Kubernetes Scheduling: Taxonomy, ongoing issues and challenges

CARMEN CARRIÓN, School of Computer Science and Engineering, Univ. Castilla-La Mancha, Spain

Continuous integration enables the development of microservices-based applications using container virtualization technology. Container orchestration systems such as Kubernetes, which has become the *de facto* standard, simplify the deployment of container-based applications. However, developing efficient and well-defined orchestration systems is a challenge.

This paper focuses specifically on the scheduler, a key orchestrator task that assigns physical resources to containers. Scheduling approaches are designed based on different Quality of Service (QoS) parameters to provide limited response time, efficient energy consumption, better resource utilization, and other things. This paper aims to establish insight knowledge into Kubernetes scheduling, find the main gaps, and thus guide future research in the area. Therefore, we conduct a study of empirical research on Kubernetes scheduling techniques and present a new taxonomy for Kubernetes scheduling. The challenges, future direction, and research opportunities are also discussed.

CCS Concepts: • **Computer systems organization → Client-server architectures**; • **General and reference → Surveys and overviews**; • **Software and its engineering → Scheduling**.

Additional Key Words and Phrases: Kubernetes, orchestration, scheduling, containers, survey

1 INTRODUCTION

Cloud-native architectures enable users and developers the flexibility to build, deploy and maintain applications independently of the underlying infrastructure. Thus, the developer can focus on the development and delivery of the application. The Cloud Native Computing Foundation (CNCF) [27] defines cloud-native as a new computing paradigm in which applications are built based on a microservices architecture, packaged as containers, and dynamically scheduled and managed by an orchestrator. In a microservice-based architecture, an application is designed into multiple microservices that are deployed and managed independently and communicate with each other over a network [79, 136, 139]. Nowadays, containers are the *de facto* standard for implementing these microservices [2, 12, 22, 98].

The growth of containers has changed how users conceive the development, deployment and, maintenance of software applications. Containers make use of the native isolation capabilities of modern operating systems with a low overhead in resource consumption and obtaining great flexibility in their deployment. Lightweight and flexible containers have given rise to microservice-based application architectures. Then, this new approach consists of packaging the different services that make up an application in separate, intercommunicating containers, which are deployed in a cluster of physical or virtual machines. The complexity of these applications has led to the need for container orchestration, that is, to provide tools that help us automate the deployment, management, scaling, interconnection, and availability of our container-based applications. Many cloud service providers offer Containers as a service (CaaS) as a cloud service model that simplifies the deployment of containerized applications in the cloud [3]. CaaS use is increasing in companies that take advantage of the portability of containers between environments, avoiding vendor lock-in. In general, a CaaS platform undertakes authentication, logging, security,

Author's address: Carmen Carrión, carmen.carrion@uclm.es, School of Computer Science and Engineering, Univ. Castilla-La Mancha, Avd. Universidad s/n, Albacete, Spain, 02071.

0360-0300/2022/6-ART $15.00
https://doi.org/10.1145/3539606

monitoring, networking, load balancing, auto-scaling, and continuous integration/continuous delivery (CI/CD) functions.

From the cloud service provider's perspective, a CaaS platform creates an abstraction layer that includes a container orchestration engine, typically based on the *de facto* standard Kubernetes [7]. Other well-known toolkits for container orchestration are Docker Swarm [88] and Mesos [48]. Usually, these orchestrators handle a cluster of physical or virtual machines to host the containerized application. In particular, the task of assigning physical resources to containers is performed by the scheduler and, the percentage of total available resources in the system and the Quality of Service (QoS) perceived by the user can be strongly affected by its implementation. Note that scheduling can focus on improving resource utilization, reducing energy consumption, or satisfying users' or applications' time requirements. But, as a rule, designing efficient container scheduling techniques is still an open issue. Even though multiple platforms exist, the wide adoption of Kubernetes [7, 27] across different branches of the industry [28] and public clouds providers (i.e. Google Kubernetes Engine (GKE), Microsoft Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (Amazon EKS)) became it as a *de facto* standard for container orchestration. So, Kubernetes is a leading open-source container orchestration platform, and scheduling in Kubernetes is an active field of research. Hence, the Scheduling Special Interest Group (SIG) [53] is a Kubernetes contributor community interested in various scheduling questions on Kubernetes. More precisely, the Kubernetes Scheduling Interest Group (k8sched) is responsible for the components that make pod placement decisions (the less unit scheduled by Kubernetes). Researching in Kubernetes scheduling for containerized applications is an area of growing interest and this paper focuses on this topic. It is crucial to study the landscape of existing Kubernetes scheduling techniques and understand their strengths and limitations to advance in this important and growing research area.

1.1 Related surveys and our contribution

There exist many surveys discussing virtual machine resource management [1, 4, 39, 60, 69, 71, 77, 82, 91] and there are also surveys dealing with container scheduling [2, 21, 49, 57, 79, 92, 103, 135, 136]. Other works have been published in recent literature, many fewer in number, related to containers' orchestration in fog/edge architectures [15], that is, architectures that provide storage and computing resources close to the place where data are generated [2, 49, 57, 71]. Table 1 summarizes the previous review and survey papers supporting the relevance of container orchestration. But, they suffer from some weaknesses: **a)** All the papers are not updated to the new current scheduling proposals. **b)** Some papers focus on specific architecture environments, such as fog or edge computing. **c)** The papers do not study the topic of scheduling problems in Kubernetes implementations, as the *de facto* standard cluster scheduling and, to the best of our knowledge, no survey covers Kubernetes scheduling techniques to identify active researching area.

The mentioned reasons above motivate us to review the primary literature on Kubernetes scheduling. In particular, this paper analyzes the proposals for the *de facto* Kubernetes orchestrator and presents an updated review from the 2016 to 2022 period. The paper describes the state-of-the-art of the Kubernetes orchestrator which complements previous surveys on container scheduling. In contrast with previous surveys, this paper reviews the works done so far in the field from a general point of view and defines a set of top-down domains; from the infrastructure domain to the application domain including a transversal performance domain. The taxonomy is accompanied by the classification of a broad range of aspects studied for Kubernetes resource scheduling, not being limited to scheduling algorithms or specific physical devices. The existing Kubernetes scheduling techniques are covered focusing on research trends and gaps. So, the paper is helpful for future research to overcome the actual limitations and challenges related to Kubernetes scheduling. To sum up, the main contributions of this paper are as follows:

- A study of the state-of-art of Kubernetes orchestrator for scheduling containerized applications is presented.

Table 1. Related works in cluster orchestration

Ref.	Survey
[1]	The authors analyze the characteristics of various workflow scheduling techniques and classify them based on their objectives (makespan, availability, reliability, energy, communication, utilization, security) and execution model (best effort, QoS constraint).
[2]	The survey classified the scheduling techniques based on the type of optimization algorithm employed to generate the schedule.
[4]	The authors present a comprehensive survey of task scheduling strategies published from January 2005 to March 2018 for cloud computing environments.
[21]	The taxonomy classifies container technologies and orchestrators according to features such as resource limit control, scheduling, load balancing, health check, fault tolerance, and autoscaling.
[22]	The survey presents an analysis of the container technology landscape following the application (HPC, Big Data, geo-distributed), performance, orchestration, and security categories.
[39]	The authors carry out a functional and performance comparison with different real topologies and use both homogeneous and heterogeneous clusters of single-board computer (SBC) devices.
[49]	The authors present a survey on architecture, infrastructure, and algorithms for resource management in fog/edge computing.
[57]	The authors analyze the suitability of Kubernetes in the fog computing model, highlight limitations and provide ideas for further research to adapt to the needs of the fog environment.
[60]	The authors provide a systematic review and classification of proposed scheduling techniques for suitable virtual machines in the field of cloud computing from 2010 to 2019.
[77]	The authors present a survey of scheduling and load balancing algorithms in cloud and fog computing environments covering only swarm-based optimization techniques.
[103]	The authors study different container orchestration platforms and apply a taxonomy based on the workloads, the scheduler architectures, cluster infrastructures, and management of the system.

- A new layered taxonomy to classify Kubernetes resource scheduling techniques is presented.
- An updated review of recent works in Kubernetes scheduling is provided.
- A list of current issues and challenges is compiled, as well as research directions in Kubernetes scheduling to promote further improvements.

1.2 Paper organization

Figure 1 outlines the structure of the survey. The article is organized as follows: Section 2 introduces the background information regarding virtualization, orchestration of containerized applications, and Kubernetes architecture. Then, Section 3 details how resource management works in Kubernetes, with a focus on scheduling. Besides, Section 4 reviews others container orchestration engines. After that, in Section 5, the new taxonomy is presented and, Subsections 5.1 to 5.5 detail the ongoing working topic related to the Kubernetes scheduling and the taxonomy described. The challenges in the area along with some future research are described in Section 6. Finally, Section 7 draws some conclusions.

2 BACKGROUND, TERMINOLOGY AND TECHNOLOGY AVAILABLE

2.1 Operating system-level virtualization

Virtualization can be defined as a technology that enables the creation of logical services by means of resources running on hardware [100]. In other words, virtualization distributes the capabilities of a resource, such as a network, storage, server, or application, among several users or environments [89]. In particular, the operating system-level virtualization technology provides isolated computing environments called containers within a common operating system. In Linux, different kernel tools such as *cgroups*, *namespaces* or *chroot* are used for this

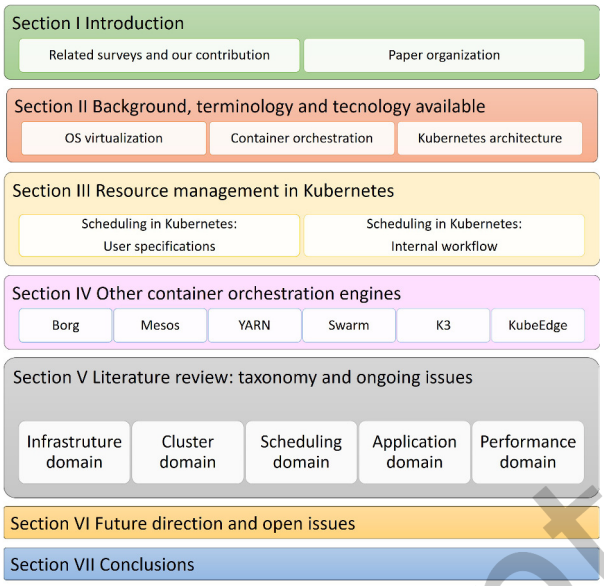
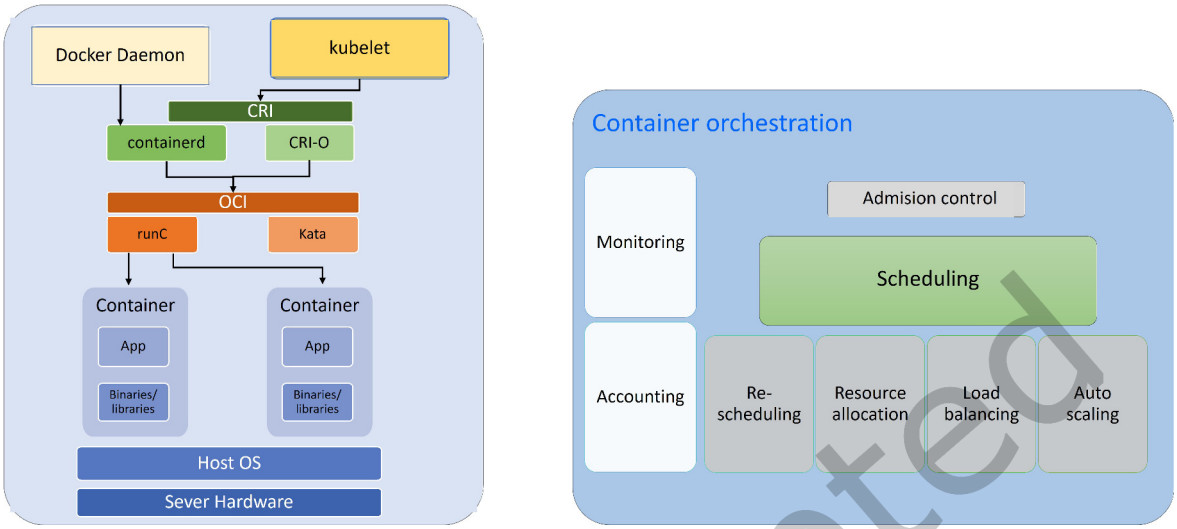


Fig. 1. Structure of the paper.

purpose. The operating system-level virtualization technology provides container-based Cloud services, also known as Container-as-a-Service (CaaS) [3]. All main cloud providers support container deployments in their infrastructure, such as Google Container Engine, Amazon Elastic Container Service (ECS), or Microsoft's Azure Container Service. It is likewise possible to set up a container cluster on private premises, leveraging popular container orchestrators such as Docker Swarm [88] or Kubernetes [7].

A container is a group of one or more processes that are isolated from the rest of the system. It comprises an application and all its library dependencies and configuration files. Moreover, containers offer reproducible execution environments, light lifecycle management, and closer-to-metal performance than classical virtual machine deployments [80, 113]. Nowadays, the most well-known container manager is Docker [5], a layered container platform that comprises several software components for developing, transporting, and running containerized applications, i. e. Docker Daemon, containerd or Docker registry [26].

Figure 2a shows the relationship between different container technologies to manage the lifecycle of containers on a node. At the lowest level of container technology rest container runtimes, such as LXC, RunC, CRun, or Kata, that create and run the container processes [115]. A high-level container runtime manages the complete container lifecycle of its host system being able to pull container images from registries, manage images and hand them over to the lower-level runtimes. Containerd and CRI-O are the most popular high-level container runtimes that implement the Open Container Initiative (OCI), a standard specification for image formats and runtimes requirements providing container portability [41]. Usually, the container runtimes are wrapped in software components called container managers or engines that increase the level of abstraction, In Figure 2a, Docker Daemon acts as a container manager with an API that simplifies the management of the lifecycle of the containers and communicates with containerd. When we scale up to a cluster, it can be extremely difficult to manage the lifecycle and management of containers, especially if the number increase with demand. Hence, container orchestration solves the problem by automating the scheduling, deployment, availability, load balancing, and networking of containers. In Figure 2a, kubelet is the daemon component of the Kubernetes orchestrator that communicates with the high-level container runtime. Note that Kubernetes works with all container runtimes that



(a) Container management architecture on a node (b) Container Orchestrator Components

Fig. 2. Terminology and Technology available

implement the standard Container Runtime Interface (CRI) [1]. More precisely, CRI defines the runtime specification required for integration with Kubernetes.

2.2 Container orchestration

As we have mentioned above, a container orchestrator manages and organizes microservice architectures at scale, dealing with the automation and lifecycle management of containers and services in a cluster. From a general point of view, end-users submit their jobs to the cluster manager master, that is, a core entity of the orchestration system. The cluster manager master is in charge of assigning the submitted tasks to the worker nodes of the compute cluster, where they are executed. It must be noted that usually jobs or applications are composed of one or more services and, services are composed of one or more heterogeneous tasks that are executed on containers. The compute cluster is an abstraction of interconnected nodes that can be physical or virtual machines in different infrastructures, such as clouds or private clusters.

Container orchestrators can be classified as on-premise or managed solutions. Borg, Mesos [48] or Kubernetes [7] are examples of on-premise orchestrators that need to be installed, configured, and managed on physical or virtual infrastructures. Managed solutions are instead offered by cloud providers as a service and need only to be partially configured. Examples of managed solutions are Google Kubernetes Engine (GKE), Microsoft Azure Kubernetes Service (AKS), and Amazon Elastic Kubernetes Service (Amazon EKS). Figure 2b shows the most characteristic components of both on-premise and managed cluster orchestrators. Next, we summarize the main functionality of these modules. More details can be found in [103] where the authors present a reference architecture for container orchestration systems.

A key element of container orchestrators is scheduling, which is the focus of this paper. The *scheduling module* is responsible for identifying the best location to complete incoming tasks. In other words, the scheduling describes a schema to place a container on a compute node of the cluster, at a given time instant. Most scheduling policies map containers based on the state of the system (resource constraints, node affinity, data location) as

[1] https://kubernetes.io/docs/concepts/architecture/cri/

well as metrics such as power consumption or response time, or makespan. Moreover, the *rescheduler component* implements a task relocation policy that determines a new location for a task. This relocation can be triggered for preemption reasons or by the system state to consolidate the load or improve resource utilization. Note that it is not necessary to implement all the components to have a fully functional container orchestration system. The *resource allocation module* reserves the cluster resources following a request-based approach that can be static or dynamic over time. The *load balancing module* is in charge of distributing tasks across container instances based on criteria such as fairness, cost-energy, or priory. The default policy for load balancing is round-robin, but other strategies can be used. The *autoscaling module* is in charge of providing horizontal and vertical scaling depending on the workload demand. In the horizontal scaling, nodes are added or deleted while in the vertical autoscaling the node resources associated with a task are increased or reduced. A simple autoscaling mechanism can be based on a threshold value on CPU or memory usage. For example, Kubernetes uses Horizontal Pod Autoscaler to perform autoscaling based on CPU and memory metrics [7]. Finally, the *admission control module* is responsible for checking that there are sufficient resources in the cluster to run the user's jobs and never exceed the quota assigned to it. And, the *accounting module* monitors the available resource for a user while the *monitoring module* keeps track of real-time resource consumption metrics for each node and collects metrics related to the health of the resources to support fault tolerance systems.

2.3 Kubernetes architecture

Kubernetes, also called as K8s, is an open-source system aimed at automating the deployment, management, and scaling of containerized applications in distributed clusters [7]. It is the evolution of Google's Borg [126] project, which had more than a decade of experience running scalable applications in production. It was released to the open-source community in 2014 and since that time it has not stopped growing.

Production systems that use K8s are complemented with tools from the K8s ecosystem to improve and facilitate their management. So, the ever-growing K8s ecosystem is composed, to name a few, of complementary tools for managing deployments (Helm, https://helm.sh/), service mesh management (Istio, http://istio.io/), monitoring (Prometheus, https://prometheus.io, Grafana, https://grafana.com) or logging the system (Kibana [25]). An updated description of the K8s ecosystem can be found in https://landscape.cncf.io. Although describing their features is beyond the scope of this paper, it is worth noting that many scheduling proposals interact with some of these tools to, for example, extract real-time information about the state of the system or predict the use of a resource [20, 86, 107, 129, 130].

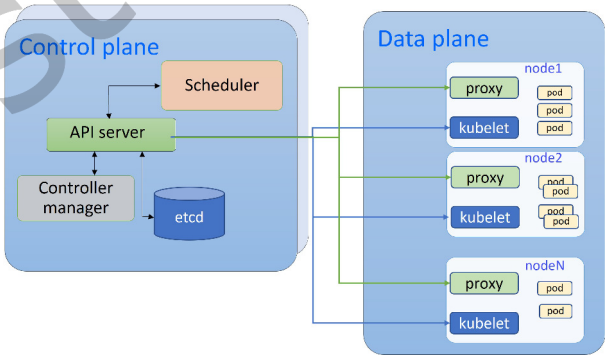


Fig. 3. Kubernetes Orchestrator: Internal architecture

From an architectural point of view, a K8s cluster consists of a set of nodes (physical or virtual machines) integrated to function as a single entity (see Figure 3). These nodes have different functions, distinguishing

between master nodes and worker nodes. Using software-defined overlay networks, such as Flannel or Calico, allows K8s to assign a unique IP address to each pod and service [97]. Note that a pod is the basic deployment unit that can be operated and managed in the cluster. The master node coordinates the cluster, and the worker nodes provide a source of resources for the cluster. A single master node is sufficient to run a cluster, although 3 master nodes are typically found in high availability topologies (HA). K8s uses an event-based declarative engine and the principle of loosely coupled components. In short, the components of the K8s master node are as follows: **a)** *etcd* is a key-value database used to synchronize the desired state of the system. **b)** *scheduler* maps each pod on a worker node. **c)** *API server* receives commands and manipulates the data for K8s objects, which are persistent entities representing the state of the cluster. The API server exposes a RESTful HTTP API to describe an object with JSON or YAML. Moreover, users can send commands to the API server by using the K8s command line interface (CLI), *kubectl*. **d)** *controller manager* monitors *etcd* and forces the system into the desired state. ReplicaSet, Deployment, Job, or DaemonSet are some of the most well-known K8s controllers that provide different functionalities such as availability, roll-back, task execution, or a pod running on each node, respectively. Independent controllers communicate through the object changes in the API server and the events that these changes trigger through informers. Controllers monitor the status of deployments and perform the necessary actions to ensure their successful execution. If the controller requires to deploy a new pod, the scheduler will perform the logic to find the most suitable worker node.

On the other hand, worker nodes are in charge of running the pods of an application. In particular, *kubelet* is the node agent responsible for the lifecycle of the deployed pods and monitoring pods and node status.

To conclude this section, it is important to mention that K8s is supported by an active research community. For example, contributions on autoscaling, authentication, architecture, security, or scheduling are supported by Special Interest Groups (SIGs), which are open source communities. In particular, the sig-scheduling [2] group focuses on contributions to scheduling pods.

3 RESOURCE MANAGEMENT IN KUBERNETES

Kubernetes automates the most efficient distribution and scheduling of containerized applications across the nodes of the cluster (physical or virtual) balancing the use of resources on each node. Previously, Subsection 2.2 presented the most characteristic components of cluster orchestrators from a general point of view. These components can also be embedded in K8s (note that not all of them need to be implemented in an orchestrator). Table 2 summarizes the resource management features implemented in K8s along with their key characteristics. Note that K8s administrators and users can configure a wide range of components to better control resource usage and final application performance. When defining an application in a cluster, the master node receives the information via the *Kubernetes API* and deploys the application to the worker node it deems appropriate. In particular, the scheduler component looks for pods that are in pending state, because they have just been created and do not yet have a worker node assigned and finds the best worker node to run the pod. The placement decision in a worker node is based both on the scheduling policy and the user specification. Subsections 3.1 and 3.2 discuss the user specifications and internal functionality of K8s scheduling in more detail.

3.1 Scheduling in Kubernetes: User specifications

Users can configure a wide range of options to specify the conditions that the scheduler should satisfy. To this end, user specifications indicate different types of constraints that play the role of a control admission. The constraints can be node-level, namespace-level, or pod-level. Next, we will provide a brief explanation.

At the node level, there is the ability to assert some control over what workloads can be run on a given set of nodes via *affinity* and *taint*. The *affinity* property attracts pods to a set of nodes, while taint does the opposite. The *taint* property allows a node to repel a set of nodes. In addition, there is a complementary property called

[2]https://github.com/kubernetes/community/tree/master/contributors/devel/sig-scheduling

Table 2. Features implemented by K8 orchestrator

Feature	**Technique**	Function
Scheduling	kube-scheduler, custom	Assign containers to worker nodes based on different policies and user specifications.
Admission control	mutating and validating controllers, custom	Intercept requests to the K8s API server before the object is persisted, but after the request is authenticated and authorized. This increases the security of the cluster.
Load balancing	round-robin, custom	Distribute the load among multiple container instances. Complex policies can be provided by external load balancing.
Health check	start-up, liveness, readiness, and shutdown probes	Control whether a container can answer to requests.
Fault-tolerance	replica controller, custom	Specify and maintain a desired number of containers.
Auto-scaling	horizontal POD autoscaler, custom	Reshape the K8s workload by automatically increasing or decreasing the number of pods. Custom metrics can be used.

tolerance. Each node has a *tolerance* and if the *taint* is above the *tolerance*, it will schedule it to another node with a higher *tolerance*.

At the pod level, we can optionally specify how much of each resource a container needs. Using the *request* and *limit* properties, we can control the minimum and maximum amount of resources requested for the pod to run. The most commonly specified resources are CPU and memory (RAM). The *request* quota is strict, while the *limit* quota is not. This means that the addition of the allocated resources of previous containers plus the new *request*resource cannot exceed the resource set of the node. But a container can exceed its *limit* quota for some time.

It must be emphasized that in some cases it may be necessary to kill the execution of a pod due to a lack of resources. For example, if a node has 2 cores and 1.5 has already been allocated, the node capacity will exceed 100% every time a new pod requests more than 0.5 cores. In this case, K8s takes different actions depending on the Quality of Service (QoS) class of the pods. In the *best-effort* class, where the pod contains neither the request nor the limit property, the pod can be eliminated if necessary. In the *guaranteed* class, the pods specify a request value equal to the limit. Thus, their survival is guaranteed. Finally, there is a third QoS class called *burstable*, in which the pods specify the requested property. *Burstable* pods are guaranteed at least a minimum amount of resources. It is important to define these values correctly because pods can be eliminated in an undesirable way at a critical moment.

Finally, at the namespace level, we can also specify the request and limit capacity of pods within a K8s namespace. These parameters can be fixed thanks to the *LimitRange* and *ResourceQuota* properties of a namespace.

3.2 Scheduling in Kubernetes: Internal workflow

The overall resource management process in K8s can be summarized as follows: When a user requests the creation of a pod with certain compute resources, the master node of K8s receives the request. The request is forwarded to the *API server* and the scheduler (*kube-scheduler*) acknowledges the request. *Kube-scheduler* determines which worker nodes (i.e. host servers) should run the pod and notifies the *kubelet* agent, running on the worker node, to create a pod. If the pod has specified computing resources in the description file, *kubelet* utilizes *cgroups* and *tc* to reserve computing resources for the pod. When the pod creation is finished, *kubelet* informs *API server* that the pod is running. At runtime, users can modify the resource configuration by submitting an update request of the YAML description file to the master node. Note that the K8s *API server* uses optimistic concurrency (when the *API server* detects concurrent write attempts, it rejects the latter of the two write operations.) In short, the scheduler runs as a pod on the master node, being part of the K8s control plane and is responsible for matching

the best node on which to run new application pods. Multiple different schedulers may be used in a cluster but *kube-scheduler* is a reference implementation provided as the *Default Scheduler* in K8s.

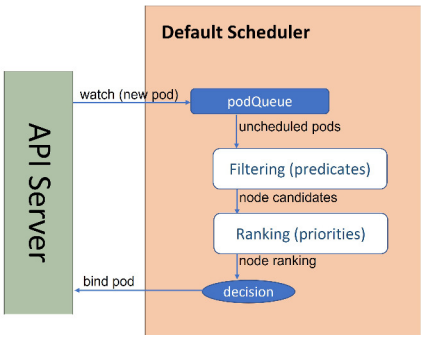


Fig. 4. Life cycle of the default scheduler in K8

3.2.1 Default Scheduler. is a core component of K8s that acts as a typical single dispatcher. It binds a pod to a specific working node. The lifecycle of the default scheduler is shown in Figure 4 and works as follows: **a)** The scheduler maintains a queue of pods called *podQueue* that keeps listening to the *API Server*. **b)** When a pod is created, the pod metadata is first written to etcd through the *API Server*. **c)** The *default scheduler*, as a controller, follows the watch state, takes action, and updates the state pattern. So, it watches the unbound pods from the *etcd* and each time an unbound pod is read from the *etcd*, the pod is added to the *podQueue*. **d)** The main process continuously extracts pods from the *podQueue* and assigns them to the most suitable nodes for running those pods. **e)** The scheduler updates the pod-node binding in the *etcd* in order to be notified to the *kubelet* on the worker nodes. **f)** The kubelet component running in the selected worker node, which monitors the object store for assigned pods, is notified that a new pod is in "pending execution" and it executes the pod. So, the pod starts running on the node. The logic of the main process iterates over the nodes in a round-robin fashion and performs per each unbinding pod the filtering and ranking substeps.

The filtering step: pre-selects workers nodes that satisfy certain requirements of the pod based on a list of policies. In particular, node filtering is based on predicates, Boolean functions that indicate whether a pod fits a worker node. The constraints of a pod are established by using node labels in the pod definition. For example, the *disktype:ssd* parameter in the YAML file of a K8s container specifies that only nodes with a solid-state hard disk are eligible for selection. Some policies supported in this step in K8s are: the *PodFitsHostPorts*, which checks if a port requested by the pod is free in the node, the *PodFitsResources* which checks if the node has the free resources needed by the pod (e.g., CPU or memory), the *PodFitsHost* that specifies the name of the node on which the pod must be deployed, or the *CheckNodeCondition* policy that checks if a node is healthy (e.g., the network is available or the kubelet is ready). **The ranking stage**: assigns a score to the remaining candidates depending on certain configurations and metrics. The better the pod matches a node, the higher the score assigned to the node. Finally, the pod is bound to the node with the highest score. In this stage, worker nodes are prioritized by considering various factors such as the state of the node, the availability of resources, and the current workload. In particular, the rank can be set statically based on predefined different policies or a user-defined policy [7, 92, 107]. Selecting the node with the highest percentage of free CPU and memory is a commonly used ranking criterion (*LeastRequestPriority* policy) (default option). Other criteria target balanced resource usage (*BalanceResourceAllocation* policy), distribution of pods of the same service among different nodes (*SelectorSpreadPriority* policy), or random distribution among hosts (*SelectorSpreadPriority* policy). In addition, predefined preferences can be set based on node affinity or anti-affinity rules (e.g., the *NodeAffinityPriority* policy).

The filtering and ranking stages can be configured using scheduling policies (before version v1.23) or scheduling profiles. In the first case, the predicates and priorities selected in the scheduler configuration file define the scheduler, and the default scheduler policy can be overridden. In the second case, a single instance of Kube-scheduler can run multiple profiles and the scheduler profiles are specified using the *KubeSchedulerConfiguration*. Listing 1 shows an example of specifying two different profiles and how plugins provide configurable behaviors at different stages of scheduling in *kube-scheduler*. By default, a pod is scheduled according to the default-scheduler profile. Specific profiles can be used to schedule a pod by specifying their name in *.spec.schedulerName*, as shown in Listing 2 for *mynginx*.

Listing 1. Defining profiles for kube-scheduler

```
apiVersion:kubescheduler.config.k8s.io/v1beta2
kind: KubeSchedulerConfiguration
profiles:
  - schedulerName: default-scheduler
  - schedulerName: new-scheduler
    plugins:
      score:
        disabled:
        - name: NodeResourcesLeastAllocated
        enabled:
        - name: NodeResourcesMostAllocated
          weight: 3
```

Listing 2. Assigning a scheduler to a pod

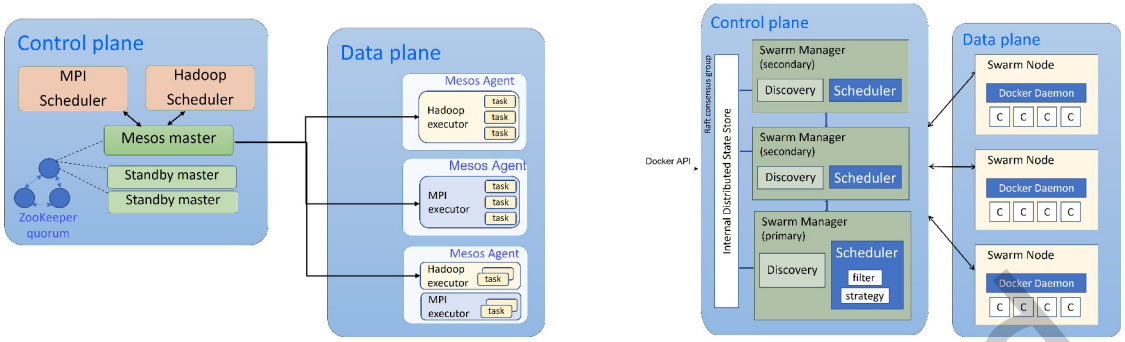
```
---
apiVersion: v1
kind: Pod
metadata:
  name: mynginx
spec:
  schedulerName: new-scheduler
  containers:
  - name: mynginx
    image: nginx:1.15.11
    ports:
    - containerPort: 80
```

3.2.2 Extending K8 scheduler. Kubernetes scheduler supports several ways to extend its functionality [107]. First, you can add new predicates and/or priorities to the default scheduler and recompile it. Second, you can implement an extender process that the default scheduler invokes as the final step. The scheduler extender is a configurable webhook that contains filter and priority endpoints that correspond to the two main phases of the scheduling cycle (filtering and ranging). In this case, the state of the entire cluster stored in the cache of the default scheduler is not shared with the scheduler extender and data communication is done through serial HTTP communication with the associated communication costs. To address these limitations, the *Scheduling framework* [3] come up as an officially recommended extension to the K8s scheduler (introduced in Kubernetes v1.15). It defines new extension points (queue sort, pre-filter, filter, score, reserve, etc.) that are integrated as plugins into the existing scheduler at compile time. This way, users can write new scheduling functions in the form of plugins. Finally, you can implement a custom scheduler process that can run instead of or alongside the default scheduler. A detailed description of how to implement a custom K8s scheduler is beyond the scope of this paper, but the Random (Marton Sereg https://banzaicloud.com/blog/k8s-custom-scheduler/) and Toy (Kelsey Hightower https://github.com/kelseyhightower/scheduler) schedulers are simple examples that show you how to write and deploy custom K8s schedulers. Detailed information on developing the code required to extend the functionality of the K8s scheduler can also be found at [134]. Note that the K8s scheduler functionality can be extended in different languages (e.g. Go, Python, Java).

4 OTHER CONTAINER ORCHESTRATION ENGINES

Nowadays, Kubernetes [7] represents the first project of the Cloud Native Computing Foundation (CNCF) [27], an organization that has more than 36 innovative projects focused on the Cloud Native Computing ecosystem. Moreover, as has been mentioned before, Kubernetes is widely adopted across industries and has become a *de facto* standard. Nevertheless, multiple frameworks are being used in today's production environments for

[3] https://github.com/kubernetes/enhancements/tree/master/keps/sig-scheduling/624-scheduling-framework



(a) Mesos Architecture. (b) Docker Swarm Architecture.

Fig. 5. Architecture of Mesos and Docker Swarm orchestrators

container orchestration, i.e. Borg [126], Mesos [48] or Docker Swarm [6]. In the following, and for the sake of comparison, the architectural characteristics, as well as the scheduling techniques of the main container-based orchestrators are briefly presented.

Borg is the first unified container-management system developed at Google. It manages both long-running services and batch jobs, which had previously been handled by two separate systems [18, 126]. The scheduler uses several techniques to scale up to tens of thousands of machines. Borg uses a master-slave architecture with a logically centralized controller and agent processes that run on each machine.

Borgmaster scheduler uses a monolithic architecture [110, 126]. This means that the scheduler is composed of a single instance that processes all the task requests and has a global state view of the system. When a job is submitted, the Borgmaster records it persistently in the Paxos store and adds the job's tasks to a pending queue. The scheduler asynchronously scans this queue and assigns tasks to machines. The scan proceeds from high to low priority, modulated by a round-robin scheme within a priority to ensure fairness across users and avoid head-of-line blocking. The scheduling algorithm first finds machines on which run the task (machines that both meet the task's constraints and have enough resources). Then, the algorithm selects one candidate machine attending to some score criteria such as the worst or best fit.

Apache Mesos is an open-source cluster manager that implements a two-level architecture (see Figure 5a). This means that a centralized master resource manager dynamically controls which resources each framework scheduler owns [62]. The frameworks (i.e. hadoop, MPI, Mesos Marathon) are in charge of scheduling at the application level.

First, users interact with the frameworks that use ZooKeeper to locate the current master [48]. Then, the frameworks coordinate with the master to schedule tasks onto agent nodes. More precisely, the master manages resource allocations using Dominant Resource Fairness (DRF) [44] as the first-level scheduler and transmits the assigned workload to slave nodes. Agent nodes execute containers and must periodically inform the master about their available resources. In this two-level architecture, applications use their own scheduling policies based on priority preservation and fairness [110].

Hadoop YARN is a cluster manager designed to orchestrate Hadoop tasks, although it also supports other frameworks such as Spark and Storm [40]. Each application framework running on top of YARN coordinates its execution flows. In YARN, there is one resource manager per cluster and one application master per framework. The application master requests resources from the resource manager and generates a physical plane from the resources it receives. The resource manager allocates containers to applications to run on specific nodes. In this context, a container is a bundle of resources bounded to run on a specific machine and for each job, there is an

application manager responsible for managing the lifecycle (i.e., resource consumption, the flow of executions). The application manager needs to harness the resources available on multiple nodes to complete a job. To obtain them, the application manager requests resources from the resource manager that contain the location preferences and properties of the containers.

Docker Swarm is the native clustering Docker's solution for the management and orchestration of a cluster where you can deploy application services as Docker containers. In fact, cluster management and orchestration are embedded as Docker Swarm mode into Docker Engine since version 1.122 [6]. Docker Swarm orchestration engine includes cluster management, scaling, and automatic fail-over but is not an enterprise scalable product solution [88]. Figure 5b shows the main components of a cluster of Docker Swarm, such as the discovery and scheduler module of the Swarm manager.

In particular, the scheduler module orchestrates the swarm nodes and executes a scheduler algorithm that chooses the machine that will execute a container based on two phases: the filtering and the scheduling strategy [29]. Filtering allows selecting nodes based on specific host properties or specific configurations, such as affinity. In the standalone Swarm, the scheduler supports three different scheduling strategies: *spread*, *binpack* and *random*. Swarm manager chooses the least loaded node to place the task under the *spread* strategy, in order to distribute the service across the swarm. In contrast, the *binpack* strategy makes Swarm choose the most loaded node to use fewer machines and to pack as many containers as it could. Under the *random* strategy, Swarm chooses nodes at random.

Table 3. Comparison of container orchestration engines

Framework	Container technology	Scheduler architecture	Scheduler algorithm	Applications
Kubernetes	CRI API, OCI-compliant	cluster centralized	round-robin dispatch, filtering and ranking	all (multiple co-located tasks)
Borg	cgroup-based	monolithic centralized	priorizated round-robin	all (indepen. tasks)
Apache Mesos	mesos contain docker,singularity	two-levels	DRF, framework-based	all (single-task)
Hadoop YARN	cgroups-based, docker	two-levels	FIFO round-robin, fair capacity preemptive	batch apps
Docker Swarm	docker	cluster centralized	filtering and scheduling (spread-binpack-random)	long-running apps

To sum up, Table 3 details several features of the orchestration platforms. Mesos allows multiple frameworks to share one cluster. It takes a distributed approach to combine multiple clusters (up to 100.000 nodes) to manage both containerized and non-containerized applications. Comparing Mesos with K8s, Mesos lacks some K8s features such as persistent volumes on external storage and assigning default IPs to containers [78]. Regarding Docker Swarm, it offers fast deployments and is pretty easy to use than K8s [81]. Nevertheless, Swarm lacks automatic scale, native logging and monitoring components, and native dashboards. Moreover, Docker Swarm uses raft consensus to manage cluster state which limits its scalability [12].

To conclude this section, it is worth pointing out that there are Kubernetes-based distributions such as K3 [93] and KubeEdge [94] that extend the use of K8s in other environments reinforcing it as the *de facto* standard orchestrator. So, **KubeEdge** is a Cloud Native Computing Foundation (CNCF) incubating project built on top of K8s for extending containerized application orchestration capabilities to hosts at edge [94]. A controller plugin for K8s, called *EdgeController*, manages edge nodes and cloud virtual machines as one logical cluster, which enables *KubeEdge* to schedule, deploy and manage container applications across edge and cloud with the same API [131]. Regarding **K3**, it is a stripped-down version of K8s for IoT and Edge application that was first released in March 2019. Scheduling, network, and cluster logic are kept the same as in K8s [93]. Total size requirements

are reduced through a reorganized plugin structure and a less resource-intensive database (sqlite3). Note that from the point of view of scheduling, the scheduler component is identical in both engines.

5 LITERATURE REVIEW: TAXONOMY AND ONGOING ISSUES

The main aim of this section is to identify and summarize the existing literature on scheduling strategies developed for implementation in K8s. To this end, we conduct a systematic review of the literature, using Kitchenham and Charters' manual as a reference [59].

First, an automated search procedure in the IEEE-Explore and Web of Science electronic databases is applied to extract the primary literature on the topic. First, an automated search procedure using the IEEE-Explore electronic and Web of Science databases is used to extract the primary literature on the topic. These are popular databases in the field of computer science and can provide potential coverage of relevant studies. More specifically, we used papers from 2016 to early 2022 for the search and the search pattern was "kubernetes scheduling" or "kubernetes scheduler". We also optimized the definition of the inclusion and exclusion criteria to identify as many relevant articles as possible. The inclusion criteria in the search for relevant studies were, on the one hand, that the article was published in an international peer-reviewed journal or conference and was written in English. On the other hand, the exclusion criteria were that the article was not accessible through university services or memberships and that K8s scheduling was not the focus of the paper, but only mentioned as an example or used by default without any improvement. It should be highlighted that the University provides free access to the browser IEEE Explore, as well as to the regular publications of Web of Science and Scopus. This paper summarizes the results of an extensive, though not omni-comprehensive literature review that analyzes and summarizes the contribution of *96* verified sources. It also classifies the literature using a taxonomy based on five domains and several subdomains. Figure 6 schematically shows the structure of this survey based on the five domains. Note that the domains refer to the following questions:

- **Infrastructure**: What infrastructures are managed by the scheduler in K8 deployments?
- **Cluster**: What cluster characteristics affect the scheduling in K8?
- **Scheduling**: How does the K8 scheduler work?
- **Application**: What kind of applications does the K8 scheduler manage?
- **Performance**: How mature are the K8 scheduling implementations in terms of their evaluation/performance?

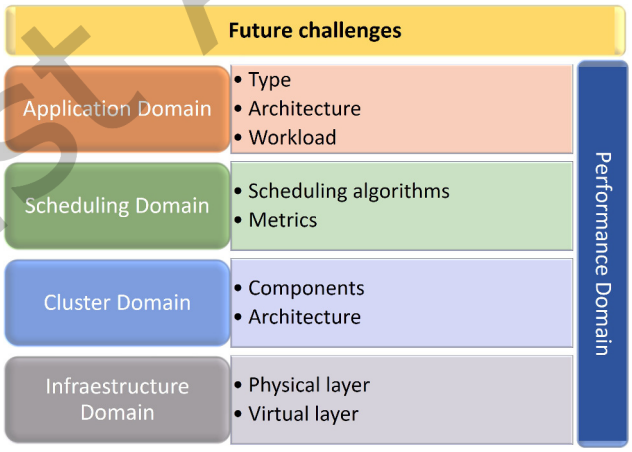


Fig. 6. Literature classification for k8 scheduling: A domain approach.

Finally, this literature review includes a discussion section that identifies the open issues and challenges, as well as the future research directions in the context of K8s scheduling. The key idea is to answer the following

question: What are important open issues and future research in Kubernetes scheduling? Even though future challenges are not considered a classification domain in this taxonomy, it has been included in Figure 6 because it is an important contribution.

5.1 Infrastructure domain

This domain is classified based on the characteristics of each physical or virtual component of the cluster comprising the computation, network, and storage resources. The subdomains are directly related to the following questions:

- What are the physical components considered in the development of the K8 infrastructure?
- What software system is designed to operate on physical devices and manage the resources of those devices?

5.1.1 Physical layer. First, we will describe the physical layer issues that affect the K8s scheduler, focusing on the hardware architecture of the worker nodes. K8s can be used as a container orchestration system in data centers whose core processing hardware components are servers (composed of CPU, disk, memory, and network interfaces). In these systems, the CPU-based compute nodes are powerful, and the network connections have consistently high throughput. However, the K8s scheduler must manage worker nodes with low resource capacities as well as different processor architectures. For example, a microcluster consists of Single-Board Computers (SBCs) and can be integrated with sensor devices to deploy IoT applications. SBCs, such as ARM-based Raspberry Pis (https://www.raspberrypi.com/) or Odroids (https://magazine.odroid.com/), are small computers designed as a single-board circuit with low processing power, reduced memory capacity, and low prices. These hardware-constrained nodes impose difficult limitations on the scheduling system and the K8s scheduler must assign resources efficiently, for example, to prevent SBCs from running out of memory when an application is running. It should also be noted that an application can become very computationally intensive during processing, such as real-time video processing or machine learning. So certain types of advanced applications can significantly speed up their processing time by using accelerators such as GPUs, TPUs, or FPGAs. Hardware-aware scheduling of worker nodes is a key aspect of K8s, as shown by some of the papers reviewed in the literature. Table 4 shows a brief overview of some papers related to the physical layer, classified as hardware-aware approaches and scheduling techniques developed specifically for GPA devices (where *Fe.* refers to *Feature*).

Table 4. Kubernetes scheduling proposals related to the physical layer

Ref. Year	Fe.	Proposal
[55] 2018	hardware	The authors tackle heterogeneous nodes with different CPU architectures, memory, and network connectivity by extending K8s with a monitoring tool. The proposed dynamic scheduler can reschedule stateless applications and provide failure recovery in case of a Raspberry Pi node failure.
[88] 2019		The authors propose to take the physical, operational, and network parameters into consideration, along with the software states to orchestrate edge-native applications dynamically on 5G. The approach extends the limits of the nodes' capabilities.
[122] 2020		The authors present an intelligent K8s scheduler that takes into account the virtual and physical infrastructure of complex data centers. More precisely, a server hardware behavior model is generated using a wind tunnel. Results carried out in real data centers show that introducing hardware modeling into the scheduler maximizes effectiveness (around 10-20%).

Continued on next page

Table 4 – continued from previous page

Ref.	Fe.	Proposal
[8] 2021		Tarema is a system that addresses task and infrastructure profiling to dynamically assign scientific workflow tasks to heterogeneous cluster resources. Groups of similar worker nodes are created considering the hardware properties of the nodes Then, K8s uses node groups and task labels to allocate tasks to available cluster resources at runtime. Tarema reduces the runtimes of real-world workflows compared to the popular shortest job next and shortest jobs to the fastest resources schedulers (around 4.54%) while providing a fair cluster usage.
[83] 2021		Polaris is built on top of KubeEdge and is sensitive to the computational capacity, hardware capacity, and energy efficiency of edge nodes. The Polaris scheduler also takes into account device locality and mobility by abstracting a cluster topology.
[114] 2018	GPAs	Gaia scheduler allocates GPU resources for Nvidia brand and a K8s plugging is used to divide a physical GPU into multiple GPUs and assign these virtual GPUs to a container.
[120] 2019		The authors built a GPU-aware resource orchestration layer, called Knot, and integrated it with K8s. Kube-Knot proposes a scheduler that leverages real-time GPU usage correlation metrics to perform safe pod co-locations, ensuring crash-free dynamic container resizing on GPUs.
[127] 2020		The authors propose an adaptive GPU scheduler that allocates and reallocates GPUs based on a sidecar that monitors the training process on the GPU.
[47] 2020		The authors present a machine learning-based scheduler to select a CPU or GPU hardware device in a K8s environment. The algorithm creates a model of runtime prediction of the applications and is also based on the nature of the processed data.
[35] 2021		The authors extend the K8s scheduler to manage the usage of hardware-accelerated nodes in multi-access edge computing (MEC) architectures. The scheduler uses information about the previous executions of a particular application to select the best CPU or GPU node.

Finally, we discuss and summarize a comparative analysis of the approaches in this subsection. In a nutshell, our review of K8s schedulers over the physical layer shows that node hardware properties are a key aspect of the K8s scheduler. K8s faces the challenge of scheduling limited computing hardware devices, such as single-board computers (SBCs) or specialized hardware accelerators. Overall, CPU hardware models enhance the K8s scheduler by including both static (RAM, vCPU) and dynamic (power consumption) information about the hardware devices. In addition, static profile-based models require hardware analysis in advance, while dynamic hardware models rely on real-time monitoring. We found that contributions to scheduling GPU devices in K8s are still limited and at an early stage. Currently, state-of-the-art GPU scheduling techniques focus on a single application type and ignore GPU utilization. Typically, the application is pre-profiled to meet the application requirements.

5.1.2 Virtual layer. First, we will describe the virtual layer issues that impact the K8s scheduler focusing on container interference and the network layer. On the one hand, virtualization allows multiple software-based entities to run on a single physical device that supports both multi-tenancy and multi-instance, as it provides resource isolation and fault-tolerance between different client organizations or tenants. In a multi-tenancy architecture, a single instance of the software runs on serving multiple tenants, while in a multi-instance model each tenant has their own virtualized instance of the application. As a rule, a publicly accessible cluster needs to be multi-tenant [49]. On the other hand, K8s nodes connect to a virtual network to provide connectivity for pods. The network model allows K8s to assign each pod and service its IP address [97]. Software-defined overlay networks, such as Flannel or Calico, are software components that decouple the physical infrastructure from networking services. Moreover, K8s network virtualization provides a software-based administrative network entity for a tenant. Software-defined networking (SDN) and network function virtualization (NFV) are options adopted for managing the network through software. While SDN leverages the separation of the control and the data plane from underlying routers and switches, NFV decouples the networking functions from the underlying

proprietary hardware covering both containerized network function (CNF) and conventional Virtualized Network Function (VNF) [49]. Next, a brief overview of the related contributions will be presented.

Table 5. Kubernetes scheduling proposals related to virtual layer

Ref. Year	Fe.	Proposal
[72] 2016	interference	The authors develop a reference net-based model (a kind of Petri net) for resource management within K8s, to better characterize performance issues and mitigate the effects of interference.
[11] 2019		*KubeSphere* presents a multi-tenant policy-driven meta-scheduler to provide fairness among competing users.
[56] 2020		The authors design an interference minimization model coupled with minimum energy utilization and maximal green energy consumption for scheduling industrial IoT applications. The proposed scheduler leads to improved energy utilization and minimal interference around 14% and 31%, respectively, in contrast to a standard first-come first-served scheduler.
[51] 2020		The authors propose a scheduling algorithm oriented to a multi-tenant model, in which team users are modeled as virtual clusters and cluster load is monitored regularly. The optimized K8s scheduling algorithm is applied to Docker-based deep learning platforms to ensure load balancing.
[123] 2020		The authors address the problem of interference in the multi-tenant cloud and design an interference-aware custom scheduler to be deployed on top of a virtualized environment. The technique extracts the system, socket, and core metrics to model the hardware (i.e, IPC, reads, writes) and the applications (i.e. reads, writes). Results showed that the custom approach improves the overall performance of the workloads while achieving a more balanced resource utilization than the default K8s scheduler.
[20] 2021		The authors extend the K8s scheduler to use a QoE-aware container scheduler and a re-scheduler for video streaming applications. They create a machine learning predictor that estimates the subjective Mean Opinion Scores (MOS) for a video session using input resource usage metrics from the worker nodes and the containers. The rescheduling algorithm is triggered in case of QoE degradation in the container due to co-location with other services.
[58] 2021		The authors evaluate the performance interference between CPU-intensive and network-intensive containers, and between multiple network-intensive containers. Results show that containers experience performance degradation by 50% due to the co-located containers.
[111] 2021		The proposed Skynet resource management approach builds a model at runtime to map a target performance-level objective for each application to resources, taking into account multiple resources and changing input load. Skynet decreases performance-level objective violations (7.4x) and increases resource utilization (2x)compared to K8s.
[107] 2019	network	The authors present a network-aware scheduler that deploys a service in the node that minimizes the Round Trip Time (RTT) as long as it has enough bandwidth for the service.
[85] 2020		ElasticFog collects real-time network traffic information and dynamically allocates resources based on this information. It adapts to changes in the state of network traffic over time to minimize latency and maximize the throughput of client requests.
[83] 2021		Polaris scheduler creates a cluster topology graph to make the scheduling aware of the state of the network infrastructure for IoT service on the edge.
[19] 2021		The authors present a network-aware scheduler that selects the node most suitable for an application within a given deadline considering the Round Trip Time (RTT) and the bandwidth of the network.

Continued on next page

Table 5 – continued from previous page

Ref. Year	Fe.	Proposal
[67] 2019	NVF	The authors present a scheduler to perform the optimal placement for service instances into a logical network by selecting the appropriate virtual network functions (VNFs) along with their service graph. The evaluations show that this design is very promising for the 5G slicing with a large-scale orchestration system.
[109] 2020		Megalos is presented as an ETSI NFV compliant architecture that supports the implementation of virtual network scenarios consisting of several networking components (VNFs). The scheduling allocates virtual network functions (VNFs) to the nodes taking into account the network topology to reduce the traffic among nodes. The proposal guarantees the segregation of each virtual LAN traffic.
[108] 2020		The authors design and implement a Service Function Chain (SFC) controller as an extension to the K8s scheduling. The SFC controller optimizes the placement of service chains in Fog environments, specifically tailored for Smart City use cases. The approach reduces the network latency when compared to the default scheduling mechanism.
[54] 2021		The authors implement K8s function scheduler plugins to deploy virtual network functions (VNFs) to nodes according to the results obtained from multiple reliable allocation models. The model is selected depending on user objectives and, new models can be added to the scheduler without restarting the scheduler.
[99] 2021		The authors explore some of the K8s features (i.e. request and limit) for resource scheduling to enhance the availability and the resilience of NFV networks. The scheduling algorithm uses genetic algorithms to profile the cloud-native network functions and allocates the appropriate memory and CPU resources with a trade-off between efficiency and risk. The proposed algorithm improves the coexistence of cloud-native network functions in the same node and reduces the interference effect.

Focusing on container interference, some guidelines are derived in [3] for container schedulers to consider when container orchestration platforms, such as K8s, are deployed on top of a virtualized IaaS layer. The evaluation results show that a suitable combination of virtual machines and containers provides maximum flexibility. However, attention must be paid to the interaction between these two virtualization layers, as inappropriate container scheduling in the virtualized infrastructure can easily lead to unacceptable performance degradation. Table 5 summarizes the contributions of some relevant papers on the topic.

On the other hand, networking is a central part of K8s and the K8s networking model sets predefined rules, i.e., each pod must have its own IP address. Several open-source CNI (Container Networking Interface) plugins are implemented to deploy a K8s production cluster, including Flannel, Weave, Calico, Cilium, or kube-router. They support overlay tunnel options in the form of virtual networks of nodes and logical connections built on the physical network of the cluster.

In [98], the authors compare different CNI plugins depending on the need for intra-host or inter-host pod-to-pod communication. The evaluation results show that there is no single, universal 'best' CNI plugin. Typically, the network topology in K8s clusters is very uniform and the K8s scheduler overlooks this network feature. Nevertheless, the cloud-edge infrastructure is not flat and it is very common that the infrastructure is hierarchically organised and contains many gateway edge devices managing hundreds of IoT devices. In this case, network-aware scheduling in K8s is particularly important, as described by the Polaris scheduler proposal in [83]. Network topology and traffic are also taken into account by the scheduler approaches proposed in [19, 85, 107]. Table 5 presents a brief description of their contributions in the context of network characteristics. In general, the results show that distributed IoT applications significantly improve the service delivery of the default K8s scheduler.

In addition, we found several recent works on NVF and SDN in the reviewed literature that relate to the virtualization network layer [54, 99, 108, 109]. In general, these K8s scheduling proposals overcome the many limitations of NFVs by managing network functions with containers that enable fast and scalable deployment. Note that in an overloaded NF, packet loss is inevitable and the compute resources used by these packets are wasted, making efficient scheduling a challenge. Table 5 sets out the proposals of the related works in the reviewed literature classified under the NVF feature.

Finally, a comparative analysis of the approaches of this subsection is discussed and summarized. The default K8s scheduler is not aware of hardware resource fragmentation and container interference leads to poor isolation of multi-tenant applications. Currently, it is difficult to achieve performance isolation for multi-tenancy on K8s clusters because this requires providing resource isolation and improving the abstraction of applications. Some scheduling mechanisms were introduced with their isolation capabilities in mind, and all of these mechanisms can provide some degree of isolation. Some scheduling strategies in K8s assign tenants to a defined set of cluster nodes so that their workload profiles do not interfere with each other and thus, avoid mutual influences on performance. However, this is only possible if the workload of the tenants is predictable. The K8s scheduler has also been extended with the development of control feedback loops that dynamically adjust the workload in the cluster and ensure the isolation of resources without requiring prior per-workload parameterization. Several of the proposed scheduling approaches reduce inference by integrating rescheduling and load balancing techniques. With respect to the virtualization network layer, based on the evaluated studies, we find that network topology plays an important role in fog/edge computing environments for IoT applications to minimize latency and maximize the throughput of client requests. In addition, some papers addressed the scheduling of flows in an NFV environment to handle the diversity of use cases envisaged by 5G technology. The proposed K8s scheduling techniques address the performance of the deployed NFs by taking into account the workload and resources of the worker nodes.

5.2 Cluster domain

In this taxonomy, the cluster domain is classified into two categories: a) the characteristics of the components and b) the environment. These subdomains are directly related to the following questions:

- What key cluster-related components impact the K8s scheduling?
- What characteristics of the computing paradigm affect the K8s scheduling?

A K8s cluster is composed of a set of nodes that run different software components and different design options may impact the final performance of the K8s scheduler. So, the *component* subdomain comprises design parameters related to the scheduler architecture and multi-cluster or federation scheduler, as well as resource management components closely linked to the scheduler in K8s. The *environment* in the cluster domain refers to the computational technologies where K8s can be deployed. They differ in their design and purpose such as cloud or fog computing. Next, more details are provided.

5.2.1 Components. First, we will describe component issues that affect the K8s scheduler focusing on the scheduler architecture, the coordination of federated clusters, and resource management controllers that impact the scheduling.

The scheduler architecture feature has been analyzed in works such as [110] or [18] for other orchestrators. In K8s, the default scheduler runs as a centralized controller on the master node that schedules only one task at a time [7]. It has a global view of the system which enables it to make better optimization decisions, but it is a single point of failure and suffers from scalability issues. So, custom schedulers extend the K8s scheduler architecture providing schedulers with modular, two-level, or distributed architectures [11, 23, 67, 137].

In [11], the authors improve the K8s monolithic scheduler to a two-level scheduler called KubeSphere. It receives tasks from users and dispatches them to a connected K8s cluster based on custom-defined policy and fairness goals. Results show that KubeSphere improves the fairness among users over the default monolithic K8s

scheduling and reduces the overall average waiting time for all users in a cluster. In [137], the authors propose a two-level asynchronous scheduling model within and between the clouds. Results show that the fair allocation mechanism proposed guarantees users' QoS. In [67], the authors propose a two-level scheduling algorithm for network slicing toward 5G. The high-level *Global Scheduler* deals with the assignment of the infrastructure resources, such as finding appropriate nodes. At the low-level, the *Slice Scheduler* performs the optimal placement for service instances considering the *Best Fit* or the *First Fit* heuristics. The evaluation results performed show that this approach is very promising for 5G slicing with large-scale orchestration systems.

In [23], the authors describe a custom distributed scheduler for K8s based on Multi-Agent System (MAS) platform that delegates the scheduling decision among agents in the processing nodes. The distribution of the scheduling task is achieved by transferring node filtering and node ranking jobs into a Multi-Agent System (MAS) that waits for the notification of the winning candidate. In [46], the authors proposed a multi-agent decentralized dispatch based on deep reinforcement learning for edge-cloud systems. They observed that the scheduling delay of centralized service orchestration is almost 9x than that of decentralized request dispatch. These delays are not trivial for some delay-sensitive service requests.

Several works in the literature focused on the design of a multi-cluster scheduler [14, 38, 50, 105, 117]. In [38], KubeFed is extended with a two-phase placement scheme compatible with the cluster federation. KubeFed is a framework generally used to split workloads on different cloud providers to avoid lock-in. The proposed scheme implements a decentralized control plane that distributes microservices among the federated infrastructure and avoids disruption of application services by providing fault tolerance. Foggy is presented in [105] as an architectural framework for workload orchestration in a multi-tier, highly distributed, heterogeneous, and decentralized cloud computing system. Foggy acts as a matchmaker between infrastructure owner and tenants. In [50], the authors present a deep reinforcement learning-based job scheduler for scheduling independent batch jobs among multiple federated cloud computing clusters adaptively. By directly specifying high-level scheduling targets, the scheduler interacts with the multi-cluster environment and automatically learns scheduling strategies from experience without any prior knowledge. In [117], the authors propose an orchestration platform for multi-cluster applications on multiple geo-distributed K8s clusters. The platform allocates the requested resources to all incoming applications while making efficient use of resources. K8s controllers are designed to automatically place, scale, and burst multi-cluster applications across multiple geo-distributed K8s clusters. In [14], the authors extend the K8s control plane for provisioning compute resources across different public cloud providers and regions. The scheduler processes end-user requests for on-demand video streaming and optimizes the use of computing resources provisioned by green energy resources.

Regarding the control plane, K8s provides controllers that work in coordination with the scheduler, such as load balancers or deployments and replica sets to heal container failures. In [34], the authors present an algorithm for scheduling heterogeneous tasks in clustered systems to ensure that all devices or processors perform the same amount of work in an equal amount of time. The approach configures a dedicated cluster to a particular task and introduces load balancing techniques using task migration. In [84], K8s is used to coordinate parked vehicles running sufficient numbers of task replicas, providing high service availability against possible failure caused by the mobility of the vehicles. In [139], the authors present a redundant placement policy for the deployment of microservice-based applications at the distributed edge based on a stochastic optimization problem. In [140], the authors develop an optimized autoscaler for HTC workloads that schedules the workload and resizes the container cluster based on the resource utilization of complete jobs, the real-time status of the job queue, and the resource initialization time of the cluster manager. In [116], the authors design a dynamic resource scheduler for TensorFlow-based training jobs which runs on top of K8s cluster. A dynamic allocation of resources is performed periodically by checking whether resource utilization in the cluster has achieved a certain threshold. In [101], the authors design a dynamic based-utilization autoscaling system in K8s. More precisely, if an application's usage is beyond 10% of the allocated resources, the container which is running the applications is re-scheduled. The

migration is based on checkpoints to avoid killing the application. In [86], K8s resource metrics and Prometheus custom metrics are collected, calculated, and fetched to the horizontal pod autoscaling. The ongoing experiments show how these metrics affect the final performance.

To sum up, a final discussion based on reviewed research is presented. Scheduler architecture is an important issue in K8s scheduler that some works have paid attention to. We observed that most of the schedulers with a two-level architecture decouples the allocation of resource and placement decision and that this feature has a high impact on cluster scalability [110]. Moreover, multi-cluster schedulers coordinate the configuration of multiple K8s clusters tackling scalability and geo-distributed environments, especially for cloud-fog-edge ecosystems. Most of the papers focus on providing a fair resource allocation while satisfying the requirements of the applications. Availability and energy-aware are also issues related to the cluster federation scheduler that some research studies have considered. Finally, it must be highlighted that many papers in the literature reviewed describe scheduling proposals together with load balancing, migration, or replication techniques to improve resource fairness, resilience, and availability.

Table 6. Cluster domain architecture of some Kubernetes scheduling proposals

Ref. Year	Fe.	Proposal
[24] 2017	cloud	The paper presents a platform that includes a comprehensive monitoring mechanism and a dynamic resource-provisioning operation
[10] 2018	cloud	OASIS is an online algorithm for admitting and scheduling asynchronous training jobs in a machine learning cluster. The number of concurrent workers and the server parameters for each job are dynamically adjusted to maximize the overall utility of all jobs based on their completion times
[67] 2019	cloud	The paper describes a multi-level scheduling solution for mobile network slicing
[66] 2019	cloud	Pigeon is a FaaS framework for private cloud that uses function-level resource scheduling and oversubscription-based static pre-warmed container pool to increase system performance
[137] 2019	cloud	The paper proposes a multi-resource fair scheduling algorithm for independent batch jobs among multiple federated cloud computing
[119] 2019	cloud	The paper presents an autonomic, Adaptive Bin Packing (ABP) algorithm to assign virtual entities to physical devices without the need to set any configuration
[50] 2020	cloud	The paper presents a deep reinforcement learning-based job scheduler for scheduling independent batch jobs among multiple federated cloud computing clusters adaptively
[37] 2020	cloud	The paper optimizes the calculation of the number of pods for Knative, a platform for serverless workloads, based on Double Exponential Smoothing
[3] 2020	cloud	The paper investigates how containers are distributed on top of virtual machines when a complex application refactored as microservices is deployed in a private cloud environment, in which containers are distributed on top of VMs
[20] 2021	cloud	The paper proposes a scheduler extension and resource rescheduling that incorporates the quality of experience (QoE) metric proposed in the ITU P.1203 standard into Service Level Objectives (SLOs). A predictor based on machine learning estimates the QoE offered by the cloud
[17] 2021	cloud	The paper presents a self-adaptive K8s cloud controller for scheduling time-sensitive applications
[105] 2017	fog	Foggy is an architectural framework and a software platform for workload orchestration and resource negotiation in a multi-tier and distributed system
[55] 2018	fog	DYSCO can run and reschedule stateless applications providing failure recovery in case of a node failure

Continued on next page

Table 6 – continued from previous page

Ref. Year	Fe.	Proposal
[23] 2019		The paper proposes a custom scheduler for K8s orchestrator that distributes the scheduling task among the processing nodes by means of a Multi-Agent System (MAS)
[107] 2019		The paper proposes a network-aware scheduling approach for container-based applications in Smart City
[85] 2020		ElasticFog runs on top of K8s and collects network traffic information in real-time to allocate computational resources proportionally to the distribution of network traffic
[104] 2020		FScaler is a reinforcement learning agent to scale and schedule containers based on defined cost functions
[131] 2018	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
[88] 2019		The paper proposes a K8s compatible scheduler that integrates real-time information about edge infrastructure components enabling dynamic and advanced orchestration and management
[56] 2020		KEIDS is a K8s-Based Energy and Interference Driven Scheduler for container management in Industrial IoT applications
[84] 2020		The paper proposes a heuristic algorithm based on min-max to task offloading problems at the edge
[36] 2021		The paper proposes a framework for deploying an edge cloud of IoT applications based on the K8s orchestrator and microservice manager
[132] 2021		KubeHICE extends K8s to orchestrate heterogeneous-ISA architectures and provide performance-aware scheduling in the edge
[46] 2021		KaiS designs a two-timescale scheduling mechanism that integrates a coordinated multi-agent algorithm to serve requests within the edge cluster and a service orchestration based on graph neural networks that gather system state
[118] 2022		The paper proposes a Joint Data Access and Task Processing (JDATP) algorithm to minimize the task response time of collaborative edge and cloud computing systems that includes data access latency and task processing latency

5.2.2 Environment. First, we will describe the considered computational architectures considered in this domain, that is, the cloud, fog, and edge paradigms. Then, we will provide a brief overview and comparative analysis of the different K8s scheduling approaches categorized by the environment. The centralized cloud computing paradigm provides on-demand resource allocation through the Internet based on a pay-as-you-go model [69]. Edge/Fog computing is assumed as a distributed system that makes storage and computing resources closer to the place where data are generated, as compared to the cloud computing-based approach. This is mainly driven by the data explosion caused by the wide adoption of the Internet of Things (IoT) [16, 82]. In particular, Fog computing [15] makes use of IoT gateways and other computing devices within the edge network, such as smart routers, SBCs, personal computers, and even micro-data centers to process IoT data. Edge computing also supports the execution of applications near the data sources but using the IoT devices and the gateways of IoT devices to process data. A related term that also introduces storage and processing power at or near the edge of the network is *Multiaccess Edge Computing* (MEC). MEC servers are owned by mobile operators and are located near the base stations so that they can be accessed over the Radio Access Network (RAN). They are mainly targeted at the support of mobile services. Edge, fog, and MEC are relatively new computing paradigms, and their evolution processes are still ongoing. So, many researchers and industries adopt different approaches. A more detailed explanation, which is out of the scope of this paper, is provided in [68]. Some K8s scheduling proposals, classified by the environment domain (*Fe.*), are described in Table 6.

According to reviewed studies, we observed that K8s scheduler has to be aware of the characteristics of the computational paradigm where is deployed to efficiently orchestrate containerized applications and provide efficient resource usage and low response time. Some reviewed papers proposed an efficient distribution of load in cloud-fog environments using the computational power of edge devices by offloading. For example, in [84], the authors provide a flexible and extended container-based architecture for online task offloading problems by leveraging the powerful onboard computing resources of parked vehicles. K8s coordinates parked vehicles to run sufficient numbers of task replicas, providing high service availability against possible failure caused by the mobility of parked vehicles. Moreover, the fair allocation mechanisms and scheduling approaches proposed for cloud computing are not always efficiently applicable in edge/fog architectures that have to deal with the new challenges of the environment, such as low computational capacity resources spread across different geographical locations [43]. In general, it can be claimed that scheduling in edge/fog computing architectures requires fully dynamic, network-aware, and fault tolerance solutions. But, having the computational resources close to the data generated by IoT devices has several benefits, such as low round-trip latency, minimizing the use of public network bandwidth, and most importantly, providing accelerated analysis and better user experience. In the reviewed literature, numerous approaches enhance the functionality of the K8s scheduler to overcome the mentioned challenges. Also, some recent proposals built K8s schedulers for edge-native applications on 5G.

5.3 Scheduling Domain

The scheduling domain is classified based on the mathematical technique used by the scheduler and on the metrics considered by the algorithm as input parameters to be optimized. The subcategories match the following questions.

- What type of mathematical model is used to select the worker node?
- What metrics are used in the design of the K8s scheduling?

Table 7 summarizes the main characteristics of the proposed solutions for K8s related to the scheduling domain.

5.3.1 Scheduling algorithms. They refer to the mathematical computation performed by the scheduler module that matches a task into a node. The scheduling problem is a NP-hard problem meaning that there is no polynomial complexity algorithm to find an optimal schedule for large-size problems. This work categorized scheduling algorithms into mathematical modeling, heuristic, meta-heuristic, machine learning and gang scheduling. These categories differ in computational cost and performance. An exhaustive survey on container-based scheduling techniques is presented in [2].

The heuristic approaches, that are widely used in traditional scheduling, are generally of low complexity, and generate a satisfactory schedule in a reasonable amount of time. Some of the most well-known heuristic strategies are the first-fit [67], First-In-First-Out (FIFO) [105], and Dominant Resource Fairness (DRF) [11, 44]. Besides, some mathematical modeling techniques analyze the scheduling problem as a set of constrained equations, such as the integer linear programming (ILP) problem, and then use standard optimization techniques to find the best solution to the problem. ILP methods present a high computational cost and they can only be used for small size problems [56, 84]. In the container-based resource scheduling problems, there are also some sophisticated proposals based on meta-heuristics including for example genetic, particle swarm, or ant colony algorithms [65]. Meta-heuristics are a popular class of population-based optimization algorithms inspired by the intelligent processes and behaviors arising in nature [52]. Two important characteristics of such algorithms are a selection of the fittest and adaptation to the environment. In general, metaheuristic algorithms can be divided into evolutionary algorithms (EAs), such as Genetic Algorithm (GA), or swarm intelligence algorithms, such as Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), and Whale Optimization among others [77]. Machine learning algorithms are considered in this classification. These techniques have the availability of learning and training from big data to make intelligent scheduling decisions based on what they have learned.

Table 7. Classifying Kubernetes scheduling proposals in the Scheduling Domain

Ref.	Alg.	Metrics						
		System Resource				App. perf.		Other metrics
		CPU	RAM	Bw	GPU	Res. time	Com. time	
[24]	heuristic	✓	✓	✗	✗	✓	✗	
[10]		✓	✓	✓	✓	✗	✓	job's utility and a price function
[67]		✓	✓	✗	✗	✗	✗	(best-fit, first-fit)
[35]		✗	✗	✗	✓	✓	✗	
[105]		✓	✓	✓	✗	✗	✗	(FCFS) disk, location, access rights
[84]		✓	✓	✓	✗	✗	✗	
[23]		✗	✗	✗	✗	✗	✗	not available
[107]		✓	✗	✓	✗	✗	✗	
[96]		✓	✓	✗	✗	✗	✗	network latency
[85]		✓	✗	✓	✗	✗	✗	location
[51]		✓	✗	✗	✓	✗	✗	
[114]		✗	✗	✗	✓	✗	✗	access cost and mean scheduling time
[42]		✗	✗	✗	✗	✗	✓	contention on workers, short-lived app
[102]		✓	✓	✗	✗	✓	✗	
[124]		✗	✓	✗	✗	✗	✗	(binpack and spread) Intel Software Guard Extensions (SGX) metrics, Enclave Page Cache (EPC) size
[108]		✗	✗	✓	✗	✗	✗	location
[55]		✗	✗	✗	✗	✓	✗	status changes in nodes, topology and network
[121]		✓	✓	✗	✗	✗	✗	latency and pod related system events
[11]		✓	✓	✗	✗	✗	✗	(DRF)
[137]		✓	✓	✗	✗	✗	✓	(WDFR)
[119]		✓	✓	✗	✗	✗	✗	(Adaptive Bin Packing (ABP)) disk storage utilization
[56]	ILP	✗	✗	✗	✗	✗	✗	(Mosek solver) interference, energy consumption
[84]		✓	✓	✓	✗	✗	✗	energy prices
[74]		✓	✓	✗	✗	✗	✗	disk and user's need
[88]		✓	✓	✓	✗	✗	✓	network access
[13]		✓	✓	✗	✗	✗	✓	(weighted OASIS scheduler) job's utility
[106]		✗	✗	✗	✗	✗	✗	latency, energy
[127]	predic.	✗	✗	✗	✗	✓	✗	
[122]		✓	✓	✓	✗	✗	✗	disk, thermal environmental, power consumption, and internally logged server metrics
[20]	ML	✓	✓	✓	✗	✗	✗	disk, file system
[104]		✓	✓	✗	✗	✗	✗	disk, user's demands
[133]		✗	✗	✗	✗	✗	✗	historical CPU and RAM usage, its prediction
[9]		✓	✓	✗	✗	✗	✗	interference, jobs' resource demands
[50]		✓	✓	✗	✗	✗	✓	user information of app
[90]		✗	✗	✗	✓	✗	✗	offline and online training

Deep machine learning is a subfield of machine learning that uses a layered structure of artificial neural networks to learn and make intelligent decisions on its own and is also applied in K8s scheduling [9, 50, 133]. For example, a deep learning-driven scheduler for deep learning clusters is presented in [90]. The proposed scheduling algorithm

streamlines overall job completion with efficient resource utilization. The proposal includes both offline supervised learning and live feedback through reinforcement learning.

Related to K8s, many scheduling algorithms have been proposed that fit in the categories mentioned above. Table 7 categorizes some proposals attending to this classification. At this point, it must be highlighted that the K8s *default scheduler* and all the previous proposals manage unbound pods following a sequential process. However, some applications will get better performance using gang scheduling techniques that operate considering bundled tasks to be scheduled. Note that several tasks can be part of the same application. Hence, the orchestrator logic can obtain a more optimal deployment of the application if it is aware of the communication and relationship that exists between the tasks. For example, deep learning training applications can potentially profit from gang scheduling, because they are resource-intensive applications with strong dependency and high inter-task communication. Designing a dependency-aware task gang scheduling will focus on reducing communication overhead and avoiding wasted resources because a task cannot start computing before all its peer tasks are launched. In the reviewed literature some proposals focus on gang scheduling techniques, such as in [13, 64]. In particular, DRAGON is presented in [13] as an extended controller for a gang scheduling in which all the tasks of a training job are scheduled as a single unit in K8s. In [13], the author combines DRAGON's and OASIS' approaches to make a scheduler with weighted autoscaling capabilities and schedule its jobs with gang scheduling. Note that OASIS is a custom scheduler on K8s that calculates the payoff of each job in the queue using the job's utility and a price function [10].

5.3.2 Metrics considered in Scheduling. The scheduling algorithms can also be classified attending to the optimization metrics. In the literature reviewed the scheduling algorithms are aware of different contextual metrics related to the infrastructure, the cluster, and the application domains. Metrics such as resource utilization (computation, storage, network), failure rate, interference, or energy are related to the infrastructure context. Then, location, mobility, stability, reliability, and availability are metrics related to the cluster domain. Finally, application requirements are usually measured in terms of response time, completion time, makespan, or resource demand (see Subsecion 5.4 for more details). These metrics allow the scheduler to be application aware.

Table 7 shows a sample of the more relevant metrics included in some of the papers reviewed in this work. The parameters most widely used by the reviewed scheduling algorithms are related to CPU and RAM utilization [9, 11, 13, 20, 121]. For example, the authors propose in [13] a dynamic resource scheduler that prioritizes jobs with higher resource requirements and tries to schedule higher priority jobs first to maximize resource utilization and reduce the need to call scale function. In [11], the authors develop a scheduling layer on top of K8s' monolithic scheduler to improve resource fairness based on Dominant Resource Fairness (DRF), user CPU and memory demand. Moreover, some efforts considered the network traffic as an important metric [19, 85, 107]. In [107] and [106], the authors address the orchestration of applications in a smart city by developing a network-aware scheduling approach. The *default scheduler* of K8s is extended to consider network infrastructure status for taking resource provisioning decisions.

On top of the K8s platform, *ElasticFog* proposed in [85] collects network traffic information in real-time and enables real-time elastic resource provisioning for containerized applications in Fog computing.

Further, we observe that several research studies on this topic have considered application requirements or user demand (see Res. time (*Response time*) and Com. time (*Completion time*) metrics in Table 7) but are less common than system context-aware metrics. Scheduling proposals that, for example, explicitly consider the completion time of the task are presented in [42, 88, 137]. Likewise, storage is a metric related to the infrastructure context, but few reviewed papers focus on it. For example, in [128] the authors present a storage service orchestration with container elasticity. Security and carbon footprint are parameters also considered in recent works.

Overall, the metrics used by the scheduling algorithms are intimately related to the monitoring module that reports the state of the system. For example, in [42], the authors propose a container placement scheme that

balances the resource contention on the worker nodes combining resource usage with expected completion time. A suite of algorithms monitors the progress of running containers and the collected data are utilized to calculate their expected completion time. In [133], the authors propose a mechanism of re-scheduling the pod on the worker nodes by predicting the utilization rate of application resources. A combination model of grey system theory with long short-term memory neural network prediction is proposed. In [37], the authors optimize the prediction of the number of pods for *Knative*. *Knative* is a popular Kubernetes-based platform for managing serverless workloads. Note that serverless is an event-driven computing model that abstracts the operational logic from the user providing fine granularity [63]. The predictive model is based on *double exponential smoothing*. In short, in the literature there are scheduling algorithms that rely on the information obtained at runtime of the system, and other algorithms that extend this set of metrics to include the expected near-future behavior of the system. That is, the scheduler system selects a worker node based not only on the current resource demand, but also on the estimate or prediction of the future resource demand. To this end, scheduling algorithms usually include predictive or profiling techniques for application behavior or resource utilization.

5.4 Application Domain

In general, the characteristics of the applications that will run in Kubernetes-controlled clusters affect scheduling, as each application has different QoS requirements that must be met [73]. In the literature reviewed, many of the scheduling proposals take into account the characteristics of the applications to be developed [19, 55, 84, 107], but other techniques are presented in a generic way [23, 34]. The application domain is classified into three subdomains according to a) the type of task or application, b) the communication architecture of the task and c) the workload generated by a set of tasks.

The subdomains match with the following questions:

- What type of applications are managed by the K8s scheduler?
- What application architectures are managed by K8s scheduler?
- How does the workload applied to the system affect the scheduling of K8s?

5.4.1 Type. The K8s default scheduling features are not designed specifically for a particular type of application. The scheduler handles a wide range of applications from High-Performance Computing (HPC), machine learning, batch, web server, IoT, or serverless applications. In the literature reviewed, we found some generic scheduling proposals, for all types of applications, but most proposals optimize their design for a specific type of application. For example, IoT applications impose specific challenges to be considered by the scheduler, such as stringent latency, capacity constraints, or uninterrupted services with intermittent connectivity. Below we will briefly mention some proposals, but Table 8 shows more examples. For instance, a scheduler for big data and training neural network models is proposed in [42], A K8s compatible scheduler for 5G edge computing applications is developed in [88], SpeCon [70] is proposed as a novel container scheduler optimized for deep learning applications, also a green energy scheduler for on-demand video streaming workloads is presented in [14], and the problem of running HPC workloads efficiently on K8s clusters is addressed in [75]. Moreover, several proposals support the Function as a Service (FaaS) paradigm in which the user executes a short-lived code without caring about the server used. In fact, Kubeless [95], OpenFaas [45], OpenWhisk [33] or Fission [76] are open-source serverless computing frameworks designed to run on top of K8s [95]. They have different architectures but all of them take responsibility for managing, scaling, and providing different resources to ensure the correct execution of functions triggered by events. The detailed serverless architecture, which is out of the scope of this paper, is provided [112]. Scheduling proposals for Kubernetes-based serverless are proposed in [37, 66]. So, Knative's open-source framework for Kubernetes-based serverless is enhanced in [37] by calculating the optimal number of pods to deploy. In [66], the authors present Pigeon, a FaaS framework for private cloud. Pigeon uses function-level resource scheduling so that the FaaS function can be directly dispatched to pre-warmed containers,

reducing limitations imposed by K8s and increasing system performance. Another novelty of this framework is the introduction of an oversubscription-based static pre-warmed container pool. This approach eliminates the necessity of dynamically creating containers during burst function trigger situations and improves system dynamics.

Based on reviewed research studies, we found that some application-aware scheduling algorithms abstract application characteristics by including metrics such as duration (completion time) or resource requirements, see Table 7. So, applications can be classified as long-lived or short-lived applications depending on their *duration*. As a rule, user-facing applications and web servers are long-lived applications, while batch and serverless applications are usually short-lived applications. For example, in serverless a function or service normally has a maximum time limit of up to fifteen minutes. On the one hand, *demand* refers to the type of resource an application or a task consume. In general, orchestrators must be aware of the hardware requirements of the applications to ensure their operation and also provide efficient resource management. For example, an application will crash and finally will be killed if it does not get the minimum memory capacity it needs. Note that memory is a non-compressible hardware requirement while the CPU requirement is a compressible resource and oversubscriptions are supported. Some reviewed papers are provided with application specification parameters in advance and the scheduler can use them to improve the scheduling. Other papers tackle the problem as soon as they are detected using re-scheduling and autoscaling.

5.4.2 Architecture. The architecture in the application domain is divided into independent and dependent tasks. A container can just run a single task, which is the simplest application case, or more than one task as a monolithic application. Note that K8s provides affinity and anti-affinity properties with pod metadata such as *nodeAffinity* and *podAffinity*. A user can define these parameters to run a pod on a node (or not) depending on where other pods are running.

Today most containerized applications make use of the microservice approach, in which multiple dependent services must be allocated in the worker nodes having into account the communication pattern. So, in [139], the authors design a redundant placement policy for the deployment of microservice-based applications with sequential combinatorial structure at the distributed edge.

In [129], the authors propose *NetMARKS* as a K8s scheduler that is aware of inter-pod dependencies and uses dynamic network metrics collected with Istio Service Mesh. They automatically discover inter-application relations to ensure efficient placement of Service Function Chains (SFCs). The authors' analysis shows that the proposal reduces application response time by up to 37 percent and saves up to 50 percent of bandwidth in a fully automated manner. The dependent function embedding problem at the serverless edge is studied in [30]. The authors design an algorithm to get the optimal edge server for each function. The algorithm represents a function as a directed acyclic graph (DAG) and tries to minimize its completion time. Nevertheless, current state-of-the-art research does not sufficiently consider the composite property of services.

5.4.3 Workload. The workload determined by the set of different applications to be run on the system poses extra challenges to the K8s orchestrator. For example, the end-user experience of user-facing service might be degraded due to sharing resources with batch jobs. Production K8s clusters typically run each type of workload on different nodes to avoid this performance interference, which leads to under-utilization. Low resource utilization of K8s cluster results in high operational cost and substantial resource wastage.

Running different types of applications on the same resource can effectively improve resource efficiency. For example, the authors in [138] extend K8s mechanisms to schedule best-effort jobs based on the real server utilization and adaptively allocate resources between best-effort batch jobs and latency-sensitive services. The evaluation results demonstrate that the average CPU utilization increases from 15% to 60% without service-level objectives (SLO) violations.

K8s may suffer from low resource utilization due to services being over-provisioned for the peak load, and the isolation between different K8s containers is limited [58, 72, 123, 138]. The impact of isolation and interference applications has been analyzed in the literature reviewed, such as we have detailed in Section 5.1.

5.5 Performance Domain

The performance domain analyses the maturity of K8s scheduling by considering the frameworks designed to evaluate the proposals and also the metrics to evaluate the performance. So, this domain is related to the following questions:

- What are the frameworks designed to evaluate the K8s scheduling?
- What are the performance metrics utilized to evaluate K8s scheduling approaches?

5.5.1 Framework. The framework developed to evaluate the K8s scheduler proposal is an important aspect of the performance domain. More precisely, the level of abstraction of the framework is a critical aspect of the evaluation results. Simulation tools for K8s schedulers tend to abstract aspects that are not relevant for their evaluation, obtaining high scalability in terms of the number of worker nodes or providing multi-clusters environments[50]. However, special attention must be paid to the degree of abstraction so that the results obtained reflect the behavior of the real system. Testbed deployments solve this problem but are usually small-scale clusters [24, 88, 133] .

Table 8 summarizes the frameworks used in some of the reviewed papers along with some other relevant properties of the performance domain, such as the platforms assessed (labeled as *System*) or the applications deployed in the framework (labeled as *App.*). While the system is related to the cluster domain described in Section 5.2, the application is connected to the application domain detailed in Section 5.4. Note that in the performance evaluation domain, we find out real applications such as IoT, web servers, or deep learning applications but also synthetic loads or traces that extract the main characteristics of real applications.

From the literature reviewed, we note that many evaluation analyses make use of simulation techniques to devise how efficient the scheduling approaches are. In all the papers reviewed, the simulator is a custom design without standard rules. Besides, we observe that most frameworks design a small testbed consisting of several physical servers (see Table 8). In addition, some proposals are developed in virtual servers. For example, the authors make use of Google Cloud Kubernetes Engine (GKE) in [13] to deploy the testbed. These frameworks address the problem of scalability but lack full control of the system.

5.5.2 Metrics. In the literature, all the proposals utilize some metrics to quantify the performance of the proposed solution. In general, the evaluation metrics are related to the objective to be optimized by the scheduling algorithms. In some papers, performance evaluation focuses on a single metric, but most examine several metrics. The more metrics used in the performance evaluation, the more complex the decision-making process will be. Therefore, different trade-offs are usually established to balance the efficiency of the proposed scheduling algorithm and the final quality of the generated solution. As a representative example, Table 8 gathers the metrics used in some of the works reviewed in the literature. From the literature reviewed, we observed that completion time is the most commonly used metric for evaluating the effectiveness of scheduling algorithms. Similarly, makespan and cost are the second and third most frequently used parameters Performance analyses that include energy consumption are less frequent.

To sum up, the metrics that evaluate the scheduler performance are related to the QoS provided to the end-user and, the cost from the resource provider's point of view. System quality of service (QoS) is measured by parameters such as total processing time or makespan and completion time, while CPU and memory resource utilization is related to system cost.

Table 8. Performance domain characteristics of some Kubernetes scheduling proposals

Ref.	System	Design & Evaluation			
		App.	Framework	Metrics	Results
[11]	cluster	synthetic	testbed (4 nodes)	number of tasks, waiting time	Resource fairness is improved over default scheduling using DRF, user resource demand and their combination
[34]			simulation (3 clusters)	makespan, throughput and response time metrics	Processing time is reduced compared to the round-robin algorithm
[50]	multi-cluster		simulation	makespan, resource utilization	The average utilization of each cluster is quite stable, the makespan sometimes increases
[56]	fog	traces .	simulation (4 clusters)	CPU utilization, energy , average interference	Energy utilization is improved, CFR reduced, and interference minimized by almost 14.42%, 47%, and 31.83%, respectively, in contrast to the conventional FCFS scheduler
[23]	multi-cluster	μbenchmark	testbed (4 clusters+ 1 coordi.)	resource utilization, completion time	Users' QoS of each CSP
[137]			testbed (VMs)	average waiting time, resource utilization	Two-level asynchronous scheduling model within and between the clouds, and introduce high priority queues in the cloud to prioritize the tasks sent by the federation scheduler
[24]	cluster	web app	testbed (1 node)	CPU and memory utilization ratios, executing pods	Evaluation of the monitoring and dynamic resource-provisioning proposals under HTTP load (using Jmeter)
[88]			testbed (5 Rpis)	time of scheduling, cluster temperature	Points of improvement regard the speed and efficiency of the scheduling process in case of stressful deployments
[133]			testbed (3 servers)	resource fragmentation, CPU and memory utilization, resource fragmentation	The proposal reduces resource fragmentation
[10]	cluster	DL	simulation and testbed	completion time of a job, concurrent workers in a training	OASIS outperforms common schedulers (FIFO, DRF, in real-world cloud systems
[13]			testbed (GKE-3 nodes)	training speed, model accuracy	The makespan increases up to 26.56% due to the autoscaling and gang scheduling, and up to 6.23% due to the weighting algorithm in comparison with the default scheduler
[42]			testbed (NSF Cloudlab)	completion time, makespan	Comparing with default scheduler, the proposal achieves up to 23.0%, reduction of completion time and an improvement of makespan for up to 37.4%
[114]	GPU cluster		testbed (5 servers)	mean time	Increase 10% resource utilization
[127]			testbed (4 nodes)	makespan, completion time	Compared to the termination-based scheduler in Optimus, the framework reduces the makespan and the average job completion time by up to 20% and 37%, respectively.

Continued on next page

Table 8 – continued from previous page

Ref.	System	App.	Framework	Metrics	Results
[67]	cluster	μservice	testbed (OpenStack 8 nodes)	deployment time of a slice, CPU and memory state	Evaluation shows that the approach is promising for 5*G* slicing with a large-scale orchestration system
[55]	fog	IoT	testbed (3 nodes, Rpis and sensors)	time rescheduling	Efficient solution to reschedule stateless applications and to recovery from failure
[107]			testbed (IDLab, Belgium)	average RTT, scheduler times	They achieve a reduction of 80% in terms of network latency compared to default K8s
[84]	MEC		simulation	task average acceptance ratios, average costs and average accumulated utility/rewards	The proposal extends the computation resources by taking advantage of powerful on-board computing resources of PVs and brings a flexible architecture to task offloading problems

6 FUTURE DIRECTION AND OPEN ISSUES

Having presented a literature review and a taxonomy on K8s scheduling, we find that there are still some open key issues worth for further discussion to improve resource management in K8s. In this section, we discuss some future research directions to address the remaining challenges.

6.1 Challenges related to infrastructure domain

6.1.1 New types of physical devices. Artificial Intelligence (AI) at the edge is now common in Fog/edge computing. AI accelerators such as Graphics Processing Unit (GPU), Field-Programmable Gate Arrays (FPGA) or Tensor Processing Units (TPU) are powerful machine learning hardware chips specifically designed to run AI and ML applications. In reviewing the literature, we found some recent work that integrates GPUs into the K8s scheduler, but no work that focuses on other accelerators. Thus, efficient resource management of AI accelerators in K8s has not yet been sufficiently explored.

6.1.2 Enhance resource virtualization techniques. Several papers analyze the impact of the so-called noisy neighbor on job scheduling. The obtained results show that the interference between virtualized resources has a great impact on the final performance. Therefore, there is a need to improve the isolation of the virtualization layer and model its behavior in order to improve container scheduling in K8s. For example, a container should never cause the node's operating system to fail. Otherwise, all the tenants running containers in the node will see their performance degraded, as well as future scheduled containers. Further research in this area is highly recommended.

With respect to network virtualization techniques, virtualized network functions must be scheduled to optimize the deployment of services. Therefore, to provide performance guarantees, it is necessary to avoid nodes with very low availability or over-loaded, and to identify bottleneck links. Algorithms and machine learning techniques can be a good solution to achieve better results.

6.2 Challenges related to cluster domain

6.2.1 Distributed control plane. The K8s scheduling architecture is a centralized system based on the master-slave philosophy. Thus, an error in the master node could result in a system failure. To overcome this problem, K8s

uses multiple master nodes and implements robust protocols [32]. Therefore, K8s scheduling should take this fact into account to support a fault-tolerant scheduler system and improve availability and resilience. Focusing on this issue, the use of distributed and coordinated scheduling techniques in K8s requires further study.

Moreover, the centralized nature of the K8s orchestration system does not align well with the needs of the distributed fog/Edge computing environments. Therefore, modifying K8s to remove centralized control is an area in need of exploration. In this context, Blockchain technology [31] has overcome its initial application in cryptocurrencies to reach over other application domains, which harness its immutable, transparent, and available information storage capability. When combined with smart contracts it can provide a distributed computer where all the nodes independently and equally contribute to a common global system state, which must be agreed upon by consensus. The integration of blockchain in task scheduling is a recent area to explore. A few steps are already taken [61, 87], however, it is still in its infancy stage.

6.2.2 Collaboration among clusters. Seamless integration and cooperation among cluster orchestrators is a core requirement for creating a federated and scalable framework. Federated frameworks extend the resilience of the system and achieve smart services. For example, if the service that controls the path of a vehicle cannot migrate to the nearest cluster orchestrator, it may be difficult to get a smart route in time. K8s scheduling should provide this collaborative feature to improve both high availability and resiliency. K8s programming must also address the pricing model for leasing resources, as well as for deciding when to outsource or in-source containers. Hence, the use of distributed and coordinated scheduling techniques in K8s requires further study.

6.2.3 Inter-operability. K8s is always framed in a layered architecture. The lower layers make use of container virtualization techniques where there are some standard efforts but, this is not the case for the upper layers. K8s schedulers need to operate with cloud providers and the communication APIs are not well defined. Hence, standard APIs are required.

6.3 Challenges related to scheduling domain

6.3.1 Scalability. K8s orchestration must consider aspect related to scalability and adaptation of the system. Most scheduling proposals analyze the quality of service parameters, such as completion time or makespan and extend K8s' functionality to work in For/Edge architectures. An important aspect of successfully running applications in these environments is supporting resource management on dynamically scalable hardware architectures. The demand for this new approach to successfully support scalable scheduling is reinforced by the requirements of mobile vehicle applications. Sophisticated and efficient re-scheduling and autoscaling approaches are required to support cluster scalability.

6.3.2 Advance context-aware scheduling algorithm. Scheduling algorithms are usually based on input metrics related to resource utilization, such as CPU, RAM, disk, and network. Considering other important input parameters is a must for scheduling algorithms. For example, sustainable scheduling techniques should be explored in depth to reduce energy consumption and ensure a lower carbon footprint. New K8s scheduling algorithms should help prevent environmental degradation without compromising the user experience. In this context, smart monitoring systems are becoming a challenge to enhance scheduling algorithms.

Moreover, scheduling should not only be considered from the static infrastructure point of view. Thus, algorithms should consider availability and reliability as first-order metrics. These metrics can be very relevant in edge environments where devices are not always on. Consumer-centric IoT applications and services, such as connected vehicles or the smart grid, are examples of trending applications that should benefit from these scheduling approaches. Enriching K8s to allow replication and automatic service redirection to the healthy entities has been explored in [125]. However, there are very few contributions in the literature on scheduling

techniques aware of the dynamic nature of the system. The dynamic nature of the resources is a critical feature to consider in new scheduling algorithms.

To sum up, to overcome the above-mentioned challenges, new filtering and affinity ranking models are required.

In addition, it can be useful to model the mobility pattern of end devices, cluster resource utilization, and application requirements. Scheduling algorithms face the challenge of enhancing their efficiency by integrating these models. In this regard, a challenging task to explore is the development of prediction techniques and runtime estimation techniques to obtain the required models.

6.4 Challenges related to application domain

6.4.1 Workflows and microservices. By default, the K8s scheduler considers one pod at a time, without considering the relationship between pods belonging to the same application. Now, with the challenge of deploying microservices-based applications, getting a global optimal scheduling solution for containerized workflows is a major issue. The development of novel Gang scheduling techniques can bring significant improvements.

6.4.2 Model workloads. Understanding the characteristics and patterns of workloads running on K8s clusters is a critical task for improving K8s scheduling. The dynamic nature of the workload suggests that predictive techniques can improve Kubernetes' scheduling techniques. Nowadays, workload prediction is still an open research topic. Moreover, in the existing literature very narrow discussion towards providing a well-established benchmark of representative applications has been provided. It is also an open research topic.

6.5 Challenges related to performance domain

6.5.1 Comparability and usability. There are many proposals related to job scheduling, but they are isolated proposals with little interaction between them, in the sense that there is no continuous improvement. For example, at the level of scheduling algorithms in K8s, new proposals are often compared to simple scheduling algorithms such as round-robin or the default-scheduler. Providing continuous integration of research would foster further progress in resource scheduling.

On the one hand, simulators specifically designed to evaluate scheduling proposals are used. In the works studied that evaluate proposals using simulation tools, there is a lack of an open-source tool that acts as a *de facto* standard. The fact that the evaluation environments are developed in an ad-hoc manner does not allow for efficient and effective analysis of the different proposals.

As for evaluations in non-simulated environments, small test environments with 3-5 physical servers are usually used. It would be desirable to have a real-world platform where new ideas can be developed and tested incrementally (i.e., adding more features or selecting the appropriate ones with the ability to compare them to other proposals).

6.5.2 Security issues. Security is one of the main concerns of distributed systems. None of the reviewed scheduling techniques developed for K8s are aware of this fact. Scheduling should take into account the behavior of system devices to reduce vulnerabilities in the system. Smart scheduling should refuse to deploy containers on nodes with suspicious behavior that pose a threat to the system, such as when DDoS attacks are detected.

7 CONCLUSIONS

Container orchestrations automate the deployment, management, scaling, interconnection, and availability of container-based applications. K8s is the open-source standard *de facto* orchestrator.

In this survey paper, we investigated the state-of-the-art of K8s functionality and reviewed related orchestration systems. The literature reviewed on K8s scheduling was studied to identify and classify the research efforts and present a taxonomy. Finally, possible future research directions to improve the orchestration in K8s clusters are identified.

ACKNOWLEDGMENTS

The Spanish Ministry of Science and Innovation (ref. RTI2018-098156-B-C52) supported this work.

REFERENCES

[1] M. Adhikari, T. Amgoth, and S.N. Srirama. 2019. A Survey on Scheduling Strategies for Workflows in Cloud Environment and Emerging Trends. *Comput. Surveys* 52, 4, Article 68 (Aug. 2019), 36 pages.
[2] I. Ahmad, M. Gh. AlFailakawi, A. AlMutawa, and L. Alsalman. 2021. Container scheduling techniques: A Survey and assessment. *Journal of King Saud University - Computer and Information Sciences* (2021).
[3] G. Ambrosino, G. B. Fioccola, R. Canonico, and G. Ventre. 2020. Container Mapping and its Impact on Performance in Containerized Cloud Environments. In *2020 IEEE Int. Conf. on Service Oriented Systems Engineering (SOSE)*.
[4] AR. Arunarani, D. Manjula, and Vijayan Sugumaran. 2019. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems* 91 (2019), 407–415.
[5] The Docker authors. 2022. Empowering App Development for Developer; Docker. https://www.docker.com/
[6] The Docker authors. 2022. Swarm mode overview; Docker Documentation. https://docs.docker.com/engine/swarm/
[7] The Kubernetes authors. 2022. Kubernetes: Production-Grade Container Orchestration. http://kubernetes.io/
[8] J. Bader, L. Thamsen, S. Kulagina, J. Will, H. Meyerhenke, and O. Kao. 2021. Tarema: Adaptive Resource Allocation for Scalable Scientific Workflows in Heterogeneous Clusters. In *2021 IEEE Int. Conf. on Big Data*. IEEE, 65–75.
[9] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep Learning-based Job Placement in Distributed Machine Learning Clusters. In *IEEE INFOCOM 2019 - IEEE Conf. on Computer Communications*. 505–513.
[10] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. 2018. Online Job Scheduling in Distributed Machine Learning Clusters. In *IEEE INFOCOM 2018 - IEEE Conf. on Computer Communications*.
[11] Angel Beltre, Pankaj Saha, and Madhusudhan Govindaraju. 2019. KubeSphere: An Approach to Multi-Tenant Fair Scheduling for Kubernetes Clusters. In *2019 IEEE Cloud Summit*. IEEE, 14–20.
[12] Ouafa Bentaleb, Adam SZ Belloum, Abderrazak Sebaa, and Aouaouche El-Maouhab. 2022. Containerization technologies: Taxonomies, applications and challenges. *The Journal of Supercomputing* 78, 1 (2022), 1144–1181.
[13] M. Bestari, A. Kistijantoro, and A. Sasmita. 2020. Dynamic Resource Scheduler for Distributed Deep Learning Training in Kubernetes. In *2020 7th Int. Conf. on Advance Informatics: Concepts, Theory and Applications (ICAICTA)*. 1–6.
[14] Yahav Biran, Sudeep Pasricha, George Collins, and Joel Dubow. 2016. Enabling green content distribution network by cloud orchestration. In *2016 3rd Smart Cloud Networks Systems (SCNS)*. 1–8.
[15] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. Association for Computing Machinery, New York, NY, USA.
[16] E. Borgia. 2014. The Internet of Things vision: Key features, applications and open issues. *Computer Communications* 54 (2014), 1 – 31.
[17] L. Bulej, T. Bureš, P. Hnětynka, and D. Khalyeyev. 2021. Self-adaptive K8S Cloud Controller for Time-sensitive Applications. In *2021 47th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*. 166–169.
[18] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59 (04 2016), 50–57.
[19] A. C. Caminero and R. Muñoz-Mansilla. 2021. Quality of Service Provision in Fog Computing: Network-Aware Scheduling of Containers. *Sensors* 21, 12 (2021), 3978.
[20] M. Carvalho and D. F. Macedo. 2021. QoE-Aware Container Scheduler for Co-located Cloud Environments. In *2021 IFIP/IEEE Int. Symposium on Integrated Network Management (IM)*. 286–294.
[21] E. Casalicchio. 2019. *Container Orchestration: A Survey*. Springer Int. Publishing, Cham, 221–235.
[22] E. Casalicchio and S. Iannucci. 2020. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience* 32, 17 (2020), e5668.
[23] O. Casquero, A. Armentia, I. Sarachaga, and et al. 2019. Distributed scheduling in Kubernetes based on MAS for Fog-in-the-loop applications. In *24th IEEE Int. Conf. on Emerging Techn. and Factory Automation (ETFA)*. 1213–1217.
[24] C. Chang, S. Yang, E. Yeh, P. Lin, and J. Jeng. 2017. A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conf.* 1–6.
[25] Saurabh Chhajed. 2015. *Learning ELK stack*. Packt.
[26] The Google cloud authors. 2021. Container Registry. https://cloud.google.com/container-registry/
[27] CNCF. 2022. Cloud native computing foundation charter. Web page. https://www.cncf.io/about/charter/
[28] M. Coté. 2020. Why Large Organizations Trust Kubernetes. Web page. https://tanzu.vmware.com/content/blog/why-large-organizations-trust-kubernetes
[29] C. Cérin, T. Menouer, W. Saad, and W. Abdallah. 2017. A New Docker Swarm Scheduling Strategy. In *2017 IEEE 7th Int. Symposium on Cloud and Service Computing (SC2)*. 112–117.

[30] S. Deng, H. Zhao, Z. Xiang, and et al. 2022. Dependent Function Embedding for Distributed Serverless Edge Computing. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (2022), 2346–2357.
[31] M. Di Pierro. 2017. What Is the Blockchain? *Computing in Science Engineering* 19, 5 (2017), 92–95.
[32] G. M. Diouf, H. Elbiaze, and W. Jaafar. 2020. On Byzantine fault tolerance in multi-master Kubernetes clusters. *Future Generation Computer Systems* 109 (2020), 407–419.
[33] K. Djemame, M. Parker, and D. Datsev. 2020. Open-source Serverless Architectures: an Evaluation of Apache OpenWhisk. In *2020 IEEE/ACM 13th Int. Conf. on Utility and Cloud Computing (UCC)*. 329–335.
[34] A. Dua, S. Randive, A. Agarwal, and N. Kumar. 2020. Efficient Load balancing to serve Heterogeneous Requests in Clustered Systems using Kubernetes. In *2020 IEEE 17th Annual Consumer Commun. Networking Conf. (CCNC)*. 1–2.
[35] G. El Haj Ahmed, F. Gil-Castiñeira, and E. Costa-Montenegro. 2021. KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters. *Software: Practice and Experience* 51, 2 (2021), 213–234.
[36] D. Ermolenko, C. Kilicheva, A. Muthanna, and A. Khakimov. 2021. Internet of Things Services Orchestration Framework Based on Kubernetes and Edge Computing. In *2021 IEEE Conf. of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. 12–17.
[37] D. Fan and D. He. 2020. Knative Autoscaler Optimize Based on Double Exponential Smoothing. In *2020 IEEE 5th Information Technology and Mechatronics Engineering Conf. (ITOEC)*. 614–617.
[38] F. Faticanti, D. Santoro, S. Cretti, and D. Siracusa. 2021. An Application of Kubernetes Cluster Federation in Fog Computing. In *2021 24th Conf. on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 89–91.
[39] R. Fayos-Jordan, S. Felici-Castell, J. Segura-Garcia, J. Lopez-Ballester, and M. Cobos. 2020. Performance comparison of container orchestration platforms with low cost devices in the fog, assisting Internet of Things applications. *Journal of Network and Computer Applications* 169 (2020), 102788.
[40] The Apache Software Foundation. 2022. Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[41] The Linux Foundation. 2022. Open Container Initiative - OCI. https://opencontainers.org/
[42] Y. Fu, S. Zhang, J. Terrero, and et al. 2019. Progress-based Container Scheduling for Short-lived Applications in a Kubernetes Cluster. In *2019 IEEE Int. Conf. on Big Data (Big Data)*. 278–287.
[43] M. Ghobaei-Arani, A. Souri, and A. Rahmanian. 2020. Resource management approaches in fog computing: a comprehensive review. *Journal of Grid Computing* 18, 1 (2020), 1–42.
[44] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symp.Networked Systems Design and Implementation (NSDI 11)*.
[45] H. Govind and H. González–Vélez. 2021. Benchmarking Serverless Workloads on Kubernetes. In *2021 IEEE/ACM 21st Int. Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 704–712.
[46] Y. Han, S. Shen, X. Wang, S. Wang, and V. Leung. 2021. Tailored Learning-Based Scheduling for Kubernetes-Oriented Edge-Cloud System. In *IEEE INFOCOM 2021 - IEEE Conf. on Computer Communications*. 1–10.
[47] I. Harichane, A. Makhlouf, and G. Belalem. 2020. A Proposal of Kubernetes Scheduler Using Machine-Learning on CPU/GPU Cluster. In *Intelligent Algo in Soft. Engineering*, Radek Silhavy (Ed.). Springer Int. Publishing, 567–580.
[48] B. Hindman, A. Konwinski, M. Zaharia, and et al. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation (NSDI'11)*. 295–308.
[49] Cheol-Ho Hong and Blesson Varghese. 2019. Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms. *ACM Computing Surveys (CSUR)* 52, 5 (2019), 1–37. https://doi.org/10.1145/3326066
[50] J. Huang, C. Xiao, and W. Wu. 2020. RLSK: A Job Scheduler for Federated Kubernetes Clusters based on Reinforcement Learning. In *2020 IEEE Int. Conf. on Cloud Engineering (IC2E)*. 116–123.
[51] S. Huaxin, X. Gu, K. Ping, and H. Hongyu. 2020. An Improved Kubernetes Scheduling Algorithm for Deep Learning Platform. In *2020 17th Int. Comp. Conf. Wavelet Active Media Tech. and Information Processing (ICCWAMTIP)*. 113–116.
[52] K. Hussain, M. N. Mohd Salleh, S. Cheng, and Y. Shi. 2019. Metaheuristic research: a comprehensive survey. *Artificial Intelligence Review* 52, 4 (2019), 2191–2233.
[53] K8. 2022. sig-scheduling blob. , 53 pages. https://bit.ly/3jbwx5O
[54] R. Kang, M. Zhu, F. He, T. Sato, and E. Oki. 2021. Design of Scheduler Plugins for Reliable Function Allocation in Kubernetes. In *2021 17th Int. Conf. on the Design of Reliable Communication Networks (DRCN)*. 1–3.
[55] F. Katenbrink, A. Seitz, L. Mittermeier, H. Müller, and B. Bruegge. 2018. Dynamic Scheduling for Seamless Computing. In *2018 IEEE 8th Int. Symposium on Cloud and Service Computing (SC2)*. 41–48.
[56] K. Kaur, S. Garg, G. Kaddoum, and et al. 2020. KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem. *IEEE Internet of Things Journal* 7, 5 (May 2020), 4228–4237.
[57] Paridhika Kayal. 2020. Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope : Invited Paper. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. 1–6.

[58] E. Kim, K. Lee, and C. Yoo. 2021. On the Resource Management of Kubernetes. In *2021 Int. Conf. on Information Networking (ICOIN)*. 154–158.
[59] Kitchenham. 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. , 53 pages. https://bit.ly/3t40kAY
[60] M. Kumar, S.C. Sharma, A. Goel, and S.P. Singh. 2019. A comprehensive survey for scheduling techniques in cloud computing. *Journal of Network and Computer Applications* 143 (2019), 1–33.
[61] W. Li, S. Cao, K. Hu, J. Cao, and R. Buyya. 2021. Blockchain-Enhanced Fair Task Scheduling for Cloud-Fog-Edge Coordination Environments: Model and Algorithm. *Security and Communication Networks* 2021 (2021).
[62] X. Li, Y. Jiang, Y. Ding, D. Wei, X. Ma, and W. Li. 2020. Application Research of Docker Based on Mesos Application Container Cluster. In *2020 Int. Conf. on Computer Vision, Image and Deep Learning (CVIDL)*. 476–479.
[63] Zijun Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo. 2021. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Survey* (dec 2021).
[64] Chan-Yi Lin, Ting-An Yeh, and Jerry Chou. 2019. DRAGON: A Dynamic Scheduling and Scaling Controller for Managing Distributed Deep Learning Jobs in Kubernetes Cluster.. In *CLOSER*. 569–577.
[65] Miao Lin, Jianqing Xi, Weihua Bai, and Jiayin Wu. 2019. Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Microservice Scheduling in Cloud. *IEEE Access* PP (06 2019), 1–1.
[66] W. Ling, L. Ma, C. Tian, and Z. Hu. 2019. Pigeon: A Dynamic and Efficient Serverless and FaaS Framework for Private Cloud. In *2019 Int. Conf. on Computational Science and Computational Intelligence (CSCI)*. 1416–1421.
[67] D. Luong, A. Outtagarts, and Y. Ghamri-Doudane. 2019. Multi-level Resource Scheduling for network slicing toward 5G. In *2019 10th Int. Conf. on Networks of the Future (NoF)*. 25–31.
[68] R. Mahmud, K. Ramamohanarao, and R. Buyya. 2020. Application Management in Fog Computing Environments: A Taxonomy, Review and Future Directions. *Comput. Surveys* 53, 4, Article 88 (July 2020), 43 pages.
[69] S. Manvi and G. Krishna Shyam. 2014. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *Journal of Network and Computer Applications* 41 (2014), 424–440.
[70] Y. Mao, Y. Fu, W. Zheng, L. Cheng, Q. Liu, and D. Tao. 2021. Speculative Container Scheduling for Deep Learning Applications in a Kubernetes Cluster. *IEEE Systems Journal* (2021), 1–12.
[71] Khaled Matrouk and Kholoud Alatoun. 2021. Scheduling algorithms in fog computing: A survey. *Int. Journal of Networked and Distributed Computing* 9, 1 (2021), 59–74.
[72] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. 2016. Adaptive Application Scheduling under Interference in Kubernetes. In *2016 IEEE/ACM 9th Int. Conf. on Utility and Cloud Computing (UCC)*. 426–427.
[73] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. A. Bañares, and O. F. Rana. 2017. Client-Side Scheduling Based on Application Characterization on Kubernetes. In *Economics of Grids, Clouds, Systems, and Services*. 162–176.
[74] Tarek Menouer. 2021. KCSS: Kubernetes container scheduling strategy. *The Journal of Supercomputing* 77 (2021), 4267–4293.
[75] C. Misale, M. Drocco, D. J Milroy, C. Gutierrez, and et al. 2021. It's a Scheduling Affair: PolarisACS in the Cloud with the KubeFlux Scheduler. In *3rd Int. WS on Containers and New Orchestration Paradigms for Isolated Env. in HPC*. 10–16.
[76] S. Mohanty, G. Premsankar, and M. di Francesco. 2018. An Evaluation of Open Source Serverless Computing Frameworks. In *2018 IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*. 115–120.
[77] Myat Myat Mon and May Aye Khine. 2019. Scheduling and load balancing in cloud-fog computing using swarm optimization techniques: A survey. In *Seventeenth Int. Conf. on Computer Applications (ICCA 2019)*. 8–14.
[78] S. Mondal, R. Pan, HM Kabir, T. Tian, and H. Dai. 2022. Kubernetes in IT administration and serverless computing: An empirical study and research challenges. *The Journal of Supercomputing* 78, 2 (2022), 2937–2987.
[79] L. Monteiro, W. H. Almeida, R. Hazin, and et al. 2018. A Survey on Microservice Security–Trends in Architecture, Privacy and Standardization on Cloud Computing Environments. *Int. Journal on Advances in Security* 11, 3-4 (2018).
[80] R. Morabito. 2016. A performance evaluation of container technologies on Internet of Things devices. In *2016 IEEE Conf. on Computer Communications Workshops (INFOCOM WKSHPS)*. 999–1000.
[81] M. Moravcik and M. Kontsek. 2020. Overview of Docker container orchestration tools. In *2020 18th Int. Conf. on Emerging eLearning Technologies and Applications (ICETA)*. 475–480.
[82] A. Musaddiq, Y. Zikria, O. Hahm, H. Yu, A. Bashir, and S. Kim. 2018. A survey on resource management in IoT operating systems. *IEEE Access* 6 (2018), 8459–8482.
[83] S. Nastic, T. Pusztai, A. Morichetta, V. Pujol, S. Dustdar, D. Vii, and Y. Xiong. 2021. Polaris Scheduler: Edge Sensitive and SLO Aware Workload Scheduling in Cloud-Edge-IoT Clusters. In *IEEE 14th Int. Conf. on Cloud Computing*. 206–216.
[84] K. Nguyen, S. Drew, C. Huang, and J. Zhou. 2020. Collaborative Container-based Parked Vehicle Edge Computing Framework for Online Task Offloading. In *2020 IEEE 9th Int. Conf. on Cloud Networking (CloudNet)*. 1–6.
[85] N.D. Nguyen, L. Phan, D. Park, S. Kim, and T. Kim. 2020. ElasticFog: Elastic Resource Provisioning in Container-Based Fog Computing. *IEEE Access* 8 (2020), 183879–183890.

[86] T. Nguyen, Y. Yeom, T. Kim, D. Park, and S. Kim. 2020. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. *Sensors* 20, 16 (2020).
[87] C. Núñez-Gómez, B. Caminero, and C. Carrión. 2021. HIDRA: A Distributed Blockchain-Based Architecture for Fog/Edge Computing Environments. *IEEE Access* 9 (2021), 75231–75251.
[88] M. C. Ogbuachi, C. Gore, A. Reale, and et al. 2019. Context-aware K8S scheduler for real time distributed 5G edge computing applications. In *2019 Int. Conf. on Software, Telecom. and Computer Networks (SoftCOM)*. 1–6.
[89] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. 2019. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* 7, 3 (2019), 677–692.
[90] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin. 2021. DL2: A Deep Learning-Driven Scheduler for Deep Learning Clusters. *IEEE Transactions on Parallel and Distributed Systems* 32, 8 (2021), 1947–1960.
[91] C. Perera, Y. Qin, J. Estrella, S. Reiff-Marganiec, and A. Vasilakos. 2017. Fog Computing for Sustainable Smart Cities: A Survey. *Comput. Surveys* 50, 3, Article 32 (June 2017), 43 pages.
[92] R. P. Prado, S. G. Galán, J. Expósito, A. Marchewka, and N. Ruiz-Reyes. 2020. Smart Containers Schedulers for Microservices Provision in Cloud-Fog-IoT Networks. Challenges and Opportunities. *Sensors* 20, 6 (2020).
[93] 2022 K3s project authors. 2022. Lightweight Kubernetes: The certified Kubernetes distribution built for IoT and Edge computing. Web page. https://k3s.io/
[94] KubeEdge project authors. 2021. A Kubernetes Native Edge Computing Framework. Web page. https://kubeedge.io/en/
[95] © Kubeless 2022 project authors. 2022. The Kubernetes Native Serverless Framework. Web page. https://kubeless.io/
[96] T. Pusztai, F. Rossi, and S. Dustdar. 2021. Pogonip: Scheduling Asynchronous Applications on the Edge. In *2021 IEEE 14th Int. Conf. on Cloud Computing (CLOUD)*. 660–670.
[97] S. Qi, S. Kulkarni, and K. Ramakrishnan. 2021. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 656–671.
[98] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan. 2020. Understanding Container Network Interface Plugins: Design Considerations and Performance. In *2020 IEEE Int. Symp. on Local and Metropolitan Area Networks (LANMAN*. 1–6.
[99] M. Rahali, C. Phan, and G. Rubino. 2021. KRS: Kubernetes Resource Scheduler for resilient NFV networks. In *GLOBECOM 2021 - IEEE Global Communications Conf.* IEEE, Madrid, Spain, 1–6.
[100] Aaqib Rashid and Amit Chaturvedi. 2019. Virtualization and its role in Cloud Computing environment. *Int. Journal of Computer Sciences and Engineering* 7, 4 (2019), 1131–1136.
[101] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari. 2019. Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes. In *2019 IEEE 12th Int. Conf. on Cloud Computing (CLOUD)*. 33–40.
[102] I. Rocha, C. Göttel, P. Felber, and et al. 2019. Heats: Heterogeneity-and Energy-Aware Task-Based Scheduling. In *2019 27th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*. 400–405.
[103] M. A. Rodriguez and R. Buyya. 2019. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience* 49, 5 (2019), 698–719.
[104] H. Sami, A. Mourad, H. Otrok, and J. Bentahar. 2020. FScaler: Automatic Resource Scaling of Containers in Fog Clusters Using Reinforcement Learning. In *2020 Int. Wireless Communications and Mobile Computing (IWCMC)*. 1824–1829.
[105] D. Santoro, D. Zozin, D. Pizzolli, F. De Pellegrini, and S. Cretti. 2017. Foggy: A Platform for Workload Orchestration in a Fog Computing Environment. In *2017 IEEE Int. Conf. on Cloud Comp. Technology and Science (CloudCom)*. 231–234.
[106] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. 2019. Resource Provisioning in Fog Computing: From Theory to Practice. *Sensors* 19, 10 (2019).
[107] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. 2019. Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications. In *2019 IEEE Conf. on Network Softwarization (NetSoft)*. 351–359.
[108] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. 2020. Towards delay-aware container-based Service Function Chaining in Fog Computing. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 1–9.
[109] M. Scazzariello, L. Ariemma, G. Battista, and M. Patrignani. 2020. Megalos: A Scalable Architecture for the Virtualization of Network Scenarios. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symp.* 1–7.
[110] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proc. of the 8th ACM European Conf. on Computer Systems (EuroSys '13)*. 351–364.
[111] Y. Sfakianakis, M. Marazakis, and A. Bilas. 2021. Skynet: Performance-driven Resource Management for Dynamic Workloads. In *2021 IEEE 14th Int. Conf. on Cloud Computing (CLOUD)*. 527–539.
[112] M. Shahrad, J. Balkind, and D. Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proc. 52nd Annual IEEE/ACM Int. Symposium on Microarchitecture (MICRO '52)*. 1063–1075.
[113] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th Int. Middleware Conf. (Middleware '16)*. New York, NY, USA, Article 1, 13 pages.

[114] S. Song, L. Deng, J. Gong, and H. Luo. 2018. Gaia Scheduler: A Kubernetes-Based Scheduler Framework. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Comp. Commun., Big Data Cloud Comp., Social Comp. Networking, Sustainable Comp. Communications*. 252–259.
[115] P. Stanojevic, S. Usorac, and N. Stanojev. 2021. Container manager for multiple container runtimes. In *2021 44th Int. Convention on Information, Communication and Electronic Technology (MIPRO)*. 991–994.
[116] R. Y. Surya and A. Imam Kistijantoro. 2019. Dynamic Resource Allocation for Distributed TensorFlow Training in Kubernetes Cluster. In *2019 Int. Conf. on Data and Software Engineering (ICoDSE)*. 1–6.
[117] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth. 2021. mck8s: An orchestration platform for geo-distributed multi-cluster environments. In *2021 Int. Conf. on Computer Communications and Networks (ICCCN)*. 1–10.
[118] J. Tang, M. M. Jalalzai, C. Feng, Z. Xiong, and Y. Zhang. 2022. Latency-Aware Task Scheduling in Software-Defined Edge and Cloud Computing with Erasure-Coded Storage Systems. *IEEE Transactions on Cloud Computing* (2022), 1–1.
[119] Asser N Tantawi and Malgorzata Steinder. 2019. Autonomic Cloud Placement of Mixed Workload: An Adaptive Bin Packing Algorithm. In *2019 IEEE Int. Conf. on Autonomic Computing (ICAC)*. 187–193.
[120] P. Thinakaran, J. R. Gunasekaran, B. Sharma, and et al. 2019. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. In *2019 IEEE Int. Conf. on Cluster Comp.(CLUSTER)*. 1–13.
[121] L. Toka. 2021. Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes. *Journal of Grid Computing* 19, 3 (2021), 1–23.
[122] P. Townend, S. Clement, D. Burdett, and et al. 2019. Invited Paper: Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes. In *2019 IEEE Int. Conf. Service-Oriented System Engineering (SOSE)*. 156–15610.
[123] A. Tzenetopoulos, D. Masouros, S. Xydis, and D. Soudris. 2020. Interference-Aware Orchestration in Kubernetes. *ISC High Performance 2020. Lecture Notes in Computer Science* 12321 (2020).
[124] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni, and C. Fetzer. 2018. SGX-Aware Container Orchestration for Heterogeneous Clusters. In *2018 IEEE 38th Int. Conf. on Distributed Computing Systems (ICDCS)*. 730–741.
[125] L. Vayghan, M. Saied, M. Toeroe, and F. Khendek. 2019. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th Int. Conf. Soft. Quality, Reliability and Security*. 176–185.
[126] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conf. on Computer Systems (EuroSys '15)*. Article 18, 17 pages.
[127] S. Wang, O. J. Gonzalez, X. Zhou, and et al. 2020. An Efficient and Non-Intrusive GPU Scheduling Framework for Deep Learning Training Systems. In *SC20: Int. Conf. for High Performance Comput., Net., Storage and Analysis*. 1–13.
[128] A. Warke, M. Mohamed, R. Engel, H. Ludwig, W. Sawdon, and L. Liu. 2018. Storage Service Orchestration with Container Elasticity. In *2018 IEEE 4th Int. Conf. on Collaboration and Internet Computing (CIC)*. 283–292.
[129] Ł. Wojciechowski, K. Opasiak, J. Latusek, and et al. 2021. NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh. In *IEEE Conf. on Computer Communications (INFOCOM-2021)*. 1–9.
[130] X. XIE and S. S. Govardhan. 2020. A Service Mesh-Based Load Balancing and Task Scheduling System for Deep Learning Applications. In *2020 20th IEEE/ACM Int. Symp. on Cluster, Cloud and Internet Computing (CCGRID)*. 843–849.
[131] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. 2018. Extend Cloud to Edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 373–377.
[132] S. Yang, Y. Ren, J. Zhang, J. Guan, and B. Li. 2021. KubeHICE: Performance-aware Container Orchestration on Heterogeneous-ISA Architectures in Cloud-Edge Platforms. In *2021 IEEE Int. Conf on Parallel, Distributed Processing with Apps, Big Data, Cloud Comput., Sustainable Comput., Communications, Social Computing, Net.* 81–91.
[133] Y. Yang and L. Chen. 2019. Design of Kubernetes Scheduling Strategy Based on LSTM and Grey Model. In *2019 IEEE 14th Int. Conf. on Intelligent Systems and Knowledge Engineering (ISKE)*. 701–707.
[134] Onur Yilmaz. 2021. Extending the Kubernetes API. In *Extending Kubernetes*. Springer, 99–141.
[135] A. Yousefpour, C. Fung, T. Nguyen, and et al. 2019. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal Systems Architecture* 98 (2019), 289 – 330.
[136] Dongjin Yu, Yike Jin, Yuqun Zhang, and Xi Zheng. 2018. A survey on security issues in services communication of Microservices-enabled fog applications. *Concurrency and Computation: Practice and Experience* 0, 0 (2018), e4436.
[137] G. Zhang, R. Lu, and W. Wu. 2019. Multi-Resource Fair Allocation for Cloud Federation. In *2019 IEEE 21st Int. Conf. on HP Comp. and Comm.; 17th Int. Conf. on Smart City; 5th Int. Conf. on Data Science and Systems*. 2189–2194.
[138] X. Zhang, L. Li, Y. Wang, E. Chen, and L. Shou. 2021. Zeus: Improving Resource Efficiency via Workload Colocation for Massive Kubernetes Clusters. *IEEE Access* 9 (2021), 105192–105204.
[139] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar. 2020. Distributed Redundancy Scheduling for Microservice-based Applications at the Edge. *IEEE Transactions on Services Computing* (2020), 1–1.
[140] C. Zheng, N. Kremer-Herman, T. Shaffer, and D. Thain. 2020. Autoscaling High-Throughput Workloads on Container Orchestrators. In *2020 IEEE Int. Conf. on Cluster Computing (CLUSTER)*. 142–152.