Orchestrating DAG Workflows on the Cloud-to-Edge Continuum

Angelo Marchese[1,†], Orazio Tomarchio[1,*,†]

[1]Dept. of Electrical Electronic and Computer Engineering, University of Catania, Catania, Italy

Abstract

Orchestrating data streaming and analytics applications presents challenges due to the increasing data volume and time-sensitive requirements. The combination of cloud and edge computing paradigms attempts to avoid their pitfalls while taking the best of both worlds: cloud scalability and compute closer to the edge where data is typically generated. However, placing microservices in such heterogeneous environments while meeting QoS constraints is a challenging task due to the geo-distribution of nodes and varying computational resources. In this paper we propose to extend Kubernetes to enable dynamic DAG workflow orchestration, taking into account both infrastructure and application states. Our approach aims to reduce QoS violations and improve application response time in Cloud-to-Edge continuum scenarios.

Keywords

DAG workflows, Containers technology, Orchestration, Kubernetes, Kubernetes scheduler

1. Introduction

The orchestration of modern data streaming and analytics applications is a complex problem to deal with, considering the increasing amount of data that needs to be processed and the requirement for deadline-constrained response times [1, 2]. These applications are typically implemented as DAG (Directed Acyclic Graph) workflows, where data collected from geographically distributed sources, like users or sensors, is moved between different processing microservices. Today, Cloud Computing offers a reliable and scalable environment to execute these applications. However, Cloud data centers are far away from the network edge and then from end users and devices. This can lead to high application response times and limited throughput. Edge Computing paradigm has emerged as a promising technology for mitigating this problem by moving computation towards the network edge [3, 4]. However, Edge environments are characterized by resource-constrained nodes and this can also have a negative impact on the application response time. Then, to take advantage of the high computational Cloud resources and the reduced network distance between data sources and Edge nodes, both Cloud and Edge infrastructure are combined together to form the Cloud-to-Edge continuum, an environment for executing distributed DAG workflows on multiple nodes organized in clusters.

ITADATA2023: The 2nd Italian Conference on Big Data and Data Science, September 11–13, 2023, Naples, Italy

*Corresponding author.

†These authors contributed equally.

angelo.marchese@phd.unict.it (A. Marchese); orazio.tomarchio@unict.it (O. Tomarchio)

0000-0002-0877-7063 (A. Marchese); 0000-0003-4653-0480 (O. Tomarchio)

However, establishing where to place each microservice of a workflow is a complex problem, considering the geo-distribution of nodes and their heterogeneity in terms of computational resources [5, 6].

Kubernetes[1] is today the de-facto orchestration platform for the deployment, scheduling and management of containerized applications in Cloud environments. Kubernetes has been initially thought for the orchestration of general purpose web services, but today it is also used for data analytics and AI training workflows [7, 8]. However, Kubernetes has not been designed for the orchestration of complex DAG workflows in geo-distributed and heterogeneous environments such as the aforementioned Cloud-Edge infrastructures [9, 10]. In particular, the default Kubernetes scheduling and orchestration strategy presents some limitations because it does not consider the ever changing resource availability on cluster nodes, node-to-node network latencies and the current application state, in terms of current resource usage of each microservice and the communication relationships between microservices [11]. Considering these factors when scheduling DAG workflows is critical in order to reduce QoS violations on the application response time.

To deal with those limitations, in this work we propose to extend the Kubernetes platform to adapt its usage on environments distributed in the Cloud-to-Edge continuum. Our approach enhances Kubernetes by implementing a dynamic DAG workflow orchestration and scheduling strategy able to consider the current infrastructure state when determining a placement for each microservice and to continuously tune the microservices placement based on the ever changing infrastructure and application states.

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

Kubernetes is a container orchestration platform which automates the lifecycle management of distributed applications deployed on large-scale node clusters [12]. A Kubernetes cluster consists of a control plane and a set of worker nodes. The control plane is made up of different management services that run inside one or many master nodes. The worker nodes represent the execution environment of containerized application workloads. In Kubernetes, minimal deployment units consist of *Pods*, which in turn contain one or more containers. In a microservices-based application, each Pod corresponds to a single microservice instance.

Among control plane components, the kube-scheduler[2] is in charge of selecting an optimal cluster node for each Pod to run them on, taking into account Pod requirements and node resources availability. Each Pod scheduling attempt is split into two phases: the scheduling cycle and the binding cycle, which in turn are divided into different sub-phases. During the

[1] https://kubernetes.io

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

scheduling cycle a suitable node for the Pod to schedule is selected, while during the binding cycle the scheduling decision is applied to the cluster by reserving the necessary resources and deploying the Pod to the selected node. Each sub-phase of both cycles is implemented by one or more plugins, which in turn can implement one or more sub-phases. The Kubernetes scheduler is meant to be extensible. In particular, each scheduling phase represents an extension point which one or more custom plugins can be registered at.

Kubernetes scheduler placement decisions are influenced by the cluster state at that point of time when a new Pod appears for scheduling. As Kubernetes clusters are very dynamic and their state changes over time, better placement decisions may be taken with respect to the initial scheduling of Pods. Several reasons can motivate the migration of a Pod from one node to another one, like for example node under-utilization or over-utilization, Pod or node affinity requirements not satisfied anymore and node failure or addition.

To this aim a descheduler component has been recently proposed as a Kubernetes sub-project[3]. This component is in charge of evicting running Pods so that they can be rescheduled onto more suitable nodes. The descheduler does not schedule replacement of evicted Pods but relies on the default scheduler for that. The descheduler's policy is configurable and includes strategies that can be enabled or disabled.

While the default Kubernetes scheduler and descheduler implementations are suitable for the orchestration of DAG workflows on centralized Cloud data centers, characterized by high and uniform computational resources and low network latencies, they present some limitations when dealing with node clusters dislocated on the Cloud-to-Edge continuum. The Kubernetes scheduler does not place microservices based on their resource and communication requirements and the current infrastructure state in terms of node resource availability and node-to-node network latencies. This means that microservices with high computational resource requirements could be placed on resource-constrained nodes and the microservices that exchange traffic between them are not always placed on nearby nodes. In the same way the Kubernetes descheduler does not reschedule Pods based on the ever changing infrastructure and application states. This can lead to higher application response times and more frequent QoS violations.

3. Proposed Approach

3.1. Overall Design

Considering the limitations described in Section 2, in this work we propose to extend the default Kubernetes orchestration strategy in order to adapt its usage to dynamic Cloud-to-Edge continuum environments. Leveraging our previous work presented in [13, 14], the main idea of the proposed approach is that in this context the orchestration and scheduling of complex DAG workflows should consider the dynamic state of the infrastructure where the workflow is executed and also the run time workflow microservices resource and communication requirements. In particular, in our approach, current node resource availability, node-to-node network latencies, resource usage of microservices and communication intensity between microservices are continuously monitored and taken into account during workflow scheduling.

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

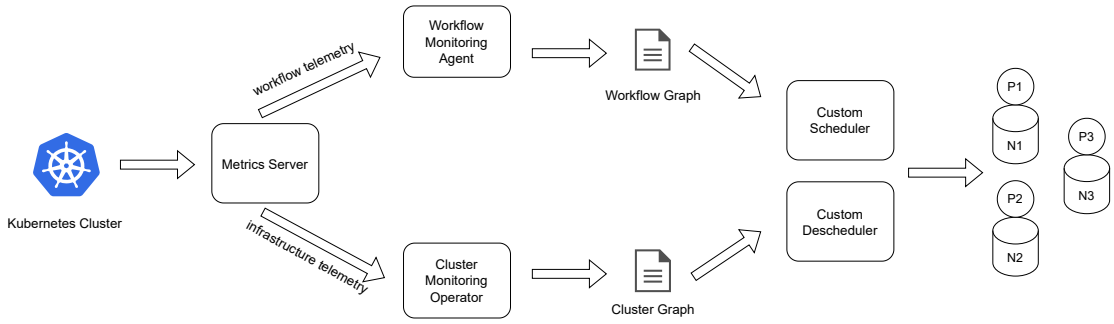

Figure 1: Overall architecture

Furthermore, a key point of our approach is the requirement to continuously tune the placement of the workflow based on the ever changing infrastructure state of Cloud-Edge environments, run time resource usage of microservices and their communication interactions.

Figure 1 shows a general model of the proposed approach. The current infrastructure and application states are monitored and all the telemetry data are collected by a *metrics server*. For the infrastructure, node resource availability and node-to-node latencies are monitored, while for the application, CPU and memory usage of microservices and the traffic amount exchanged between them are monitored. Based on the infrastructure telemetry data the *cluster monitoring* agent determines a *cluster graph* with the set of available resources on each cluster node and the network latencies between them. Similarly, the *workflow monitoring* agent uses microservices telemetry data to determine a *workflow graph* whose nodes represent microservices with their current resource usage and the edges the communication channels between them each with a specific weight that indicates the respective traffic amount sent through that channel. The cluster and workflow graphs are then used by the *custom scheduler* to determine a placement for each application Pod, and the *custom descheduler* to take Pod rescheduling actions if better scheduling decisions can be done. Further details on the components of the proposed approach are provided in the following subsections.

3.2. Workflow and Infrastructure Monitoring

The cluster monitoring agent runs as a controller in the Kubernetes control plane. It periodically determines the cluster graph with the currently available CPU and memory resources on each cluster node and the node-to-node network latencies.

During each execution the list of cluster *Node* resources is fetched from the Kubernetes API server. First, for each node n_i the CPU and memory values currently available on it, cpu_i and mem_i respectively, are determined. These values are fetched by the agent from a Prometheus[4] metrics server, which in turn collects them from node exporters executed on each cluster node. The cpu_i and mem_i parameters are then assigned as values for the *available-cpu* and *available-memory* annotations of the node n_i.

Then, for each pair of nodes n_i and n_j their network cost $nc_{i,j}$ is determined. The $nc_{i,j}$ parameter

[4]https://prometheus.io/

is proportional to the network latency between nodes n_i and n_j. Network latency metrics are fetched by the operator from the Prometheus metrics server, which in turn collects them from network probe agents executed on each cluster node as Pods managed by a DaemonSet. These agents are configured to periodically send ICMP traffic to all the other cluster nodes in order to measure the round trip time value. For each node n_i the operator assigns to it a set of annotations *network-cost-*n_j, with values equal to those of the corresponding $nc_{i,j}$ parameters. Finally, the cluster graph with the updated CPU and memory available resources and the network cost values for each node is then submitted to the Kubernetes API server.

The workflow monitoring agent runs also as a controller in the Kubernetes control plane and periodically determines the workflow graph with the current CPU and memory usage for each microservice and the traffic amounts exchanged between them.

During each execution the list of *Deployment* resources that constitute a DAG workflow are fetched from the Kubernetes API server. First, for each Deployment D_i its CPU and memory usage, cpu_i and mem_i respectively, are determined. These values are equal to the average CPU and memory consumption of all the Pods managed by the Deployment D_i and are fetched by the agent from the Prometheus metrics server, that in turn collects them from CAdvisor[5] agents. These agents are executed on each cluster node and monitor current CPU and memory usage for the Pods executed on that node. The cpu_i and mem_i parameters are then assigned as values for the *cpu-usage* and *memory-usage* annotations of the Deployment D_i.

Then for each Deployment D_i, the traffic amounts $traff_{i,j}$ with all the other Deployments D_j of the workflow are determined. The $traff_{i,j}$ parameter is proportional to the traffic amount exchanged between microservices μ_i and μ_j. Traffic metrics are fetched by the agent from the Prometheus metrics server, which in turn collects them from the Istio[6] platform. Istio is a service mesh implementation, whose control plane is installed in the Kubernetes cluster. The Istio control plane injects a sidecar container running an Envoy proxy on each Pod when they are created. All the traffic between Pods is intercepted by their corresponding Envoy proxies that in turn expose traffic statistics through metrics exporters that can be queried by the Prometheus server.

Each $traff_{i,j}$ parameter is assigned by the agent as the value for the annotation *traffic-*D_j of the Deployment D_i. The workflow graph with the set of CPU and memory usage and the traffic amounts for each Deployment is then submitted to the Kubernetes API server.

3.3. Custom Scheduler

The proposed custom scheduler extends the default Kubernetes scheduler by implementing two additional plugins, the *ResourceAware* and *NetworkAware* plugins that extend the node scoring phase of the default Kubernetes scheduler. For each Pod to be scheduled, each of the two plugins assigns a partial score to each candidate node of the cluster that has passed the filtering phase. The ResourceAware plugin takes into account the values of the *cpu-usage* and *memory-usage* annotations of the Deployment associated with the Pod to be scheduled and the values of the *available-cpu* and *available-memory* annotations of the node to be scored. The NetworkAware plugin takes into account the values of the *traffic* annotations of the Deployment associated

[5] https://github.com/google/cadvisor
[6] https://istio.io

with the Pod to be scheduled and the values of the *network-cost* annotations of the node to be scored. The node scores calculated by the ResourceAware and NetworkAware plugins are added to the scores of the other scoring plugins of the default Kubernetes scheduler.

Algorithm 1 Custom scheduler node scoring function

Input: p, cpu_{p_i}, mem_{p_i}, n, cpu_{n_i}, mem_{n_i} cns, nc, $traff$
Output: $score$

$rascore \leftarrow \alpha \times \frac{cpu_{n_i} - cpu_p}{cpu_p} \times 100 + \beta \times \frac{mem_{n_i} - mem_p}{mem_p} \times 100$
$cmc \leftarrow 0$
for cn in cns **do**
    $pcmc \leftarrow 0$
    for cnp in $cn.pods$ **do**
        $pcmc \leftarrow pcmc + nc_{n,cn} \times traff_{p,cnp}$
    end for
    $cmc \leftarrow cmc + pcmc$
end for
$nascore \leftarrow -cmc$
$score \leftarrow \gamma \times rascore + \delta \times nascore$

The algorithm takes as inputs the following arguments:

- p: the Pod to be scheduled.
- cpu_p: the CPU usage of Pod p.
- mem_p: the memory usage of Pod p.
- n: the node to be scored.
- cpu_n: the CPU available on node n.
- mem_n: the memory available on node n.
- cns: the set of nodes in the cluster, including node n.
- nc: the network costs between node n and all the other nodes cns.
- $traff$: the traffic amounts between the Pod p and all the other Pods in the workflow.

The algorithm starts by calculating the value of the variable *rascore*. This variable represents the partial contribution to the final node score given by the ResourceAware scheduler plugin. Its value is given by the weighted sum of the percentage differences between the CPU and memory currently available on node n and the respective usage values of Pod p. The α and β parameters are in the range between 0 and 1 and their sum is equal to 1. By changing the values of these parameters, a different contribution to the *rascore* variable value is given by the respective CPU and memory percentage differences. The higher the difference between available resources on node n and those used by Pod p, the greater the score assigned to node n. This allows to effectively balance the load between cluster nodes and then to reduce the shared resource interference between Pods resulting from incorrect node resource usage estimation and then its impact on application performances.

Then the partial contribution to the final node score given by the NetworkAware scheduler plugin is calculated. First the variable *cmc* is initialized to zero. This variable represents the total cost of communication between the Pod p and all the other Pods in the application when the Pod p is placed on node n. This variable represents the total cost of communication between the Pod p and all the other Pods of the application when the Pod p is placed on node n. The algorithm iterates through the list of cluster nodes *cns*. For each cluster node *cn* the *pcmc* variable value is calculated. This variable represents the cost of communication between the Pod p and all the other Pods *cn.pods* currently running on node *cn* when the Pod p is placed on node n. For each Pod *cnp* running on node *cn* the $traf_{p,cnp}$ parameter value is multiplied by the network cost $nc_{n,cn}$ between node n and node *cn* and added to the *pcmc* variable. The *pcmc* variable value is then added to the *cmc* variable. The final partial node score contribution of the NetworkAware scheduler plugin is assigned to the variable *nascore* as the opposite of the *cmc* variable value. The *nascore* variable value is assigned in such a way that the Pod p is placed on the node, or in a nearby node in terms of network latencies, where the Pods with which the Pod p exchanges the greatest amount of traffic are executed.

Then the final node score is calculated as the weighted sum between the *rascore* and *nascore* variables values, where the γ and δ parameters are in the range between 0 and 1 and their sum is equal to 1. By changing the values of these parameters, a different contribution to the *score* variable value is given by the *rascore* and *nascore* values.

3.4. Custom Descheduler

The custom descheduler runs as a controller in the Kubernetes control plane. The main business logic of the custom descheduler is implemented by a descheduling function that is called periodically for each application Pod to establish if that Pod should be rescheduled or not. Inside the descheduling function the same node scoring function implemented by the custom scheduler and showed in Algorithm 1 is invoked for each cluster node in order to assign them a score based on the current cluster and workflow graphs determined by the cluster and workflow monitoring agent respectively. If there is at least one node with a higher score than that of the node where the Pod is currently executed, the descheduler evicts the Pod. As in the case of the default Kubernetes descheduler, the proposed custom descheduler does not schedule a replacement of evicted Pods but relies on the custom scheduler for that. The use of the proposed custom descheduler is aimed at giving the running application Pods the possibility to be rescheduled on the basis of the current cluster network latencies and computational resources availability on each node and the traffic exchanged between microservices and their computational resources usage, thus allowing to optimize the application placement at run-time. By evicting currently running Pods and then forcing them to be rescheduled, application scheduling can take into account the ever changing cluster and application states with the latter mainly influenced by the user request load and patterns. One limitation of the proposed approach is that Pod eviction can cause downtime 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. To reduce the impact of Pod rescheduling, for each execution the descheduler evicts one Pod at most among the replicas of a single Deployment.

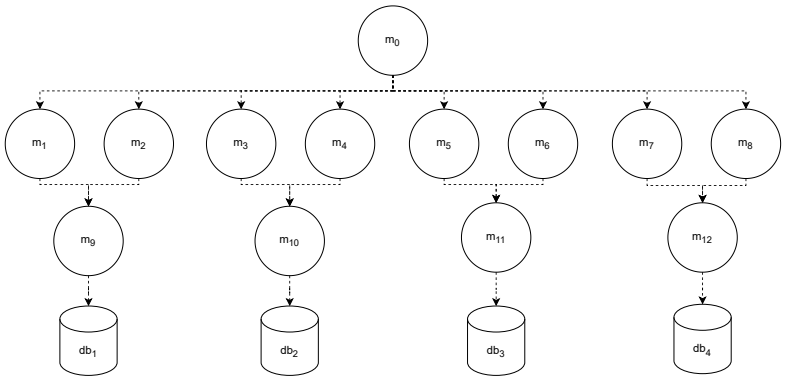

Figure 2: Sample DAG workflow application

4. Evaluation

The proposed solution has been validated using a sample DAG workflow application executed on a test bed environment. The application, whose structure is depicted in Figure 2, is composed of different microservices and database servers. Microservice m_0 represents the entry point for external traffic coming from end users and input data sources. This service represents the entry point for external user requests that are served by backend microservices that interact between them by means of network communication.

The test bed environment for the experiments consists of a Kubernetes cluster with one master node and five worker nodes. These nodes are deployed as virtual machines on a Proxmox[7] physical node and configured with 8GB of RAM and 2 vCPU. In order to simulate a realistic Cloud-to-Edge continuum environment with geo-distributed nodes, network latencies between cluster nodes are simulated by using the Linux traffic control (*tc*[8]) utility. By using this utility network latency delays are configured on virtual network cards of the cluster nodes.

We conduct black box experiments by evaluating the end-to-end response time of the workflow 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[9]. Each experiment consists of 10 trials, during which the k6 tool sends requests to the microservice m_0 for 40 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 cluster and workflow monitoring agents and custom scheduler and descheduler components are deployed on the cluster and when only the default Kubernetes scheduler is present. We consider three different scenarios based on the network latency between the cluster nodes: 10ms, 100ms and 200ms. In all the scenarios the α and β parameters of the ResourceAware plugin of the custom scheduler are assigned the same value of 0.5 in order to make the CPU and memory percentage differences between the respective resource availability on cluster nodes

[7] https://www.proxmox.com
[8] https://man7.org/linux/man-pages/man8/tc.8.html
[9] https://k6.io

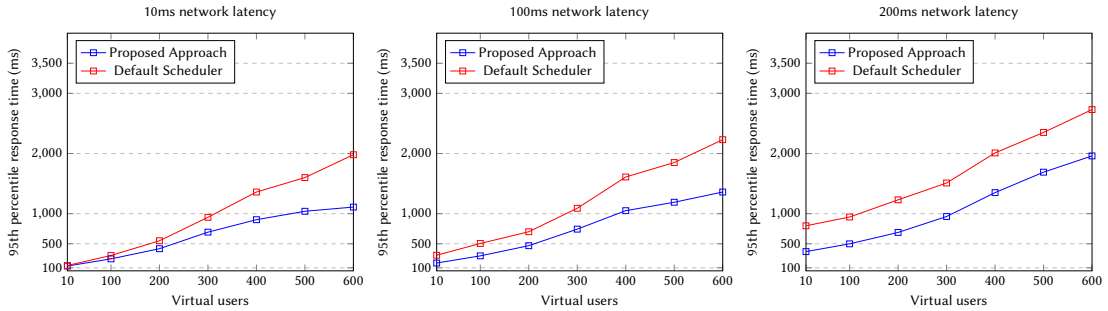

Figure 3: Experiments results

and the resource usage of Pods contribute equally to the *rascore* variable value. Similarly the γ and δ parameters are assigned the same value of 0.5 in order to make the *rascore* and *nascore* variables values, determined by the ResourceAware and NetworkAware plugins respectively, equally contribute to the final node score.

Figure 3 illustrate the results of the three experiments performed, each for a different scenario, showing the 95th percentile of the application response time as a function of the number of virtual users that send requests to the application in parallel. In all the cases, the proposed approach performs better than the default Kubernetes scheduler with average improvements of 39%, 56% and 66% in the three scenarios respectively. In the first scenario, network communication has no significant impact on the application response time because of the low node-to-node network latencies. Thus, the proposed network-aware scheduling strategy does not lead to high improvements in the application response time. Furthermore, for a low number of virtual users the proposed approach has similar performances to the default scheduler. This is because of the limited shared resource interference between Pods though they are placed on the same nodes by the default scheduler. However, when the number of virtual users increases, the proposed approach performs better than the default scheduler, with higher improvements for higher numbers of virtual users. The response time in the case of the default scheduler grows faster than in the case of the proposed approach. This is because of the proposed resource-aware scheduling strategy that distributes Pods on cluster nodes based on their run time resource usage, then reducing the shared resource interference between Pods. In the other scenarios, network communication becomes a bottleneck for the application response time and the lack of a network-aware scheduling strategy leads to high response times. In these scenarios our approach performs better than the default scheduler for low numbers of virtual users also, with higher improvements for higher node-to-node network latencies. One consequence of the combination of both a resource-aware and a network-aware scheduling strategy in our approach is that when the number of virtual users increases the response time grows much faster for high network latencies, though it remains lower than the response time in the case of the default scheduler. This can be explained by the fact that, for a higher number of virtual users and then for a higher request load, the average resource usage of microservices increases and then the distribution of Pods among cluster nodes caused by the resource-aware scheduling strategy is higher. This leads to an increase in the network latency between application microservices and

then in the end-to-end application response times.

5. Related Work

In the literature, there is a variety of works that propose to extend the Kubernetes platform in order to adapt its usage for the orchestration of microservices-based applications and in particular DAG workflows on the Cloud-to-Edge continuum [15, 16].

NetMARKS [17] is a Kubernetes scheduler extender that uses dynamic network metrics collected with Istio Service Mesh to ensure an efficient placement of Service Function Chains, based on the historical traffic amount exchanged between services. The proposed scheduler however does not consider run-time cluster network conditions in its placement decisions.

The authors of [18] propose to leverage application-level telemetry information during the lifetime of an application to create service communication graphs that represent the internal communication patterns of all components. The graph-based representations are then used to generate colocation policies of the application workload in such a way that the cross-server internal communication is minimized. However, in this work scheduling decisions are not influenced by the cluster network state.

In [19] a scheduling framework is proposed which enables edge sensitive and Service-Level Objectives (SLO) aware scheduling in the Cloud-Edge-IoT Continuum. The proposed scheduler extends the base Kubernetes scheduler and makes scheduling decisions based on a service graph, which models application components and their interactions, and a cluster topology graph, which maintains current cluster and infrastructure-speci□c states. However, this work does not consider historical information about the traffic exchanged between microservices in order to determine their run time communication affinity.

In [20] Nautilus is presented, a run-time system that includes, among its modules, a communication-aware microservice mapper. This module divides the microservice graph into multiple partitions based on the communication overhead between microservices and maps the partitions to the cluster nodes in order to make frequent data interaction complete in memory. While the proposed solution migrates application Pod if computational resources utilization is unbalanced among nodes, there is no Pod rescheduling in the case of degradation in the communication between microservices.

In [21] Pogonip, an edge-aware scheduler for Kubernetes, designed for asynchronous microservices is presented. Authors formulate the placement problem as an Integer Linear Programming optimization problem and define a heuristic to quickly find an approximate solution for real-world execution scenarios. The heuristic is implemented as a set of Kubernetes scheduler plugins. Also in this work, there is no Pod rescheduling if network conditions change over time.

In [22] an extension to the Kubernetes default scheduler is proposed that uses information about the status of the network, like bandwidth and round trip time, to optimize batch job scheduling decisions. The scheduler predicts whether an application can be executed within its deadline and rejects applications if their deadlines cannot be met. Although information about current network conditions and historical job execution times is used during scheduling decisions, communication interactions between microservices are not considered in this work.

6. Conclusions

In this work we proposed to extend the Kubernetes platform to adapt its usage for the orchestration of complex DAG workflows executed on the Cloud-to-Edge continuum. The main goal is to overcome the limitations of the Kubernetes static scheduling policies when dealing with the placement of DAG workflows on highly distributed environments. The idea is to make the Kubernetes scheduler aware of the run time communication intensity between the workflow microservices and their resource usage, and the cluster network conditions to make scheduling decisions that aim to reduce the overall workflow response time. Furthermore, a descheduler is proposed to dynamically reschedule microservices if better scheduling decisions can be made based on the ever changing application and cluster network states. As a future work we plan to improve the proposed scheduling and descheduling strategies, by using AI techniques, in particular those in the field of Reinforcement Learning, in order to design more sophisticated algorithms that take into account historical information about both the infrastructure and application run time states.

References

[1] F. A. Salaht, F. Desprez, A. Lebre, An overview of service placement problem in fog and edge computing, ACM Comput. Surv. 53 (2020). doi:10.1145/3391196.

[2] D. Calcaterra, G. Di Modica, O. Tomarchio, Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks, Journal of Cloud Computing 9 (2020). doi:10.1186/s13677-020-00194-7.

[3] B. Varghese, E. de Lara, A. Ding, C. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele, P. Willis, Revisiting the arguments for edge computing research, IEEE Internet Computing 25 (2021) 36–42. doi:10.1109/MIC.2021.3093924.

[4] X. Kong, Y. Wu, H. Wang, F. Xia, Edge computing for internet of everything: A survey, IEEE Internet of Things Journal 9 (2022) 23472–23485. doi:10.1109/JIOT.2022.3200431.

[5] M. Goudarzi, M. Palaniswami, R. Buyya, Scheduling iot applications in edge and fog computing environments: A taxonomy and future directions, ACM Comput. Surv. 55 (2022). doi:10.1145/3544836.

[6] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, A. Ahmed, Edge computing: A survey, Future Generation Computer Systems 97 (2019) 219–235. doi:10.1016/j.future.2019.02.050.

[7] M. Riedlinger, R. Bernijazov, F. Hanke, AI Marketplace: Serving Environment for AI Solutions using Kubernetes, in: Proceedings of the 13th International Conference on Cloud Computing and Services Science - CLOSER, 2023, pp. 269–276. doi:10.5220/0000172900003488.

[8] A.-A. Corodescu, N. Nikolov, A. Q. Khan, A. Soylu, M. Matskin, A. H. Payberah, D. Roman, Big data workflows: Locality-aware orchestration using software containers, Sensors 21 (2021). doi:10.3390/s21248212.

[9] P. Kayal, Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope : Invited paper, in: 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), 2020, pp. 1–6. doi:10.1109/WF-IoT48130.2020.9221340.

[10] S. Böhm, G. Wirtz, Towards orchestration of cloud-edge architectures with kubernetes, in: S. Paiva, X. Li, S. I. Lopes, N. Gupta, D. B. Rawat, A. Patel, H. R. Karimi (Eds.), Science and Technologies for Smart Cities, Springer International Publishing, Cham, 2022, pp. 207–230.

[11] I. Ahmad, M. G. AlFailakawi, A. AlMutawa, L. Alsalman, Container scheduling techniques: A survey and assessment, Journal of King Saud University - Computer and Information Sciences (2021). doi:j.jksuci.2021.03.002.

[12] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade, Queue 14 (2016) 70–93. doi:10.1145/2898442.2898444.

[13] A. Marchese, O. Tomarchio, Extending the Kubernetes Platform with Network-Aware Scheduling Capabilities, in: Service-Oriented Computing: 20th International Conference, ICSOC 2022, Springer-Verlag, Seville, Spain, 2022, p. 465–480. doi:10.1007/978-3-031-20984-0_33.

[14] A. Marchese and O. Tomarchio, 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), SciTePress, 2023, pp. 261–268. doi:10.5220/0011972600003488.

[15] Z. Rejiba, J. Chamanara, Custom scheduling in kubernetes: A survey on common problems and solution approaches, ACM Comput. Surv. 55 (2022). doi:10.1145/3544788.

[16] C. Carrión, Kubernetes scheduling: Taxonomy, ongoing issues and challenges, ACM Comput. Surv. 55 (2022). doi:10.1145/3539606.

[17] L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, M. Hong, Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh, in: IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, 2021, pp. 1–9. doi:10.1109/INFOCOM42981.2021.9488670.

[18] L. Cao, P. Sharma, Co-locating containerized workload using service mesh telemetry, in: Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 168–174. doi:10.1145/3485983.3494867.

[19] S. Nastic, T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, D. Vii, Y. Xiong, Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters, in: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), 2021, pp. 206–216. doi:10.1109/CLOUD53861.2021.00034.

[20] K. Fu, W. Zhang, Q. Chen, D. Zeng, X. Peng, W. Zheng, M. Guo, Qos-aware and resource efficient microservice deployment in cloud-edge continuum, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2021, pp. 932–941. doi:10.1109/IPDPS49936.2021.00102.

[21] T. Pusztai, F. Rossi, S. Dustdar, Pogonip: Scheduling asynchronous applications on the edge, in: IEEE 14th International Conference on Cloud Computing (CLOUD), 2021, pp. 660–670. doi:10.1109/CLOUD53861.2021.00085.

[22] A. C. Caminero, R. Muñoz-Mansilla, Quality of service provision in fog computing: Network-aware scheduling of containers, Sensors 21 (2021). doi:10.3390/s21123978.