Author copy of paper published at *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*

Polaris Scheduler: Edge Sensitive and SLO Aware Workload Scheduling in Cloud-Edge-IoT Clusters

Stefan Nastic
Reinvent Labs GmbH
Vienna, Austria
snastic@reinvent-group.at

Thomas Pusztai
Andrea Morichetta
Víctor Casamayor Pujol
Schahram Dustdar
Distributed Systems Group, TU Wien
Vienna, Austria
lastname@dsg.tuwien.ac.at

Deepak Vij
Ying Xiong
Futurewei Technologies, Inc.
Santa Clara, CA, USA
firstname.lastname@futurewei.com

***Abstract*—Application workload scheduling in hybrid Cloud-Edge-IoT infrastructures has been extensively researched over the last years. The recent trend of containerizing application workloads, both in the cloud and on the edge, has further fueled the need for more advanced scheduling solutions in these hybrid infrastructures. Unfortunately, most of the current approaches are not fully sensitive to the edge properties and also lack adequate support for Service Level Objective (SLO) awareness. Previously, we introduced software defined gateways (SDGs), which enable managing novel edge resources at scale. At the same time Kubernetes was initially released. In spite of not being specifically developed for the edge, Kubernetes implements many of the design principles introduced by our SDGs, making it suitable for building SDG extensions on top of it. In this paper we present Polaris Scheduler – a novel scheduling framework, which enables edge sensitive and SLO aware scheduling in the Cloud-Edge-IoT Continuum. Polaris Scheduler is being developed as a part of Linux Foundation's Centaurus project. We discuss the main research challenges, the approach, and the vision of SLO aware edge sensitive scheduling.**

***Index Terms*—Edge Computing, Service Level Objectives, Elasticity, Container Scheduling**

I. Introduction

The next generation clouds are already expanding beyond traditional data centers into the far edges of the network. This is completely transforming how we perceive the notion of computing infrastructures and other digital resources such as storage and network. Additionally, novel types of resources, e.g., Edge-based gateways are becoming an integral part of this novel computational fabric. We are witnessing a paradigm shift, in which digital resources are becoming truly ubiquitous and first-class citizens available across the entire Cloud-Edge-IoT (CEI) continuum. This requires us to rethink how we perform scheduling and placement of novel cloud- and edge-native workloads.

As the CEI systems become more complex and more dependent on third-party services, Service Level Objectives (SLOs) become increasingly important. Typically, SLOs define specific and measurable capacity guarantees of a workload, e.g., available memory to a provisioned VM. Next generation SLOs aim to go a step further and provide guarantees about workload's performance, which is easier to understand, communicate and generally more relevant [1], [2]. Unfortunately, for application workloads, which run in edge environments, traditional SLO enforcement and violation mitigation mechanisms, such as elastic scaling are not easily attainable as they are for more traditional cloud-native workloads. Therefore, for such edge-native workloads it is important to address SLOs as early as possible in their orchestration and management lifecycle. Typically, this means considering SLO constraints and requirements already in the CI/CD pipeline, i.e., during the deployment phase.

Since its inception in 2014[1], Kubernetes has put itself forward as a de facto standard for orchestrating and managing containerized workloads (pods) in the cloud. While Kube scheduler and other Kubernetes-based schedulers are well optimized for the cloud environment, where compute nodes are powerful and network connections have consistently high throughput, they largely lack features, which are needed for scheduling containers in edge environments. Recently, various solutions have been emerging to offer Kubernetes distributions, which are specifically tailored for the edge. Examples include KubeEdge[2] [3], MicroK8S[3] and K3S[4]. Unfortunately, such and similar solutions largely lack adequate support for considering SLOs for workload scheduling at the edge.

In our previous work we introduced software defined gateways (SDGs) [4], [5] which enable abstracting and managing novel edge resources at scale. At the same time Kubernetes was initially released. In spite not being specifically developed for the edge, Kubernetes implements many of the design principles introduced by our SDGs. This and its inherent extensibility makes it a perfect candidate for building SDG extensions on top of it.

In this paper, we present a novel Polaris Scheduler. It is being developed as a part of Linux Foundation's Centaurus

This work is supported by Futurewei's Cloud Lab. as part of the overall open source initiative.

[1]https://github.com/kubernetes/kubernetes/commit/2c4b3a562ce34cddc3f8218a2c4d11c7310e6d56
[2]https://kubeedge.io/
[3]https://microk8s.io
[4]https://k3s.io

project[5] and it builds on our SDGs to enrich them with SLO awareness for the edge-native scheduling. In particular, we discuss the main research challenges to achieve optimal scheduling in the novel Cloud-Edge-IoT continuum, vision and approach to the SLO aware edge scheduling.

The remainder of this paper is structured as follows: Section II provides further motivation for our work, by discussing the main research challenges. It also provides an overview of our background work in this area. Section III presents the main approach and the architecture of our Polaris Scheduler. In Section IV, we introduce our SLO-aware scheduling pipeline and describe main Polaris Scheduler extension plugins. In Section V we discuss the related approaches, which are most relevant to Polaris Scheduler. Finally in Section VI we conclude our paper and provide a future outlook.

II. Motivation and Background

In scope of our work on Linux Foundation's Centaurus project, we identified several research challenges, which are critical missing pieces for enabling effective and efficient workload scheduling. To further motivate our work, subsequently we discuss the most important research challenges of workload scheduling and placement in the Cloud-Edge-IoT continuum. After that we provide an overview of our main projects, which conceptually and practically underpin this work.

A. Research Challenges

Our approach aims to address main research challenges, which include:

RC-1 *Scheduling based on dynamically changing data:* Most of contemporary schedulers work based on apriori static or only limited dynamic information, such as availability of (physical) nodes, network topology, and so forth. To be able to deal with the high dynamicity in Cloud-Edge-IoT continuum an access to fine-grained, real-time monitoring information is required during scheduling. Such information can be used to continuously update the information on the state of the underlying infrastructure.

RC-2 *User input aware scheduling process:* Scheduling is usually performed on a fixed set of attributes, with little support for user inputs. This is particularly important for workloads, which are associated with SLOs, as it is inefficient and costly to do scheduling without considering user-provided SLOs. For example, this can lead to early SLO violations, immediate after scheduling, which would then trigger mitigation actions, effectively forcing a rescheduling of the workload.

RC-3 *Considering workload's dependencies and internal structure:* Most of the schedulers view a workload as a black box, meaning that they do not consider any information pertaining to workload's internal structure. This has an obvious disadvantage that the scheduling cannot be optimized for particular properties of a workload, such as that Service A needs to communicate with Service B. This type of QoS data, which is internal to the workload, holds valuable information that can be useful in preventing premature SLO violations. Finally, being able to guarantee all dependencies, e.g., a virtual sensor is attached to a node, are satisfied before starting a workload on the node ensures correct readiness behaviour.

RC-4 *Hybrid infrastructure placement and scheduling:* Usually the Cloud-Edge-IoT infrastructure is not flat. Instead, it is mainly organized ad hoc, usually in a hierarchical manner. For example, it is a common case to have a so called gateway node, which manages a portion of the edge network and mediates its communication with the cloud. This means that the scheduler might not have "direct access" to all the nodes in the logical Cloud-Edge-IoT cluster. This obviously poses a big issue for the traditional schedulers and placement approaches, which assume having complete information about the cluster's nodes, network topology and auxiliary (virtual) devices.

RC-5 *Scheduler scalability:* Scheduling resources on top of a very large, hyper distributed infrastructure can be very costly in terms of determining suitable nodes based on workload's scheduling score and constraints. This is an issue for optimization heuristics that rely on static data to select nodes, but more so for the approaches, such as ours, which also consider and account for dynamic infrastructure changes.

RC-6 *Opportunistic resource exploitation:* Opportunistic scheduling on mobile and constrained devices is another challenge, which is caused by a highly dynamic nature of edge infrastructures. This means that and Cloud-Edge-IoT scheduler not only needs to account for the dynamically changing state of the edge (see RC-1), but also for dramatic changes to the infrastructure itself, such as devices coming and going, dynamic network changes, e.g., due to cellular tower handovers, and so forth.

RC-7 *Considering edge-specific locality properties:* Generally, the main goal of any scheduling approach is to find an optimal utilization of available resources, while satisfying certain constraints. Contrary to the traditional systems, we are usually required to make different decisions and efficient trade-offs regarding data and computation movement at the edge. Some of the edge-specific locality constraints, which are important to consider for optimal scheduling include: locality of data storage nodes, proximity of the container registry, and locality in the Cloud-Edge-IoT compute continuum, e.g., cloud-nodes vs. edge-edge).

RC-8 *Embedding intelligence in the scheduler:* Optimal workload scheduling is one of the main tasks for efficient operations of edge-native applications and jobs. As the intelligence permeates through the edge, applications are significantly benefiting. Additionally, such edge intelligence will play an important role to realizing next generation operations, management and orchestration. One of the key challenges to realize this AI-driven EdgeOps is more intelligent scheduling and placement of workloads

[5]https://www.centauruscloud.io

across the Cloud-Edge-IoT continuum.

RC-9 *Energy aware scheduling:* Both in the cloud data centers and for the edge devices, energy consumption plays a very important role. Power consumption of the Cloud-Edge-IoT infrastructure and the scheduled workloads has broad implications. They span from environmental impacts of power sources to actual functional limitations of specific devices. Therefore, a scheduling algorithm needs to consider a cost model, which can optimize resource utilization, but at the same time minimize the overall energy consumption of the entire Cloud-Edge-IoT infrastructure. Finally, when considering power needs when scheduling a workload, it is also important to factor in unreliable and dirty power sources, as well as the fact that some edge devices are not always on.

B. Background

The Polaris Scheduler builds on amalgamation of a number of our previous and ongoing projects and collaborations. Next, we summarize the work most pertinent to Polaris Scheduler.

1) Project Centaurus: Centaurus is a Linux Foundation project, which provides a cloud infrastructure platform, which can be used to build public or private clouds. Centaurus unifies the orchestration, network provisioning and management of cloud compute and network resources at regional scale. Polaris Scheduler utilizes several subprojects (SIGs) of Centaurus Project. These include: Arktos [6] and Polaris SLO Cloud[6] [2].

The Polaris SLO Cloud project [2], [7] is part of Centaurus and aims to make complex SLOs first class citizens in Cloud Computing. It features high-level abstractions and a framework for defining and implementing complex metrics that aggregate lower-level metrics, complex SLOs that operate based on these metrics, and elasticity strategies that reach beyond simple horizontal or vertical scaling.

Arktos is designed as a multi-tenant and large scale cloud platform evolved from Kubernetes [8] to schedule, provision, and manage resources for VM, Container, and Serverless functions. Specifically, the vision of Arktos is to 1) manage a very large compute cluster, in the order of 100K compute nodes, this allows us to have the scale to experiment various scheduling algorithms for increasing resource utilization for cloud providers; 2) unify the technology stack to manage various resource types such as bare metal servers, virtual machines, and lightweight containers, which allows the platform to manage resources more efficiently at different granularity level, thus achieving the true elasticity of the platform; 3) provide true multi-tenant computing environment with strong isolation so that resources can be shared without impacting other tenants. This further improves resource utilization by using techniques such as resource reclamation among tenants.

2) Software Defined Gateways: Software defined gateways [4] encapsulate the cloud and edge resources and abstract their provisioning and governance. Their main purpose is to support virtualizing compute resources in Cloud-Edge-IoT continuum. Their main aim is to provide managed execution environments for application workloads, which can be easily orchestrated.

To achieve this, SDGs encapsulate functional aspects (e.g., communication capabilities or sensor poll frequencies) and non-functional aspects (e.g., quality attributes, elasticity capabilities, costs and ownership information) of the Edge resources and expose them to the SDG provisioning middleware [5]. The functional, provisioning and governance capabilities of the units are exposed via well-defined APIs, which support provisioning and controlling the SDGs at runtime. Our conceptual model also allows for composing and interconnecting SDGs, in order to dynamically orchestrate and deliver the Cloud-Edge-IoT resources.

The Polaris Scheduler builds on these concepts. It also adopts the notion of a provisioning daemon and gateway profiler from SDG provisioning middleware.

3) Project KubeEdge: KubeEdge [3] is an open source system for extending native containerized application orchestration capabilities to hosts at Edge. It is built on top of Kubernetes and it provides fundamental infrastructure support for network, application deployment and metadata synchronization between cloud and edge.

KubeEdge architecture includes a network protocol stack called KubeBus, a distributed metadata store and synchronization service, and a lightweight agent (EdgeCore) for the edge. KubeBus is designed to have its own implementation of OSI network protocol layers, which connects nodes at edge and in the cloud as one virtual network. KubeBus provides a unified multitenant communication infrastructure with fault tolerance and high availability. The distributed metadata store and sync service is designed to support the offline scenario when edge nodes are not connected to the cloud. EdgeController manages remote edge and cloud nodes as one logical cluster, which enables KubeEdge to schedule, deploy and manage container applications across edge and cloud with the same API.

The Polaris Scheduler builds on top of KubeEdge and aims to enhance it with a novel approach to workload scheduling in Cloud-Edge-IoT continuum, which is sensitive to the edge properties and also SLO aware.

III. Polaris Scheduler Approach

A. Main Concepts and Objectives

The main aim of our Polaris Scheduler is to facilitate SLO-aware scheduling of workloads in hybrid Cloud-Edge-IoT continuum clusters. This entails reaching the following *objectives*:

1) Polaris Scheduler Framework aims to enable *capturing intrinsic interdependencies among a workload's services*, in order to enable users to specify their QoS and SLOs constraints and requirements. This objective addresses: RC-2 and RC-3.
2) With Polaris Scheduler we aim to *support continuous monitoring of the physical infrastructure nodes and dynamic cluster properties*. This information needs to be constantly fed to the scheduler in order to support the

[6]https://polaris-slo-cloud.github.io

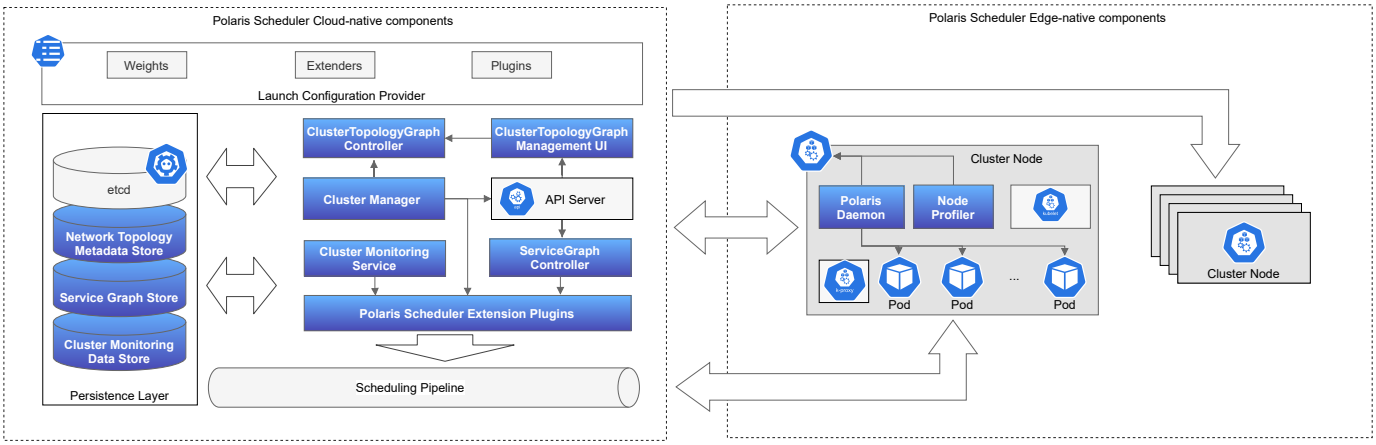

Fig. 1: Polaris Scheduler Architecture Overview.

main scheduling process. This objective addresses RC-1 and RC-6.

3) Polaris Scheduler provides *a structured approach to dealing with infrastructure properties, inherent to the Cloud-Edge-IoT continuum*. To this end, it aims to provide a comprehensive set of extensions (plugins), which can capture and interpret edge-specific properties, reflecting them in the main scheduling process. This objective addresses: RC7 and RC9.
4) We take a pragmatic approach with Polaris Scheduler and go beyond proposing a new algorithm and heuristics to *provide a complete framework that can be used in any Cloud-Edge-IoT cluster*, which is based on Kubernetes or any of its distributions. Our framework is rooted in main cloud- and edge-native principles such as hierarchical decomposition, horizontal scalability and high degree of automation. This objective addresses RC-4, RC-5 and RC-8.

To achieve these objectives Polaris Scheduler defines and implements the following main concepts:

- A *Service Graph*, which models workload's components and their interactions. It provides enough metadata about the workload to the scheduling pipeline to make it possible to optimally schedule the workload at the edge. This achieves Objective 1.
- A *Cluster Topology Graph*, which maintains the cluster- and infrastructure-specific states in order to make the scheduling pipeline edge sensitive and SLO aware. To keep these states up to date, in face of highly dynamic Cloud-Edge-IoT continuum, Polaris Scheduler implements an Infrastructure Monitoring Service that works in cooperation with Polaris Daemon and Node Profiler. This achieves Objective 2.
- *Edge aware and SLO aware extension plugins* which are responsible to implement edge- and SLO-awareness in the scheduling pipeline. For example, Polaris Scheduler has specific plugins, which support reasoning about the locality and proximity properties. This achieves Objective 3.
- Polaris scheduler is distributed across the cloud and the edge. Its control plain is inherently cloud-native and highly scalable. Additionally, it contains a set of agents and profilers, which run inside the nodes. This achieves Objective 4.

In the continuation, we provide an architecture overview of our Polaris Scheduler.

B. Architecture Overview

Figure 1 shows a high-level view of the framework architecture. The architecture is organized in two main parts. The main layers and components of the first, cloud-native part are shown on the left-hand side of the figure. The cloud-native portion of Polaris Scheduler can be seen as its control plain and it is intended to fully run in the cloud. Similarly, the right-hand side of Figure 1 shows the edge-native portion of our framework. It includes pods/containers (application workload), but also services, daemons and agents, which execute on each node in the cluster. It is important to note that these components need to be optimized for the edge, but they can, naturally, run in the cloud as well, i.e., across the entire Cloud-Edge-IoT continuum.

Polaris Scheduler is built on top of KubeEdge, which is a Kubernetes distribution specially designed for the edge[7]. Figure1 clearly illustrates the components developed by Polaris Scheduler (marked as purple boxes) and the components which are adopted from KubeEdge/Kubernetes (indicated as gray boxes with blue pictograms). The main components comprising Polaris Scheduler are: *1) Scheduling Pipeline, 2) Persistence Layer, 3) Cluster Topology Graph Controller, 4) Cluster Manager, 5) Service Graph Controller, 6) Infrastructure Monitoring Service, 7) API Server, 8) Launch Configuration Provider, 9) PolarisDaemon, 10) NodeProfiler, 11) Kubelet and 12) Network Proxy*.

The *Scheduling Pipeline* is the core part of our Polaris Scheduler. It implements the scheduling algorithm and defines a set of extension plugins. The Polaris Scheduler provides

[7]When referring to the components common to both orchestrators, we use Kubernetes and KubeEdge interchangeably.

these plugins to customize the scheduling pipeline workflow, in order to make it SLO aware and sensitive to the infrastructure properties that are specific to the edge, such as locality, device mobility and so forth. To achieve this, the Polaris Plugins tap into the metadata, which is specifically provided by custom Polaris components. Additionally, the plugins maintain their own configuration models, which capture information pertinent to the workload, cluster and SLOs. We discuss the Scheduling Pipeline in more detail in Section IV.

The *Persistence Layer* of Polaris Scheduler is responsible for storing all shared state and metadata, which is used during the main scheduling process. In addition to the Etcd cluster, which is the default key-value store used in most Kubernetes distributions, Polaris Scheduler uses three additional data stores. They are used to persist infrastructure models (Cluster Topology Graph Store), main dependencies among workload's services (Service Graph Store), real time monitoring information (Cluster Monitoring Data Store). We discuss this and the corresponding controllers and services in more detail subsequently.

The *API Server* is a component of the Kubernetes control plain that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plain. Polaris Scheduler adopts the Kubernetes API server, which can be used to submit a workload for scheduling. Additionally, the API Server is used as the central gateway that facilitates access to the cluster's shared state (Persistence Layer) through which all other components interact.

The *Cluster Monitoring Service* in coordination with the Node Profiler (which runs on each cluster node) executes a sequence of runtime profiling actions to complete the dynamic node profile. For example, the profiling actions include: currently available RAM, firewall settings, environment information, list of processes and daemons, and list of currently running pods. As soon as the latest version of the node profile snapshot is complete, it is communicated back to the Cluster Monitoring Service. The monitoring service stores the snapshots in the Cluster Monitoring Data Store. This data is then used during the main scheduling process to determine if a workload can be scheduled on a specific node and to check the SLOs.

The *Cluster Manager* is responsible for managing all the cluster resource for Polaris Scheduler. This involves providing information about the available resources, such as persistent volume claims, but also updates about the changes in the infrastructure, such as to its network connections. The former can be easily obtained from Kubernetes API Server, via its List and Watch APIs[8]. The later is more complex and requires a custom solution. For this reason Cluster Manager works together with the API Server and Cluster Topology Graph controller. The *Cluster Topology Graph controller* is our custom Kubernetes controller, which is responsible for managing Cluster Topology Graph Custom Resource Definition (CRD). The details of the CRD are presented in a later section. At the moment, it is important to note that, since the Cluster Topology Graph is implemented as a Kubernetes CRD it can be updated through the Kubernetes API. This is a nice side effect, which enables the Cluster Manager to hook into the Cluster Monitoring Service, receive updates about the infrastructure changes and consequently update the store metadata, which is used during workload scheduling. Finally, the Cluster Topology Graph can be manually specified or updated through the Cluster Topology Graph Management UI. For example, a field engineer can use the management UI to configure the node topology when bootstrapping the physical infrastructure or to update it at a later stage.

The *Service Graph Controller* is another custom Kubernetes controller, which is responsible for managing the Service Graph CRD. The Service Graph Controller enables attaching a custom service dependency graph to a workload. This service graph is used to enable users to attach custom SLO mappings [1] to their workloads. This effectively enables users to associate specific Quality of Service (QoS) and SLO constraints and requirements on a sub-workload granularity, e.g., for a communication bus connecting two services. More details on Service Graph CRD are presented later in the paper.

The *Launch Configuration Provider* is inherited from Kubernetes. The scheduler policy can be configured to specify which weights, extenders, and plugins are used in the main scheduling process. The latter play a very important role for Polaris Scheduler, as it uses a set of custom plugins to enrich the main scheduling process with SLO-aware considerations (cf. Section IV). Therefore, this enables specifying per-cluster scheduling configuration models. Currently, the launch configuration of the scheduler policy supports three formats: configuration file, command-line parameter, and ConfigMap.

The *Polaris Daemon* is responsible to make the Polaris Scheduler aware of the edge-specific properties. The Kubelet is the primary node agent that runs on each node in the cluster. Polaris Daemon is the edge-native node agent, which runs alongside Kubelet. The Polaris Daemon extends our SDG daemon to collect edge-specific metadata about the node, its compute capabilities, networking, auxiliary hardware (e.g., accelerators, sensors, etc.) and other node capabilities [5], [9], [10]. This is done by periodically probing the edge node and communicating this information back to the Polaris Scheduler cloud-based control plain. More specifically, the Cluster Topology Graph Controller processes this metadata and makes it readily available to the Scheduling Pipeline.

C. Cluster Topology Graph

Typically in cloud-only Kubernetes clusters, the node and network topology is very uniform, thus the Kubernetes scheduler has no notion of such a topology and assumes it is completely flat. While this makes sense in such environments, it is not sufficient to capture complex relationships among the constituents of the Cloud-Edge-IoT continuum. For example, at the edge it is very common to have the infrastructure hierarchically organized, typically containing many gateway edge devices. Such gateways can manage hundreds of IoT

[8]https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19/

devices or even integrate entire subsystems, such as factory floors.

To make the *scheduling workflow edge sensitive and SLO aware*, the Polaris Scheduler needs an explicit representation of the cluster node (physical) topology and the network QoS parameters for the links between the nodes. To this end, we created a cluster topology abstraction that can be configured to represent the topology of the cluster. For example, this can be done by a field engineer through the Cluster Topology Graph Management UI (cf. Figure 1). The Infrastructure Monitoring Service and Polaris Daemon continuously monitor the state of the underlying infrastructure and reflect its changes in the Cluster Topology Graph CRD. This metadata is then used by the scheduling pipeline plugins.

The Cluster Topology Graph models the topology of the cluster, i.e, it describes the infrastructure model. Its links record the network QoS parameters of the corresponding network connections. This graph is critical for the NetworkType and NetworkQoS plugins (discussed in Section IV). While the former only needs to check single network links, the latter has to be able to compute the shortest path between any two nodes of the cluster, which is why a graph representation is required.

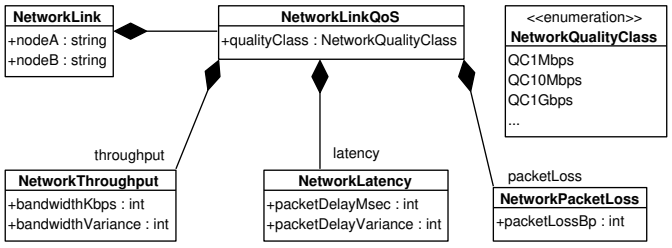

Fig. 2: Cluster Topology Graph formal model (NetworkLink partial view).

Since Kubernetes offers no solution for directly storing a graph resource, each link in the topology is stored as a separate NetworkLink CRD instance, whose UML class diagram is shown in Figure 2. The network topology graph itself is constructed by the Polaris Scheduler at runtime, based on these network links. This approach also facilitates updates of the cluster's network topology, because a change in a single network connection, requires updating only one NetworkLink resource, as opposed to updating a large graph resource.

D. Service Graph

In the Cloud-Edge-IoT continuum, application workloads typically execute across network boundaries and on a very heterogeneous devices and nodes. To account for such variability Polaris Scheduler introduces Service Graph CRD. The Service Graph is crucial piece which provides enough metadata about the workload to the scheduling pipeline plugins to make scheduling the workload possible. It also captures the necessary information to make the scheduling SLO-aware.

The service graph abstraction used in Polaris Scheduler models workload's components and how they interact with each other. Each node in the service graph represents an application component that can be deployed as a set of pods (a service node should not be confused Kubernetes Service), while each directed link between two component indicates that there is a communication link between the two nodes. Each service graph link can be annotated with the network QoS requirements, i.e., SLOs, for the connection between the nodes it connects. Additionally, the service graph enables specifying comprehensive SLO constraints and requirements making the Polaris Scheduler fully aware of workload's SLOs.

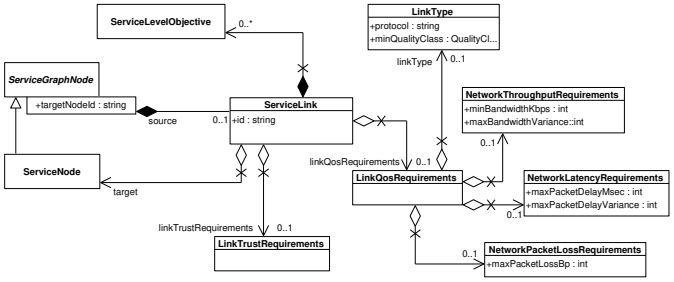

Fig. 3: Service Graph formal model (Service Link partial view).

For example, the service graph can be used for defining SLO constraints on the network links. Each service graph node will have an ID that can be referenced by a Kubernetes-native deployment to allow the scheduler to find the appropriate service graph node and constraints during scheduling (similar to how a Persistent Volume Claim can be referenced by a Pod). This assumes that the service graph is submitted to the cluster before any of the deployments. FInally, usage of the service graph is optional, i.e., deployments that do not reference a service graph are scheduled without SLO optimizations.

IV. Edge Sensitive and SLO Aware Workload Scheduling

A. Scheduling Pipeline of the Polaris Scheduler

Workload scheduling lifecycle represents a set of steps and interactions that need to be performed to schedule a workload in a Cloud-Edge-IoT environment. For the simplest example where a workload equals one pod, the workload scheduling lifecycle entails the steps necessary to bind the pod to a node, from the time some deployment controller signals that a new (replica) pod needs to be created. Figure 4 gives an overview of the scheduling workflow. The same figure (on the right-hand side) illustrates the most important steps in the Polaris Scheduler scheduling pipeline.

The *Scheduling Pipeline* implements the core phases of workload scheduling. Similarly to most multi-criteria decision making (MCDM) online scheduling algorithms, Polaris Scheduler also consists of two main phases: the scoring phase and the binding phase. We further divide these phases into four main steps: i) filter, ii) score host, iii) select host and iv) bind pod. Polaris Scheduler implements its scheduling pipeline based on Kubernetes scheduling framework, extending it with a number of edge-specific plugins.

In Figure 4, the main steps of the workload scheduling lifestyle are marked with purple circles and are numbered accordingly. The circles marked with a star represent continuous

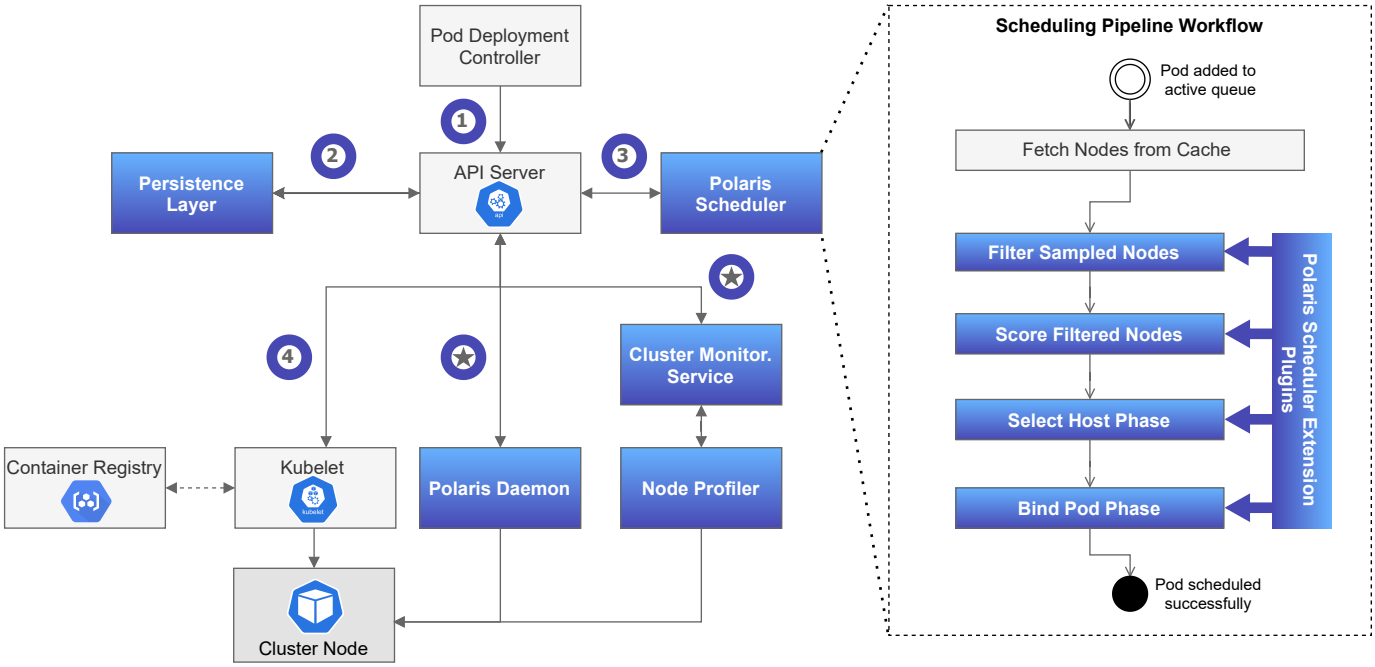

Fig. 4: Overview of workload scheduling lifecycle and key phases of scheduling pipeline.

flow of data. The boxes depicted in purple represent components specific to the Polaris Scheduler. Other components are provided by Kubernetes or another orchestrator. In the simplest case (workload = pod), the scheduling process starts when a pod deployment controller creates a new (replica) pod. In the first step, the controller communicates a (updated) pod specification (e.g., a YAML file) to the API Server. In the second step, the API Server makes sure to persist the pod specification in the Persistence Layer etcd. In the third step, the Polaris Scheduler is triggered and it puts the pod in the active queue for scheduling and starts the *scheduling pipeline*. The actual steps of the scheduling pipeline are described subsequently. After a suitable node is found, the scheduler triggers the API Server which invokes the node's primary agent (Kubelet) to spin up the pod (shown as step four).

Finally, there are two additional interactions, which are crucial for performing pod scheduling at the edge. These are marked with purple circles and a star. First, we have the Polaris Daemon, which runs on each node of the cluster, continuously collecting edge-specific metadata about the node and delivering it to the Cluster Topology Graph Controller, which stores this information as *Cluster Topology Graph* CRD. Second, the Node Profiler also runs on each cluster node and continuously executes a sequence of runtime profiling actions to create a node profile snapshot. This information is also stored as Cluster Topology Graph CRD. The CRD is cached for better performance. These specific interactions make the scheduling pipeline edge aware and the scheduler sensible to the inherent edge properties and constraints.

Figure 4, on the right-hand side, also zooms in on the scheduling pipeline and shows the main phases of its workflow. These phases include: *i) Fetch nodes from cache; ii) Filter phase; iii) Score phase; iv) Select Host phase; and v) Bind pod phase*. The figure also depicts how Polaris Scheduler extends these phases with a set of custom plugins to make the whole scheduling pipeline sensitive to the edge-properties and SLO aware.

In the *fetch nodes from cache* phase the scheduler retrieves all the node data from cache. This data includes the nodes themselves with appropriate locality labels, such as 'edge-node', but also all other metadata which is captured by Polaris Cloud custom CRDs. Finally, at this point we combine the latest profiling snapshots with the rest of the metadata.

In the *filter phase* the scheduler analyzes the cluster nodes, retrieved in the previous phase, one-by-one, until sampling rate/scope has been reached. Sample scope is a cluster-specific metric, which determines the minimum number of nodes that should be analyzed before moving to the scoring phase. As seen in the Figure4, Polaris Scheduler provides custom plugins to make this phase sensitive to edge properties and SLO aware. In a nutshell, the plugins in this phase are used to filter out nodes that cannot run the pod. For example, a plugin in this phase might use the information provided in the Service Graph to filter out nodes which do not provide required LinkType, such as 5G network capability.

In the *score phase*, the nodes are sorted by the Polaris Scheduler plugins. These plugins are used to rank the nodes that have passed the filtering phase. At the end of this phase, the node with the highest score becomes the Select Host. Currently, our plugins particularly focus on capturing the locality and proximity properties. In the next section, we analyze different properties which also need to be considered by future scoring plugins to enable full-fledged, edge- and SLO-aware scheduling.

TABLE I: Example Polaris Scheduler Plugins for SLO Awareness.

Plugin Name	Scheduling Pipeline Phase	Functionality Overview
GraphSort	Sort	Sorts the queued service instances that belong to the same ServiceGraph, according to a breadth-first traversal of the graph. Instances that belong to different graphs, are sorted according to their creation timestamps.
ServiceGraphManager	PreFilter, Reserve	Loads the ServiceGraph of the application that the service instance belongs to and updates it with the selected node.
NetworkType*	Filter	Filters out nodes that do not support the LinkType (WiFi, 4G, 5G, etc.) and QualityClass (10 Mbit/s, 1 Gbit/s, 10 Gbit/s, etc.) demanded by the ServiceLinks of the service instance.
NetworkQoSFilter**	Filter	Filters out nodes that do not meet the QosRequirements (i.e., throughput, packet delay, jitter, and packet loss) of the service instances' ServiceLinks. This entails looking up the nodes of already scheduled service instances that have a dependency on the current instance and calculating the QoS of the connection to these nodes.
NodeCost*	PreScore, Score	Assigns higher scores to cheaper nodes.
PodsPerNode*	PreScore, Score, Normalize Score	Increase colocation of different services of the same application on a node to reduce latency between the services.
NetworkQoSScore**	Score	Assigns higher scores to nodes that are likely to maintain the network requirements for a prolonged period of time.
WorkloadType**	PreScore, Score	Uses insights from previous deployments of this workload type to assign more suitable nodes a higher score.
AtomicDeployment*	Permit	Ensures that when a new Service Graph is deployed, either all its services are deployed or none at all. After the initial deployment, i.e., when new service instances are added due to scaling, this plugin does nothing.

When the pipeline reaches the *Select Host phase*, a host node has already been selected and the scheduler waits for the node to prepare for the arrival of the pod. This phase exists to prevent race conditions while the scheduler waits for the bind to succeed. Additionally, Polaris Scheduler uses this phase to update workload's service graph. This has to be done in this phase, as the scheduling is done one pod at a time and we need to propagate latest version of the service graph for each pod. In case the binding phase fails, Polaris Scheduler provides mechanisms to rollback the service graph to the last stable version.

Finally, the *Bind Pod phase* runs plugins that are used to perform any work required before the pod is bound. For example, a plugin may provision a virtual sensor on the target node before binding a pod to run on it.

B. Polaris Scheduler Extension Plugins

To realize the SLO-Aware scheduling behaviour Polaris Scheduler takes advantage of Kubernetes' highly-flexible scheduling extension points. These are essentially a set of hooks, which can be used to extend or customize the scheduling lifecycle. To this end, Polaris Scheduler defines a set of plugins which capture SLO specific behaviour and integrate it with the scheduling lifecycle.

Table I lists most important examples of the plugins added by Polaris Scheduler. Generally, we categorize Polaris Scheduler plugins into static-data and dynamic-data plugins, depending on the type of data they operate with. Further, based on their general optimization goal or constraint type, we can classify plugins into the ones that focus on optimizing workload's performance (should be generally applicable to edge workloads) and the ones that specifically facilitate meeting SLOs. For example, in Table I Polaris Scheduler plugins, which are marked with an asterisk ('*') operate on mostly static data and aim to increase the performance of edge workloads. The plugins marked with '**' operate on data that may change dynamically and are directly related to SLOs, such as throughput, packet delay, jitter, and packet loss.

C. Properties for SLO aware scheduling

As previously discussed, the edge has many specific properties which are not typical for the Cloud. In this regard, next we highlight examples of such properties, which need to be considered by specific plugin objectives to achieve fully fledged, SLO aware and edge aware scheduling. The Polaris Scheduler is currently taking into consideration locality, but the road-map of the project includes at least all the following:

- *Locality*: The distribution of the edge fabric through the geographical space implies that applications can require that the deployment of their containers is done at specific geographic locations. Therefore, it can be required to the scheduler that some data is stored next to the processor units that have use it, and not in a cloud storage far from them.
- *Device*: The heterogeneity of devices the the edge requires a careful specification of the requirements for the deployment in terms of hardware capacity. In this regard, given the constrained capacities of some edge devices, it can be required to the scheduler to deploy on certain hardware that is specifically proficient for the task, even if in general terms is not as suitable as other available devices.
- *Mobility*: Some edge devices can be mobile, in this regard, some application can require that the deployment hardware moves to a certain region or that it keeps moving around a certain area. There are applications that require to sense specific areas, where might not be any sensor. This type of SLOs can require the scheduler to only deploy the task to the hardware able to move to that specific location.

- *Availability*: Edge devices are, in general, constrained. Therefore their availability depends on two main factors: energy and connectivity. The deployment might require a minimum time of availability for the application, therefore, the device is required to have enough energy. Similarly, some edge devices might have only network connection during a specific period of the day, which needs to be taken into account for the deployment. Therefore, the specific availability of the resource will be required to be given as an SLO in order to balance the device requirement with its availability.
- *Network*: Similarly as for the devices, a manifold of different networks can connect the edge devices, and each with its own idiosyncrasies. In this sense, a deployment can specify to the scheduler a specific type of connection for its application, for instance, to ensure enough bandwidth or that the latency is below a certain threshold.
- *Provider*: The edge tier is composed by different providers, for business or economic reasons a deployment can have a preference over a specific provider. This is business oriented, however, it can be very useful from that point of view in order to enable emerging applications.
- *Energy efficiency*: Given the heterogeneity of devices and topologies of the edge tier, there can be several configurations that match the deployment requirements. In this sense, the user can chose the configuration that is more efficient in terms of energy consumption, aiming at reducing its deployment cost or to increase the time of availability for the deployment. Therefore, the scheduler requires such a plugin in order to offer this type of SLO.
- *Sustainability*: Given the urgent climatic emergency, some deployments can prioritize the use of configurations with characteristics such as: the lowest carbon footprint, the lowest energy consumption or the one that can be shared and re-used.

It is clear that these plugins can have some conflicts between them, so that optimizing the device does not matches with having the required network. Such conflicts can be mitigated by associating weights with the plugins using the Launch Configuration Provider (cf. Section III and Figure 1).

To sum up, these are properties that the Polaris Scheduler aims at incorporating as plugins in the near future. With the development of more CEI systems and the evolution of the society needs more of them can be added to the road-map.

V. Related Work

Container Schedulers: Kubernetes default scheduler is an online scheduler, which at its core implements a greedy multi-criteria decision making (MCDM) algorithm. This scheduling algorithm performs well in the typical Cloud environments, but it lacks features which are needed for scheduling containers in edge environments, i.e., where compute nodes are resource constrained, network connections can be unreliable, compute fabric is highly heterogeneous, device nodes are geographically dispersed, and the infrastructure is highly dynamic (devices can suddenly come and go). Additionally, other container schedulers, such as Docker Swarm and Apache Mesos and Apache Hadoop YARN, implement a similar greedy MCDM algorithm. A key phase of MCDM is the so-called scoring phase. In that phase, the algorithm calculates the score of feasible node, and selects the highest scoring node for scheduling. Therefore, our general approach is complementary to these approaches as we also rely on a similar algorithm, particularly extending the scheduling pipeline with a set of plugins. These plugins make the scheduling pipeline sensitive to the edge properties and SLO aware.

Service Placement Problem (SPP): In the context of Service Placement Problems (SPP), the works usually formulate the task as an optimization problem and an algorithm is implemented to solve an instance of the problem heuristically by leveraging assumptions within the system mode. For example, Wang et al. [11] developed a reinforcement-learning-based hierarchical service tree placement strategy, aiming to optimize the net utility, defined in their work as achieved utility minus network congestion. Placing services also involves disseminating data. Aral and Ovatman [12] handled it, in the context of edge computing, as a facility location problem. They covered the dynamic creation, replacement, and removal of replicas, where the decisions were guided by the continuous monitoring of data requests from the edge nodes. We do not compete with them, as we do not propose a novel algorithm or solution to the optimization problem.

QoS-aware scheduling: Regarding QoS-aware scheduling, Delimitrou et al. [13] developed Paragon, a recommendation system in the context of data centers, based on collaborative filtering techniques, to classify unknown workload and assign the best configuration according to previously scheduled applications. Still, in the frame of data manipulation, Cardellini et al. [14] propose a QoS-aware and self-adaptive scheduler as an extension of Storm. The scheduler monitors the data rate exchanged, and thanks to a distance-based selection mechanism, proposes to schedule applications with complex topologies in geographically distributed environments.

Another stream of research for QoS-aware scheduling involves the IoT scenario. In this regard, Li et al. [15] proposed a three-layer QoS scheduling model for service-oriented IoT, based on the use of self-defined metrics for each layer and Markov Decision Model solutions. Hong et al. [16] developed a QoS-aware network resource management framework to obtain a better bandwidth control with containers for bandwidth-sensitive applications. They propose three scheduling policies, i.e., proportional share scheduling, minimum bandwidth reservation, maximum bandwidth limitation, that can be combined to improve the network usage. Murtaza et al. [17] introduced a QoS-aware service provisioning for Internet of Everything (IoE) tasks. The approach analyzes the various service types of IoE requests and presents strategies to allocate the most appropriate available fog resource. Scarlett et al. [18] first formulate a formal model for fog resources and applications, defining the fog landscape scenario. They then propose an optimization solution for the Fog Service Placement Problem, solved as an integer linear programming model, with the aim

of maximizing the utilization of the fog landscape given the application's requirements, considering the active involvement of the cloud resources. Brogi et al. first [19] developed a prototype for better deploying fog applications, estimating their QoS-assurance and Fog resource consumption considering latency and throughput in their simulations. Then, they extended the work, including a model and predictive analytics to predict resource consumption and cost [20], [21].

Workload scheduling on hybrid cloud-edge/cloud-fog infrastructures: Towards the cloud-edge continuum scheduling, Fu et al. [22] proposed a reinforcement-learning-based online resources manager and load-aware microservice scheduler. The system's goal is to ensure optimization of both computational and network resources for microservices while guaranteeing the QoS. A reinforcement learning technique is also proposed by [23] to place Virtual network functions in the cloud-edge continuum optimally. Mahmud et al. [24] developed a service placement for Industry 4.0 IoT systems in a fog environment, proposing a heuristic methodology. This work considers the context IoT devices, for example, the data size and sensing frequency, the details of various industry 4.0-oriented applications (I4OAs), including execution model, time and space complexity, dependency, and resource requirements.

Workload scheduling in mobile cloud: Mobile edge computing scheduling has to deal with delays caused by users' mobility, which happens when they move away from serving edge clouds. Wang et al. [25] solved it as a coordination problem using the Markov decision process framework. Furthermore, they proposed a reinforcement-learning-based online microservice coordination algorithm to learn the optimal strategy. Tian et al. [26], in their paper, they target the Cloudlet platform proposing a scheduling workflow with QoS enhancement. They solved this as a multi-objective function. Chunlin et al. [27] inspect how to guarantee QoS for users of mobile devices, analyzing the cost and the benefits of mobile cloud, proposing to solve the multiple mobile/cloud context for service scheduling as a utility optimization problem. Wang et al. [28] propose a scheduling algorithm in Mobile Cloud Computing (MCC) for obtaining higher profit and low energy consumption. They consider multi-tasks with time constraints, and they map the problem with a heuristic algorithm based on Ant Colony Optimization.

The approaches described mainly deal with specific problems, designing accurate algorithms that solve a limited spectrum of cases. In our proposal, instead, the scope is broader. We present the design of a complete framework that can operate in the whole Cloud-Edge-IoT continuum, including the capture of the system's intrinsic properties and working on every Kubernetes-based (or any other orchestrator) cluster.

VI. Conclusion and Future Work

In this paper we presented Polaris Scheduler, which is a novel scheduling framework that enables edge sensitive and SLO aware scheduling in Cloud-Edge-IoT Continuum. Polaris Scheduler is being developed as a part of Linux Foundation's Centaurus project. It brings SLO awareness and edge sensitivity to the Cloud-Edge-IoT continuum. Polaris Scheduler implements a Service Graph, which models workload's components and their interactions. It provides enough metadata about the workload to the scheduling pipeline to make it possible to optimally schedule the workload in at the edge. Polaris Scheduler introduces a Cluster Topology Graph, which maintains the cluster- and infrastructure-specific states in order to make the scheduling pipeline edge sensitive and SLO aware. To keep these states up to date, in face of a highly dynamic Cloud-Edge-IoT continuum, Polaris Scheduler implements an Infrastructure Monitoring Service that works in cooperation with Polaris Daemon and Node Profiler. Finally, Polaris Scheduler implements a set of extension plugins for enriching the workload scheduling pipeline. These plugins are responsible for implementing edge- and SLO-awareness in the scheduling pipeline. For example, by using its specific plugins, Polaris Scheduler enables reasoning about the locality and proximity properties.

In the future, we aim to continue developing Polaris Scheduler as a part of Linux Foundation's Centaurus Project[9]. We plan to continue enriching its plugins ecosystem, to include support for considering other infrastructure properties, which are relevant in the Cloud-Edge-IoT continuum. Another line of research will include, adding support for sustainable and energy-aware scheduling. Finally, we aim to provide better integration of scheduling SLOs with the elastic scaling strategies, to provide better support for elasticity, but also SLO violations prevention and mitigation at the edge.

References

[1] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "Sloc: Service level objectives for next generation cloud computing," *IEEE Internet Computing*, vol. 24, no. 3, pp. 39–50, 2020.

[2] T. Pusztai, S. Nastic, A. Morichetta, V. Casamayor Pujol, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "A novel middleware for efficiently implementing complex cloud-native slos," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[3] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 373–377.

[4] S. Nastic, S. Sehic, D.-H. Le, H.-L. Truong, and S. Dustdar, "Provisioning software-defined iot cloud systems," in *2014 international conference on future internet of things and cloud*. IEEE, 2014, pp. 288–295.

[5] S. Nastic *et al.*, "A Middleware Infrastructure for Utility-based Provisioning of IoT Cloud Systems," in *The First IEEE/ACM Symposium on Edge Computing*, 2016.

[6] Futurewei Technologies Inc., "Arktos open source project repository," URL: https://github.com/futurewei-cloud/arktos, 2020, [Online; accessed April-2020].

[7] T. Pusztai, S. Nastic, A. Morichetta, V. Casamayor Pujol, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "Slo script: A novel language for implementing complex cloud-native elasticity-driven slos," in *2021 IEEE International Conference on Web Services (ICWS)*, 2021.

[8] "Kubernetes," URL: https://kubernetes.io/, 2020, [Online; accessed April-2020].

[9] S. Nastic, M. Voegler, C. Inziger, H.-L. Truong, and S. Dustdar, "rtGovOps: A Runtime Framework for Governance in Large-scale Software-defined IoT Cloud Systems," in *Mobile Cloud 2015*, 2015.

[10] S. Nastic, G. Copil, H.-L. Truong, and S. Dustdar, "Governing elastic iot cloud systems under uncertainty," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, pp. 131–138.

[9] https://www.centauruscloud.io

[11] Y. Wang, Y. Li, T. Lan, and N. Choi, "A reinforcement learning approach for online service tree placement in edge computing," in *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 2019, pp. 1–6.

[12] A. Aral and T. Ovatman, "A decentralized replica placement algorithm for edge computing," *IEEE transactions on network and service management*, vol. 15, no. 2, pp. 516–529, 2018.

[13] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[14] V. Cardellini *et al.*, "On qos-aware scheduling of data stream applications over fog computing infrastructures," in *IEEE Symposium on Computers and Communication (ISCC)*, 2015, pp. 271–276.

[15] L. Li, S. Li, and S. Zhao, "Qos-aware scheduling of services-oriented internet of things," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1497–1505, 2014.

[16] C.-H. Hong, K. Lee, M. Kang, and C. Yoo, "qcon: Qos-aware network resource management for fog computing," *Sensors*, vol. 18, no. 10, p. 3444, 2018.

[17] F. Murtaza, A. Akhunzada, S. ul Islam, J. Boudjadar, and R. Buyya, "Qos-aware service provisioning in fog computing," *Journal of Network and Computer Applications*, vol. 165, p. 102674, 2020.

[18] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards qos-aware fog service placement," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 14.05.2017 - 15.05.2017, pp. 89–96.

[19] A. Brogi and S. Forti, "Qos-aware deployment of iot applications through the fog," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1185–1192, 2017.

[20] A. Brogi, S. Forti, and A. Ibrahim, "Deploying fog applications: How much does it cost, by the way?" in *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER'18)*, 2018.

[21] *Predictive analysis to support fog application deployment*. Wiley, 2019.

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

[23] M. Bunyakitanon, X. Vasilakos, R. Nejabati, and D. Simeonidou, "End-to-end performance-based autonomous vnf placement with adopted reinforcement learning," *IEEE Transactions on Cognitive Communications and Networking*, vol. 6, no. 2, pp. 534–547, 2020.

[24] R. Mahmud, A. N. Toosi, K. Ramamohanarao, and R. Buyya, "Context-aware placement of industry 4.0 applications in fog computing environments," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 11, pp. 7004–7013, 2019.

[25] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, 2019.

[26] W. Tian, R. Gu, R. Feng, X. Liu, and S. Fu, "A qos-aware workflow scheduling method for cloudlet-based mobile cloud computing," in *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2019, pp. 164–169.

[27] L. Chunlin, Y. Xin, Z. Yang, and L. Youlong, "Multiple context based service scheduling for balancing cost and benefits of mobile users and cloud datacenter supplier in mobile cloud," *Computer Networks*, vol. 122, pp. 138–152, 2017.

[28] T. Wang, X. Wei, C. Tang, and J. Fan, "Efficient multi-tasks scheduling algorithm in mobile cloud computing with time constraints," *Peer-to-Peer Networking and Applications*, vol. 11, no. 4, pp. 793–807, 2018.