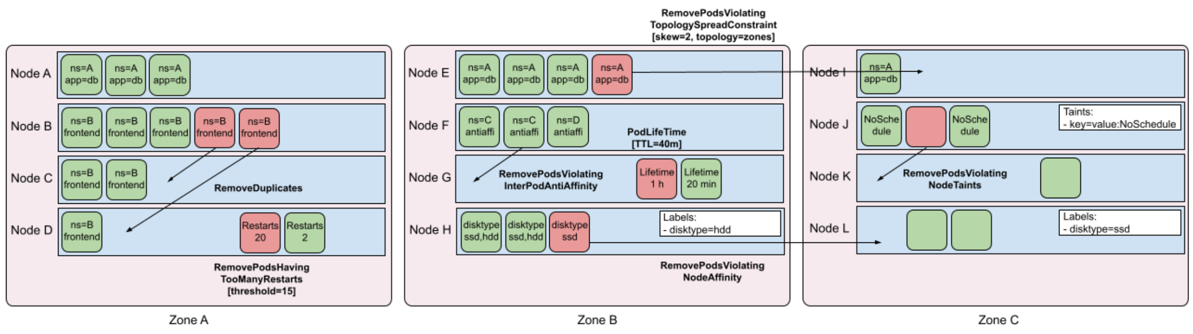

Figure 2: Kubernetes descheduling strategies.

able nodes. It's important to note that the descheduler itself does not schedule the replacement of evicted Pods; rather, it relies on the default scheduler for that task. The descheduler's policy is configurable and includes default eviction plugins that can be enabled or disabled. It features a common eviction configuration at the top level, along with configuration options from the Default Evictor plugin. Figure 2 displays the various default eviction plugins available in the Kubernetes descheduler at the time of writing. Similar to the Kubernetes scheduler, the descheduler is designed to be extensible, allowing integration of custom eviction plugins.

Among the default eviction plugins, the *LowNodeUtilization* and *HighNodeUtilization* evict Pods based on the resource utilization of cluster nodes. The LowNodeUtilization plugin identifies underutilized nodes and evicts Pods, with the expectation that recreating evicted pods will be scheduled on these underutilized nodes. Conversely, the HighNodeUtilization plugin targets nodes with high utilization, evicting Pods in the hope of scheduling them more compactly onto fewer nodes.

It's worth noting that, similar to the NodeResourcesFit and NodeResourcesBalancedAllocation scheduler plugins, these eviction plugins determine resource usage on each node as the ratio between the weighted sum of requested resources of Pods running on nodes and the total allocatable resources on those nodes. Consequently, the actual resource usage on nodes is not considered, potentially leading to inefficient rescheduling decisions.

3 PROPOSED APPROACH

3.1 Overall Design

Considering the limitations described in Section 2, this work proposes an extension to the Kubernetes platform. The aim is to incorporate a load-aware scheduling and descheduling strategy, rendering the platform suitable for orchestrating microservices-based applications with dynamic resource usage patterns. The primary concept underlying this approach is that the scheduling and descheduling processes for complex microservices-based applications should take into account both the dynamic state of the infrastructure and the real-time resource requirements of the microservices. Although various scheduling and descheduling strategies could be defined based on specific optimization goals, our focus in this work is to minimize runtime shared resource interference among microservices Pods. This, in turn, contributes to improving application response time. A notable distinction from the default Kubernetes scheduler and descheduler components lies in our proposed approach's utilization of runtime telemetry data for microservices resource usage profiling. Unlike statically defined Pod resource requests, this dynamic profiling lessens the burden on application architects to predict resource usage and communication relationships between microservices ahead of time when defining Pod resource requirements.

The overall architecture of the proposed approach, which is based on our previous work (Marchese and Tomarchio, 2023), is depicted in Figure 3. The continuous monitoring of runtime infrastructure and application microservices resource usage is facilitated through a *metrics server*. This server collects telemetry data, including CPU, memory, network, and disk bandwidth resources for both cluster nodes and Pods within the cluster. The *node monitor* operator, leveraging infrastructure telemetry data, annotates each cluster node with resource usage information, providing a runtime view of the cluster state. Similarly, the *application monitor* operator utilizes application telemetry data to annotate each application microservice Deployment with resource usage information, determining the runtime view of the application state. The *custom scheduler* employs these runtime cluster and application states to determine optimal placements for each application Pod. Meanwhile, the *custom descheduler* is responsible for taking Pod rescheduling actions if better scheduling decisions can be made. Further details regarding the components of the proposed approach are provided in the following subsections.

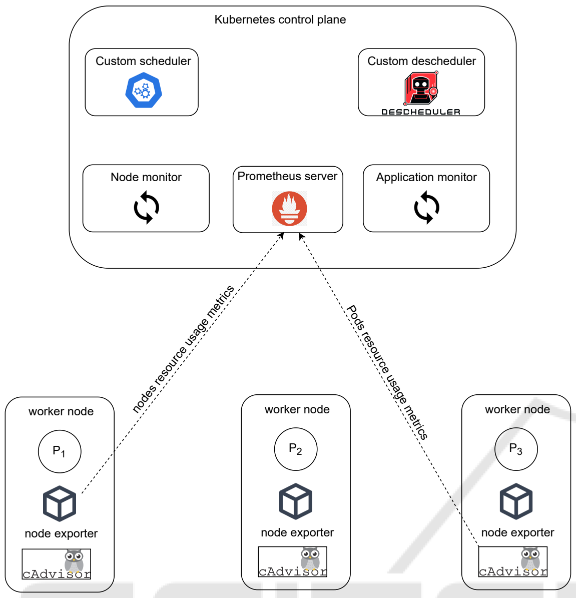

Figure 3: General architecture of the proposed approach.

3.2 Node Monitor

The node monitor component periodically determines the runtime total usage of CPU, memory, network, and disk bandwidth resources on each cluster node.

This component is a Kubernetes operator written in the Java language using the Quarkus Operator SDK[6] and runs as a Deployment in the Kubernetes control plane. Being a Kubernetes operator, it is triggered by a Kubernetes custom resource, specifically the *Cluster* custom resource, whose schema is shown in Listing 2.

```
apiVersion: v1alpha1
kind: Cluster
metadata:
  name: sample-cluster
spec:
  runPeriod: 30
  nodeSelector: {}
```

Listing 2: Example of a Cluster custom resource.

A Cluster resource includes a *spec* property with two sub-properties: *runPeriod* and *nodeSelector*. The *runPeriod* property determines the interval in seconds between two consecutive executions of the operator logic. Meanwhile, the *nodeSelector* property acts as a filter, selecting the list of nodes in the cluster that should be monitored by the operator.

The primary logic of the node monitor is executed by the ClusterReconciler class, specifically within its *reconcile()* method. In this method, the list of Node resources that satisfy the *nodeSelector* condition is retrieved from the Kubernetes API server. Subsequently, the *updateResourceUsage()* method of the ClusterReconciler class is invoked. This method determines, for each node n, the runtime total usage of CPU, memory, network, and disk bandwidth resources on that node, denoted as cpu_n, mem_n, net_n, and $disk_n$, respectively.

These values are determined from metrics fetched by the operator from a Prometheus[7] metrics server. The Prometheus server collects these metrics from node exporters[8], which run as DaemonSets on each cluster node. A DaemonSet is a Kubernetes resource that manages identical Pods each running on a different cluster node. The metrics, stored on the Prometheus server, exist as time series collections of measures. The runtime usage of each resource type is calculated as the average over a configurable period of the corresponding metric.

The parameters cpu_n, mem_n, net_n, and $disk_n$ are then assigned by the operator as the values for the annotations *cpu-usage*, *memory-usage*, *net-usage*, and *disk-usage* of node n, respectively.

3.3 Application Monitor

The application monitor component periodically determines the runtime usage of CPU, memory, network, and disk bandwidth resources for each application microservice running in the cluster. Similar to the node monitor, this component is a Kubernetes operator written in the Java language using the Quarkus Operator SDK. It runs as a Deployment in the Kubernetes control plane and is activated by a Kubernetes custom resource, specifically the *Application* custom resource, whose schema is shown in Listing 3.

An Application resource contains a *spec* property with three sub-properties: *runPeriod*, *namespace*, and *deploymentSelector*. The *runPeriod* property determines the interval between two consecutive executions of the operator logic. The *namespace* and *deploymentSelector* properties are used to select the set of Deployment resources that compose a specific microservices-based application. The Deployments selected by the Application resource are those created

[6]https://github.com/quarkiverse/quarkus-operator-sdk

[7]https://prometheus.io/

[8]https://github.com/prometheus/node_exporter

```
apiVersion: v1alpha1
kind: Application
metadata:
  name: sample-application
spec:
  runPeriod: 30
  namespace: default
  deploymentSelector: {}
```

Listing 3: Example of an Application custom resource.

in the namespace specified by the *namespace* property and with labels that satisfy the *deploymentSelector* property condition.

The primary logic of the application monitor is executed by the *ApplicationReconciler* class, specifically by its *reconcile()* method. In this method, the list of *Deployment* resources selected by the *namespace* and *deploymentSelector* properties is fetched from the Kubernetes API server. Then, the *updateResourceUsage()* method of the *ApplicationReconciler* class is invoked to determine, for each Deployment d, its runtime usage of CPU, memory, network, and disk bandwidth resources denoted as cpu_d, mem_d, net_d, and $disk_d$, respectively. These values, representing the average CPU, memory, network, and disk bandwidth consumption of all the Pods managed by the Deployment d, are fetched by the operator from the Prometheus metrics server. The metrics server, in turn, collects them from *CAdvisor*[9] agents. These agents run on each cluster node, monitoring the runtime usage of each resource type for the Pods executed on that node.

Similar to the node monitor, the application monitor determines the runtime usage of each resource type as the average over a configurable period of the corresponding metric. The parameters cpu_d, mem_d, net_d, and $disk_d$ are then assigned by the operator as the values for the annotations *cpu-usage*, *memory-usage*, *net-usage*, and *disk-usage* of Deployment d, respectively.

3.4 Custom Scheduler

The custom scheduler operates as a Deployment in the Kubernetes control plane and enhances the default Kubernetes scheduler by implementing a custom *LoadAware* scoring plugin written in the Go language. This custom plugin is based on the Kubernetes scheduler framework.

For each Pod scheduled, the plugin assigns a score to each candidate node in the cluster that has passed the filtering phase. This is achieved by executing a scoring function. The scores computed by the custom plugin are then aggregated with the scores from other scoring plugins within the default Kubernetes scheduler.

During the scoring phase, the *Score()* function of the LoadAware scheduler plugin is invoked to assign a score to each cluster node n when scheduling a Pod p. This function assesses the following parameters:

- cpu_p: the runtime CPU usage of Pod p, as specified by the value of the *cpu-usage* annotation of the Deployment owning the Pod.
- mem_p: the runtime memory usage of Pod p, as specified by the value of the *memory-usage* annotation of the Deployment owning the Pod.
- net_p: the runtime network bandwidth usage of Pod p, as specified by the value of the *network-usage* annotation of the Deployment owning the Pod.
- $disk_p$: the runtime disk bandwidth usage of Pod p, as specified by the value of the *disk-usage* annotation of the Deployment owning the Pod.
- cpu_n: the runtime total CPU usage on node n.
- mem_n: the runtime total memory usage on node n.
- net_n: the runtime total network bandwidth usage on node n.
- $disk_n$: the runtime total disk bandwidth usage on node n.
- $totcpu_n$: the total amount of allocatable CPUs on node n.
- $totmem_n$: the total amount of allocatable memory on node n.
- $totnet_n$: the total amount of allocatable network bandwidth on node n.
- $totdisk_n$: the total amount of allocatable disk bandwidth on node n.

The score of node n for the Pod p is determined by Equation (1):

$$score(p,n_i) = (1 - std(cpur_n, memr_n, netr_n, diskr_n)) \times MaxNodeScore \tag{1}$$

where:

$$MaxNodeScore = 100 \tag{2}$$

$$cpur_n = \frac{cpu_p + cpu_n}{totcpu_n} \tag{3}$$

$$memr_n = \frac{mem_p + mem_n}{totmem_n} \tag{4}$$

$$netr_n = \frac{net_p + net_n}{totnet_n} \tag{5}$$

[9]https://github.com/google/cadvisor

$$diskr_n = \frac{disk_p + disk_n}{totdisk_n} \qquad (6)$$

The resulting score is higher on nodes where the ratio between the runtime usage and the total allocatable amount of different resource types is more balanced. The runtime usage of a resource type on a node is determined as the sum of the usage of that resource of the Pod p and the total usage of Pods running on that node. The lower the standard deviation between each resource type usage ratio on a node, the more heterogeneous the Pods running on that node are, and consequently, the higher the node score. The fundamental rationale behind the proposed scheduling strategy is to reduce shared resource interference among Pods to minimize QoS violations on application response time. To achieve this, Pods competing for the same resource type should be placed on different nodes whenever possible. Unlike the default Kubernetes scheduler NodeResourcesFit and NodeResourcesBalancedAllocation scheduler plugins, which consider microservices' resource requirements statically determined by application architects ahead of time to assign node scores, the proposed LoadAware scoring plugin utilizes runtime resource usage telemetry data. This approach allows tuning scheduling decisions based on dynamically changing Pod resource usage patterns.

3.5 Custom Descheduler

The custom descheduler operates as a CronJob in the Kubernetes control plane and enhances the default Kubernetes descheduler by implementing a custom *LoadAware* evictor plugin. This custom plugin is written in the Go language and is based on the Kubernetes descheduler framework.

The descheduler CronJob is configured to periodically run Jobs that execute the descheduling logic, comprising both the default evictor plugin and the proposed LoadAware evictor plugin. The custom descheduler is set to evict at most one Pod for each iteration.

During each execution of the descheduler CronJob, the *Deschedule()* function of the LoadAware plugin is invoked. This function takes as input the list of nodes in the cluster and the Pods running on them, determining the Pod to be evicted. For each node, a score is assigned as the standard deviation of the runtime total usage of CPU, memory, network, and disk bandwidth resources. These values are determined based on the annotation values of the corresponding Node resource.

Starting from the node with the highest score, Equation 1 is used to assign node scores for Pods running on that node. For each Pod, if there is at least one node with a higher score than the node where the Pod is currently executed, the Pod becomes a candidate for eviction. The Pod that is evicted, if any, is the one with the highest difference between the score of the node where it is currently executed and another node in the cluster. These operations are repeated for the other nodes until a Pod to be evicted is found.

The configuration of the Default evictor plugin ensures that nodes where the Pod cannot be executed due to scheduling constraints are not considered as candidate nodes for migrating the Pod. This prevents the Pod from being unable to be rescheduled after eviction. Similar to the default Kubernetes descheduler, the proposed custom descheduler does not schedule replacements for evicted Pods; instead, it relies on the custom scheduler for that task.

The purpose of the proposed custom descheduler is to provide running Pods the opportunity to be rescheduled based on their runtime resource usage patterns. This approach aims to optimize application placement at runtime. By evicting currently running Pods and subsequently forcing them to be rescheduled, the balance between the usage of different resource types on each node can be maintained. This helps reduce the impact of shared resource interference among Pods on the application response time.

One limitation of the proposed approach is that Pod eviction can cause performance degradation in the overall application. However, it should be considered that cloud-native microservices are typically replicated, so the temporary shutdown of one instance generally causes only a graceful degradation of the application quality of service. Furthermore considering that the descheduler is configured to evict at most one Pod for each iteration, no downtime for microservices is caused by Pod eviction if more than one replica is executed for each of them.

4 EVALUATION

The proposed solution has been evaluated by using a sample microservices-based application generated using the *µBench* benchmarking tool (Detti et al., 2023). µBench enables the generation of service-mesh topologies with multiple microservices, each running a specific function. Among the prebuilt functions in µBench, the *Loader* function models a generic workload that stresses node resources when processing HTTP requests. When invoked, the Loader function computes an N number of decimals of π. The larger the interval, the greater the complexity and stress on the CPU. Additional stress on node

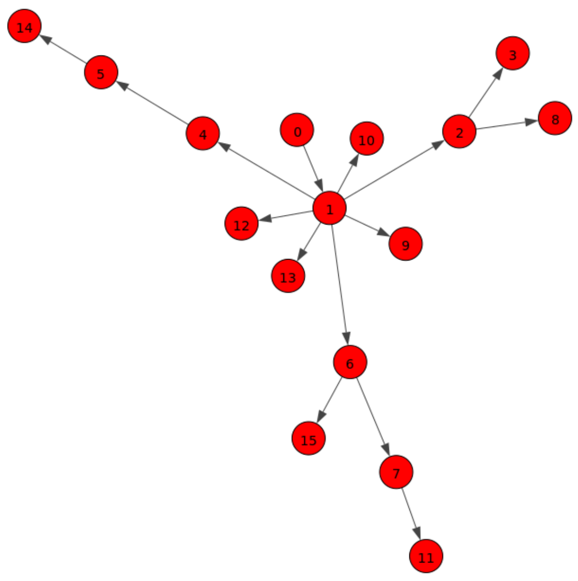

Figure 4: Sample application topology.

memory and disk bandwidth can be configured by adjusting the amount of memory and the number of disk read and write operations required by the function for each computation. Finally, function network bandwidth usage can be configured by adjusting the number of bytes returned by the function in response to each request.

Figure 4 depicts the sample application topology created using the µBench tool. The application comprises sixteen microservices, each running two replicas. Microservice m_0 serves as the entry point for external user requests, which are then handled by backend microservices. These backend microservices interact with each other through the exchange of HTTP requests. The sixteen microservices are grouped into groups of four items, where microservices within the same group run the Loader function with the same configuration parameters. Four versions of the Loader function are defined, f_0, f_1, f_2 and f_3, each configured to stress a different resource type, including CPU, memory, network, and disk bandwidth.

The test bed environment for the experiments consists of a Rancher Kubernetes Engine 2 (RKE2)[10] Kubernetes cluster with one master node and five worker nodes. These nodes are deployed as virtual machines on a Proxmox[11] environment and configured with 2 vCPU, 8GB of RAM, a Gigabit virtual network adapter and a virtual disk with read and write bandwidth of around 500 MB/s.

Black box experiments are conducted by evaluating the end-to-end response time of the sample application when HTTP requests are sent to the microservice m_0 with a specified number of virtual users each sending one request every second in parallel. Requests to the application are sent through the k6 load testing utility[12] from a node inside the same network where cluster nodes are located. This setup minimizes the impact of network latency on the application response time. Each experiment consists of 10 trials, during which the k6 tool sends requests to the microservice m_0 for 30 minutes. For each trial, statistics about the end-to-end application response time are measured and averaged with those of the other trials of the same experiment. For each experiment, we compare both cases when our node and application monitor operators and custom scheduler and descheduler components are deployed on the cluster and when only the Kubernetes scheduler is present with the default configuration. We consider three different scenarios based on different configurations for the four loader functions reported in Table 1.

Table 1: Loader functions configurations for the three scenarios.

	Scenario 1	Scenario 2	Scenario 3
f_0 (number of decimals of π generated)	1000	1500	2000
f_1 (memory usage)	100MB	200MB	400MB
f_2 (network bytes returned per request)	10KB	20KB	40KB
f_3 (bytes written on disk per request)	10MB	20MB	40MB

Figure 5 illustrates the results of three experiments, each representing a different scenario. The graph depicts the 95th percentile of the application response time in relation to the number of virtual users concurrently sending requests to the application. In all scenarios, the proposed approach consistently outperforms the default Kubernetes scheduler, showcasing average improvements of 23%, 31%, and 37%, respectively. For a low number of virtual users, the proposed approach exhibits similar performance to the default scheduler due to limited shared resource interference between Pods placed on the same nodes by the default scheduler. However, as the number of virtual users increases, the proposed approach surpasses the default scheduler, with more substantial improvements observed at higher virtual user counts. The response time with the default scheduler grows faster compared to the proposed approach. Furthermore, the disparity in response time becomes higher between the three scenarios as the resource usage of each function increases.

[10]https://docs.rke2.io
[11]https://www.proxmox.com
[12]https://k6.io

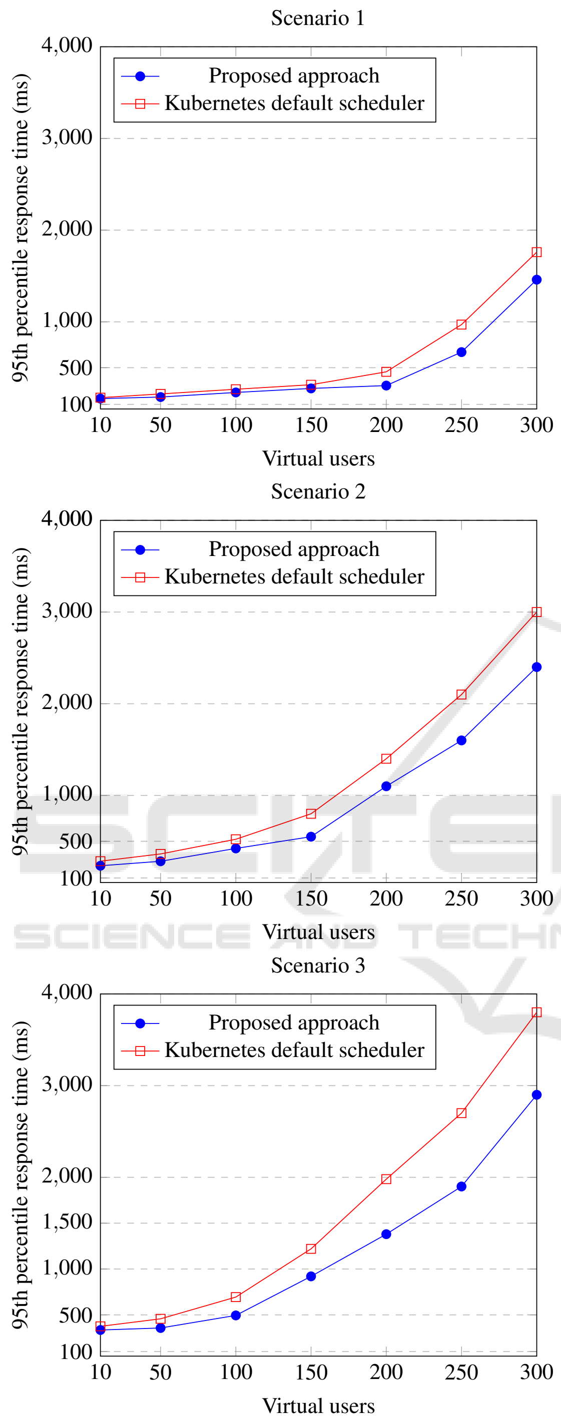

Figure 5: Experiments results.

5 RELATED WORK

In the literature, various works propose extending the Kubernetes platform to address the limitations of its static orchestration strategy when applied to microservices-based applications sharing the same node cluster (Senjab et al., 2023).

A novel approach for scheduling the workloads in a Kubernetes cluster, which are sometimes unequally distributed across the environment or deal with fluctuations in terms of resources utilization, is presented in (Ungureanu et al., 2019). The proposed approach looks at a hybrid shared-state scheduling framework model that delegates most of the tasks to the distributed scheduling agents and has a scheduling correction function that mainly processes the unscheduled and unprioritized tasks. The scheduling decisions are made based on the entire cluster state which is synchronized and periodically updated by a master-state agent.

In (Fu et al., 2021) Nautilus is presented, a runtime system that includes, among its modules, a contention-aware resource manager and a load-aware microservice scheduler. On each node, the resource manager determines the optimal resource allocation for its microservices based on reinforcement learning that may capture the complex contention behaviors. The microservice scheduler monitors the QoS of the entire service and migrates microservices from busy nodes to idle ones at runtime.

Boreas (Lebesbye et al., 2021) is a Kubernetes scheduler which is designed to evaluate bursts of deployment requests concurrently. Boreas finds the optimal placements for service containers with their deployment constraints by utilising a configuration optimiser.

In (Jian et al., 2023) DRS is proposed, a deep reinforcement learning enhanced Kubernetes scheduler, to mitigate the resource fragmentation and low utilization issues caused by the inefficient policies of the default Kubernetes scheduler. The Kubernetes scheduling problem is modeled as a Markov decision process with designed state, action, and reward structures to increase resource usage and decrease load imbalance. Then, a DRS monitor is designed to perceive parameters concerning resource utilization and create a thorough picture of all available resources globally. Finally, DRS is configured to automatically earn the scheduling policy through interaction with the Kubernetes cluster, without relying on expert knowledge about workload and cluster status.

In (Kim et al., 2024) a dynamic resource management and provisioning scheme for Kubernetes infrastructure is presented, which is capable of dynamically adjusting the resource allocation of Pods while overcoming the weakness of the existing resource restriction problem.

Finally, in our previous works (Marchese and Tomarchio, 2022b) and (Marchese and Tomarchio, 2022a) a network-aware Kubernetes scheduler is proposed, aimed to reduce the network distance among the microservices with a high degree of communication to improve the application response time. The load-aware scheduler plugin proposed in this work is complementary to the network-aware one and both can be used together on the same scheduler.

6 CONCLUSIONS

In this work we proposed to extend the Kubernetes platform with a real load-aware orchestration strategy aimed at reducing the shared resource interference among distributed microservices-based applications running on the same clusters in order to minimize QoS violations on their response times. The main goal is to overcome the limitations of the Kubernetes static scheduling and descheduling policies that require ahead of time knowledge of computational resource requirements of each microservice to make optimal container placement and rescheduling decisions. Considering the dynamic nature of distributed microservices applications, the idea is to extend the Kubernetes scheduler and descheduler components with custom plugins that make use of runtime microservices resource usage telemetry data to make their decisions. In this way, the effort for static application resource usage profiling can be reduced, while at the same time guaranteeing the expected application performances.

As a future work, we plan to improve the proposed custom scheduling and descheduling strategies by using time series analysis techniques in order to design more sophisticated algorithms that take into account long-term telemetry data to improve application resource usage predictions.

ACKNOWLEDGEMENTS

This work was partially funded by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, Mission 4 Component C2 Investment 1.1 - Call for tender No. 1409 of 14/09/2022 of Italian Ministry of University and Research - Project ”Cloud Continuum aimed at On-Demand Services in Smart Sustainable Environments” - CUP E53D23016420001.

REFERENCES

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93.

Detti, A., Funari, L., and Petrucci, L. (2023). µbench: An open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980.

Fu, K., Zhang, W., Chen, Q., Zeng, D., Peng, X., Zheng, W., and Guo, M. (2021). Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 932–941.

Gannon, D., Barga, R., and Sundaresan, N. (2017). Cloud-native applications. *IEEE Cloud Computing*, 4:16–21.

Goudarzi, M., Palaniswami, M., and Buyya, R. (2022). Scheduling iot applications in edge and fog computing environments: A taxonomy and future directions. *ACM Comput. Surv.*, 55(7).

Jian, Z., Xie, X., Fang, Y., Jiang, Y., Lu, Y., Dash, A., Li, T., and Wang, G. (2023). Drs: A deep reinforcement learning enhanced kubernetes scheduler for microservice-based system. *Software: Practice and Experience*, n/a(n/a).

Kayal, P. (2020). Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope : Invited paper. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, pages 1–6.

Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I., and Ahmed, A. (2019). Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235.

Kim, J., No, J., and Park, S.-s. (2024). Effective resource provisioning scheme for kubernetes infrastructure. In Nagar, A. K., Jat, D. S., Mishra, D., and Joshi, A., editors, *Intelligent Sustainable Systems*, pages 75–85, Singapore. Springer Nature Singapore.

Kong, X., Wu, Y., Wang, H., and Xia, F. (2022). Edge computing for internet of everything: A survey. *IEEE Internet of Things Journal*, 9(23):23472–23485.

Lebesbye, T., Mauro, J., Turin, G., and Yu, I. C. (2021). Boreas – A Service Scheduler for Optimal Kubernetes Deployment. In Hacid, H., Kao, O., Mecella, M., Moha, N., and Paik, H.-y., editors, *Service-Oriented Computing*, pages 221–237, Cham. Springer International Publishing.

Luo, Q., Hu, S., Li, C., Li, G., and Shi, W. (2021). Resource scheduling in edge computing: A survey. *CoRR*, abs/2108.08059.

Manaouil, K. and Lebre, A. (2020). Kubernetes and the Edge? Research Report RR-9370, Inria Rennes - Bretagne Atlantique.

Marchese, A. and Tomarchio, O. (2022a). Extending the kubernetes platform with network-aware scheduling capabilities. In *Service-Oriented Computing: 20th International Conference, ICSOC 2022, Seville, Spain, November 29 – December 2, 2022, Proceedings*, page 465–480, Berlin, Heidelberg. Springer-Verlag.

Marchese, A. and Tomarchio, O. (2022b). Network-aware container placement in cloud-edge kubernetes clusters. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 859–865, Taormina, Italy.

Marchese, A. and Tomarchio, O. (2023). Sophos: A Framework for Application Orchestration in the Cloud-to-Edge Continuum. In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 261–268. SciTePress.

Oleghe, O. (2021). Container placement and migration in edge computing: Concept and scheduling models. *IEEE Access*, 9:68028–68043.

Salaht, F. A., Desprez, F., and Lebre, A. (2020). An overview of service placement problem in fog and edge computing. *ACM Comput. Surv.*, 53(3).

Senjab, K., Abbas, S., Ahmed, N., and Khan, A. u. R. (2023). A survey of kubernetes scheduling algorithms. *Journal of Cloud Computing*, 12(1):87.

Ungureanu, O.-M., Vlădeanu, C., and Kooij, R. (2019). Kubernetes cluster optimization using hybrid shared-state scheduling framework. In *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, ICFNDS '19, New York, NY, USA. Association for Computing Machinery.

Varghese, B., de Lara, E., Ding, A., Hong, C., Bonomi, F., Dustdar, S., Harvey, P., Hewkin, P., Shi, W., Thiele, M., and Willis, P. (2021). Revisiting the arguments for edge computing research. *IEEE Internet Computing*, 25(05):36–42.