Università degli Studi di Catania

Dottorato di ricerca in Ingegneria dei Sistemi
Energetica, Informatica e delle Telecomunicazioni

Angelo Marchese

Orchestrating applications on the Cloud-to-Edge continuum

Supervisor: Prof. Orazio Tomarchio

XXXVI Cycle

Contents

1 The Cloud-to-Edge continuum **9**

1.1 Cloud computing 10

1.2 Edge computing 15

1.3 Orchestration of Cloud-Edge architectures 17

2 Kubernetes **21**

2.1 Design principles 21

2.2 Kubernetes architecture 22

2.3 Kubernetes objects 25

2.4 Kubernetes scheduler 28

2.5 Kubernetes descheduler 32

2.6 Kubernetes controllers and operators 34

2.7 Kubernetes in the Edge 36

3 Motivation and state of the art **39**

3.1 Kubernetes infrastructure and application models 39

3.2 Kubernetes limitations 44

3.3 Kubernetes extension proposals 47

4 Sophos framework **51**

4.1 Overall design 51

4.2 Cluster monitoring operator 56

4.3 Application topology modeling 59

4.4 Application monitoring operator 62

4.5 Custom scheduler 66

4.6 Custom descheduler 72

5 Evaluation **74**

5.1 Application and test bed environment 74

5.2 Experiments and results 80

List of Figures

1.1 Cloud computing architecture 10
1.2 Map of Microsoft Azure data centers 12
1.3 Edge computing architecture 14

2.1 Kubernetes architecture 23
2.2 Kubernetes scheduler 29
2.3 Kubernetes operator 35

3.1 Kubernetes infrastructure model 40
3.2 Kubernetes application model 42

4.1 Overall architecture 55
4.2 Cluster monitoring operator 57
4.3 Relationship types hierarchy 61
4.4 Application monitoring operator 63

5.1 Sock Shop application 75
5.2 Sock Shop application TOSCA service template 76
5.3 Test bed environment 78
5.4 Experiments results (Scenario 1) 80

5.5 Experiments results (Scenario 2) 81
5.6 Experiments results (Scenario 3) 81
5.7 Experiments results (Scenario 4) 82
5.8 Experiments results (Scenario 5) 82

Abstract

Orchestrating modern distributed microservices applications presents challenges due to the increasing complexity of these applications and their time-sensitive requirements. The combination of Cloud and Edge Computing paradigms attempts to avoid their pitfalls while taking the best of both worlds: cloud scalability and compute closer to the edge where data is typically generated. However, placing microservices in such heterogeneous environments while meeting QoS constraints is a challenging task due to the geo-distribution of nodes and varying computational resources. In particular, Edge infrastructure is more dynamic and unstable than that of Cloud data centers and is characterized by higher network latencies and more frequent node failures and network partitions. Kubernetes is today the de-facto standard for container orchestration on Cloud data centers. However, its static container scheduling strategy is not suitable for the placement of complex and distributed microservices-based applications on Edge environments. Current infrastructure network conditions and resource availability neither run time application state are taken into account when scheduling microservices. To deal with these limitations in this work the Sophos framework is proposed, as an extension of the Kubernetes platform in order to implement an effective

application and infrastructure-aware container scheduling and orchestration strategy. In particular, an extension of the default Kubernetes scheduler is proposed that considers application and infrastructure telemetry data when taking scheduling decisions. Furthermore, a descheduler component is also proposed that continuously tunes the application placement based on the ever changing application and infrastructure states. An evaluation of the proposed approach is presented by comparing it with the default Kubernetes orchestration and scheduling strategy.

Chapter 1

The Cloud-to-Edge continuum

The Cloud computing paradigm has been as of now the way to go for the execution and management of complex distributed applications. However, Cloud data centers are far away from the network edge and then from end users and devices. This can have a negative impact on the application response time and throughput, critical quality-of-service (QoS) requirements for modern applications. Edge Computing paradigm has emerged as a promising technology for mitigating this problem by moving computation towards the network edge. In this context, both Cloud and Edge infrastructure are combined together to form the Cloud-to-Edge continuum, an environment for executing distributed applications. In this chapter both paradigms are first described. Then some insights on the problem of orchestrating applications on the Cloud-to-Edge continuum are given.

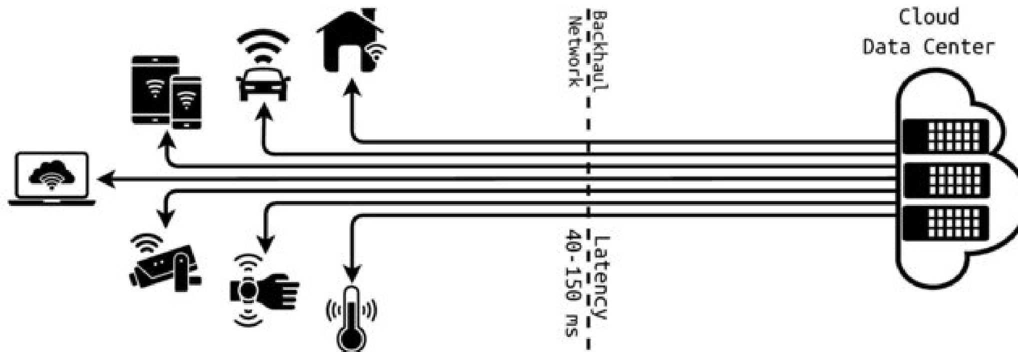

Figure 1.1: Cloud computing architecture

1.1 Cloud computing

Cloud computing offers services from large data centers to end users. It adopts a so-called pay-as-you-go model in which customers acquire services or resources based on their demand and are only charged for their actual usage, which realizes the long-held dream of making computing the fifth utility [1]. Due to economies of scale and multiplexing, since its birth, Cloud computing has been creating substantial monetary values for both Cloud providers and Cloud customers. From the users' perspective, it not only gives them an option to avoid investing in costly proprietary data center infrastructures and their maintenance to reduce the IT expenditure, but also allows them to fully focus on their main business.

Based on the services provided, NIST classified Cloud computing into three categories [2]:

- Infrastructure as a Service (IaaS), which leases computing resources, such as processing, storage, and network.
- Platform as a Service (PaaS), that offers services to host and manage

applications.

- Software as a Service (SaaS), which directly provides applications to the end customers.

As a general purpose computing platform, Clouds can host a broad spectrum of applications, such as scientific simulations, data analytics, and web applications. In the past web applications were mainly hosted in either proprietary infrastructures or rented server rooms. In addition to the high upfront investment cost, these approaches were difficult to scale, leaving applications either over-provisioned or under-provisioned under a dynamic workload. The elasticity feature of Clouds that allows users to acquire/release a virtually unlimited amount of resources dynamically makes Cloud the ideal platform to host web applications whose workload are fluctuant in nature. Due to its advantages compared to traditional solutions, many organizations providing web applications have shifted or been moving their applications to Cloud. In 2019, Flexera surveyed 786 technical professionals across a broad cross-section of organizations and found that 94% of respondents use the Cloud one way or another [3].

As Cloud providers are building more data centers around the world increasingly, it opens the opportunity for web application providers to utilize this globally available Cloud infrastructure to boost the performance of their applications. Deploying web applications in multiple geographically distributed data centers brings the following significant benefits:

- it improves availability of applications even under unexpected data center outages.

Figure 1.2: Map of Microsoft Azure data centers

- it provides balanced and satisfactory QoS to geographically dispersed customers.
- it helps to comply with data regulations.
- it avoids vendor lock-in.
- it enables cost optimization among different vendors.

Cloud servers are typically powerful nodes co-located in huge data centers that are based in different locations across the globe. As an example Figure 1.2 shows a map of the Microsoft Azure global network made up of more than 200 data centers distributed over more than 60 regions[4].

The location and the size of the data centers follows an economical model that decreases the cost of operation by concentrating a big number of nodes, and by targeting locations with lower electricity price. The nodes inside the

data centers are connected via low-latency and high-bandwidth links that handle the ever-rising internal traffic. Nowadays fiber optics technology is making its way to data centers to meet this demand. Thus, the inter-node latency within data centers can be considered negligible.

However, this centralized architecture constrains end users to communicate with the application services over long-distance Internet links. For a wide range of applications, communicating over the core network does not affect the user-perceived quality-of-experience (QoE). For example, storage and web hosting applications can offer decent performance when deployed on a Cloud platform since they can tolerate relatively long network latencies. In contrast, modern latency-sensitive applications require the response time to be lower than a strict threshold, and will perform poorly under such conditions.

The upsurge in the fields of Artificial Intelligence (AI), autonomous vehicles, and most prominently IoT have changed the nature of end users. Rather than having people behind devices transmitting requests to the Cloud, it is estimated that by 2025 55% of all the data will be generated by 21.5 billion IoT devices [5] (approximately 63% of all the connected devices [6]). The shift of end users' nature was combined with an emergence of new set of IoT applications and other applications like stream processing and Virtual Reality (VR). Such applications are demanding in terms of latency and bandwidth. This growth in the number of connected devices and the introduction of new requirements have created new challenges for the Cloud architecture:

- Network round trip time: the round-trip time needed for an end user to access a Cloud application is estimated as 40 ms for a wired connection

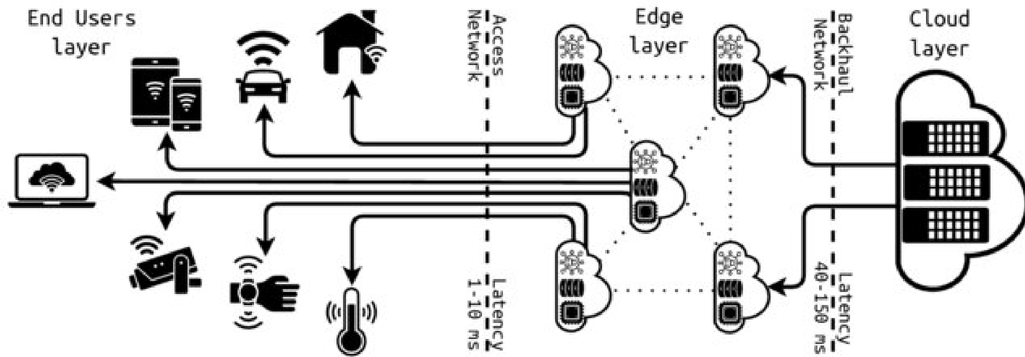

Figure 1.3: Edge computing architecture

and as high as 150 ms for a 4G connection [7]. Meanwhile, applications like VR and gaming can only tolerate end-to-end response times (including network and computation delays) of 20 ms maximum [8]. Such low latency cannot be supported even by 5G where the user-to-Internet latency is reported as 30-40 ms.

- Bandwidth: The majority of IoT devices use very little bandwidth, however the massive number of IoT connected devices can easily congest network links, specifically those of the Wide Area Network (WAN) [9]. Since the number of devices is ever increasing, and the technologies that use them is advancing it is evident that the WAN bandwidth should increase to accommodate them. Yet, the improvements in the field of network technology are slow compared to the growth of Internet traffic produced by the IoT devices.

1.2 Edge computing

Edge computing adds an additional layer to the Cloud by placing computational resources close to data generating devices, like sensors, actuators, or other entities. This new placement strategy aims to reduce latency and improve bandwidth capabilities [10, 11]. In contrast to the Cloud, where all data is transmitted to and stored in a centralized way, Edge computing provides additional resources to take the load from the Cloud [12]. Especially low-latency application fields, like real-time analytics or video surveillance, can benefit from the increased bandwidth and reduced latency realized by Edge computing [13]. Also, the geographically distributed nature of large IoT networks is prone to unstable network connections [14]. The core requirement of Edge computing, minimizing the latency and increasing bandwidth for real-time services, might be violated by long distances between client and servers. The length of the physical distance between clients and servers has a coherence with latency. Edge computing as general technology has different types, also called Edge technologies [15]. Common types are Mobile Edge Computing (MEC), Cloudlet, Micro Data Center (mDC), and Fog. All of these offer different provision models.

MEC was introduced by Nokia and placed computational resources (e.g., computing, network, and storage) mainly next to mobile Radio Access Network (RAN) stations. This type of Edge computing aims to provide low latency network access to low-power devices, often to process time-critical tasks, like real-time analytics. The placement next to mobile RAN stations enables fast and dynamic provisioning of applications, which offer real time services. Accordingly, the provision of applications near the data-generating

devices reduces the traffic sent to the Cloud and reduces network congestion. There is no common understanding if MEC is a substitute for the Cloud. It is unclear whether data must or must not be forwarded to the Cloud in any case.

Cloudlets form virtualized clusters with a set of decentralized devices for running low-latency applications. They are self-managing, fast and easy to deploy by local administrators and aim to provide computational resources close to data-generating devices. Workloads and applications are supposed to be transmitted as VM overlays. The overlays are executed on top of a base image that is already available on the target device. It is necessary to shift these VM overlays rapidly, for example, if the service-requesting devices change their position continuously, for example, in smart traffic control systems. Therefore, it is inevitable that cloudlets have a reliable network connection with a high bandwidth available [16].

Similar to cloudlets, mDCs want to reduce response times by collaborating with the Cloud as an additional layer. mDCs support multi-tenancy and need, therefore, a strong hardware- and software-based protection against unauthorized access. They run in an isolated and secured unit in terms of physical and virtual access. For data exchange with the Cloud, they usually have a reliable, fast, and durable connection.

Fog computing, often also called Edge computing follows similar principles. Although there is a high similarity between Fog computing and the formerly presented Edge technologies, Fog computing can be interpreted as one step closer to data-generating devices and is explicitly designed in a decentralized way. In Fog computing, large-scale networks of heterogeneous

devices cooperate and communicate to realize low latency for the lower layers. It is still an unanswered question if there is a substantial discrimination between Edge and Fog computing. Since Edge and Fog computing share common goals for many devices and act as an additional layer to the Cloud, one can assume both terms can be treated equally.

1.3 Orchestration of Cloud-Edge architectures

The combination of both the Cloud and Edge infrastructures offers a distributed environment for the execution of complex microservices applications. The process of executing and managing the lifecycle of these applications on this environment is called orchestration. Also for Cloud-only environments the problem of application and resource orchestration is a challenging one [17]. Cloud resource orchestration regards complex operations such as selection, deployment, monitoring, and run-time control of resources. The overall goal of orchestration is to guarantee full and seamless delivery of applications by meeting QoS goals of both cloud application owners and Cloud resource providers. Resource orchestration is considered to be a challenging activity because of the scale dimension that resources have reached, and the proliferation of heterogeneous cloud providers offering resources at different levels of the cloud stack. Utilizing the full architecture of Cloud-Edge environments, also called the Cloud-to-Edge continuum, different techniques on how to deploy applications have emerged over time. The most common, so-called provision or orchestration models, are scaling, offloading and Edge-only deployments, even if the Edge layer is not supposed to replace the Cloud

completely. Scaling to the Edge, also called distributed offloading, keeps the application running in the Cloud and Edge layer simultaneously. This provision model applies especially if the Cloud and Edge layers need to work together because both are running out of resources. For example, the Edge node may reach the available amount of storage and the Cloud may violate the latency requirement [18]. Offloading from Cloud to Edge and vice versa is the most frequently discussed and used approach. Applications and tasks are moved to the Edge layer to achieve better response times, e.g., due to periodic high load. Equally, applications may be shifted to the Cloud when the load decreases [19]. Edge-only deployments are also possible, where the applications are placed only on the Edge layer and moved across different Edge nodes, to fulfill the needed latency. Offloading is not necessarily destined for this way of placement. However, many solutions perform a mixed approach even if they focus on Edge placement in specific [20].

Using Cloud-Edge architectures for running a large set of different services comes along with new complexities. These complexities arise because of the additional Edge layer, different Edge technologies, and provision models. Cloud-Edge orchestration is responsible for assigning workloads to the Cloud, Edge, and IoT layer based on a particular set of objectives.

The most important aspect that must be covered is an efficient placement of applications dependent on the origin of potential requests. Also, the required real-time latency and bandwidth of devices must be considered to achieve the claimed application response times. For this, complex decision and orchestration models are required that consider demand and supply of resources like CPU, memory, disk, and network utilization [12]. Fault-tolerance

and resilience is a further core requirement of Cloud-Edge architectures [21]. As already mentioned in the former section, Cloud-Edge systems are used in critical areas, like smart traffic control systems. Outages because of broken nodes in the architecture should not happen. The complexity of managing those architectures is further intensified due to the decentralized, distributed, and large-scale nature of the Edge layer. In addition, a heterogeneous set of devices, often low-power devices, needs to be managed appropriately [22, 19]. Dynamically scaling down and up the number and types of nodes in Cloud-Edge architectures must be supported to realize large-scale deployments in the right way [23]. Also noteworthy are security considerations like supporting multi-tenancy for Edge technologies like cloudlets and mDCs, which are offering a shared model. Security mechanisms like authentication and authorization are strongly required to operate Cloud-Edge systems, especially if they are publicly accessible [22, 21].

The aforementioned provision models present several challenges that must be met. Workloads should be fast and easy to deploy and moved across the layers in the architecture. Meanwhile, container technology is the de facto standard running workloads in Cloud-Edge architectures. Containers follow the principle of lightweight virtualization and contain the application, libraries, and the runtime environment. They are executed in an isolated way by a so-called container engine which restricts the amount and type of resources a particular container can use (i.e., CPU, memory, and storage). The container engine itself runs on an operating system and shares the kernel with the container instances. Container technology has been widely accepted for this area because containers are small in size, have a fast startup time

compared to traditional VMs [24] and can be easily ported to other physical nodes [25, 26]. Today, different container technologies are available, each with its own container runtime implementation, like Docker [1], containerd [2] and Podman [3].

However, the usage of container solutions alone for the execution of complex applications on distributed Cloud-Edge environments can result difficult to be adopted; this justifies the introduction of a higher containerization layer known as container orchestration. Container orchestrator engines (or container orchestrators) automate container provisioning and management including resource scheduling, coordination, and communication across containers, and resource booking and accounting [27, 28]. Currently, different container orchestrators are available like Docker Swarm [4], Apache Mesos [5] and Kubernetes [6], with the last one being the most adopted solution today and that will be described in detail in chapter 2.

[1] `https://www.docker.com`
[2] `https://containerd.io`
[3] `https://podman.io`
[4] `https://docs.docker.com/engine/swarm`
[5] `https://mesos.apache.org`
[6] `https://kubernetes.io`

Chapter 2

Kubernetes

Kubernetes is today the de-facto standard container orchestration platform used for the execution and lifecycle management of Cloud-native applications. Kubernetes was initially thought to be used on Cloud environments, but today its adoption also has been extended for Edge environments also. In this chapter the Kubernetes platform and its components are first described. Then examples of Edge-oriented Kubernetes distributions are provided.

2.1 Design principles

Kubernetes is a container orchestration platform that was initially developed by Google from lessons learned in a decade of running the Borg job manager [13]. It is primarily designed as a distributed and a scalable system to automate the life cycle of container-based applications on a cluster of machines. It provides a set of high-level abstractions to help DevOps easily deploy their applications, automatically scale in/out a particular service based on its load,

perform rolling updates without inducing offline times, do DNS based service discovery, manage application configuration and perform a plethora of other services. Kubernetes is a growing ecosystem with rich functionality thanks to its design principles and its inherent extensibility.

Kubernetes is built around a set of principles that leads to its design and architecture. At the heart are declarative configuration, immutable infrastructure and online self-healing. The central idea behind the declarative configuration is to let the system act on behalf of the user intent: the DevOps describes the desired state of the cluster and Kubernetes sits in an endless loop (reconciliation loop) trying to drive the system to the desired state continually. This involves fetching container images from central repositories, allocating and mounting storage spaces, configuring networks, configuring, starting, scaling and deleting containers, etc. In Kubernetes, system changes are not performed incrementally. Although technically possible, it is considered as an anti-pattern to imperatively mutate the state of a container. Instead, container immutability is preferred by encouraging the operators to rebuild their images and restart their containers.

2.2 Kubernetes architecture

The Kubernetes platform is deployed on top of a cluster. A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In pro-

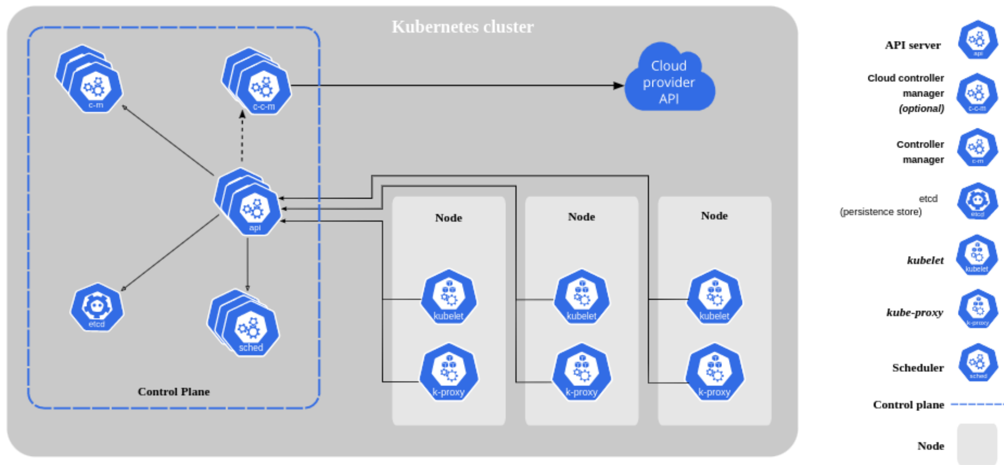

Figure 2.1: Kubernetes architecture

duction environments, the control plane usually runs across multiple nodes and a cluster usually runs multiple worker nodes, providing fault-tolerance and high availability.

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new Pod in case of failure of another Pod). Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. The components that make up the control plane of a Kubernetes cluster are the following:

- kube-apiserver: a server that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane and it is designed to scale horizontally.

- etcd: a consistent and highly-available key value store used as the Kubernetes backing store for all cluster data.
- kube-scheduler: a component that watches for newly created Pods with no assigned node, and selects a node for them to run on.
- kube-controller-manager: a component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.
- cloud-controller-manager: a component that embeds cloud-specific control logic. The cloud controller manager allows to link the cluster into the Cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with the cluster. The cloud-controller-manager only runs controllers that are specific to the Cloud provider. If Kubernetes is run on premises the cluster does not have a cloud controller manager.

Node components run on every node, maintaining running Pods and providing the Kubernetes runtime environment. The node components of a Kubernetes cluster are the following:

- kubelet: an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.
- kube-proxy: a component that maintains network rules on nodes. These network rules allow network communication to Pods from network sessions inside or outside the cluster. Kube-proxy uses the operating sys-

tem packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

- container runtime: the software that is responsible for running containers on a node. Kubernetes supports container runtimes such as Docker, containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

2.3 Kubernetes objects

Kubernetes provides abstractions and high level concepts to manage deployed microservices as generic as possible in order to perform various operations (discovery, scaling, load-balancing, rolling updates, etc.). The following are the main Kubernetes objects:

- Pods: a Pod is the basic element of operations in Kubernetes. Technically speaking, a Pod is a collection of containers that share the same process and network address spaces (virtual network interfaces, ports, IP addresses and shared memory). Processes inside a Pod can communicate with each other using sockets listening on the loop-back interface or with inter-process communication facilities such SysV shared-memory.

- Labels and selectors: labels are key-value pairs used to keep track of objects inside a Kubernetes cluster. All objects in Kubernetes can be labeled. A DevOps can query to find all the objects with a particular label or issue a command that will affect all of those objects in a single

shot. Selectors are the complementary concept that select objects based on a set of given labels.

- Services: a service provides load-balancing and DNS-based naming to applications running on the cluster. There are many types of services. In a ClusterIP service, a static virtual IP address is used as an iptables target. All requests to this target are forwarded to one of the corresponding Pods. The load-balancing is a feature of the netfilter subsystem of the Linux kernel. On the other hand, a NodePort service uses the host IP instead. The service IP and port are provided as environment variables once the container is started and the service name is resolvable using the DNS server provided by Kubernetes (kube-dns).

- ReplicaSets: a ReplicaSet is responsible of maintaining the desired number of replicas of a particular Pod. ReplicaSets are the best example of a reconciliation loop, they keep watching the cluster's state and schedule new Pods or remove existing ones in response to scalability demands or cluster failures.

- ConfigMaps and Secrets: they are key-value objects stored in Kubernetes. ConfigMaps provide configuration information to the workloads. They are an essential part to make Pods reusable components. They can be mounted as a file (keys map to files and values map to files contents) or provided as environment variables. For example, an Apache or a Nginx Pod can be contextualized by providing the server's configuration as a ConfigMap mounted as a file. Secrets are a mean to provide sensitive information to the container like database credentials

at container creation time instead of being stored on the container's file-system image.

- Deployments: Deployment is a higher level abstraction and it is the preferred API object to deploy a micro-service on Kubernetes. Besides managing a ReplicaSet, it also provides features to effectively perform application upgrades. More specifically RollingUpdates that ensures the application Pods are smoothly replaced without inducing an offline time.

- StatefulSets: StatefulSet is the workload API object used to manage stateful applications. Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of its Pods. These Pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

- DaemonSets: A DaemonSet ensures that all (or some) nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

- Namespaces: a Namespace is an abstraction that defines a logical scope for resources. They can be used to implement multi-tenancy and resource isolation between development teams or to create multiple virtual clusters on top of a single infrastructure.

2.4 Kubernetes scheduler

The Kubernetes scheduler, also called *kube-scheduler*[1], is in charge of selecting an optimal cluster node for each Pod to run them on, taking into account different constraints related to Pod requirements and node resources availability. Pods are taken from a scheduling queue and each Pod scheduling attempt is split into two phases: the scheduling cycle and the binding cycle. During the scheduling cycle a node for the Pod to schedule is selected, while during the binding cycle the scheduling decision is applied to the cluster by reserving the necessary resources and deploying the Pod to the selected node. Together, a scheduling cycle and a binding cycle are referred to as a "scheduling context". Scheduling cycles are run serially, while binding cycles may run concurrently. A scheduling or binding cycle can be aborted if the Pod is determined to be unschedulable or if there is an internal error. In this case the Pod will be returned back to the queue and retried.

Both the scheduling cycle and the binding cycle are divided into different phases, also called extension points. Each phase of both cycles is implemented by one or more plugins, which in turn can implement one or more phases to perform more complex or stateful tasks. Some plugins can influence the scheduling decisions while other are informational only.

Figure 2.2 shows the scheduling context of a Pod and the extension points that the scheduling framework exposes. The plugins of the PreEnqueue phase are called prior to adding Pods to the scheduling queue, where Pods are marked as ready for scheduling. Only when all PreEnqueue plugins return "Success", the Pod is allowed to enter the active queue. Otherwise, it's placed

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

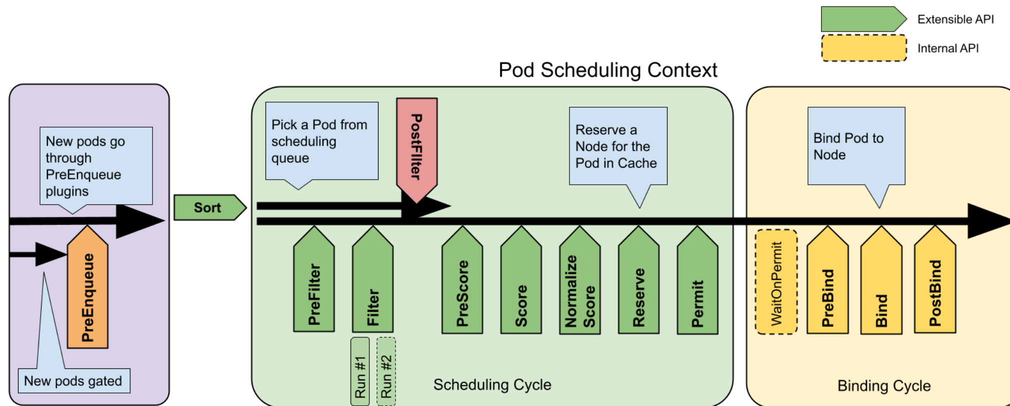

Figure 2.2: Kubernetes scheduler

in the internal unschedulable Pods list, and doesn't get an "Unschedulable" condition. The plugins of the QueueSort phase are used to sort Pods in the scheduling queue. A queue sort plugin essentially provides a *Less(Pod1, Pod2)* function. Only one queue sort plugin may be enabled at a time.

The plugins of the PreFilter phase pre-process info about the Pod, or check certain conditions that the cluster or the Pod must meet. If a PreFilter plugin returns an error, the scheduling cycle is aborted. These plugins are used to filter out nodes that cannot run the Pod. For each node, the scheduler will call filter plugins in their configured order. If any filter plugin marks the node as infeasible, the remaining plugins will not be called for that node. Nodes may be evaluated concurrently. The plugins of the Filter phase filter out nodes that cannot run the Pod. For each node, the scheduler will call filter plugins in their configured order. If any filter plugin marks the node as infeasible, the remaining plugins will not be called for that node. Nodes may be evaluated concurrently. The plugins of the PostFilter phase are called after Filter phase, but only when no feasible nodes were found for the Pod.

Plugins are called in their configured order. If any plugin marks the node as "Schedulable", the remaining plugins will not be called. A typical PostFilter implementation is preemption, which tries to make the Pod schedulable by preempting other Pods.

The plugins of the PreScore phase are used to perform "pre-scoring" work, which generates a sharable state used by the plugins of the Score phase. If a PreScore plugin returns an error, the scheduling cycle is aborted. The plugins of the Score phase rank nodes that have passed the filtering phase. The scheduler will call each scoring plugin for each node. There will be a well defined range of integers representing the minimum and maximum scores. After the NormalizeScore phase, the scheduler will combine node scores from all plugins according to the configured plugin weights. Among the default scoring plugins, the *NodeResourcesFit* plugin checks if a node has sufficient CPU and memory resources to satisfy Pod resource requirements, while the *InterPodAffinity* plugin evaluates inter-Pod affinity rules. Each affinity rule reflects a communication relationship between two Pods and states that those Pods should be placed on the same topology domain. A topology domain consists of a single node or a set of nodes that share the same label, specified by the *topologyKey* parameter of the affinity rule, and that are typically located on the same Cloud provider region or availability zone. The *weight* parameter of an affinity rule represents the priority of that rule. The greater this value, the greater the need to schedule the two microservices near to each other. The plugins of the NormalizeScore phase are used to modify scores before the scheduler computes a final ranking of Nodes. A plugin that registers for this extension point will be called with the score results from

the same plugin. This is called once per plugin per scheduling cycle. If any NormalizeScore plugin returns an error, the scheduling cycle is aborted.

The plugins that implement the Reserve extension have two methods, namely Reserve and Unreserve, that back two informational scheduling phases called Reserve and Unreserve, respectively. Plugins which maintain runtime state should use these phases to be notified by the scheduler when resources on a node are being reserved and unreserved for a given Pod. The Reserve phase happens before the scheduler actually binds a Pod to its designated node. It exists to prevent race conditions while the scheduler waits for the bind to succeed. The Reserve method of each Reserve plugin may succeed or fail. If one Reserve method call fails, subsequent plugins are not executed and the Reserve phase is considered to have failed. If the Reserve method of all plugins succeed, the Reserve phase is considered to be successful and the rest of the scheduling cycle and the binding cycle are executed. The Unreserve phase is triggered if the Reserve phase or a later phase fails. When this happens, the Unreserve method of all Reserve plugins will be executed in the reverse order of Reserve method calls. This phase exists to clean up the state associated with the reserved Pod.

The plugins of the Permit phase are invoked at the end of the scheduling cycle for each Pod, to prevent or delay the binding to the candidate node. A permit plugin can do one of the three things:

- approve: once all Permit plugins approve a Pod, it is sent for binding.
- deny: if any Permit plugin denies a Pod, it is returned to the scheduling queue. This will trigger the Unreserve phase in Reserve plugins.

- wait: if a Permit plugin returns "wait", then the Pod is kept in an internal "waiting" Pods list, and the binding cycle of this Pod starts but directly blocks until it gets approved. If a timeout occurs, wait becomes deny and the Pod is returned to the scheduling queue, triggering the Unreserve phase in Reserve plugins.

The plugins of the PreBind phase perform any work required before a Pod is bound. For example, a pre-bind plugin may provision a network volume and mount it on the target node before allowing the Pod to run there. If any PreBind plugin returns an error, the Pod is rejected and returned to the scheduling queue. The Pods of Bind phase the are used to bind a Pod to a node. Bind plugins will not be called until all PreBind plugins have completed. Each bind plugin is called in the configured order. A bind plugin may choose whether or not to handle the given Pod. If a bind plugin chooses to handle a Pod, the remaining bind plugins are skipped. Finally, the plugins of the PostBind phase are called after a Pod is successfully bound. This is the end of a binding cycle, and can be used to clean up associated resources.

2.5 Kubernetes descheduler

The scheduler's decisions are influenced by its view of a Kubernetes cluster at that point of time when a new Pod appears for scheduling. As Kubernetes clusters are very dynamic and their state changes over time, there may be desire to move already running Pods to some other nodes for various reasons:

- some nodes are under or over utilized.

- the original scheduling decision does not hold true any more, as some requirements are not satisfied any more.
- some nodes failed and their Pods are moved to other nodes.
- new nodes are added to clusters.

Consequently, there might be several Pods scheduled on less desired nodes in a cluster. The Kubernetes descheduler[2] finds Pods that can be moved and evicts them based on a set of criteria configurable through a policy. The descheduler policy includes default strategy plugins that can be enabled or disabled. It includes a common eviction configuration at the top level, as well as configuration from the Evictor plugin (Default Evictor, if not specified otherwise). Top-level configuration and Evictor plugin configuration are applied to all evictions. The Default Evictor Plugin is used by default for validating, filtering, grouping or sorting Pods before processing them in a strategy plugin, or for applying a PreEvictionFilter of Pods before eviction. Strategy plugins are grouped in Deschedule and Balance plugins. The Deschedule plugins process Pods one by one, and evict them in a sequential manner. The Balance plugins process all Pods, or groups of Pods, and determine which Pods to evict based on how the group was intended to be spread.

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

2.6 Kubernetes controllers and operators

Kubernetes controllers are control loops that watch the state of the cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state. A controller tracks at least one Kubernetes resource type. These objects have a spec field that represents the desired state. The controller(s) for that resource are responsible for making the current state come closer to that desired state. The controller might carry the action out itself; more commonly, in Kubernetes, a controller will send messages to the API server that have useful side effects.

The Job controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server. Job is a Kubernetes resource that runs a Pod, or perhaps several Pods, to carry out a task and then stop. Once scheduled, Pod objects become part of the desired state for a kubelet. When the Job controller sees a new task it makes sure that, somewhere in the cluster, the kubelets on a set of nodes are running the right number of Pods to get the work done. The Job controller does not run any Pods or containers itself. Instead, the Job controller tells the API server to create or remove Pods. Other components in the control plane act on the new information (there are new Pods to schedule and run), and eventually the work is done. After a new Job is created, the desired state is for that Job to be completed. The Job controller makes the current state for that Job be nearer to the desired state: creating Pods that do the work desired for that Job, so that the Job is closer to completion. Controllers also update the objects that configure them. For example once the work is done for a Job, the Job controller updates that Job object to mark it Finished.

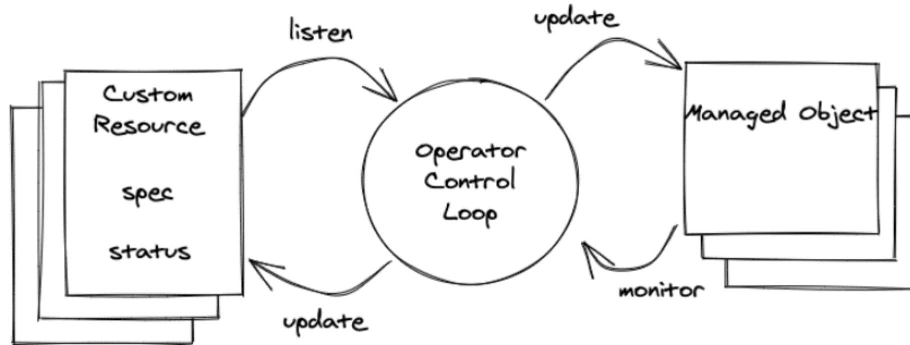

Figure 2.3: Kubernetes operator

In contrast with Job, some controllers need to make changes to things outside of the cluster. This is the case of controllers that make sure there are enough nodes in the cluster, then these controllers need something outside the current cluster to set up new nodes when needed. The controllers that interact with external state find their desired state from the API server, then communicate directly with an external system to bring the current state closer in line.

The control plane of Kubernetes is meant to be extensible and new custom controllers can be executed as operators. Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. The operator pattern aims to capture the key aim of a human operator who is managing a service or set of services. Human operators who look after specific applications and services have deep knowledge of how the system ought to behave, how to deploy it, and how to react if there are problems. People who run workloads on Kubernetes often like to use automation to take care of repeatable tasks. The operator pattern

captures how code can be written to automate a task beyond what Kubernetes itself provides. Kubernetes' operator pattern concept allows to extend the cluster's behaviour without modifying the code of Kubernetes itself by linking controllers to one or more custom resources. Operators are clients of the Kubernetes API that act as controllers for a custom resource.

2.7 Kubernetes in the Edge

As mentioned before, Kubernetes is meant to be extensible and this property has been exploited to adapt the platform also for Edge computing scenarios. Different Edge-oriented Kubernetes distributions have been recently proposed, with KubeEdge, MicroK8s and K3s being the most important ones.

KubeEdge[3] is an open source system extending native containerized application orchestration and device management to hosts at the Edge. It is built upon Kubernetes and provides core infrastructure support for networking, application deployment and metadata synchronization between Cloud and Edge. It also supports MQTT and allows developers to author custom logic and enable resource constrained device communication at the Edge. KubeEdge consists of a Cloud part and an Edge part, which consist of Cloud Core and Edge Core structures, unlike the Kubernetes master node and worker node structures.

MicroK8s[4], maintained by Canonical[5], aims to simplify the usage of Kubernetes on public and private Clouds by providing a lightweight and fully

[3] `https://kubeedge.io`
[4] `https://microk8s.io`
[5] `https://canonical.com`

compliant Kubernetes distribution, especially for low–end application areas like IoT. By default, microK8s enables all basic components of Kubernetes (like api–server, scheduler, or controller-manager) to make the cluster available. Further add-ons (e.g., DNS, ingress, or the metrics–server) can be enabled with one single command. The realization of high-availability where multiple nodes carry the control plane and the datastore, can be achieved with a few commands.

Rancher[6] offers K3s[7] a lightweight Kubernetes distribution, also with focus on low–end application areas. It is also fully compliant to Kubernetes, contains all basic components by default, and targets a fast, simple, and efficient way to provide a highly available and fault-tolerant cluster to a set of nodes. The deployment takes place via one single and small binary including dependencies. Similar to microK8s, etcd is replaced by another datastore, here sqlite3[8]. Also, in–tree storage drivers and Cloud provider components are removed to keep the size small. K3s tries to lower the memory footprint by a reorganization of the control plane components in the cluster. The K3s master and worker nodes, also called server and agents, encapsulate all the components in one single process. K3s is installed via a shell script that allows it to be run as a server or agent node. To achieve high-availability, new worker nodes can be easily added to the cluster by running a few commands. The minimum hardware requirements are at least 1 vCPU and 512 MB of memory.

K0s[9], maintained by Mirantis, is another free and open-source lightweight

[6]`https://www.rancher.com`
[7]`https://k3s.io`
[8]`https://www.sqlite.org`
[9]`https://k0sproject.io`

Kubernetes distribution. Similar to k3s, it is provided with core Kubernetes components as a single binary without host operating system dependencies and aims at bare metal, Edge, IoT, and Cloud scenarios. K0s is easy to install with only a few commands. By default, it isolates the control plane and only deploys application workloads to worker nodes. k0s uses etcd as control plane storage for multi-node clusters and SQLite for single-node clusters, but supports custom storages via kine[10]. K0s has a memory footprint of 510 MB RAM. Mirantis recommends controller nodes with at least 1 GB RAM and worker nodes with at least 512 MB RAM.

Although Kubernetes started to be extended also for Edge computing scenarios, still it is not yet ready to be fully adopted in the Cloud-to-Edge continuum, due to some limitations on its default orchestration and scheduling strategy that will be discussed in chapter 3.

[10] https://github.com/k3s-io/kine

Chapter 3

Motivation and state of the art

In this chapter the drawbacks of the Kubernetes platform that limit its adoption in the Cloud-to-Edge continuum and that motivate this work are first described. Then examples in the literature of Kubernetes extensions that aim to overcome these limitations are presented.

3.1 Kubernetes infrastructure and application models

To better understand the reason why Kubernetes is not yet ready to be fully adopted in the Cloud-to-Edge continuum, an explanation of how both the infrastructure and the application are modeled by the Kubernetes control plane needs to be done.

Figure 3.1 shows an example of a Kubernetes cluster. Each node N_i provides a set of allocatable CPU and memory resources, indicated as cpu_i and mem_i. The information about allocatable resources on each node is used

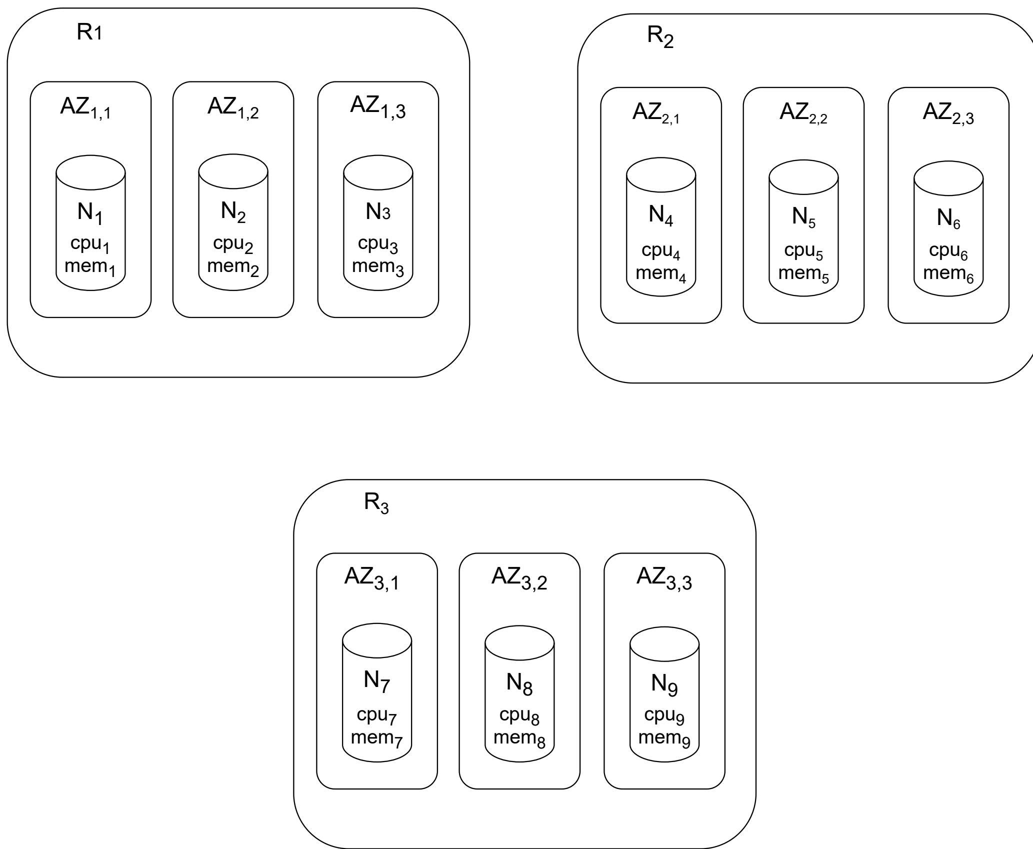

Figure 3.1: Kubernetes infrastructure model

by the Kubernetes scheduler to establish if a Pod can be scheduled on that node, based on the Pod resource requirements and the resources available on the node. Available resources on a node are evaluated by the scheduler as the difference between the total allocatable resources of the node and the sum of resources requested by each Pod executed on that node. Nodes are grouped in topology domains, in particular availability zones $AZ_{i,j}$ and regions R_i. The concept of topology domain is used by the Kubernetes scheduler as an indicator about the network distance among nodes. Nodes on the same topology domain are considered by the scheduler to be near to each other in terms of network distance. Both allocatable resources on each node and topology domains are static cluster metadata that are defined by cluster administrators by assigning labels to nodes.

Figure 3.2 shows an example of application as intended by Kubernetes. An application is made up of different microservices μ_i, each represented by a Deployment D_i, which in turn contains a Pod template Pt_i. A Pod template of a Deployment is a template for the Pods managed by that Deployment, each representing a replica of the corresponding microservice.

Listing 1 shows an example of Pod template. A Pod template Pt_i contains a specific set of resource requirements, respectively cpu_i and mem_i for the amount of CPU and memory resources that have to be reserved to that Pod and a set of inter-Pod affinity rules. Each affinity rule $aff_{i,j}$ reflects a communication relationship between microservices μ_i and μ_j and states that Pods of the two microservices should be placed on the same topology domain. The topology domain (eg. region or availability zone) is specified by the *topologyKey* parameter of the affinity rule. The *weight* parameter of

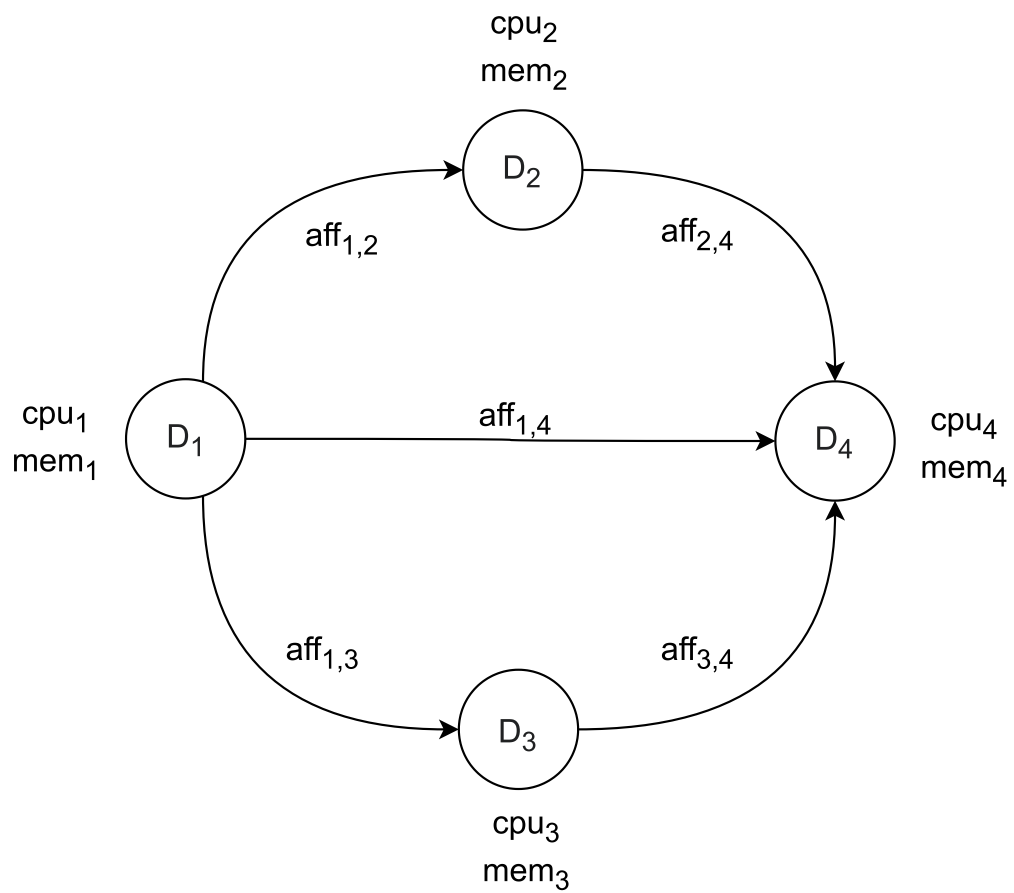

Figure 3.2: Kubernetes application model

```yaml
spec:
  containers:
  - name: m1
    image: nginx
    resources:
      requests:
        memory: 300Mi
        cpu: 500m
    affinity:
      podAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 50
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - m2
            topologyKey: topology.kubernetes.io/region
```

Listing 1: Example of a Pod template spec section

an affinity rule represents the priority of that rule. The greater this value, the greater the need to schedule the two microservices near to each other. An affinity rule can be used to give information to the Kubernetes scheduler about the need to place microservices with a high communication degree near to each other. As in the case of the infrastructure, the application graph, with the set of microservices with their resource requirements and affinity rules, needs to be specified before the run time phase by application developers. The submitted application graph is then used by the Kubernetes scheduler to find an optimal placement for each Pod.

3.2 Kubernetes limitations

The Kubernetes infrastructure and application models and the scheduler and descheduler policies based on these models are the main limitations in the adoption of Kubernetes in the Cloud-to-Edge continuum.

Regarding the infrastructure, the main limitation is related to the fact that Kubernetes does not consider the run time state of the infrastructure when scheduling and evicting Pods for rescheduling. Cloud-only environments are characterized by homogeneous nodes with high computational resources and with high speed network connectivity. On the other hand the Cloud-to-Edge continuum is an heterogeneous evironment, with the Edge infrastructure being more dynamic and unstable than that of Cloud data centers and characterized by more frequent node failures, limited computational resources, network partitions and variations in the network latencies. Considering the run time cluster state and network conditions during Pod scheduling and rescheduling decisions is critical in dynamic environments like the Cloud-to-Edge continuum, where node resource availability and network latencies are unpredictable and highly variable factors.

However Kubernetes does not monitor the current CPU and memory utilization on each node. To evaluate the utilization level of a node resource, Kubernetes considers the sum of the values required for that resource by each Pod running on that node. This means that the estimated resource usage may not match its run time value. If resource usage on a node is underestimated, more Pods end up being scheduled on that same node, causing an increase in the shared resource interference between Pods and consequently a decrease in the overall application performances. Furthermore, the current network

latencies between cluster nodes are not considered when evaluating inter-Pod affinities. This means that although an inter-Pod affinity rule is satisfied by placing the respective Pods on the same topology domain, the two Pods are not guaranteed to communicate with low network latencies. This because a topology domain does not always corresponds to a latency-constrained domain, especially in the case of Edge environments. In a distributed application, high communication latencies between microservices lead to high end-to-end application response times.

Regarding the application, two main limitations characterize the Kubernetes orchestration strategy. The first one is that Kubernetes is not application-aware, in the sense that no application topology information is used during the scheduling of the application itself. Kubernetes implements a declarative language that allows to model each microservice as an independent component, but does not offer a mechanism to model at a high level the microservice-to-microservice communication relationships, in particular the communication patterns and protocols. Neither this information is used during the scheduling of Pods to minimize the network distance between those microservices that interact through communication channels that can represent a bottleneck for the application performances. This is the case, for example, of HTTP and gRPC synchronous communication channels whose response times contribute to the end-to-end application response time. The second limitation is related to the fact that Kubernetes does not evaluate the current application state, in terms of resource usage of each microservice and communication intensity between microservices, when taking its scheduling and rescheduling decisions. Only statically defined Pod resource

requirements and inter-Pod affinity rules are considered. Application developers need to predict ahead of time how many resources are required by each microservice and what are the most involved microservices communication channels, in order to determine the CPU and memory requirements of each Pod and the inter-Pod affinity rules and their weights.

However, this is a complex task, considering that microservices resource requirements and communication affinities are dynamic parameters that strongly depend on the run time load and distribution of user requests. Defining all Pod resource requirements and inter-Pod affinities before the run time phase can lead to inefficient scheduling decisions and then reduced application performances. Overestimating resource requirements for Pods reduces the probability that these are scheduled on constrained Edge nodes near to end users, while underestimating resource requirements increase Pod density on cluster nodes and then their interference. Errors in estimating inter-Pod affinities can lead to situations where microservices that communicate more are not placed near to each other. Thus, considering run time resource usage of each microservice and the communication intensity between them and not only statically defined Pod resource requirements and inter-Pod affinity rules is a critical requirement to define an effective scheduling strategy able to minimize QoS violations for complex distributed applications.

Furthermore, Kubernetes does not implement scheduling policies that consider the end user location for the placement of microservices, neither the interaction between microservices and other external services like. This is a critical aspect considering that users are typically geo-distributed and the network distance between microservices, especially the frontend ones, and

the end user can affect the application response time.

Finally, another important limitation is related to the fact that Kubernetes does not evaluate the interaction between microservices and other external services like database servers. Considering also these interactions and scheduling microservices based on the position of the external services is critical to avoid bottleneck in the application response time.

3.3 Kubernetes extension proposals

In the literature, there is a variety of works that propose to extend the Kubernetes platform in order to adapt its usage to Cloud-Edge environments [29, 30].

A network-aware scheduler is proposed in [31], implemented as an extension of the filtering phase of the default Kubernetes scheduler. The proposed approach makes use of round-trip time labels, statically assigned to cluster nodes, in order to minimize the network distance of a specific Pod with respect to a target location specified on its configuration file.

The authors of [32] present an orchestration tool that extends Kubernetes with adaptive autoscaling and network-aware placement capabilities. The authors propose a two-step control loop, in which a reinforcement learning approach dynamically scales container replicas on the basis of the application response time, and a network-aware scheduling policy allocates containers on geo-distributed computing environment. The scheduling strategy uses a greedy heuristic that takes into account node-to-node latencies to optimize the placement of latency-sensitive applications.

In [33], the authors propose a solution for resource allocation in a Kubernetes infrastructure hosting network service. Te proposed solution aims to avoid resource shortages and protect the most critical functions. Te authors use a statistical approach to model and solve the problem, given the random nature of the treated information.

In [34] an extension to the Kubernetes default scheduler is proposed that uses information about the status of the network, like bandwidth and round trip time, to optimize batch job scheduling decisions. The scheduler predicts whether an application can be executed within its deadline and rejects applications if their deadlines cannot be met.

The authors of [35] propose to leverage application-level telemetry information during the lifetime of a distributed application to create service communication graphs that represent the internal communication patterns of all components. The graph-based representations are then used to generate colocation policies of the application workload in such a way that the cross-server internal communication is minimized.

In [36] Pogonip, an Edge-aware scheduler for Kubernetes, designed for asynchronous microservices is presented. Authors formulate the placement problem as an Integer Linear Programming optimization problem and define a heuristic to quickly find an approximate solution for real-world execution scenarios. The heuristic is implemented as a set of Kubernetes scheduler plugins.

In [37] a Kubernetes scheduler extender is proposed that uses application traffic historical information collected by Service Mesh to ensure efficient placement of Service Function Chains (SFCs). During each Pod scheduling,

nodes are scored by adding together traffic amounts, averaged over a time period, between the Pod's microservice and its neighbors in the chain of services executed on those nodes.

In [38] a scheduling framework is proposed which enables Edge sensitive and Service-Level Objectives (SLO) aware scheduling in the Cloud-Edge-IoT Continuum. The proposed scheduler extends the base Kubernetes scheduler and makes scheduling decisions based on a service graph, which models application components and their interactions, and a cluster topology graph, which maintains current cluster and infrastructure-specific states.

In [39] Nautilus is presented, a run-time system that includes, among its modules, a resource contention aware resource manager and a communication-aware microservice mapper. The first, by using a RL algorithm to capture the relationship between microservices resource contention and the overall performance, make optimal resource allocation decisions to lower the resource usage. The latter divides the microservice graph into multiple partitions based on the communication overhead between microservices and maps the partitions to the cluster nodes in order to make frequent data interaction complete in memory.

The authors of [40] propose an improved design for Kubernetes scheduling that takes into account physical, operational, and network parameters in addition to software states in order to enable better orchestration and management of Edge computing applications. They compare the proposed design to the default Kubernetes scheduler and show that it offers improved fault tolerance and dynamic orchestration capabilities.

In [41] a custom Kubernetes scheduler is proposed that takes into account

delay constraints and Edge reliability when making scheduling decisions. The authors argue that this type of scheduler is necessary for Edge infrastructure, where applications are often delay-sensitive, and the infrastructure is prone to failures. The authors demonstrate their Kubernetes extension and release the solution as open source.

Finally, [42] presents a Kubernetes Edge-scheduler that considers inter-node network latencies and services communication requirements in order to optimize, using an heuristic algorithm, the placement of containerized applications in geographically distributed clusters. A re-scheduler is also proposed that is responsible for container migration in order to improve resource utilization in the cluster.

Chapter 4

Sophos framework

In this chapter, the Sophos framework is presented as an extension of the Kubernetes platform in order to adapt its usage to dynamic Cloud-to-Edge continuum environments. The Sophos framework has been designed and implemented in several steps, each one presented on a separate published work like [43, 44, 45]. First, an overview of the framework is given. Then each framework component is described in detail.

4.1 Overall design

Considering the limitations described in chapter 3, the Sophos framework aims to extend the Kubernetes platform with a dynamic application and infrastructure-aware orchestration and scheduling strategy, in order to adapt its usage to dynamic Cloud-to-Edge continuum environments. The main goal of the Sophos framework is to reduce the impact of the instability of the Edge infrastructure on the end-to-end application response time.

To do this, the orchestration and scheduling strategy of Sophos is guided by three basic policies. The first one is based on the idea of reducing the shared resource interference between the microservices of an application. The responsiveness of a microservice is strongly related to the amount of available computational resources on the node where it is executed. The higher the request load on a microservice, the higher the amount of computational resources it requires for performing a computation with fast response times. This means that the shared resource interference between microservices can have a negative impact on their response times and this requires a scheduling strategy that places microservices on nodes based on the microservices resource requirements and the nodes resource availability. The second one is based on the idea of reducing the network distance between the microservices that communicate through critical communication channels. In a microservices application a generic user request consists of a chain of sub-requests and its end-to-end latency is affected by the latencies of each service call in the chain. This means that in order to obtain lower end-to-end application response times it is necessary to consider the whole application topology graph and try to schedule each microservice on the basis of the other microservices placement. In particular, critical communication channels are those that represent bottlenecks in a chain of requests. Critical communication channels are those channels that use synchronous and blocking communication protocols or those through which a high amount of traffic is exchanged. Scheduling microservices that communicate through critical channels on the same node or in nodes with a limited network latency allows to reduce the end-to-end application response time. Finally, the third one is based on the idea of

scheduling the application considering also the position of end users and the external services with which the microservices interact. In a Cloud-to-Edge continuum scenario, external services, like database servers, are typically located on Cloud data centers, while end users are geographically distributed on the Edge layer. Though scheduling the application microservices near to each other, the application response time cannot be improved if the overall application is executed far away from the locations where the highest amount of user requests originate or the most requested external services are located.

In general, the main idea of the Sophos framework is that the orchestration and scheduling of complex microservices applications should consider the application topology and the dynamic states of both the application and the infrastructure where the application is executed. These are critical information to consider in order to reduce the impact of the instability of the Edge infrastructure on the end-to-end application response time.

To this aim, in Sophos, a way for application architects to model the application topology is provided and at the same time current node resource availability, node-to-node network latencies, resource usage of microservices and communication intensity between microservices and between the application and the end users and external services are continuously monitored and taken into account during application scheduling. This reduces the effort for application architects to predict ahead-of-time resource usage of each microservice and the run time communication relationships between microservices, in order to define Pod resource requirements and inter-Pod affinity rules. Furthermore, a key point in the Sophos framework is the requirement to continuously tune the placement of the application based on the

ever changing infrastructure state of Cloud-Edge environments, run time resource usage of microservices and their communication interactions.

Figure 4.1 shows a general model of the proposed framework. The current infrastructure and application states are monitored and all the telemetry data are collected by a *metrics server*. For the infrastructure, node resource availability and node-to-node latencies are monitored, while for the application, CPU and memory usage of microservices and the traffic amount exchanged between them are monitored. Based on the infrastructure telemetry data the *cluster monitoring* operator determines a *cluster graph* with the set of available resources on each cluster node and the network latencies between them. Application architects model the static *application group graph* with the set of microservices and external services and their static communication relationships, in terms of the communication patterns and protocols. On the other hand the *application monitoring* operator uses application telemetry data to determine the dynamic application group graph whose nodes represent microservices with their current resource usage and the edges their dynamic communication relationships, in terms of the respective traffic amount exchanged between them.

The dynamic application group graph also includes a set of nodes each representing an input or an output proxy. An input proxy consists of a Pod managed by a DaemonSet that intercepts all the user requests sent to the node where it is executed and forwards them to a frontend application microservice, like for example an API gateway microservice. An edge in the dynamic application group graph that connects an input proxy to a microservice represents the traffic flow to that microservice coming from the node where

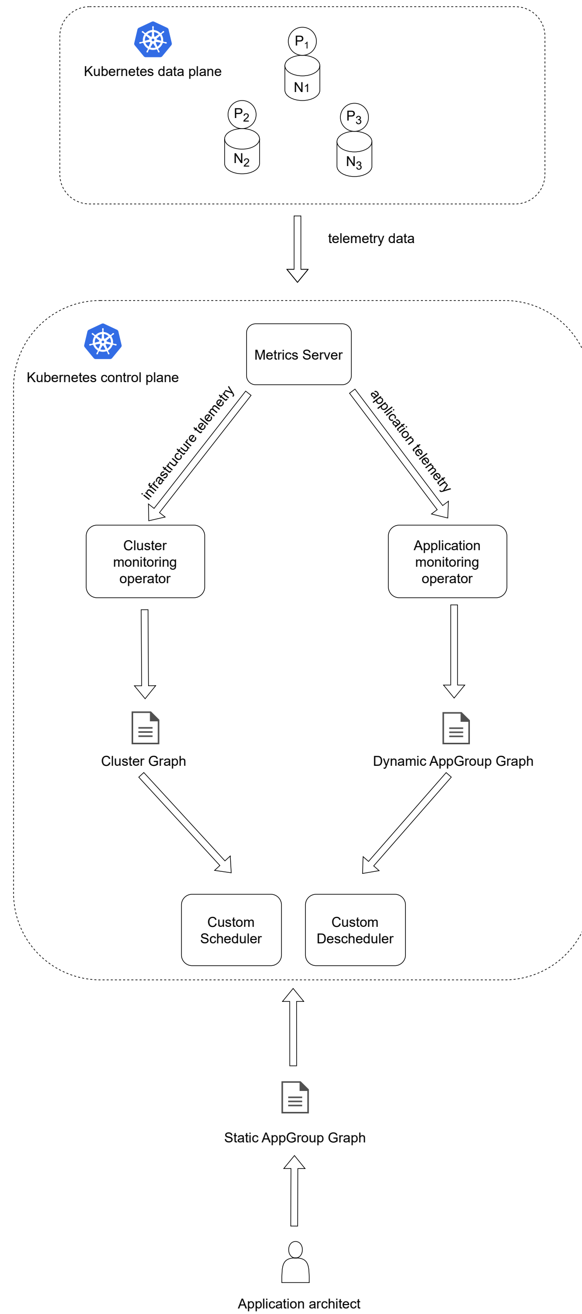

Figure 4.1: Overall architecture

the proxy service is executed. An output proxy consists of a Pod managed by a Deployment that intercepts all the requests sent by microservices to a specific external service. An edge in the dynamic application group graph that connects a microservice to an output proxy represents the traffic flow between that microservice and the external service. Different output proxies are associated with different external services and each output proxy is located on a node with limited network distance to the corresponding external service.

The cluster and application group graphs are then used by the *custom scheduler* to determine a placement for each Pod in the application, and the *custom descheduler* to take Pod rescheduling actions if better scheduling decisions can be done. A prototype implementation of the Sophos framework publicly available[1]. Further details on the components of the framework are provided in the following sections.

4.2 Cluster monitoring operator

The cluster monitoring operator periodically determines the cluster graph with the currently available CPU and memory resources on each cluster node and the node-to-node network latencies. Figure 4.2 shows how it works. This component is a Kubernetes operator written in the Java language by using the Quarkus Operator SDK[2]. As a Kubernetes operator it is activated by a Kubernetes custom resource, in particular the *Cluster* custom resource. A

[1] `https://github.com/unict-cclab/sophos`

[2] `https://quarkus.io/extensions/io.quarkiverse.operatorsdk/quarkus-operator-sdk`

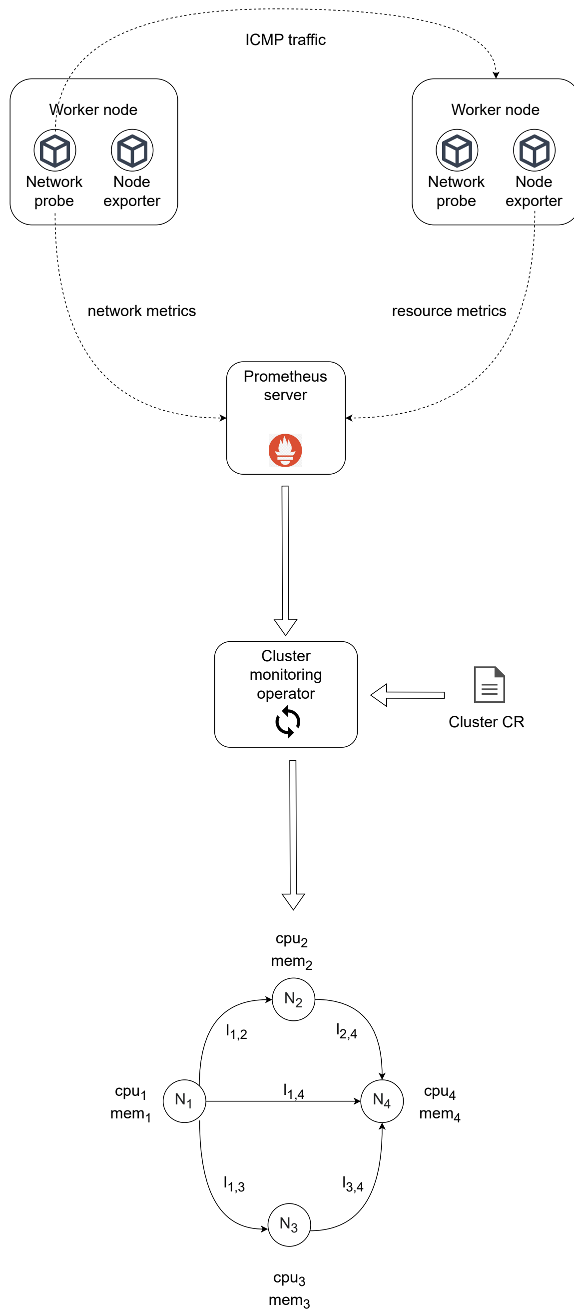

Figure 4.2: Cluster monitoring operator

Cluster resource contains a *spec* property with two sub-properties: *runPeriod* and *nodeSelector*.

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

Listing 2: Example of a Cluster custom resource

Listing 2 shows an example of a Cluster custom resource. The runPeriod property determines the interval between two consecutive executions of the operator logic. The nodeSelector property represents a filter that selects the list of nodes in the cluster that should be monitored by the operator.

During each execution the list of Node resources that satisfy the condition specified by the nodeSelector property is fetched from the Kubernetes API server. First, for each node N_i the CPU and memory values currently available on it, cpu_i and mem_i respectively, are determined. These values are fetched by the operator from a Prometheus[3] metrics server, which in turn collects them from node exporters executed on each cluster node. The cpu_i and mem_i parameters are then assigned as values for the *available-cpu* and *available-memory* annotations of the node N_i.

Then, for each pair of nodes N_i and N_j their network latency $l_{i,j}$ is determined. The $l_{i,j}$ parameter is proportional to the network latency between nodes N_i and N_j. Network latency metrics are fetched by the operator from the Prometheus metrics server, which in turn collects them from network

[3]https://prometheus.io

probe agents executed on each cluster node as Pods managed by a DaemonSet. These agents are configured to periodically send ICMP traffic to all the other cluster nodes in order to measure the round trip time value. For each node N_i the operator assigns to it a set of annotations l-n_j, with values equal to those of the corresponding $l_{i,j}$ parameters. Finally, the cluster graph with the updated CPU and memory available resources and the network cost values for each node is then submitted to the Kubernetes API server.

4.3 Application topology modeling

The first step necessary for the implementation of an effective application-aware orchestration and placement strategy consists on the modeling of the application topology graph, namely the microservices that compose an application, the external services requested by the application and their communication channels. What is known before the run time phase by application architects about the application composition and the communication patterns between its microservices and the external services represents an important information that can be used for the initial placement of Pods, when no information about the dynamic communication interactions between Pods is given.

Unlike the base Kubernetes scheduler implementation that mainly considers low level resource requirements, Sophos allows to specify higher level requirements that relate with the application composition and its microservices communication patterns. The basic idea is that a scheduling strategy based on application architecture and topology information can improve the

quality and effectiveness of microservice deployments and allows to realize scheduling strategies customized for the specific application needs. High-level requirements can also relate to the resiliency, availability and reliability of an application. However, Sophos considers those related with the communication aspects, because they are the most critical when dealing with communication-intensive and latency sensitive applications.

A fundamental requirement in the application modeling task is to give application architects a tool that allows them to model the application topology in a high level and qualitative manner, following an intent modeling approach [46]. This way application architects have not to deal with quantitative parameters and low level details, but they only need to know the microservices that compose an application, the requested external services and the protocols they use to communicate between them. For this purpose Sophos uses the TOSCA standard [47] for modeling the application topology graph. The application topology graph is modeled as a TOSCA service template where node templates represent microservices and external services and the relationship templates the communication channels between them. The TOSCA service template is then compiled into a set of Kubernetes deployment artifacts by using the Puccini compiler[4] and its TOSCA profile for Kubernetes.

In particular, this profile defines the *Service* node type that we use to model a generic microservice or external service. This node type exposes the capabilities *metadata*, *service* and *deployment*, whose types are *Metadata*, *Service* and *Deployment* respectively. Furthermore, a *route* requirement is defined, whose instances can be associated with *Service* capabilities of other

[4] `https://puccini.cloud`

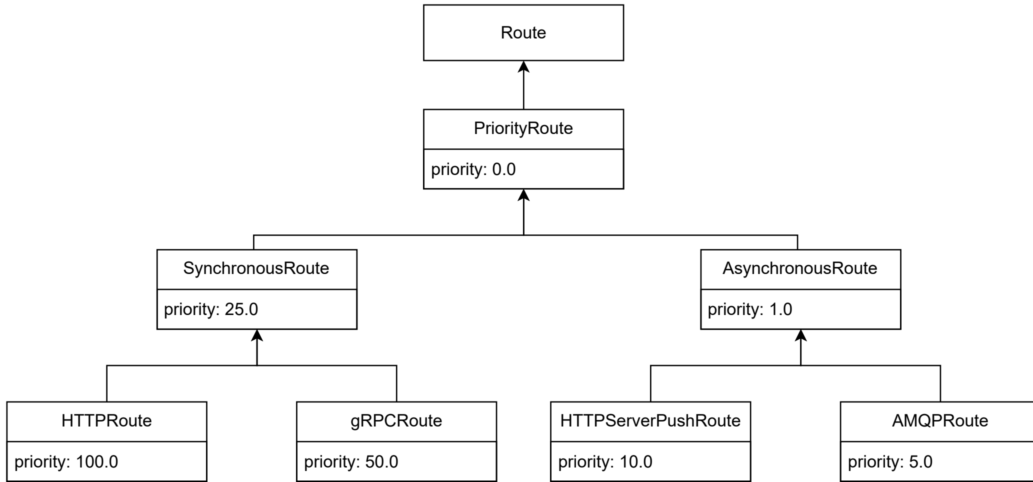

Figure 4.3: Relationship types hierarchy

Service node templates. The association between *route* requirement instances and *Service* capabilities can be done through relationship templates of the *Route* type.

To model the various types of communication channels that characterize typical microservice applications, a custom hierarchy of relationship types is defined, shown in Figure 4.3, as an extension of the *Route* relationship type. The relationship type *PriorityRoute* is the base type for which the property *priority* is defined. This property represents the priority level associated with the corresponding communication channel and for each type in the hierarchy a default value is assigned. The higher this value, more critical the communication channel is. The relationship types hierarchy is defined in such a way that a higher priority value is associated with the synchronous and heavyweight communication protocols. The relationship types *Synchronous-Route* and *AsynchronousRoute* extend the type *PriorityRoute* and represent synchronous and asynchronous communication channels respectively. The re-

lationship types *HTTPRoute* and *gRPCRoute* extend the type *Synchronous-Route* and represent HTTP and gRPC communication channels respectively. The relationship types *HTTPServerPushRoute* and *AMQPRoute* extend the type *AsynchronousRoute* and represent HTTP Server Push and AMQP communication channels respectively. The proposed relationship types hierarchy is only a sample hierarchy that models typical communication patterns in a microservices application. It can be extended with new custom types and the default priority values can be overridden in each relationship template inside a service template.

The Puccini compiler takes as input the application service template and maps each Service node template nt_i with a Kubernetes Deployment D_i. The node template nt_i associated to an external service is mapped by the compiler with a Deployment D_i of the corresponding output proxy. In the Pod template section of a Deployment D_i an annotation named *saff-*D_j is set for each Route relationship between node templates nt_i and nt_j. The value associated with this annotation is equal to the priority attribute value of the relationship and represents the static contribution of the communication affinity between microservices μ_i and μ_j or between a microservice μ_i and an external service es_j.

4.4 Application monitoring operator

The application monitoring operator periodically determines the dynamic application group graph with the current CPU and memory usage for each microservice and the traffic amounts exchanged between microservices, be-

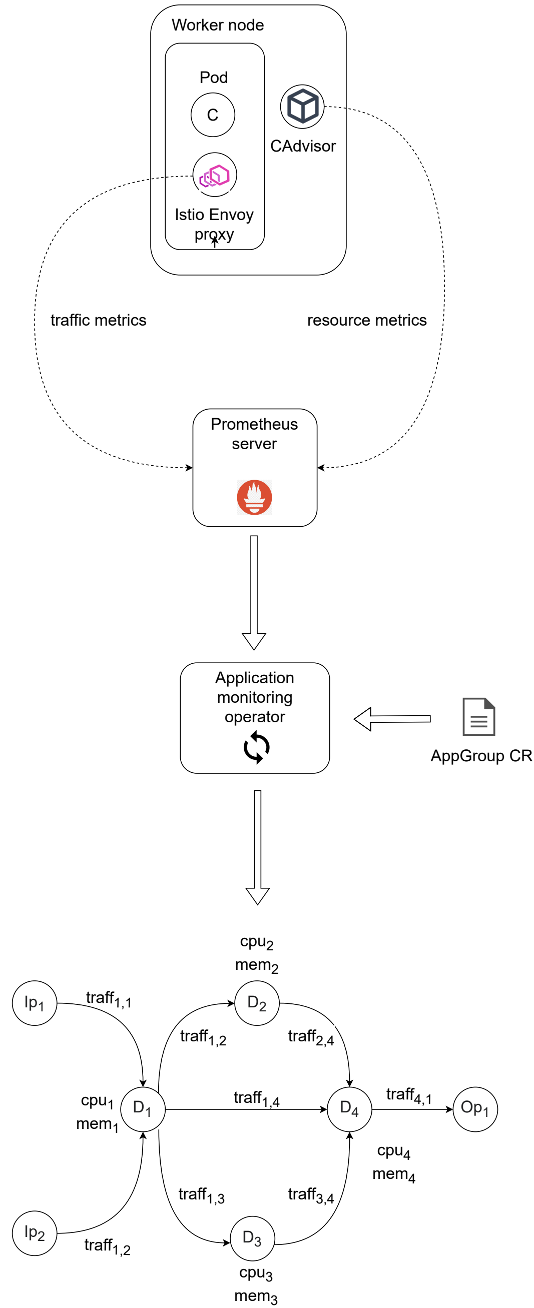

Figure 4.4: Application monitoring operator

```yaml
apiVersion: unict.it/v1alpha1
kind: AppGroup
metadata:
  name: sample-app-group
spec:
  runPeriod: 30
  name: sample-app-group
  namespace: default
```

Listing 3: Example of an AppGroup custom resource

tween the frontend microservices and the input proxies executed on each cluster node and between the backend microservices and the output proxies. Figure 4.4 shows how it works. As in the case of the cluster monitoring operator, this component is a Kubernetes operator written in the Java language by using the Quarkus Operator SDK. This operator is activated by an instance of the *AppGroup* Kubernetes custom resource. An AppGroup resource contains a *spec* property with three sub-properties: *runPeriod*, *name* and *namespace*. Listing 3 shows an example of an AppGroup custom resource.

The runPeriod property determines the interval between two consecutive executions of the operator logic. The name and namespace properties are used to select the set of Deployment resources that compose a specific microservices application. The Deployments selected by the AppGroup custom resource are those created in the namespace specified by the namespace property of the custom resource and with a label *app-group* whose value is equal to the value of the name property in the custom resource.

During each execution the list of Deployment resources selected by the name and namespace properties are fetched from the Kubernetes API server.

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

Then for each Deployment D_i, the traffic amounts $traff_{i,j}$ with all the other Deployments D_j of the application are determined. The $traff_{i,j}$ parameter is proportional to the traffic amount exchanged between microservices μ_i and μ_j. If a Deployment D_i is associated with a frontend microservice, then the set of traffic amounts $traff_{i,j}$ with all the input proxy Pods Ip_j are determined also. While, if a Deployment D_i is associated with a backend microservice and this microservice interacts with some external services, then the set of traffic amounts $traff_{i,j}$ with the corresponding output proxy Pods Op_j are determined also. Traffic metrics are fetched by the operator from the Prometheus metrics server, which in turn collects them from the Istio[6] platform. Istio is a service mesh implementation, whose control plane is installed in the Kubernetes cluster. The Istio control plane injects a sidecar container running an Envoy proxy on each Pod when they are created. All the traffic between Pods is intercepted by their corresponding Envoy proxies that in turn expose traffic statistics through metrics exporters that can be

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

queried by the Prometheus server. By injecting Envoy proxies also on the input proxy Pods the source of user requests and the distribution of user traffic among the different sources can be traced. While, in the case of the output proxy Pods, the traffic between microservices and the external services can be traced.

Each $traff_{i,j}$ parameter is assigned by the operator as the value for the annotation *daff-*D_j (*daff-*Ip_j in the case $traff_{i,j}$ corresponds to the traffic amount exchanged with the input proxy Pod Ip_j or *traff-*Op_j in the case $traff_{i,j}$ corresponds to the traffic amount exchanged with the output proxy Pod Op_j) of the Deployment D_i. The value associated with this annotation represents the dynamic contribution of the communication affinity between microservices μ_i and μ_j. The application group graph with the set of CPU and memory usage and the traffic amounts for each Deployment is then submitted to the Kubernetes API server.

4.5 Custom scheduler

The proposed custom scheduler extends the default Kubernetes scheduler by implementing three additional plugins. The custom scheduler is a program written in the Go language and is based on the Kubernetes *scheduling framework*. In particular, the *ResourceAware* and *NetworkAware* plugins that extend the node scoring phase of the default Kubernetes scheduler and the *QueueSort* plugin that extends the Pod queue sorting phase are proposed. The QueueSort plugin is executed when a new Pod arrives to the scheduling queue and it establishes an ordering for that Pod. For each Pod to be sched-

uled, each of the two plugins assigns a partial score to each candidate node of the cluster that has passed the filtering phase. The ResourceAware plugin takes into account the values of the *cpu-usage* and *memory-usage* annotations of the Deployment associated with the Pod to be scheduled and the values of the *available-cpu* and *available-memory* annotations of the node to be scored. The NetworkAware plugin takes into account the values of the *saff* and *daff* annotations of the Deployment associated with the Pod to be scheduled and the values of the *l* annotations of the node to be scored. The node scores calculated by the ResourceAware and NetworkAware plugins are added to the scores of the other scoring plugins of the default Kubernetes scheduler.

Algorithm 1 Custom scheduler node scoring function

Input: p, cpu_p, mem_p, n, cpu_{n_i}, mem_{n_i} cns, l, $saff$, $daff$

Output: $score$

1: $rascore \leftarrow \alpha \times \frac{cpu_{n_i} - cpu_p}{cpu_p} \times 100 + \beta \times \frac{mem_{n_i} - mem_p}{mem_p} \times 100$
2: $cmc \leftarrow 0$
3: **for** cn in cns **do**
4:     $pcmc \leftarrow 0$
5:     **for** cnp in $cn.pods$ **do**
6:         $pcmc \leftarrow pcmc + l_{n,cn} \times (saff_{p,cnp} + daff_{p,cnp})$
7:     **end for**
8:     $cmc \leftarrow cmc + pcmc$
9: **end for**
10: $nascore \leftarrow -cmc$
11: $score \leftarrow \gamma \times rascore + \delta \times nascore$

The algorithm takes as inputs the following arguments:

- p: the Pod to be scheduled.
- cpu_p: the CPU usage of Pod p.
- mem_p: the memory usage of Pod p.
- n: the node to be scored.
- cpu_n: the CPU available on node n.
- mem_n: the memory available on node n.
- cns: the set of nodes in the cluster, including node n.
- l: the network latencies between node n and all the other nodes cns.
- $saff$: the static communication affinities between the Pod p and all the other Pods in the application.
- $daff$: the dynamic communication affinities between the Pod p and all the other Pods in the application and the input and output proxy Pods.

The algorithm starts by calculating the value of the variable $rascore$. This variable represents the partial contribution to the final node score given by the ResourceAware scheduler plugin. Its value is given by the weighted sum of the percentage differences between the CPU and memory currently available on node n and the respective usage values of Pod p. The α and β parameters are in the range between 0 and 1 and their sum is equal to

1. By changing the values of these parameters, a different contribution to the *rascore* variable value is given by the respective CPU and memory percentage differences. Unlike the default Kubernetes scheduler that estimates computational resources availability on a node based on the sum of the requested resources of each Pod running on that node, the ResourceAware plugin considers current node resource usage based on the run time telemetry data. The higher the difference between available resources on node n and those used by Pod p, the greater the score assigned to node n. This allows to effectively balance the load between cluster nodes and then to reduce the shared resource interference between Pods resulting from incorrect node resource usage estimation and then its impact on application performances.

Then the partial contribution to the final node score given by the NetworkAware scheduler plugin is calculated. First the variable *cmc* is initialized to zero. This variable represents the total cost of communication between the Pod p and all the other Pods in the application when the Pod p is placed on node n. This variable represents the total cost of communication between the Pod p and all the other Pods of the application (or those corresponding to input or output proxies) when the Pod p is placed on node n. The algorithm iterates through the list of cluster nodes *cns*. For each cluster node *cn* the *pcmc* variable value is calculated. This variable represents the cost of communication between the Pod p and all the other Pods *cn.pods* currently running on node *cn* when the Pod p is placed on node n. For each Pod *cnp* running on node *cn* the sum of the values of the parameters $saff_{p,cnp}$ and $daff_{p,cnp}$ is multiplied by the network latency $l_{n,cn}$ between node n and node *cn* and added to the *pcmc* variable. The *pcmc* variable value is then

added to the *cmc* variable. The final partial node score contribution of the NetworkAware scheduler plugin is assigned to the variable *nascore* as the opposite of the *cmc* variable value. The *nascore* variable value is assigned in such a way that the Pod p is placed on the node, or in a nearby node in terms of network latencies, where the Pods with which the Pod p has the highest communication affinity are executed. By considering the current node-to-node network latencies and the communication affinities between microservices the proposed NetworkAware plugin implements a dynamic Pod scheduling strategy differently from the default inter-Pod affinity plugin that implements a static scheduling strategy. For each affinity rule the default Kubernetes scheduler assigns a score different from zero only to the nodes that belong to a topology domain matched by the *topologyKey* parameter of the rule. The NetworkAware plugin instead scores all the cluster nodes based on the current relative network distance between them. This allows to implement a more fine-grained node scoring approach able to take into account the ever changing network conditions in the cluster instead of using node labels statically assigned before the run time phase. Furthermore, differently from the inter-Pod affinity plugin that takes into account only statically assigned communication affinity weights between Pods, the NetworkAware plugin also evaluates the traffic between Pods to determine the effective communication affinities between them that can change at run time. By considering also the communication affinities between the application Pods and those corresponding to input or output proxies, with the last ones placed near to the corresponding external services, the custom scheduler is able to schedule the application near the end users from which the highest amount of requests

originates or the most requested external services.

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

The QueueSort plugin is invoked during the sorting phase in order to determine an ordering for the Pod scheduling queue. This function takes as input two Pods and returns *true* if the value of the *index* label on the first Pod is lesser than the value of the same label on the second one, otherwise it returns *false*. This way Pods with lower values of the index label are scheduled first. The index label is used to determine a topological sorting of the microservices graph and application developers have to assign a value for this label on the Pod templates of each application Deployment, with lower values given to the frontend microservices. By scheduling frontend microservices first, communication affinities between these microservices and the input proxies are evaluated earlier during the scoring phase, resulting in the application placement following the end user position. If backend microservices are instead scheduled before the frontend ones, only communication affinities between internal application microservices are evaluated during the scoring phase and this can lead to situations where the frontend microservices are placed far away from the end user.

4.6 Custom descheduler

The custom descheduler runs as a controller in the Kubernetes control plane. This component is a Kubernetes operator written in the Go language by using the Operator SDK framework[7]. This operator is activated by an instance of the same *AppGroup* Kubernetes custom resource that activates the application monitoring operator. The runPeriod property of the AppGroup resource determines the interval between two consecutive executions of the descheduler logic. The name and namespace properties are used to select the set of Pods that compose a specific microservices application.

The main business logic of the custom descheduler is implemented by a descheduling function that is called periodically for each application Pod to establish if that Pod should be rescheduled or not. Pods are evaluated based on the same ordering defined by the QueueSort plugin. Inside the descheduling function the same node scoring function implemented by the custom scheduler and showed in Algorithm 1 is invoked for each cluster node in order to assign them a score based on the current cluster and application group graphs. If there is at least one node with a higher score than that of the node where the Pod is currently executed, the descheduler evicts the Pod. As in the case of the default Kubernetes descheduler, the proposed custom descheduler does not schedule a replacement of evicted Pods but relies on the custom scheduler for that. The use of the proposed custom descheduler is aimed at giving the running application Pods the possibility to be rescheduled on the basis of the current cluster network latencies and computational resources availability on each node and the traffic exchanged

[7] `https://sdk.operatorframework.io`

between microservices and their computational resources usage, thus allowing to optimize the application placement at run-time. By evicting currently running Pods and then forcing them to be rescheduled, application scheduling can take into account the ever changing cluster and application states with the latter mainly influenced by the user request load and patterns. One limitation of the proposed approach is that Pod eviction can cause downtime in the overall application. However, it should be considered that cloud-native microservices are typically replicated, so the temporary shutdown of one instance generally causes only a graceful degradation of the application quality of service. To reduce the impact of Pod rescheduling, for each execution the descheduler evicts one Pod at most among the replicas of a single Deployment.

Chapter 5

Evaluation

5.1 Application and test bed environment

The proposed framework has been validated using the Sock Shop application[1] executed on a test bed environment. As shown in Figure 5.1, this application is composed of multiple microservices and exposes different APIs that can be requested by external users from the *frontend* service. An external request for an API consists of multiple sub-requests between a subset of the application microservices. Then the application can be thought of as composed of different microservice chains, each activated by a specific application API. In this application three different communication patterns between microservices are present. The communication between the *frontend* service and the other backend services and between the *order* and *shipping* services is a synchronous HTTP communication. The *order*, *user*, *catalogue* and *cart* services store their data in different database servers and the communication

[1] `https://microservices-demo.github.io`

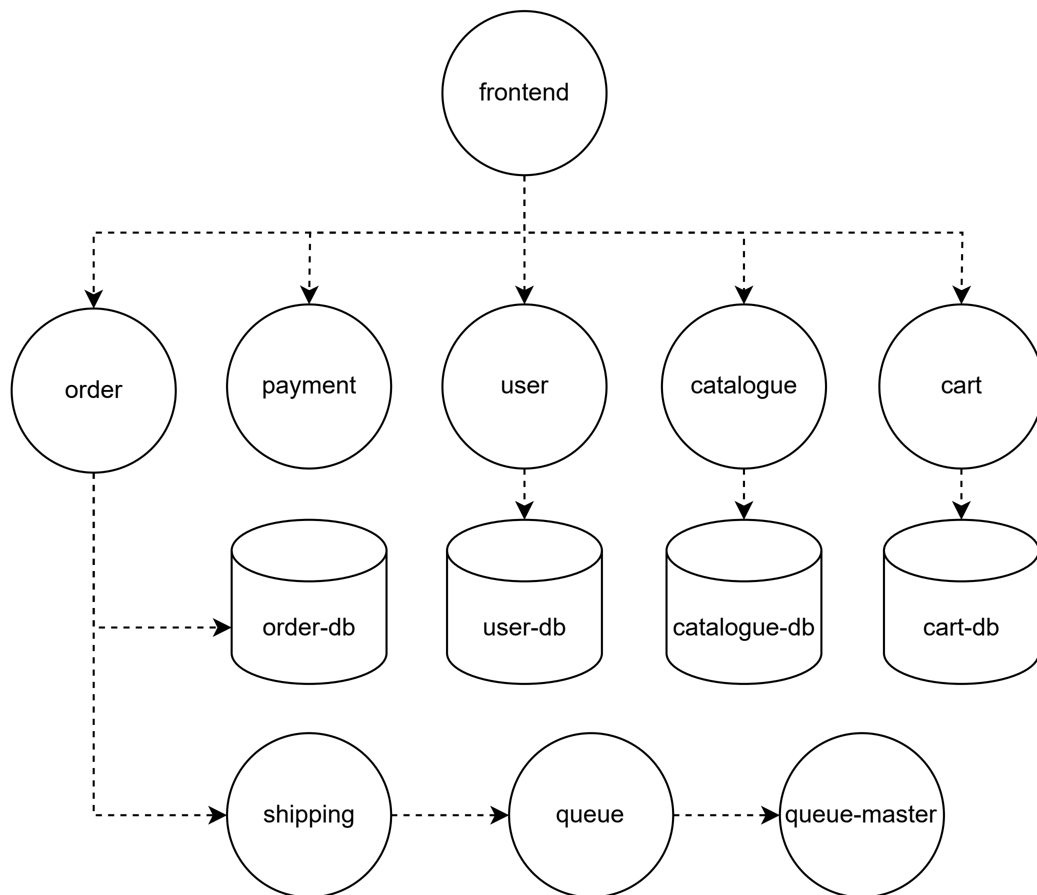

Figure 5.1: Sock Shop application

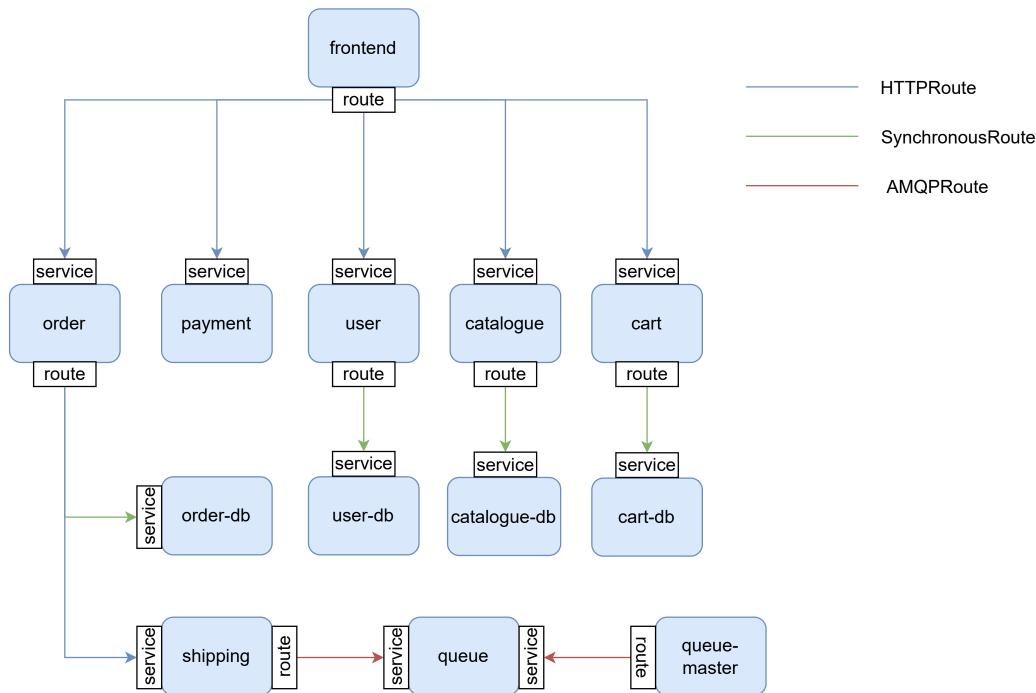

Figure 5.2: Sock Shop application TOSCA service template

with them is a synchronous TCP communication. The *queue* service is a message broker and the communication between this service and the *shipping* and *queue-master* services is an asynchronous AMQP communication.

Figure 5.2 shows the TOSCA service template of the Sock Shop application. Each service is modeled as a node template, while the communication channels between services are modeled as relationship templates. Each node template is an instance of the *Service* node type, defined in the Puccini TOSCA profile, and each relationship template, that associates a *route* requirement with a *service* capability, is an instance of one of the relationship types presented in chapter 4. Starting from the TOSCA service template, the Puccini compiler creates the corresponding Deployment and Service re-

sources and assigns static affinity labels to each Pod template based on the relationship templates of the service template. For each node template representing a database server the compiler creates a Deployment and a Service for the corresponding output proxy.

Figure 5.3 shows the test bed environment for the experiments. It consists of a node where the database servers are executed and a Kubernetes cluster with one master node and six worker nodes being part of three different layers: Cloud, Fog and Edge. These nodes are deployed as virtual machines on a Proxmox[2] physical node and configured with different resources: 16GB of RAM and 4 vCPU for the node in the Cloud layer, 8GB of RAM and 2 vCPU for the two nodes in the Fog layer and 4GB of RAM and 1 vCPU for the three nodes in the Edge layer. In order to simulate a realistic Cloud-to-Edge continuum environment with geo-distributed nodes, network latencies between nodes being part of different layers and between Edge nodes are simulated by using the Linux traffic control (*tc*[3]) utility. By using this utility network latency delays are configured on virtual network cards of the nodes. Output proxies associated with the corresponding database servers are forced to be scheduled on the Cloud layer worker node by using a node affinity rule.

Black box experiments are conducted by evaluating the end-to-end response time of the Sock Shop application when HTTP requests are sent to the frontend service with a specified number of virtual users each sending one request every second in parallel. Requests to the application are sent through the k6 load testing utility[4]. Each experiment consists of 10 trials,

[2]`https://www.proxmox.com`
[3]`https://man7.org/linux/man-pages/man8/tc.8.html`
[4]`https://k6.io`

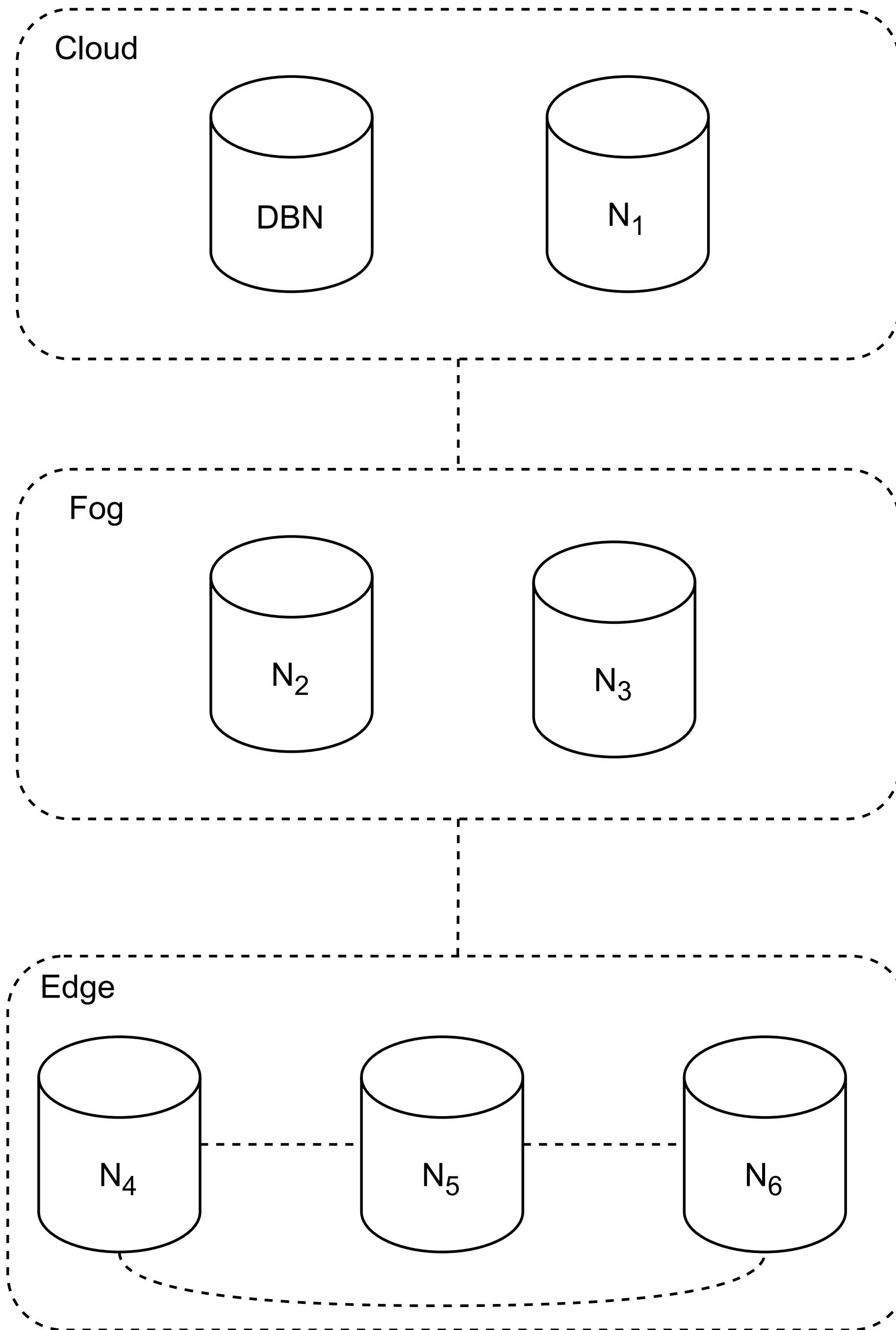

Figure 5.3: Test bed environment

during which the k6 tool sends requests to the frontend service for 40 minutes. For each trial, statistics about the end-to-end application response time are measured and averaged with those of the other trials of the same experiment. The trial interval is partitioned into 5 minutes sub-intervals, during which the k6 tool sends requests to a different application API. This way a realistic scenario with variability in the user requests distribution and the application usage patterns is simulated by activating different microservice channels at different time intervals. Furthermore, during each sub-interval the k6 tool sends requests to a different worker node and in the last sub-interval requests are equally distributed to all the nodes. By sending user requests to different nodes geo-distribution of users is simulated. For each experiment we compare both cases when our cluster and application monitoring operators and custom scheduler and descheduler components are deployed on the cluster and when only the default Kubernetes scheduler is present. We consider five different scenarios based on the network latency between the nodes of the different layers and between the Edge nodes: 10ms, 100ms, 200ms, 300ms and 500ms. In all the scenarios the α and β parameters of the ResourceAware plugin of the custom scheduler are assigned the same value of 0.5 in order to make the CPU and memory percentage differences between the respective resource availability on cluster nodes and the resource usage of Pods contribute equally to the *rascore* variable value. Similarly the γ and δ parameters are assigned the same value of 0.5 in order to make the *rascore* and *nascore* variables values, determined by the ResourceAware and NetworkAware plugins respectively, equally contribute to the final node score.

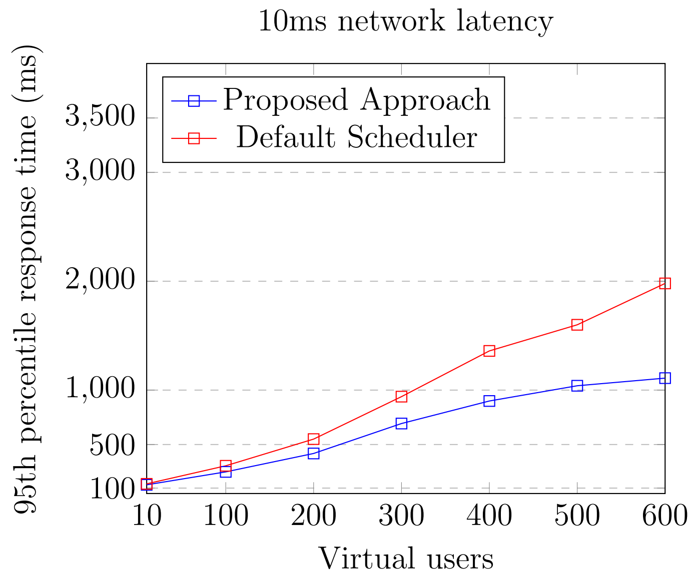

Figure 5.4: Experiments results (Scenario 1)

5.2 Experiments and results

Figures 5.4, 5.5, 5.6, 5.7 and 5.8 illustrate the results of the five experiments performed, each for a different scenario, showing the 95th percentile of the application response time as a function of the number of virtual users that send requests to the application in parallel. In all the cases, the proposed approach performs better than the default Kubernetes scheduler with average improvements of 39%, 56%, 66%, 68% and 70% in the five scenarios respectively. In the first scenario, network communication has no significant impact on the application response time because of the low node-to-node network latencies. Thus, the proposed network-aware scheduling strategy does not lead to high improvements in the application response time. Furthermore, for a low number of virtual users the proposed approach has similar performances

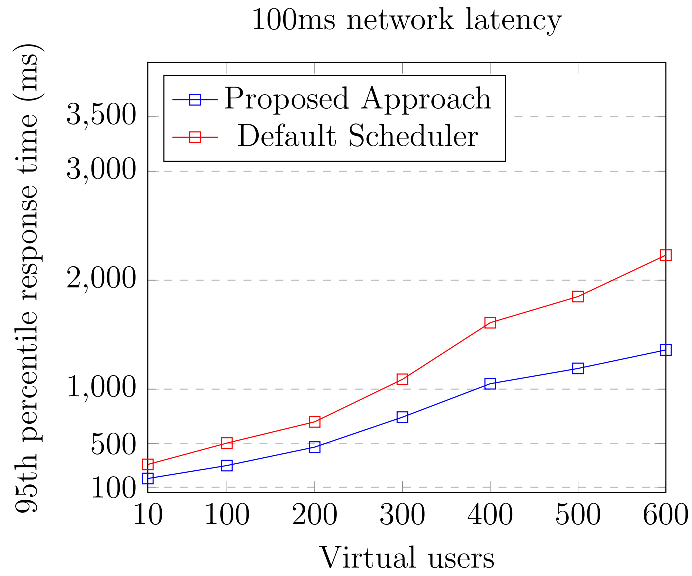

Figure 5.5: Experiments results (Scenario 2)

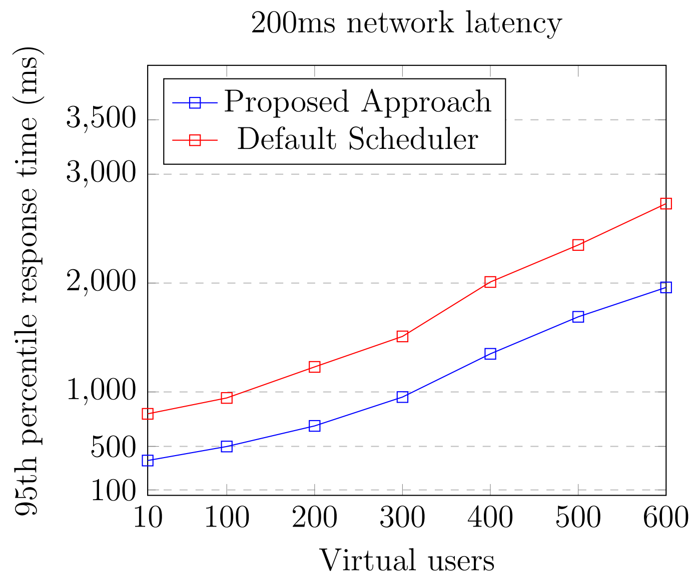

Figure 5.6: Experiments results (Scenario 3)

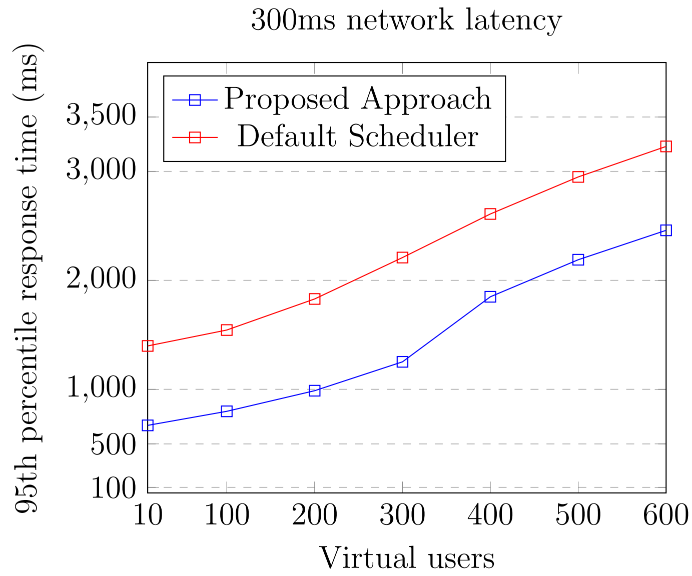

Figure 5.7: Experiments results (Scenario 4)

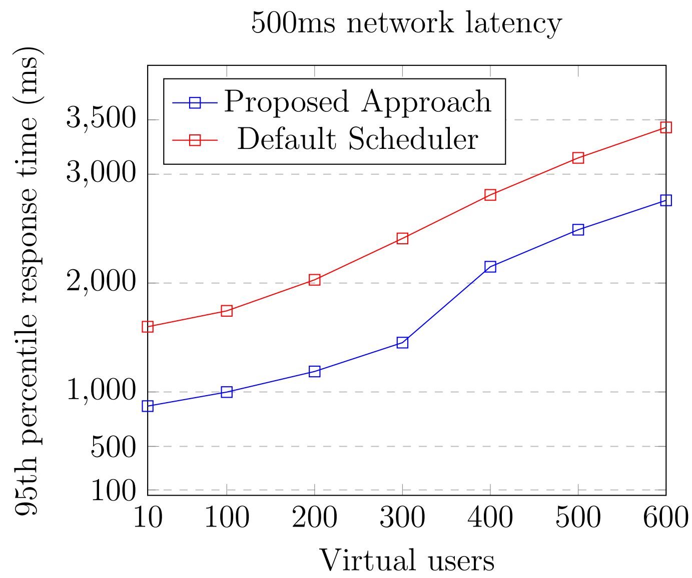

Figure 5.8: Experiments results (Scenario 5)

to the default scheduler. This is because of the limited shared resource interference between Pods though they are placed on the same nodes by the default scheduler. However, when the number of virtual users increases, the proposed approach performs better than the default scheduler, with higher improvements for higher numbers of virtual users. The response time in the case of the default scheduler grows faster than in the case of the proposed approach. This is because of the proposed resource-aware scheduling strategy that distributes Pods on cluster nodes based on their run time resource usage, then reducing the shared resource interference between Pods. In the other scenarios, network communication becomes a bottleneck for the application response time and the lack of a network-aware scheduling strategy leads to high response times. In these scenarios our approach performs better than the default scheduler for low numbers of virtual users also, with higher improvements for higher node-to-node network latencies. One consequence of the combination of both a resource-aware and a network-aware scheduling strategy in our approach is that when the number of virtual users increases the response time grows much faster for high network latencies, though it remains lower than the response time in the case of the default scheduler. This can be explained by the fact that, for a higher number of virtual users and then for a higher request load, the average resource usage of microservices increases and then the distribution of Pods among cluster nodes caused by the resource-aware scheduling strategy is higher. This leads to an increase in the network latency between application microservices and then in the end-to-end application response times.

Conclusions

In this work we proposed Sophos, in order to extend the Kubernetes platform with an application and infrastructure-aware orchestration strategy. The main goal is to overcome the limitations of the Kubernetes static scheduling policies when dealing with the placement of microservices-based applications that are executed in the Cloud-to-Edge continuum. The idea is to make the Kubernetes scheduler aware of the run time communication intensity between microservices and their resource usage, and the cluster network conditions to make scheduling decisions that aim to reduce the overall application response time. Furthermore, a descheduler is proposed to dynamically reschedule microservices if better scheduling decisions can be made based on the ever changing application and cluster network states.

As a future work we plan to extend the scheduler and descheduler components in order to define SLO-aware orchestration strategies. By specifying an SLO in the application performances it is possible to define an equilibrium condition that is reached when that SLO is guaranteed and reschedule the application only when that condition is not met. Furthermore, we plan to improve our scheduling strategy by improving the analysis of the application telemetry data. At the moment only the relationships between Pods

that communicate directly are evaluated by measuring the traffic exchanged between them. However, considering the fact that in a microservices-based application a user request traverses a chain of requests, placing Pods that belong to the same request chain near to each other may improve the application response time. In this context, we plan to make use of distributed tracing techniques to find relationships between Pods that do not communicate directly.

Bibliography

[1] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[2] Nist. The nist definition of cloud computing 2011. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf, 2011.

[3] Flexera. Rightscale state of the cloud report 2019. https://bit.ly/RightScaleReport, 2019.

[4] Azure. Microsoft azure global infrastructure. https://azure.microsoft.com/en-in/explore/global-infrastructure, 2023.

[5] Intel. Intelligent decisions with intel internet of things. https://intel.ly/32ybEs2, 2018.

[6] IoT Analytics. State of the iot 2018. http://bit.ly/State_IoT, 2018.

[7] CLAudit. Planetary-scale cloud latency auditing platform 2017. http://claudit.feld.cvut.cz, 2017.

[8] Mohammed S. Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. Toward low-latency and ultra-reliable virtual reality. *IEEE Network*, 32(2):78–84, 2018.

[9] Nik Bessis and Ciprian Dobre. *Big Data and Internet of Things: A Roadmap for Smart Environments*. Springer Publishing Company, Incorporated, 2014.

[10] Paul Wood, Heng Zhang, Muhammad-Bilal Siddiqui, and Saurabh Bagchi. Dependability in edge computing. *CoRR*, abs/1710.11222, 2017.

[11] Asif Laghari, Awais Jumani, and Rashid Laghari. Review and state of art of fog computing. *Archives of Computational Methods in Engineering*, 28, 02 2021.

[12] Sergej Svorobej, Malika Bendechache, Frank Griesinger, and Jörg Domaschka. *Orchestration from the Cloud to the Edge*, pages 61–77. Springer International Publishing, Cham, 2020.

[13] Asif Laghari, Rashid Laghari, Asif Wagan, and Aamir Umrani. Effect of packet loss and reorder on quality of audio streaming. *ICST Transactions on Scalable Information Systems*, 7, 09 2019.

[14] Gopika Premsankar, Mario Francesco, and Tarik Taleb. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal*, PP:1–1, 02 2018.

[15] Kashif Bilal, Osman Khalid, Aiman Erbad, and Samee U. Khan. Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. *Computer Networks*, 130:94–120, 2018.

[16] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[17] D. Calcaterra, G. Di Modica, and O. Tomarchio. Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. *Journal of Cloud Computing*, 9(49), 2020.

[18] Mohammad Aazam, Sherali Zeadally, and Khaled A. Harras. Offloading in fog computing for iot: Review, enabling technologies, and research opportunities. *Future Generation Computer Systems*, 87:278–289, 2018.

[19] Cheol-Ho Hong and Blesson Varghese. Resource management in fog/edge computing. *ACM Computing Surveys*, 52(5):1–37, sep 2019.

[20] Daniel Maniglia A. da Silva, Godwin Asaamoning, Hector Orrillo, Rute C. Sofia, and Paulo M. Mendes. An analysis of fog computing data placement algorithms. In *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous '19, page 527–534, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Karima Velasquez, David Perez Abreu, Marcio R. M. Assis, Carlos Senna, Diego F. Aranha, Luiz F. Bittencourt, Nuno Laranjeiro, Marilia Curado, Marco Vieira, Edmundo Monteiro, and Edmundo Madeira. Fog orchestration for the internet of everything: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 9(1):14, Jul 2018.

[22] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. Challenges and opportunities in edge computing. In

2016 IEEE International Conference on Smart Cloud (SmartCloud), pages 20–26, 2016.

[23] Claus Pahl, Nabil El Ioini, Sven Helmer, and Brian Lee. An architecture pattern for trusted orchestration in iot edge clouds. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 63–70, 2018.

[24] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34, 2015.

[25] Roberto Morabito. Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, 5:8835–8850, 2017.

[26] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures – a technology review. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 379–386, 2015.

[27] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[28] Emiliano Casalicchio. *Container Orchestration: A Survey*, pages 221–235. Springer International Publishing, Cham, 2019.

[29] Zeineb Rejiba and Javad Chamanara. Custom scheduling in kubernetes: A survey on common problems and solution approaches. *ACM Comput. Surv.*, 55(7), dec 2022.

[30] Carmen Carrión. Kubernetes scheduling: Taxonomy, ongoing issues and challenges. *ACM Comput. Surv.*, 55(7), dec 2022.

[31] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Towards network-aware resource provisioning in kubernetes for fog computing applications. In *IEEE Conference on Network Softwarization (NetSoft)*, pages 351–359, 2019.

[32] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications*, 159:161–174, 2020.

[33] Mohamed Rahali, Cao-Thanh Phan, and Gerardo Rubino. Krs: Kubernetes resource scheduler for resilient nfv networks. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2021.

[34] Agustín C. Caminero and Rocío Muñoz-Mansilla. Quality of service provision in fog computing: Network-aware scheduling of containers. *Sensors*, 21(12), 2021.

[35] Lianjie Cao and Puneet Sharma. Co-locating containerized workload using service mesh telemetry. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 168–174, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Thomas Pusztai, Fabiana Rossi, and Schahram Dustdar. Pogonip: Scheduling asynchronous applications on the edge. In *IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 660–670, September 2021.

[37] Lukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–9, 2021.

[38] Stefan Nastic, Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Deepak Vii, and Ying Xiong. Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 206–216, 2021.

[39] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 932–941, 2021.

[40] Michael Chima Ogbuachi, Chinmay Gore, Anna Reale, Péter Suskovics, and Benedek Kovács. Context-aware k8s scheduler for real time distributed 5g edge computing applications. In *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6, 2019.

[41] David Haja, Mark Szalay, Balazs Sonkoly, Gergely Pongracz, and Laszlo Toka. Sharpening kubernetes for the edge. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, SIGCOMM Posters and Demos '19, page 136–137, New York, NY, USA, 2019. Association for Computing Machinery.

[42] László Toka. Ultra-reliable and low-latency computing in the edge with kubernetes. *Journal of Grid Computing*, 19(3):31, July 2021.

[43] Angelo Marchese and Orazio Tomarchio. Network-aware container placement in cloud-edge kubernetes clusters. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 859–865, 2022.

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

[45] Angelo Marchese. and Orazio Tomarchio. Sophos: A framework for application orchestration in the cloud-to-edge continuum. In *Proceedings of the 13th International Conference on Cloud Computing and Services Science - CLOSER*, pages 261–268. INSTICC, SciTePress, 2023.

[46] Damian A. Tamburri, Willem-Jan Van den Heuvel, Chris Lauwers, Paul Lipton, Derek Palma, and Matt Rutkowski. Tosca-based intent modelling: goal-modelling for infrastructure-as-code. *SICS Software-Intensive Cyber-Physical Systems*, 34(2):163–172, Jun 2019.

[47] OASIS. Topology and Orchestration Specification for Cloud Applications Version 2.0. http://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html, October 2020.