UNIVERSIDADE FEDERAL DE MINAS GERAIS
Departamento de Ciências da Computação
Programa de Pós-graduação em Ciências da Computação

Marcos Magno de Carvalho

QOE-AWARE CONTAINER SCHEDULING FOR CO-LOCATED CLOUD APPLICATIONS

Belo Horizonte
2021

Marcos Carvalho

QoE-Aware Container Scheduling for Co-located Cloud Applications

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Daniel Macedo

Belo Horizonte

2021

Carvalho, Marcos Magno de.

C331q QoE-aware container scheduling for co-located cloud applicationss [manuscrito] / Marcos Magno de Carvalho — 2021.
111 f. il.

Orientador: Daniel Fernandes Macedo.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituro de Ciências Exatas, Departamento de Ciência da Computação
Referências: f.100-111

1. Computação – Teses. 2. Computação em nuvem – Teses. 3. Aprendizado profundo – Teses. 4. Transmissão de vídeo – Teses. I. Macedo, Daniel Fernandes. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*22 (043)

Ficha Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

QoE-Aware Container Scheduling For Co-Located Cloud Applications

MARCOS MAGNO DE CARVALHO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. DANIEL FERNANDES MACEDO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. JOSÉ MARCOS SILVA NOGUEIRA
Departamento de Ciência da Computação - UFMG

PROF. MAGNOS MARTINELLO
Departamento de Informática - UFES

Belo Horizonte, 1 de Outubro de 2021.

To my mother, Maria Lucia, my brother, Mateus Henrique, and my beloved Taciana Cristina, for their unconditional support and trust.

Agradecimentos

To my mother for her love and prayers. The comfort that the difficult days will be worth all our effort. My brother, for your care and inspiration, to continue on my way. Thank you for understanding during this period when I was away from you.

To my girlfriend and future wife for her affection and support. How many weekends without seeing each other?! I am immensely grateful for the fellowship and for believing in my dreams. You strengthened me to make it to the end.

To my advisor Daniel, for the invaluable assistance and guidance. We started working together before me becoming a master's student at UFMG. You have always been solicited and encouraged to participate in my academic journey. Thanks for everything else.

To my colleagues at the Winet laboratory, especially Vinicius Fonseca, Erik, Luis Henrique Cantelli, Henrique Moura, Gilson, Racyus (thanks for the mattress in the lab), among others who welcomed me to this university. You were my support in various difficulties encountered along the way and provided several good moments.

To the DCC professors and staff, for their teaching and support, especially to Sônia and professors José Marcos, Flip, and Flop.

To the CNPq research funding agency for making my dedication to my studies financially viable.

To all those who contributed in some way to the accomplishment of the work.

"Research is to see what everybody else has seen, and to think what nobody else has thought."

(Albert Szent-Györgyi)

Resumo

A Computação em nuvem tem sido bem-sucedida em fornecer recursos de Computação para implantar aplicações altamente disponíveis para vários provedores de conteúdo (clientes em nuvem). Nesse caso, para melhorar o uso de recursos, o provedor de nuvem tende a compartilhar seus recursos computacionais entre diferentes clientes, co-localizando aplicações no mesmo servidor. No entanto, aplicações co-localizadas geram interferência entre elas, o que pode causar degradação nas aplicações. Além disso, cada aplicação exige um diferente tipo de recurso e desempenho, o que torna o gerenciamento de recursos ainda mais complexo.

Para mitigar isso, o processo de escalonamento de contêineres utiliza métricas de Qualidade de Serviço (Quality of Service - QoS), que são pré-estabelecidas e especificadas nos Objetivos de Nível de Serviço (Service Level Objectives - SLO). No entanto, para aplicações em que a experiência dos usuários é importante e mensurável, o SLO baseado em QoS é insuficiente para garantir aos usuários finais uma boa qualidade de experiência (QoE). Isso ocorre porque as métricas de QoS não refletem corretamente a experiência dos usuários.

A proposta desta dissertação trata desse problema, propondo um agendador/reagendador de contêiner ciente da QoE em um ambiente onde as aplicações estão co-localizadas. Para tanto, propomos uma nova abordagem que considera as métricas da nuvem para estimar a QoE que a nuvem pode oferecer. Além disso, propomos o uso de QoE como uma métrica de desempenho no SLO e um algoritmo que usa a estimativa da QoE para realizar o agendamento/reagendamento dos contêineres. Finalmente, realizamos uma avaliação experimental da nossa proposta considerando duas aplicações diferentes de streaming de vídeo. Os resultados obtidos mostram que o agendamento com reconhecimento da QoE pode aumentar a QoE dos usuários, além de melhorar outros fatores da QoE, como travamento e mudanÃ§a de resolução. Além disso, nossos resultados mostraram que nosso agendador/reagendador foi capaz de reduzir a quantidade de recursos utilizados.

Keywords: Computação em Nuvem, Agendador de Container, Aprendizado Profundo, Transmissão de Vídeo.

Abstract

Cloud computing has been successful in providing computing resources to deploy highly available applications for multiple content providers (cloud customers). In this case, to improve resource usage, the cloud provider tends to share its computing resources between different customers, co-locating applications on the same server. However, co-located applications generate interference with each other, which can cause degradation of the applications. Furthermore, each application demands a different type of resource and performance, which makes resource management even more complex.

To mitigate this, the container scheduling process uses metrics based on Quality of Service (QoS), which are pre-established and specified in the Service Level Objectives (SLO). However, for applications where users' experience is important and measurable, QoS-based SLO is insufficient to guarantee end-users good Quality of Experience (QoE). This is because the QoS metrics do not correctly reflect the users' experience.

The proposal of this dissertation deals with this problem, proposing a QoE-aware container scheduler/rescheduler in an environment where applications are co-located. To that end, we propose a new approach that considers cloud metrics to estimate the QoE that the cloud can offer. Furthermore, we propose using QoE as a performance metric in SLO and an algorithm that uses QoE estimation to perform the container scheduling/rescheduling. Finally, we carried out an experimental evaluation of our proposal considering two different streaming video applications. The results obtained show that QoE-aware scheduling can increase users' QoE, in addition to improving other QoE factors, such as stall event and resolution change. Furthermore, our results showed that our scheduler/reschedule was able to reduce the amount of resources used.

Keywords: Cloud Computing, Container Scheduler, Deep Learning, Video Streaming.

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has become one of the most widespread technologies in recent years, and consequently, cloud-native applications have been growing such as the Internet of Things (IoT), machine learning-driven applications, and streaming audio/video applications [Ahmad et al., 2021]. Furthermore, with the advent of 5G networks, more applications will be hosted on the cloud, such as Augmented Reality (AR), Virtual Reality (VR), and Internet of Vehicles applications [Krogfoss et al., 2020; Qureshi et al., 2020].

The cloud computing ecosystem comprises two agents: first, the cloud provider such as AWS, Google Cloud, and Microsoft Azure, which provides the resources (CPU, memory, disk, and network). Second, cloud customers (e.g., Netflix, Dropbox, Spotify), which use cloud resources to host the applications to serve their clients. With that, the cloud providers render their resources to their customers as Infrastructure As a Services (IaaS) and employ a pay-per-use model [Madni et al., 2017].

The IaaS model is characterized by computing resources shared among many cloud customers simultaneously. This means that the cloud providers deploy different co-located applications via virtualization technologies, such as container-based virtualization [Tosatto et al., 2015]. This technology allows multiple applications to run on the same server, in which they are co-located to improve resource utilization [Ren et al., 2018]. In addition to sharing cloud resources, another essential characteristic of the IaaS model is horizontal elasticity [Al-Dhuraibi et al., 2017]. In this case, IaaS permits the use of resources according to demand, which means that the number of containers allocated can be adjusted to deal with the applications' workload variations.

Although the cloud provider wants to improve its resources usage through co-locating applications, in practice, there is a trade-off between high resource utilization and interference between applications precisely because they are co-located. In other

words, co-located applications cause interference between them, which leads to performance degradation when an application demand exceeds the resources available on the shared host [Medel et al., 2017]. Hence, the likelihood of performance degradation increases with the degree of application co-location [Guo et al., 2019]. In addition, latency-sensitive (online) applications are the most affected [Zhao et al., 2019]

Despite this problem, the cloud provider needs to guarantee the performance level to their customers, which is pre-established in the Service Level Objectives (SLOs) specified in the Service Level Agreement (SLA). Traditional cloud management systems use different Quality of Services (QoS) metrics for each type of application on the SLOs configuration [Elhabbash et al., 2019]. This is because each application requires a different type of resource; for example, while some applications need more compute resources (e.g., data analysis, artificial intelligence, and data warehousing), others require more network resources (e.g., media streaming, web applications) [Mei et al., 2020]. To deal with this heterogeneity, the applications are distributed between the various cloud servers by the container scheduling process [Zhong et al., 2020]. However, as mentioned, the co-location of different workloads with different QoS requirements generates interference among them, especially in latency-sensitive applications.

Furthermore, a QoS-based SLO is often insufficient for applications where a good Quality of Experience (QoE) is essential for the users and fundamental for the content provider's profits [Haouari et al., 2019]. This is because the QoS metrics themselves do not accurately reflect the end-user experience [Juluri et al., 2015]. In other words, it is hard to define the optimal values of QoS metrics to reach the desired QoE [Kafetzakis et al., 2012]. In contrast, direct measuring QoE has become a more robust method to understand end-user experience and engagement [De Cicco et al., 2019]. The most common concept to measure QoE is Mean Opinion Score (MOS) values, proposed by ITU-T Recommendation P. 10/G.100. This proposal maps the MOS score continuously from 1 to 5, where the satisfaction level increases with the MOS value.

Nevertheless, several works explore QoE management applied to network management, omitting how QoE can be measured and then exploited within the cloud computing environment to improve the end-to-end users' QoE [Barakabitze et al., 2019]. Therefore, it is essential to create QoE-aware cloud management techniques, particularly considering QoE in the container scheduling process.

1.1 Motivation

Several content providers, such as Netflix, Youtube, and Spotify, are migrating their applications to the cloud environment due to its advantages in terms of cost and available computing resources [Abdallah et al., 2018; Soldani et al., 2018]. However, as mentioned, cloud providers share their computing resources among their customers, which can cause degradation among latency-sensitive applications [Zhao et al., 2019]. Nevertheless, several works use QoS metrics and different objectives (such as maximization resource usage) to perform container scheduling to mitigate this degradation, but these approaches do not reflect a good end-user experience [Juluri et al., 2015]. Besides that, the works presented in the literature that aiming to improve users' QoE do not consider the degradation caused by the co-located application in the final user's QoE. In general, these works measure QoE in another part of the users' end-to-end path, such as in Wi-Fi networks.

Therefore, the motivation of this dissertation is based on the lack of QoE-aware container-based solutions, as will be described in Chapter 3. In addition, the use automatic container scheduling for QoE improvement may be of interest to the cloud providers, which can be a differentiator when it comes to offering their services.

1.2 Objectives

The general objective of this work is to propose a QoE-aware container scheduler for multiple colocated QoE-aware applications on the cloud environment. To achieve this goal, the general objective was broken down into the following specific objectives:

- Propose an architecture for container scheduling using QoE estimators.
- Evaluate and develop machine learning models to estimate the user's QoE for different applications.
- Develop algorithms to improve the user's QoE by choosing the best server for container deployment.

1.3 Contributions

This dissertation proposes using QoE estimators to perform container scheduling and rescheduling in a cloud environment. The scheduler is evaluated in an experimental environment considering two different video transmission applications; video-on-demand

and a specific type of live stream application. This work considers the video application due to its high growth on the internet in recent years [Abdallah et al., 2018]. Also, as forecasting suggests, global video traffic will be 82% of all IP traffic by 2022 [Cisco, 2017]. Despite that, other applications could be supported as well, as long as there is a suitable QoE predictor for that application, for example, Web browsing and audio streaming. The main contributions present in the proposed work are presented below.

- A QoE-aware container scheduling and rescheduling for co-located cloud applications.
- A management system that uses QoE objectives as SLO metrics and extends the Kubernetes Scheduler.
- Two novel machine learning estimators to predict users' QoE on video-on-demand and live virtual classroom application in cloud environments, following the ITU-P Recommendation P.1203 [ITU Telecommunication Standardization Sector, 2017]. To the best of our knowledge, this is the first work that uses a set of cloud computing resources (CPU, memory, disk, network) to predict QoE within the cloud.
- The quantitative evaluation of our proposals in an experimental environment.

1.4 Organization

This work is structured into seven chapters, including this introduction. The next chapter introduces background concepts, covering cloud computing, Kubernetes Engine, QoE, and ITU-T Recommendation P.1203. Also, we detail the two applications used in this work and the Machine Learning techniques used to create each estimator. Chapter 3 presents related work about QoE-aware cloud resource management and QoE-aware container scheduling. Chapter 4 describes the proposed system architecture, including the cloud environment as a data plane and our scheduler as the control plane. Chapter 5 presents the data collection process and model building methodology as well as the model result. Chapter 6 shows our evaluation in an experimental environment. Finally, Chapter 7 presents the final considerations, the conclusions, and future work.

Chapter 2

Background

This chapter discusses some basic concepts related to this dissertation. First, in section 2.1, we introduce the cloud computing architecture and cloud resource management (2.1.1). Then, section 2.2 presents the Kubernetes platform, its components, and the horizontal auto-scaling and resource scheduling processes. Next, section 2.3 presents concepts related to the Quality of Experience (QoE) and ITU-T Recommendation P1203. Section 2.4 presents the two applications used in this work, being VoD and Live Virtual Classrooms respectively in 2.4.1 and 2.4.2.

A series of QoE influencing events such as rebuffering and rate adaptation makes the QoE dynamic and continuously time-varying and, therefore, makes the QoE prediction a challenging task that requires sophisticated machine learning methods. Because of this, we performed a pre-study to define the best technique to deal with continuous QoE prediction and presented this technique in this chapter. First, we discussed the basic machine learning concepts in 2.5 and, finally, we presented the Deep Learning technique used in this work in 2.5.1.

2.1 Cloud Computing

New technologies have rapidly developed for data processing, storage, and data transfer over the internet. Moreover, computing resources have become cheaper and more advanced than ever (Zhang et al. [2010]). With those characteristics emerged a new computing model called Cloud Computing [Maenhaut et al., 2020]. The National Institute of Standards and Technology (NIST)[1] establishes cloud computing as a model that allows access to a shared set of cloud resources, such as servers, storage, and network.

[1]https://csrc.nist.gov/publications/detail/sp/800-145/final

These resources are provisioned on-demand and released with minimal management effort or cloud provider cooperation.

Cloud computing can be separated into different types, being: *public cloud*, *private cloud*, and *hybrids* [Maenhaut et al., 2020]. Each type has its own peculiarities, which will be discussed below. In the *public cloud*, there are two agents: the cloud provider (e.g., AWS, Google Cloud, and Microsoft Azure), which provides the computing and network resources; and the cloud customers (e.g., Netflix, Youtube, Dropbox), which use these resources to provide an application to their clients. In a public cloud, several cloud customers share the cloud resources among them. On the other hand, the *private cloud* consists of cloud computing resources used exclusively by one cloud customer (organization) on a private network. Besides that, the private cloud can be deployed within either the customer's infrastructure or on a third-party cloud provider [Zhang et al., 2016]. In the first case, the cloud provider and cloud customer are the same agents. Lastly, a *hybrid cloud* comprises of private and public clouds, which means that this cloud type combines the local infrastructure or private cloud with the public cloud, sharing data among them.

In terms of business model, the cloud allows their customers to use computing/networking resources in the form of three services: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [Zhang et al., 2010]. This set of services is divided into four layers, being: (1) hardware layer, (2) infrastructure, (3) platforms, and (4) applications, as illustrated in Figure 2.1.

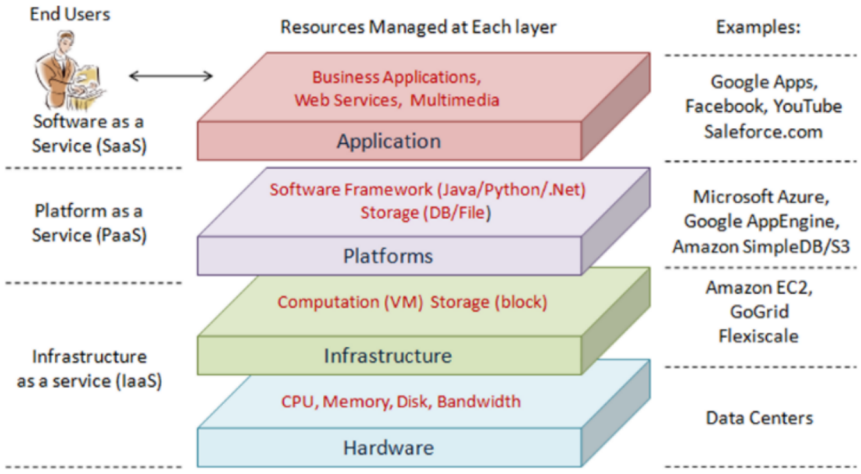

Figure 2.1. Cloud computing architecture (Zhang et al. [2010])

The **Hardware layer** refers to cloud physical resources, including servers, routers, switches, computing and network resources, such as CPU, disk and bandwidth. These resources are located in data centers. At a higher level, the **Infrastructure**

layer contains computing/network resources virtualized by Virtual Machines (VM) using software like Kernel-based Virtual Machine (KVM)[2] or emerging container technologies such as Docker[3] and Linux Containers (LXC)[4]. Section 2.2 begins by discussing the difference between VMs and containers.

The Platform layer is a software framework that bridges cloud customers and the cloud provider resources. It is through this layer that the cloud customers are authenticated and create instances of their resources. Finally, the applications offered by cloud customers to their clients are in the **Application layer**. Cloud applications take advantage of better performance from automatic resource scalability and lower operating cost, ensuring on-demand delivery, unlike traditional applications.

The main factor motivating to use cloud computing is its ability to provide computing and network resources according to their customer's needs [Al-Dhuraibi et al., 2017], such as on-demand resource scalability, and pay-per-use models. Besides that, cloud providers guarantee customers a performance level established in a Service Level Agreement (SLA) that specifies a set of Service Level Objectives (SLOs). The SLA is a document that contains the terms for the provided services, whereas the SLOs are usually composed of one or more Quality of Service (QoS) measurements [Elhabbash et al., 2019].

To illustrates, Table 2.1 shows an example of SLAs content and its respective SLOs adapted from Elhabbash et al. work. The table resulted from a survey of relevant ISO standards, NIST, and ETSI publications and surveys.

Furthermore, different types of applications need different QoS metrics as well as SLOs. For example, some applications require more computing resources, and others require more network resources. Therefore, one way to guarantee the SLO for each application is to use cloud scalability, balancing the application's resources between the various cloud servers by scheduling resources. However, this creates an environment where applications are co-located in the same server, generating interference among them, especially in latency-sensitive applications. Therefore, cloud management is essential in this scenario to guarantee the SLO agreement and improve cloud resource usage. The following section discusses the main terms of cloud management, such as resource scheduling, and then the next section details the most used container-based cloud platform, named as Kubernetes.

[2]www.linux-kvm.org/page/MainPage
[3]https://www.docker.com/
[4]https://linuxcontainers.org/pt$_br$/

Table 2.1. Cloud SLA contents and their SLOs

Content	Sub-content	SLOs
Availability	n/a	Availability, Interval-availability Maximum Processing Time
Performance	Response time	Maximum Response Time Response Time Mean Response Time Variance
	Capacity	Disk space CPU power Memory size, Throughput Bandwidth
	Elasticity	Elasticity speed threshold Elasticity Precision threshold
Reliability	Service resilience / fault tolerance	Time to Service Recovery Mean Time to Service Recovery
Cost	n/a	Cost per Time Unit
Support	n/a	Support Hours

2.1.1 Cloud Resource Management

Next-generation cloud systems will require new methodologies to deal with cloud resource management. Conventional control and management systems face considerable challenges, mainly to offer appropriate QoS and QoE levels [Al-Dhuraibi et al., 2017].

In terms of cloud resource management, the literature proposes three main categories [Singh and Chana, 2016a], as in Figure 2.2, namely, (i) Resource Provisioning, (ii) Resource Scheduling, and (iii) Resource Monitoring. Below we present each one.

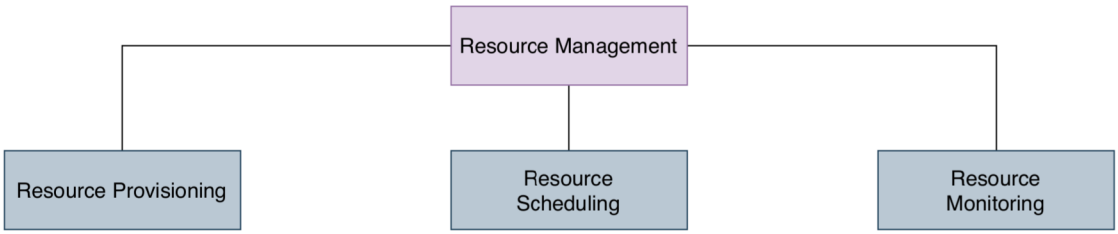

Figure 2.2. Taxonomy of resource management in the cloud.

Resource provisioning: it allocates virtualized resources to cloud customers. When the cloud provider accepts a request to create new resources (such as VM or container), the provisioning system allocates it for the customer. Besides that, provi-

sioning permits the use of resources according to demand; that is, if the demand for the cloud resources at a given time exceeds the initially reserved value, then additional resources are provisioned. When it occurs, the system can scale the resources into two forms: either vertically, adding more CPU cores, memory, or bandwidth into the already allocated resources, or, horizontally, that consist in adding new instances of the same resources associated with an application [Qu et al., 2018]. Thus, the two types of scaling are known as elasticity, one of the main characteristics of cloud computing. Lastly, provisioning has different purposes, such as improving performance and resource utilization, saving energy, reducing cost, and ensuring availability [Al-Dhuraibi et al., 2017].

Resource scheduling: is a dynamic allocation process of virtualized resources (VM/container) after its provisioning Adhikari et al. [2019], which corresponds to placing a set of newly instantiated virtual resources into the cloud's physical infrastructure. This can be formally explained as follows: The cloud environment consists of n servers that are represented as $S = \{S_1, S_2, S_3 \dots S_n\}$. Each server may have m virtualized resources that serve the cloud customers, represented as $R = \{R_1, R_2, R_3 \dots R_m\}$. Besides, the cloud system dynamically receives multiple requests $T = \{T_1, T_2, T_3 \dots T_x\}$ to allocate new resources R_i to deal with workload variation. In this scenario, resource scheduling has one main responsibility: choose in which server S_i to allocate a resource R_i.

The mapping of R_i with an appropriate server is done from the viewpoint of cloud provider efficiency [Zhan et al., 2015]. In addition to that, several works consider resource scheduling based on the user's quality of service (QoS) requirements ([Singh and Chana, 2016b]). This means that resource scheduling has to meet various QoS parameters, such as high availability, computing capacity, security, execution time, and cost. Besides that, to ensure SLA/SLO upon system/resource failures, including unpredictable crashes and performance degradation due to resource availability, the cloud system performs a process known as **Resource Rescheduling** [Yao et al., 2016]. This process deletes the resource (VM / Container) from the original server experiencing some technical problem and schedules it again on another server.

Resource monitoring: refers to physical or virtual resource measurement in real-time. The monitoring information includes available and total resource usage for each resource component, such as CPU, memory, network, and storage. Monitoring resource usage is used to take care of important QoS requirements like security, availability, and performance. Besides that, resource monitoring is essential to provide adequate QoS and QoE for consumers [da Rosa Righi et al., 2019].

2.2 Kubernetes: Container orchestration

There are architectural differences between VMs and containers, as well as performance differences. While each VM includes a complete operating system (OS), containers share a single host OS, including the kernel and files, making them advantageous over full hardware virtualization [Kavitha and Varalakshmi, 2017]. Containers were designed to perform isolation with low overhead and fast start-up time. Because of that, containers are gaining widespread popularity among cloud providers [Al-Dhuraibi et al., 2017].

Kubernetes[5] is the most popular open-source container-based virtualization solution to manage Docker-based containers. It implements the main features of the cloud, such as automated deployment and scaling. Figure 2.3 shows the Kubernetes architecture in high level. The architecture is divided into a Control Plane, for managing the overall containerized workloads, and a Data Plane, which provides virtualized computing/networking resources. As shown in the architecture, the Control Plane is formed by a Master Node, while the Data Plane is composed of multiple Worker Nodes. Each node is either a physical or virtual machine. The node components are described below.

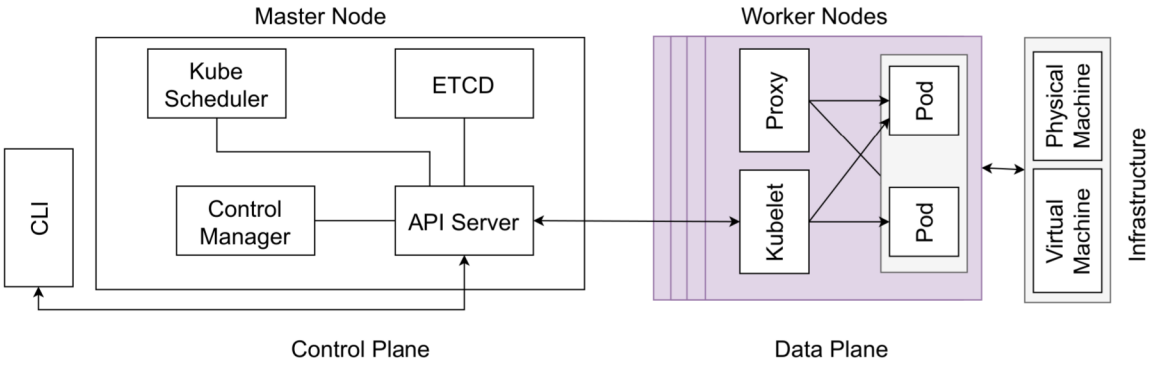

Figure 2.3. Kubernetes Architecure

Worker Node: In the nomenclature used in Kubernetes, Worker Node refers to a server (virtual/physical). Also, *pod* is another term explicitly used in Kubernetes. It is the primary deployment unit in Kubernetes, into which one or more containers can be created and grouped. This creates a cluster, which is a set of worker nodes running containerized applications inside pods. The containers inside a pod share the same IP Address, port space (namespace), and data volumes. Meanwhile, pods are isolated from each other. The end user's request is distributed to worker nodes according to load balancing rules. The *proxy* receives these requests and forwards them to *pods*.

[5]www.kubernetes.io/

Furthermore, *Kubelet* is the node agent that monitors the cluster state and reports it to the Control Plane.

Master Node: The Master Node manages the cluster functionality, such as pod creation, resource scheduling, and horizontal auto-scaling (discussed below). The cloud administrator can use the *kubectl* Command-line interface (CLI) and client libraries, such as Kubernetes Python Client[6] and client-Go[7], to control the clusters through an *API Server*.

Besides that, the *Control Manager* monitors the *etcd* storage component, which stores the cluster state. For example, if a *pod* stopped abruptly, the *Controller Manager* makes the necessary changes to restore the previous state, in this case, scaling another pod to replace it.

Finally, the *kube scheduler* (KS) is the default scheduling component, which schedules each container on a specific node in the data plane. It will be described below.

Kubernetes Network Model: Network communication in the Kubernetes cluster is pod-level communication, which means that communication is not done directly between containers. Also, Kubernetes itself does not implement the underlying networking, requiring a third-party plugin [Xu et al., 2018].

Kubernetes supports two kinds of network plugins: i) *Kubenet plugin*,which is a simple network plugin that works on Linux only. In this case, *Kubenet* creates a Linux bridge with the worker node. However, the cloud provider needs to set up routing rules for communication between nodes. ii) The second option is the *CNI (Container Networking Interface)* family of plugins, which is the most used solution for Kubernetes clusters [Kapočius, 2020]. CNI creates a network interface into the container network namespace and on the host, connecting the pods [Xu et al., 2018]. Figure 2.4(a) shows the network architecture for Kubernetes using a CNI plugin.

Several projects create CNI plugins, such as Calico[8], Weave[9], and Flannel[10]. Flannel is the most used network solution for the Kubernetes cluster [Zeng et al., 2017], [Park et al., 2018], and because of that, this work employs it in the experimental environment. Flannel provides network virtualization on layer 2 overlaid on a layer 3 network, using VXLAN as default installation and UDP overlay as choices. Operating in VXLAN mode, packets cross the boundary only once from the pod to the host [Qi et al., 2020]. Figure 2.4(b) shows the Flannel plugin used on Kubernetes. Flannel

[6]https://github.com/kubernetes-client/python
[7]https://github.com/kubernetes/client-go
[8]https://www.projectcalico.org/
[9]https://www.weave.works/docs/net/latest/kubernetes/
[10]https://github.com/flannel-io/flannelflannel

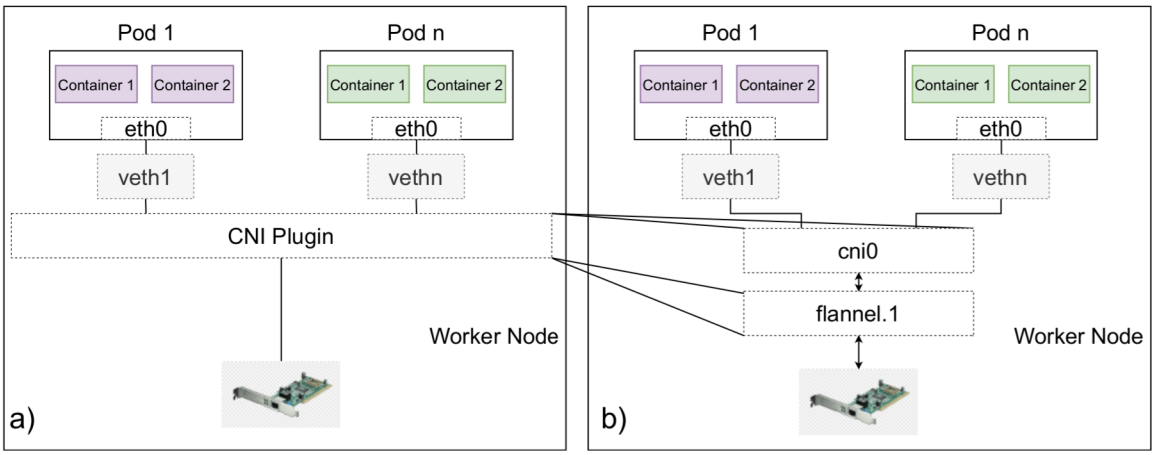

Figure 2.4. Network Architecture for Kubernetes - Adapted from [Xu et al., 2018]

creates two network interfaces, being *cni0* and *flannel.1*. The first one creates network connectivity of containers, and the second allocates a subnet to each worker node.

Kubernetes horizontal autoscaling: Autoscaling methods automatically and promptly provision and de-provision cloud resources without human intervention in response to dynamic fluctuations in workload [Imdoukh et al., 2020]. Thus, autoscaling reduces management overhead compared with an operator that monitors the system's performance and decides when adding or removing resources. In the context of Kubernetes, this automatic process is known as Horizontal Pod Autoscaling (HPA).

HPA performs reactive horizontal scaling (automatically) based on threshold-based rules referring to resource utilization (e.g., CPU and memory) [Al-Dhuraibi et al., 2017]. The threshold values are set in the HPA configuration. HPA is implemented as a control loop, where in each iteration, the *Controller Manager* requests (through an API Server) the Pod's resource usage to compare with the target value. When the current resource usage is greater than the target value, HPA allocates new replicas of the container. The containers to be allocated enter a waiting queue and remain in a *pending* status until the Kubernetes scheduler (KS) allocates them to a worker node.

Algorithm 1 Kubernetes Replicas Algorithm

Input: $U_{target}, ActivePods$
Output: P ▷ The number of Pods to deploy

1: **while** True **do**
2: **for all** $i \in ActivePods$ **do** ▷ $ActivePods$ for application j
3: $U_i \leftarrow getAverageMemoryUtilizations(i)$
4: $\mathbf{U} = \mathbf{U} \cup \{U_i\}$
5: $P \leftarrow ceil(currentReplicas * (sum(U)/U_{target})$
6: $wait(\tau)$ ▷ Wait τ seconds, the control loop period

However, to decide how many containers to scale, the HPA checks the ratio between the current *pod* metric and the target value, multiplied by the number of already allocated containers. Algorithm 1 shows how Kubernetes calculates the number of additional replicas to be deployed. For instance, consider the application j with one container already allocated, which average memory utilization is 200MB, and the target value is 100MB. In this case, the number of ideal replicas will be two, since (1 * (200/100)). This means that one more container will be deployed, and the workload will be balanced among the replicas through a *proxy* or an external load balancer [Takahashi et al., 2018].

Kubernetes Resource Scheduling (KS): Every container needing allocation is added into a waiting queue, which KS monitors. As the containers are added to the waiting queue, KS searches for a suitable work node to deploy them. This process is divided into three steps. First, KS verifies which worker node can receive the new container using a set of filters, also known as *predicates*. The purpose of filtering is to consider nodes meeting all pod requirements further in the scheduling process [Santos et al., 2019]. The second step is named scoring, where the KS ranks the filtered nodes based on pre-defined *priorities* and then finds the best worker node based on one or more scheduling algorithms.

Below we exemplify the *predicates* available on Kubernetes, taken from the official website of the Kubernetes Scheduler [11].

- **PodFitsResources**: Checks if the Node has free resources (eg, CPU and Memory) to meet the requirement of the Pod.
- **PodFitsHostPorts**: Checks if a Node has free ports. For instance, if the pod requires to bind the application on port 80, but another pod is already using that port on the node, this node will not be considered.
- **NoDiskConflict**: This predicate checks if a Pod has conflicts on a Node due to the volumes it requests, and those that are already mounted.
- **MatchNodeSelector**: This uses node labels to define a particular set of nodes where the pod can be deployed. It is known as node-affinity. On the other hand, a pod should not be allocated on a node with certain pods already deployed, which is a pod-anti-affinity.

After applying the filters, the KS knows which nodes are suitable for the pod deployment and then starts the scoring process. For example, given a pod that requires

[11] https://kubernetes.io/docs/reference/scheduling/policies/

half a core (0.5) CPU, the *PodFitsResources* predicate returns False for a node that only has 0.4 CPU free. However, if the filtering process does not find any worker node, the pod remains unscheduled until a suitable worker node is available. On the other hand, when several worker nodes are candidates, the KS triggers the node scoring process based on *priorities*. Next, some *priorities* also are shown.

- **SelectorSpreadPriority**: Spreads Pods across hosts, considering Pods that belong to the same application.
- **EqualPriority**: Gives an equal weight to all nodes.
- **LeastRequestedPriority**: Favors nodes with fewer requested resources. The more Pods that are scheduled to a Node, and the more resources those Pods use, the lower the ranking this policy will give.
- **NodePreferAvoidPodsPriority**: Prioritizes nodes according to node annotations, for example, which operating system version is running on the Node, or the name that specifies a Node. This priority can be used to ensure that different Pods shouldn't run on the same Node.

A score is calculated based on the priorities. After that, as the third step, the KS selects a group of nodes with the highest score and then selects the most appropriate node in a round-robin fashion to equally divide the load among the machines.

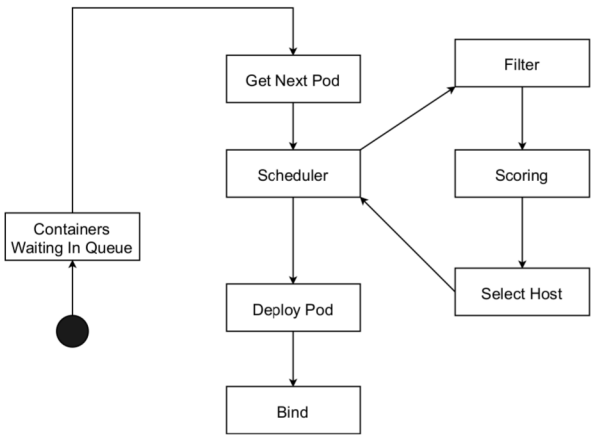

Figure 2.5. KS Flowchart - Adapted from [Xiao Yuan, 2019]

Figure 2.5 summarizes the KS algorithm, which starts in the *Kubernetes Containers Waiting Queue* state. As default, the three steps on Kubernetes resource scheduling consider mainly metrics from CPU or/and memory. However, several works have proposed some other QoS metric, such as network bandwidth [Santos et al., 2019], response

time [Priya et al., 2019], and more. Nevertheless, as discussed in the next section, the QoS metrics themselves do not reflect the user experience, which drives this work to propose the QoE-aware container scheduling.

2.3 Quality of Experience (QoE) and ITU-T Recommendation P1203

User experience becomes a crucial factor in end-user engagement and, therefore, on the cloud customer's revenue [Seufert et al., 2014]. As such, understanding the user's experience is vital to cloud customers, who use costly cloud computing resources, and to end-users, who may choose/change application providers based on their experience [Guarnieri et al., 2017]. However, for further improvements in end-to-end user's QoE, cloud providers need to evolve, exploiting advances in QoE-aware and cognitive operations in cloud resource management [Skorin-Kapov et al., 2018], [Barakabitze et al., 2019].

Quality of Experience (QoE) has been defined as the degree of pleasure or annoyance a user feels towards an application or service [Patrick Le Callet and Andrew Perkis, 2012]. In the past, the traditional network Quality of Service (QoS) metrics, such as packet loss, delay, and throughput, were used to define the level of satisfaction/performance of an application/service [Barakabitze et al., 2019]. However, although users can see the effects of these metrics on their experience, QoS metrics are insufficient to assess the user experience [Juluri et al., 2015].

There are two main QoE assessment methodologies, namely, subjective and objective. The Mean Opinion Score (MOS) is the most popular subjective measurement scale, which collects data directly from users based on their experience with the application [Barman and Martini, 2019]. For example, the user watches a video and rates it on a discrete five-point scale: 1-bad, 2-poor, 3-fair, 4-good, and 5-excellent. However, this is challenging because collecting these values requires surveying users about their experience, which is costly and does not scale [Guarnieri et al., 2017].

On the other hand, objective metrics can be quantified with a measurement tool [Juluri et al., 2015]). The difficulties in measuring subjective QoE motivated the development of objective tools that estimate subjective QoE from physical characteristics [Alreshoodi and Woods, 2013]. Several works have been proposed to map objective metrics into subjective ones (mainly for MOS), using different approaches (machine learning, correlation, etc) and suitable[12] for each application in which the user's QoE

[12]Note: Each application type can have its method for measuring the user's QoE. For example,

can be measured. For example, the authors Hora et al. [2016] propose a ML-based MOS predictor using the Wi-Fi parameters as features to estimate the user's QoE on the Web. In the same way, the authors Miranda et al. [2020] proposed MOS estimates using machine learning for video on demand application; however, considering the Internet Control Message Protocol (ICMP) probing metrics as inputs.

In the context of video streaming, HTTP adaptive streaming (HAS) is the most common technology used to deliver content to the end-users, but there is no widely accepted QoE model for them. Nevertheless, ITU-T provides a parametric bitstream-based quality assessment model, specifically, on ITU-T P.1203 [Skorin-Kapov et al., 2018]. Below, we describe briefly the ITU-T Recommendation P.1203 [ITU Telecommunication Standardization Sector, 2017] concepts and the software developed by authors Robitza et al. [2018] that implement ITU-T Rec. P.1203, which is used in this work.

P.1203 Standardization: The ITU Telecommunication Standardization Sector (ITU-T) develops international standards for information and communication technologies. These standards are presented as ITU-T Recommendations, which define how telecommunication networks operate, the quality of services, methods for objective and subjective assessment for audio and video quality, and more. The ITU-P Rec. 1203 is a set of objective parametric quality assessment modules for multimedia streaming applications. Those modules can predict the impact of audio and video encodings and IP network parameters on user's quality. Briefly, ITU-P Rec. 1203 creates a set of 30 subjective database ratings on the 5-point scale. Out of the 30 databases, 17 were used for model development, and 13 were used for model validation and selection. Also, for the ITU-P Rec. 1203 model, the video was encoded with H.254 codec in Full HD (1920X1080), and audio was encoded with the AAC codec. More information about the model can be found in the official documentation [13].

P.1203 is composed of three modules, being P.1203.1, P.1203.2, and P.1203.3. The first one, P.1203.1, is used in this work, referring to the video quality estimation module. P.1203.2 refers to the audio quality estimation module, and P.1203.4 is the audiovisual integration module.

P.1203.1 performs four models of operation that can vary to attend to different levels of available input information. Figure 2.6 shows each one. First, Model 0 comprises metadata-based information, such as delay, stall event, and media information (codec and resolution). Next, Model 1 adds frame information, and then Model 2 ac-

QoE measured in the video on demand is different from the method used to measure QoE on websites or audio streaming.

[13]https://www.itu.int/rec/T-REC-P.1203-201611-S/en

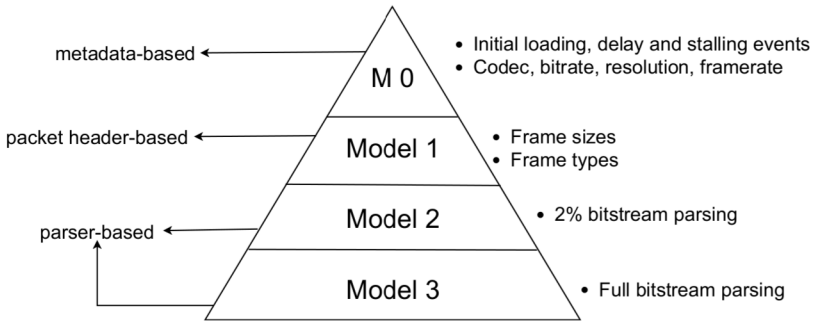

Figure 2.6. P.1203.1 Models of operation - Adapted from [Robitza et al., 2018]

cesses 2% of the video bitstream information. Finally, Model 3 accesses the complete bitstream information.

Open Software for ITU-T P.1203: The authors [Robitza et al., 2018] implemented the FULL P.1203 standard in Python. The software is available on a Github[14] repository, and after installation, it works either by command line on some terminal or by an API client. As input, the software receives one or more audiovisual segments or a JSON-formatted specification and which model (from Figure 2.6) will be used for evaluation. The output results are per-second audio/video quality scores, measured as MOS, and an overall audiovisual integrated quality score. The Appendix B shows an example of JSON-input and the JSON-output.

2.4 Video Streaming Applications

Video streaming has grown dramatically during the past few years using internet-based video applications [Li et al., 2020], presented in various forms, such as video on demand (VoD), live streaming, and personal broadcasting through social networks and e-learning systems. As a result, according to [Cisco, 2017], the global IP video traffic will be 82% of all IP traffic by 2022, and as video streaming applications grow, they demand more computing resources as well [Wu et al., 2017]. Because of that, video content providers are migrating to the cloud due to its advantages (high scalability, pay-per-use, etc.) [Ferdaus et al., 2017]. The following subsections describe the two video streaming applications used in this work: Video on Demand and Live Streaming.

2.4.1 Video on Demand

VoD systems stream video content to users as they make requests. This type of application has features such as start over, pause, rewind, and forward, enabling the end-user

[14]1https://github.com/itu-p1203/itu-p1203/

to control and select part of the video [Juluri et al., 2015]. Hence, the video content is exhibited as an asynchronous mode.

As mentioned, HTTP adaptive streaming (HAS) is the most common technology used to deliver video content to the end-user [Skorin-Kapov et al., 2018]. In this context of adaptability, the original video file is split into multiple segments, often known as chunks. These correspond to a few seconds of the original video, encoded in different bitrates and resolution quality [Bampis et al., 2017]. Usually, each chunk has 2-5 seconds.

The division of the video into chunks is used to adapt the video presentation to the users as internet connectivity changes, for example, as bitrate increases/decreases. In this case, the chunks are larger for each bitrate and, consequently, provide better video quality. This also prevents video stall events when the bitrate decreases, which means that the smaller video chunk will be downloaded – this always maintains chunks on the playback buffer to be played (as described below). The user's QoE is highly correlated to video quality. In short, when video quality is better, users tend to have a good experience, although it also depends on other factors. Besides that, stall events lead to a poor user experience [Ghadiyaram et al., 2017].

Proprietary formats implement HAS for VoD, such as *Microsoft Smooth Streaming (MSS)*[15] and *Adobe HTTP Dynamic Streaming (HDS)*[16] [Barman and Martini, 2019]. However, in 2011, the International Organization for Standardization (ISO) endorsed a new open standard called MPEG-DASH, or simply DASH (Dynamic Adaptive Streaming Over HTTP) [Stockhammer, 2011]. DASH has been widely adopted by video content providers, such as Netflix, Youtube, Amazon, and Hulu [Quinlan et al., 2016].

To illustrate how DASH works, Figure 3 shows an example of communication between client and server. First, the original video is pre-split into different bitrates (X, Y, Z), each containing n chunks that are stored on the server's side. Then, the communication between DASH client and server is established using the HTTP protocol. When the user requests a video, the server sends a Media Presentation Description (MPD) file. The MPD file contains a manifest with the list of all available representations, information about video/audio resolution and their respective bitrates, as well as the duration of each video chunk [Juluri et al., 2015]. It also contains the URLs used to send HTTP-GET requests to get video chunks while watching the video. Each chunk is then stored on a playback buffer to be played on the client's device. Finally, on the client's side, there is the Adaptive Bit Rate (ABR) module, which implements the

[15]https://www.microsoft.com/silverlight/smoothstreaming/
[16]https://www.adobe.com/devnet/hds.htm

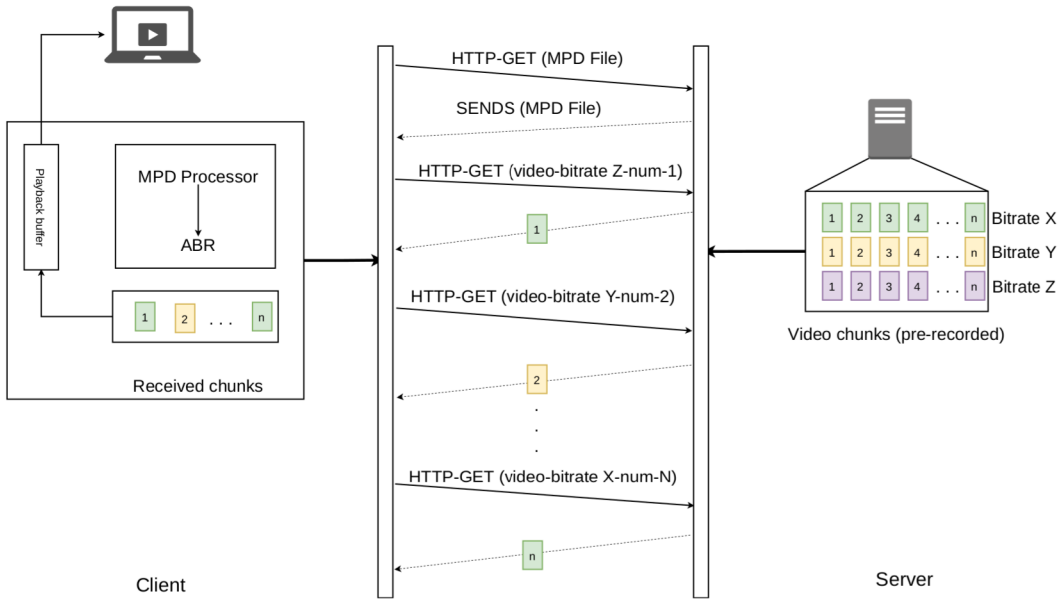

Figure 2.7. VoD architecture using DASH standard

algorithm to choose the appropriate bitrate based on the client-side link. The bitrate to be chosen is taken from the MPD file, processed by the MPD processor.

To summarize, streaming a video on demand means watching the current video chunk while progressively downloading future chunks, adapting video quality as network conditions change. Furthermore, VoD allows the user to navigate among the video duration as they want.

2.4.2 Live Streaming - Virtual Classroom application

In the live streaming application, the video contents are streamed to the users (viewer(s)) as images are captured from a source (camera, shared desktop) [Li et al., 2020]. Unlike VoD, this type of application does not allow users to start over, rewind, or forward while the video is transmitted. This means that the streaming is exhibited in synchronous mode, that is, in real-time.

Live streaming has numerous purposes, which can have four architecture variations [Li et al., 2020]. In short, (a) One-to-one (unicast), when a user streams video content to another user, such as a video call. (b) Many-to-many, that means every user streaming video content among them, such as in a videoconference. (c) Many-to-one occurs when several sources capture scenes and send them to one user. An example is multi-camera video surveillance, which streams them to one user. Finally, (d) One-to-many (multicast) is when one user streams video content to many other users, such as live broadcasts in social media (large audiences) and virtual classrooms (e-learning

system, lower audiences) [Khan and Salah, 2020]. Those architectures are based on three main processes that implement live streaming applications, being (i) real-time transcoding, (ii) packaging, and (iii) transmission [Aral et al., 2019]. They will be detailed below in the context of virtual classrooms.

This work employs a one-to-many architecture, specifically for live virtual classroom applications. The live virtual classroom is characterized by the use of videoconferencing technologies, in which it has the unique ability to simulate the wealthiest form of human interaction, namely, face-to-face. Also, it usually contains a lower audience localized in the same region, which does not demand large content distribution centers or multi-cloud. Several platforms can be used as virtual classrooms, such as Zoom, Microsoft Meeting, and Google Hangout. Although this type of application can be designed as many-to-many, we simplify it, considering that only the teacher (one source) is transmitting video content while students are watching (many).

The following explains those processes applied to a cloud-based live virtual classroom environment. The first process is video transcoding, which consists of re-encoding and converting each stream instance (from the source) into different bitrates, resolutions and then stores them. However, unlike VoD, this process is performed in real-time as a stream is received from the source. For example, as Figure 2.8 illustrates, the source video from a teacher is transcoded into three different segments (resolution/bitrate). It is a computation-intensive task and consumes a massive amount of resources [Wei et al., 2016]. Because of that, live streaming content providers have been using the cloud to host their applications, where they can use the extensive resource capacity and the elasticity of the cloud [Li et al., 2020].

The second process is packaging, which is a set of operations executed on the already encoded data as soon as possible so that video players can interpret the stream. This is also known as the video file format. These operations include (i) cutting the video data into chunks, (ii) setting all information about the timing and structure of the video, and (iii) generating the manifest file, which is used for progressive download [Li et al., 2020]. Transcoder and packing is essentially the core operation for HAS content preparation [Seufert et al., 2014].

The packaging process can be used with any HAS implementation (HDS, MSS, DASH); however, *HTTP Live Streaming (HLS)* has mostly been used for the live streaming application [Chakraborty et al., 2015]. HLS has the same approach as DASH, which splits the video into various bitrates/resolutions (high, mid, and low), each one in multiple chunks, and delivers a manifest file so that the client performs HTTP-GET progressively based on the network conditions. Each HLS chunk is usually 10 seconds in duration and has the extension .ts. The HLS manifest file has the extension .m3u8

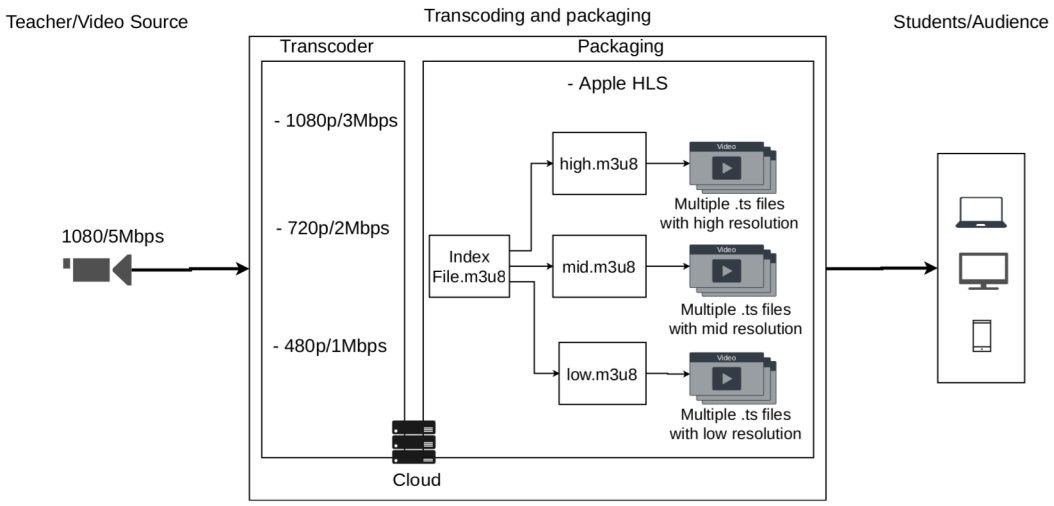

Figure 2.8. A cloud-based Live Virtual Classroom Architecture - Adapted from [Aral et al., 2019]

[La et al., 2020]. After created files (.ts) for HLS, finally, the video chunks are delivered to the audience (students) devices, where they are sorted and merged for playback in a media player [Aral et al., 2019]

2.5 Machine Learning

Machine learning (ML) has been identified in the literature and some market products as the main tool to implement autonomous adaptability and improve decision-making [Moysen and Giupponi, 2018]. ML algorithms create models that can learn to make decisions directly from the data without following predefined rules [Wang et al., 2017]. These algorithms are separated into four learning approaches: *Supervised Learning*, *Unsupervised*, *Semi-Supervised* and *Reinforcement Learning* (Xie et al. [2018]).

Supervised learning takes an already labeled dataset, which is known as the training dataset. This dataset is organized into an input vector (x) and the desired output value (y) to develop a predictive model by inferring a function $f(x)$, returning the predicted output $\hat{y}$. This learning uses two statistical methods, namely: *Regression* and *Classification*. Algorithms that use *Regression* have their output value defined by a numerical value belonging to an interval in the set of real numbers, whether finite or not. Regression approximates a continuous function from a set of examples. Finally, algorithms using *Classification* have their output value defined by a set of finite classes. These values can be numeric (integer values) or categorical. The classes are directly linked to the problem addressed, and the goal of the learning algorithm is to build a classifier that can correctly determine which class the new unlabeled examples belong

to.

Unsupervised learning receives a set of unlabeled inputs (that is, no output). Thus, the objective of this type of learning is to find patterns, structures, or knowledge in unlabeled data, grouping sample data according to its similarity [Xie et al., 2018]. *Semi-Supervised Learning* is a combination of the first two forms presented above. That is, it starts learning with a labeled training base and, during system operation, it makes refinements at runtime using the entry of new data. Finally, *Reinforcement Learning (RL)* involves an agent, a set of states S and an action space A. The agent is a learning entity that interacts with the environment through actions. Thereby, the agent is on the state S_t and makes an action A_t in time t, which can change the environment and take the agent to another state S_{t+1}. Each action generates a reward R_{t+1} (positive or negative). In order to maximize the reward, the reward R_{t+1} shows to the agent how much good or bad was the choice in the action A_t when the agent was in state S_t.

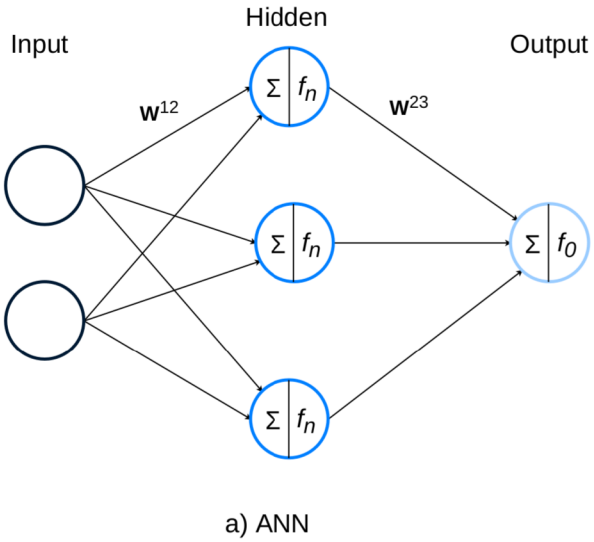

Figure 2.9. (a) Example of Artificial Neural Network Architecture - Adapted from [Gao et al., 2019].

Several algorithms have been developed to perform these learning approaches. Among these algorithms, Artificial Neural Networks (ANN) are the most important [Chen et al., 2019a]. An ANN is composed of several neurons in parallel, which are inspired by the human brain. Furthermore, they consist of the *input* layer and processing units (neurons) organized in *hidden* and *output* layers, as shown in Figure 2.9. Succinctly, neurons in parallel that are on an intermediary layer form a hidden layer on the neural network. Also, the weights W^{12} and W^{23} are the weights of the outputs from the previous layers, and element-wise activation functions, f_n, and f_0 implement the nonlinear transformations in the hidden and output layers, respectively [Gao et al.,

2019].

There are various types of ANNs, and next, we will show one of them called Recurrent Neural Network (RNN). However, before that, we illustrate the concepts of deep learning from an ANN example. Then, we present the RNN and how it can be modeled as a Deep Recurrent Neural Network (DRNN). Finally, we show an RNN-specific architecture called Long Short-Term Memory (LSTM).

2.5.1 Deep Learning and Recurrent Neural Networks

Traditional machine learning methods obtain satisfactory results with the assistance of experts in feature engineering (people having profound knowledge on the domain). However, this is insufficient as some features in interactions (combinations of features) that are implicit could not be learned by traditional methods [Yue et al., 2020]. On the other hand, Deep Learning (DL) methods are better suited to deal with this due to the capability of learning features (feature interactions) and extraction of hierarchical features [Erfani et al., 2016]. This is a consequence of the deep learning ability to represent the data as a nested hierarchy within hidden layers of an artificial neural network [Shrestha and Mahmood, 2019],[Chalapathy and Chawla, 2019].

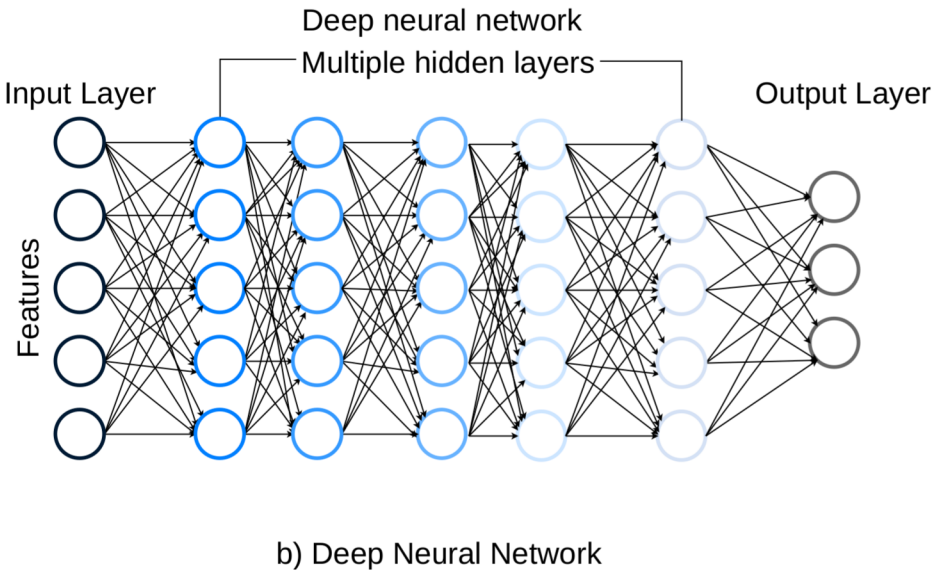

Figure 2.10. Example of a DNN architecture.

An example of DL is an ANN composed of multiple hidden layers of processing units, in which, in each layer, units are connected to units in adjacent layers. These multiple hidden layers form a Deep Neural Network (DNN), as Figure 2.10 illustrates. In this architecture, information flows in just one direction, from the *input* layer, through the *hidden* layers to the *output* layer. Because of that, ANN and DNN cannot capture

sequential information in the input data, which is needed for dealing with sequence data or time series forecasting.

Other ANN types have been developed to solve the ANN/DNN limitation in different domains, such as Object Detection, Text Classification, and Natural Language Processing (NLP). One of them is the Recurrent Neural Network (RNN) which also can be modeled as a Deep Recurrent Neural Network (DRNN) [Pascanu et al., 2013]. Below we describe the RNN architecture and how it overcomes ANN/DNN limitation and how to make them DRNN.

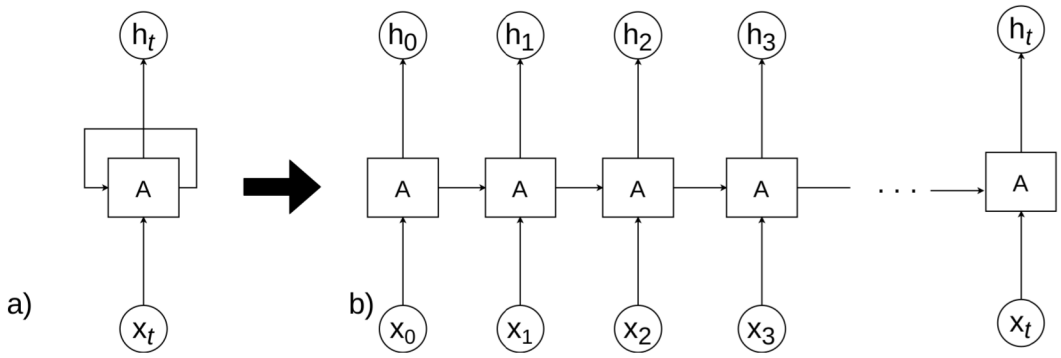

Figure 2.11. (a) Example of RNN Architecture; (b) The unrolling of RNN in t times

In RNN, the processing units form a cycle based on a feedback loop, in which the output for a layer becomes the input to the next layer. In addition, the cyclic connections in RNNs exploit a self-learned amount of temporal context, which makes RNNs better suited for sequence modeling tasks or time-series data [Li and Wu, 2015]. This allows the network to have a memory of the previous states [Shrestha and Mahmood, 2019]. For instance, Figure 2.11 (a) shows a single RNN architecture and the unrolling of RNN in time in Figure 2.11 (b). In this example, the RNN was unrolled t times into t-layer RNN.

In DNN, the *depth* is defined as having multiple hidden layers between the input and output layers. Assuming this, a single RNN can be considered deep since any RNN can be expressed as a composition of multiple layers when unfolded in time [Pascanu et al., 2013]. However, the authors Pascanu et al. [2013] show other approaches to extending an RNN into a DRNN, introducing *functions*, separated into four approaches: (1) Input-to-Hidden Function - consists of adding nonlinear hidden layers before the RNN hidden layers. It can be used to extract features and, thus, improve the performance of the model. (2) hidden-to-hidden Function - consist of adding new fixed-length nonlinear hidden layers among the RNN layers. As the author's example, one option is to use Multi Layer Perceptron (MLP) with one or more hidden layers. (3) Hidden-to-Output Function - in this case, the deep can be made by adding nonlinear

hidden layers before the RNN output layer. For example, add an MLP. This makes the model able to summarize the history of previous inputs more efficiently. Finally, (4) Stack of Hidden States Function - consists of stacking multiple recurrent hidden layers on top of each other.

RNNs can remember information from the recent past; however, it is challenging to learn the long-term dependency in the data, which is the major limitation of RNN [Shrestha and Mahmood, 2019]. This is known as the vanishing gradient problem [Gupta and Dinesh, 2017]. However, to overcome this limitation, the authors Hochreiter and Schmidhuber [1997] proposed a new implementation of RNN, called LSTM (Long Short-Term Memory).

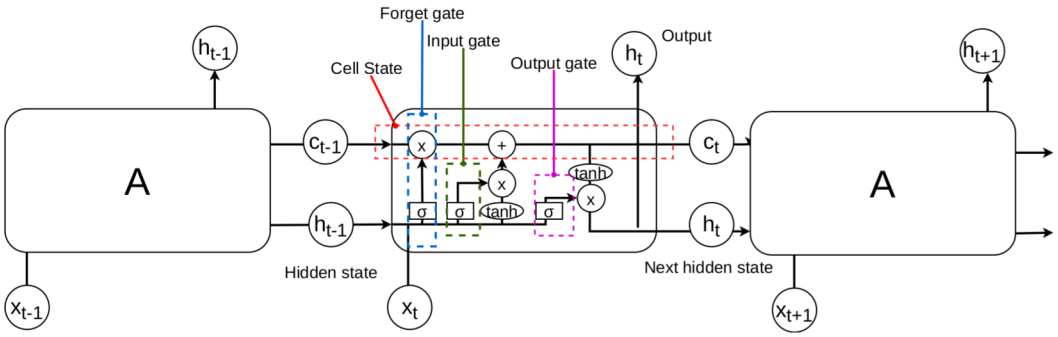

Figure 2.12. LSTM Architecture

LSTM was designed to memorize very long-term temporal dependencies through memory cells containing different types of mechanisms that are called gates and state [Malakar et al., 2021], as illustrated in Figure 2.12. These gates are: (i) *Forget gate*. The main idea behind a forget gate is to decide what information will be retained. A sigmoid layer produces either 0 (discard information) or 1 (retain information). (ii) The *input gate* decides what new information will be added to the current cell state. (iii) The *output gate*, through a sigmoid activation function, determines long-term state output. In short, these gates control what is stored, read and written on the cell [Shrestha and Mahmood, 2019].

In terms of state, there are two: (i) The *cell state* is the main LSTM component, where the information moves. The current cell state C_{t-1} depends on the previous cell state and new information that is added. Finally, (ii) the *hidden state* is the value obtained from the output gate multiplied by the cell state, which produces the current hidden state.

Another RNN implementation is Gated Recurrent Unit (GRU). GRU was proposed by Cho et al. [2014] and is similar to LSTM but simpler in terms of computing demand and implementation [Fu et al., 2016]. GRU cells have two gates: (i) Update

gate, which controls memory update, and (ii) reset gate, which is used to define how the new input will be combined with GRU's contents.

2.6 Summary

This chapter presented the fundamental concepts necessary to understand the proposal of this dissertation. First, the cloud computing paradigm focusing on the architecture and business model was presented. Then, three main cloud resource management categories, including resource scheduling. Furthermore, we presented the Kubernetes platform as well as discussed its components and network model. Also, the Kubernetes horizontal autoscaling and scheduler were presented. It is essential to know the Kubernetes scheduler because it will be one of our baselines in the evaluation.

Other concepts discussed in this chapter were QoE assessment methodologies and ITU-T Recommendation P.1203 used to measure user's QoE in both video streaming applications considered in this work. These applications also were discussed, focusing on the architecture and the adaptive protocol used in each one (DASH and HAS, respectively). Finally, we discussed machine learning and Deep Learning concepts, as well as the RNN and LSTM architectures.

We use these concepts in the following chapters to show our proposal, which is to perform QoE-aware resource scheduling in a cloud environment, employing deep learning techniques to estimate user's QoE on VoD and live virtual classroom applications, which are co-located with other applications/services.

Chapter 3

Related Work

This chapter summarizes the main works found in the literature related to the theme of this dissertation. The first section of this chapter (3.1) shows the literature search methodology, including keywords used to search the works and the criteria to select them. The following section (3.2) describes works that perform QoE-aware cloud resource management but do not include the scheduling process. Next, the works related to container scheduling are shown in Sections 3.3 and 3.4. Finally, Section 3.5 presents a summary of this chapter's contents.

3.1 Methodology

We surveyed the literature using Google Scholar (https://scholar.google.com.br/), which returns papers from several scientific editors, including IEEE Xplore Digital Library (https://ieeexplore.ieee.org) and ACM Digital Library (https://dl.acm.org/).

The search was refined by selecting the papers with 10 or more citations in the last 6 years or 5 or more citations in the last 2 years. This citation quantity restriction was applied only for works related to our, presented in Section 3.3 and 3.4. For works presented in section 3.2 we do not apply any citation restriction.

There have been many studies on virtual machine scheduling on the cloud [Liu and Qiu, 2016]. For works related to ours we restricted the search to those proposals that consider only containers because: (1) our work specifically addresses container scheduling problems. (2) We consider the growing cloud containerization, and it has become an emerging research topic [Adhikari et al., 2019], [Masdari and Khoshnevis, 2020].

The search results were grouped into three categories described below. First, the works that use QoE for resource management without considering the scheduling

process. Then, two other categories that consider QoE and the container scheduling process, however, using different approaches. Below we detail each one, and in the following sections, we describe the works.

1. **QoE-Aware Cloud Resource Management** In this category, we selected works that employ QoE to perform cloud resource management. The works consider both container or VM as a resource. None of them perform resource scheduling. The results were obtained employing the following keywords:
   - QoE-Aware cloud resource management
   The results ares shown in Section 3.2.

2. **Computing Resource-Based Container Scheduling:** Considering computing resources as parameters for making scheduling decisions. This means that computing resources such as CPU, memory, and disk are relevant to decide where the container will be scheduled. The results were obtained employing the following keywords:
   - Container scheduling and "cloud"
   The results are shown in Section 3.3.

3. **QoE/QoS-aware and ML-Based Container Scheduling:** We selected works that consider QoS or QoE to perform container scheduling. Considering QoS, these works aim to minimize or maximize some QoS parameters, such as response time and bandwidth. In general, these QoS parameters are configured into an SLO (Table 2.1). These works employ several techniques, including machine learning. In terms of QoE, we have not found any work that considers the user's QoE measured inside the cloud to perform container scheduling. To the best of our knowledge, our proposal is the first work that makes QoE-aware resource scheduling considering QoE measured into a cloud computing environment. This search employed the following key-words:
   - "QoS-aware/drive" and container scheduling
   -" QoE-aware/drive" and container scheduling
   The results are shown in Section 3.4.

3.2 QoE-Aware Cloud Resource Management

On the end-to-end user path, cloud resource management is critical to the overall user experience. However, as mentioned, ensuring a good quality of service does not reflect a good quality of experience. Thereby, some works in the literature propose QoE-aware

cloud resource management. This section shows some of these works that perform resource management techniques but do not consider the resource scheduling process.

The authors [Dutta et al., 2016] proposed QoE-aware resource management, which performs resource elasticity analyzing the VM's CPU/memory usage and the end-user QoE. The proposed scheme decides on whether to trigger elasticity or not based on QoE perceived by users. The authors consider vertical and horizontal scaling to deal with the workload while keeping the user's QoE. The evaluation was made as an experimental approach using a testbed. Also, the proposed scheme was incorporated into the ETSI MANO framework[1]. As a result, the authors showed that there is a significant tradeoff between the amount of allocated resources and the user's QoE.

The authors [Slivar et al., 2019] proposed QoE-aware resource allocation to game applications hosted on the cloud. Specifically, the authors denoted bandwidth as a resource and used a QoE model developed in their previous work to estimate the MOS scores based on the video encoding parameters [Slivar et al., 2016]. Besides that, the authors proposed an algorithm that, in the first step, defines the lowest bitrate value possible for each player and gradually allocates more bandwidth to the player with the largest MOS gain. As in our work, the authors consider the QoE fairness, modeled by the equation proposed by Hoßfeld et al. [2018].

Another work that considers QoE-aware resource allocation is proposed in [Haouari et al., 2019]. The proposal creates a prediction-driven resource allocation framework to maximize the user's QoE and minimize resource allocation costs on live streaming applications over a geo-distributed cloud. The authors consider storage as resources and, unlike our work, do not consider the live streaming transcoder process, which means that the intensive CPU and memory used are not considered. Also, the authors created an estimator that predicts the number of viewers in each region, not the user's QoE, as is proposed in our work. Based on the predicted viewers, the algorithm decides the amount of storage at each cloud to ensure the user's QoE (in terms of startup delay) and minimize the cost per allocated GB. Moreover, the viewers are served from their closest cloud site.

Autonomic ConTainerized Service Scaler (ACTS) is an autonomic system proposed by [Santos et al., 2020] to QoE-aware horizontally and vertically scale containers to different workloads. Horizontal scaling is performed by altering the number of instances, while vertical scaling modifies the CPU and memory allocated to the current instances of a web application. The experimental evaluation demonstrates that ACTS keeps the user's QoE metrics within the limits set in the SLA. However, the authors'

[1]https://www.etsi.org/technologies/open-source-mano

proposal does not consider the user's QoE on the scheduling process, which could further improve the user's QoE.

The works mentioned above differ from our work because the main proposal does not involve resource scheduling. Despite that, it is worth mentioning those works to show that there is an effort in the literature towards QoE-aware cloud resources management. Nevertheless, the literature also showed a lack of works that employ QoE-aware cloud resource scheduling, which motivated our proposal.

3.3 Computing Resource-Based Container Scheduling

This section shows a set of works that consider computing resources as parameters to perform container scheduling.

The authors in [Medel et al., 2017] proposed a client-side container scheduling based on the application's resource demands that extends the Kubernetes platform. Considering the client-side demand, the application provider has to define and classify their application's resource usage, dividing the applications into two categories: high and low resource usage. The scheduler proposed uses this classification to balance the number of applications in each worker node. It avoids more than one container with high resource usage (i.e., CPU or disk) in the same worker node. Also, the scheduler considers this classification to decide on what worker node to deploy the container to. For example, if an application uses more network than CPU resources, it should be allocated to a server with more available bandwidth than processing power. Notice that, like our work, the authors take into account the degradation in the applications due to co-location. However, it is difficult to estimate the application's resource usage beforehand [Masdari and Khoshnevis, 2020].

The authors [Liu et al., 2018] propose a multi-objective container scheduling algorithm considering factors from the server's side, being (1) the server's CPU and (2) memory usage, and (3) the time to transmit the container image over the cloud network. Besides that, (4) the author's proposal also needs to classify the application's resource demand. With that, the scheduler calculates the matching relationship between containers and available worker nodes. As in the previous work, computation-intensive containers should be allocated to worker nodes with more CPUs, while network-intensive containers are allocated to worker nodes with high bandwidth. The last factor is (5) the clustering of containers, which means that the authors consider the characteristics of the application to reduce the network transmission consumption between interre-

lated containers (for example, in a multi-tier architecture). The interrelated containers are scheduled to the same node whenever possible (clustering of containers). However, the authors overlooked the effect of application co-location, which can cause overload in the node or damage the containers. Besides that, their proposal does not use any mechanisms to mitigate overloads, for example, container rescheduling. On the other hand, our work reschedules the containers when the user's QoE is degraded.

As with previous works, the authors [Mao et al., 2017] propose resource-aware scheduling called DRAPS (Dynamic and Resource-Aware Placement Scheme). DRAPS performs scheduling based on the currently available node's resources. In addition, DRAPS performs container cluster monitoring to identify the dominant resource type that is most used by each cluster (belonging to the same application). It is used to balance resource usage among the nodes, which means that the containers will be scheduled in a complementary way in terms of resource usage, preventing containers that use the same resource intensively from being on the same node. However, the proposal requires resource demands specification for each application type. As mentioned, it is challenging to estimate resource demand beforehand. Furthermore, the authors' proposal performs container migration based on resource bottlenecks on the node. In this case, cluster monitoring checks when a resource type becomes a bottleneck and identifies the most resource-intensive container. After that, this container is migrated to the most appropriate worker node and killed from the node to release the resources. This migration can avoid user's QoE degradation after the resource-intensive container is migrated, releasing more resources to the user's QoE container. However, the authors did not specify the threshold value to determine a bottleneck, which may be hard to decide for each application type. In contrast, our proposal uses a sophisticated ML method to determine when the user's QoE is degraded due to resource contention on the worker node caused by co-located applications. Furthermore, instead of rescheduling the resource-intensive containers with no defined QoE method, we decided to prioritize rescheduling those containers where QoE can be measured while other applications/services run in best-effort mode.

A Kubernetes Container Scheduling Strategy called KCSS was proposed by [Menouer, 2021]. KCSS can use a set of criteria to select a worker node based on the Priority Order For Similarity to the Ideal Solution (TOPSIS)[Lai et al., 1994] algorithm. The author's criteria are related to the state of the cloud, considering three work nodes' resource usage metrics: CPU usage, disk usage and memory usage. Thus, the aim is to compact the containers to use the maximum resources possible from the worker node. Also, the authors used criteria to minimize power consumption, minimize the number of running containers on the worker node, and minimize the time of trans-

mitting the image to the worker node. TOPSIS chooses a a worker node whose distance from the best solution and the worst solution is minimal using the n-dimensional Euclidean distance. This means that TOPSIS aggregates all criteria into a single rank and selects the worker node with the highest rank.

Table 3.1. KCSS Example

	CPUs	Memory	Disk	Power	Containers	Images
Worker Node 1	60%	70%	50 %	140 watt/s	2	1
Worker Node 2	20%	30%	80%	150 watt/s	3	0

Table 3.1 shows the exact example illustrated by the authors. In this example, two worker nodes are considered, which CPU, memory, and disk utilization rate are represented for each one. Also, power consumption, number of containers already allocated, and whether the image is in the worker node (1) or not (0). For this example, after KCSS runs, Worker Node 1 will be selected to execute a new container. Although this proposal compacts the containers and improves the worker node resource usage, the authors do not consider the interference caused by the co-located applications. In this example, resource contention could be caused if the container achieves more than 40% of the worker node CPU.

Further, KCSS is generic, allowing the use and combination of other criteria and resources. Due to this generalization, we use this algorithm to compare to our work, in which we implemented two KCSS criteria versions (maximize and minimize), considering the same metrics as used in our proposal.

3.4 QoS-aware Container Scheduling

This section shows a set of works that consider QoS metrics as parameters to perform container scheduling.

The authors [Santos et al., 2019] proposed an extension of the Kubernetes Scheduler (KS) for latency-sensitive applications in a distributed cloud solution (fog computing). The proposal adds new predicates and/or priorities to KS. As mentioned, the KS default scheduler uses only CPU and/or memory information to execute container scheduling. However, Santos et al. [2019] proposal includes two QoS requirements that are assigned to each worker node as a label, being: Round Trip Time (RTT) and bandwidth available. Moreover, the authors proposed a Network-Aware Scheduler (NAS) algorithm that selected the suitable worker node based on the minimization of the RTT and checks if the best candidate worker node has enough bandwidth. However, this

approach to extending the Kubernetes Scheduler by adding new predicates and/or priorities is more complex than using the Kubernetes API. This is because it is necessary to recompile Kubernetes. Unlike this strategy, our proposal extends the Kubernetes Scheduler by creating a control plane over Kubernetes that uses its API. Also, our approach can be transported to another platform, such as Docker Swarm.

The authors [Guo and Yao, 2018] proposed container scheduling to optimize system performance based on interrelated containers (as in [Liu et al., 2018]), workload balancing, and response time. Thus, the proposal schedules containers with dependence between them into the same server or a neighbor server. This approach decreases the network calls across the cluster and decreases the user's response time. Meanwhile, our work does not consider container dependence or intercommunication. Indeed, this is a limitation of our work that is discussed in Section 4.5. Additionally, Guo and Yao [2018] considered the workload on the servers. In this case, the scheduler also balances the container cluster workload among neighbor servers. In short, this means that the proposal considers the container configuration (amount of resources needed), and the new container will be scheduled to the server that will keep the servers' workload more balanced. Furthermore, the proposal regards container migration based on communication information between the container in the cluster and whether the load imbalance degree exceeds a threshold value. This keeps the overall system balanced. Hence, Guo and Yao [2018] presents good results using a simulated computational environment. However, it is important to observe that the simulation approach may not represent the real characteristics of the cloud computing environment, such as dynamic workload variation over time and user behavior. Also, this approach can impact the generalization of the results when applied in the real world.

Finally, Yang et al. [2018] conducted a study to show the effectiveness of ML in cloud resource scheduling. The authors proposed a classifier using unsupervised and supervised learning to determine which servers are most suitable (with sufficient resources) to guarantee the application's QoS, such as throughput and response time. They used a public dataset to create the model and validate their proposal. The classifier training was separated into two steps – first, clustering and labeling, which separates the servers with similar performance into groups, using the k-means technique. Also, the clusters were labeled to represent their performance level. Second, the authors trained models based on several ML techniques, in which Random Forest and XGBoost achieved 92.86% accuracy to predict the performance level for a given server. This prediction is used to rank the servers and select the most suitable server for a specific application. However, model accuracy depends on the (unsupervised) clustering step, which is complex and difficult to be replicated in other environments. On

the other hand, our work proposes to use only supervised learning, labeling the data (with a QoE value), which is simpler and easier to transport to other environments. Besides that, the authors did not employ their model in an experimental environment like in our work. However, they argue and show that the ML methods can help with revealing non-trivial correlations between application demands and server status.

Table 3.2. Comparison between related works

	Co-location	Rescheduling	Max Objective	Source of metrics	ML	QoE
[Medel et al., 2017]	Yes	No	Execution time	Server App.	No	No
[Liu et al., 2018]	No	No	Execution time	Server App	No	No
[Mao et al., 2017]	Yes	Yes	Scalability	Server App	No	No
[Menouer, 2021]	No	No	Computing resource usage	Server	No	No
[Santos et al., 2019]	No	No	Network QoS	Network	No	No
[Guo and Yao, 2018]	No	Yes	Response time	Container Server	No	No
[Yang et al., 2018]	No	No	Application's QoS	Server	Yes	No
Our proposal	Yes	Yes	User's QoE	Container/Server	Yes	Yes

Table 3.2 lists the main characteristics of the works described in the last two sections of this chapter and compares them with our proposal. As shown in the table, only Medel, Mao and our proposal take the interference caused by co-located applications into account. Furthermore, as in our work, Mao and Guo perform container rescheduling to recompose the cluster when there is poor performance. In our case, we consider the user's QoE degradation as poor performance while they consider QoS metrics.

None of these works aim to improve user's QoE. Instead, the ultimate objective is to maximize cloud resource utilization or some QoS parameters from the SLO. Unfortunately, QoS-based SLO is usually insufficient, because the QoS metrics themselves reflect poorly the end-user experience. Because of that, we propose the QoE value as an SLO metric to guarantee the end user's QoE. Naturally, the QoE can be put into performance content (in Table 2.1), and the QoE-based SLO could be measured in terms of its minimum, mean, and variance.

Most works do not consider the use of machine learning. The main advantages of using ML in cloud management are providing intelligence and autonomous adaptability, minimizing human intervention in cloud resource management, and improving QoS and QoE. Although Yang used an ML approach to container scheduling, cloud resource management can be improved with more sophisticated methods.

3.5 Summary

This chapter showed that there are works in the literature that use QoE as a parameter to perform cloud resources management but do not consider resource scheduling. These works were presented in section 3.2. In addition, some works that perform container scheduling using two different approaches were presented. The first approach considers computing resources as parameters (Section 3.3), while the second uses QoS metrics (Section 3.4).

However, this chapter also showed, employing the methodology described in section 3.1, the lack of work that considers the user's QoE measured within the cloud to perform the container scheduling. The following chapters show the proposal of this dissertation to cover this gap and the results obtained.

Chapter 4

QoE-aware Container Scheduler

This chapter discusses a generic architecture for container schedulers/reschedulers for co-located applications. Section 4.1 shows the problem statement. Next, section 4.2 describes the proposed system architecture. Section 4.3 describes the ML-based QoE Monitor module, while the general definition of ML algorithm and Model Input used in this work are explained in Section 4.3.1. The generic Model Output is defined in Section 4.3.2, and the specific Model Output for VoD application and Live Classroom is described in sections 4.3.3 and 4.3.4, respectively. Section 4.5 discusses the limitations of the proposed architecture. Finally, section 4.6 summarizes this chapter.

4.1 Contextualization and Problem Statement

This work considers a cloud infrastructure that uses container-based applications, which is currently a widely adopted solution to deploy applications [Maenhaut et al., 2020]. In this scenario, new containers are created automatically without human intervention and are deployed among the cloud worker nodes. This distributed deployment creates an environment where the containers are co-located with other applications/services in the same worker node.

Figure 4.1 shows the components considered in this work, and next we contextualize each one in a top-down approach. The cloud system maintains a queue of new containers to be deployed represented as $C = \{c_1, c_2, c_3 \dots c_k\}$. This queue employs the FIFO (First-In, First-Out) algorithm and serves only applications where the QoE can be estimated. Each of these new containers will be scheduled into one of the worker nodes of the set $W = \{w_1, w_2, w_3 \dots w_m\}$. Besides that, containers already allocated can be rescheduled to another worker node due to either system and hardware failures or any cloud provider policy. The containers to be rescheduled are deleted, and new

containers are put into the queue to be scheduled to a new worker node of the same set.

Furthermore, users' connections are distributed among these containers through a load balancing service. This creates a different quantity of users per container, represented as $U = \{u_1, u_2, u_3 \dots u_n\}$, where U is the set of all the users for all containers and u_x represents the number of users for each container. Also, each container's users experience a different QoE, expressed as $Q_x = \{q_{x,1}, q_{x,2}, q_{x,3} \dots q_{x,n}\}$, where $q_{x,i}$ is the QoE of user i for container x.

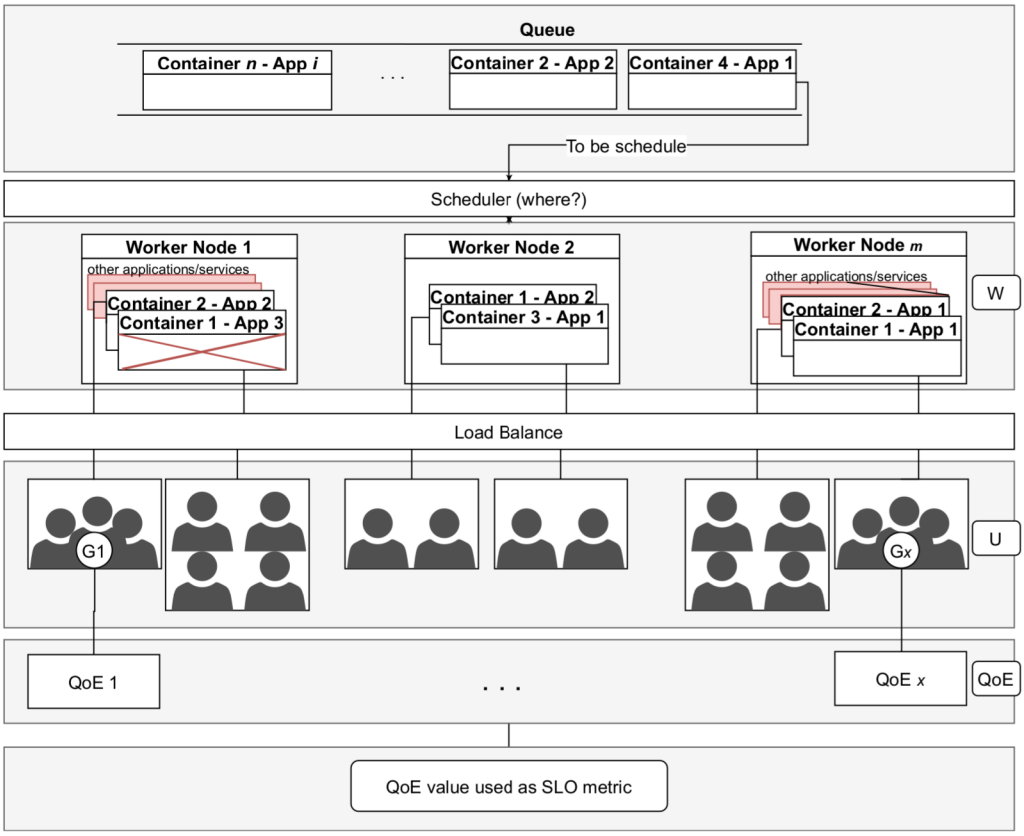

Figure 4.1. Problem Formulation

After contextualizing the components considered in this work, naturally, two questions arise: (i) *What is the best worker node W_i to deploy a new container in order to maximize the QoE and ensure the agreed SLO?* Moreover, (ii) *Given a container's users c_x already deployed with degraded QoE q_x, n, how can this QoE be improved so that there is no SLO violation?* Consequently, in response to these questions, first, we want to choose the best worker node to deploy a new container to either improve or keep the users' QoE above an SLO threshold. Second, we perform container rescheduling to deploy the container c_x into another suitable worker node. For this, we need to

estimate how the current cloud resource usage affects the users' QoE in their respective applications.

One way to do that is to use machine learning techniques to map the relationship between cloud resource usage and the users' QoE. Hence, this work defines QoE predictors based on a set of compute and network metrics observed over time. The machine learning models serve as oracles that answer the following question: *For a new container, what will be the users' QoE in that container if it is deployed in a specific worker node?* These oracles are used in a *Scheduler Decision Algorithm* that ranks worker nodes according to the QoE values. All of those steps will be described in more detail in the following sections. Before that, we describe the relevant elements of the system architecture.

4.2 System Architecture Overview

Figure 4.2 shows in general lines the proposed system architecture. It is divided into two planes, the Data Plane formed by the cloud computing platform and the Control Plane, composed of three modules developed in this work, being: *Cloud Resource Monitor*, *ML-Based QoE Monitor*, and *Scheduler Decision Algorithm*. Below we explain the cloud environment used in this work, and then we describe the modules in the Control Plane.

As represented in Figure 4.2, the Data Plane can be composed of several other cloud platforms, such as OpenStack[1], Docker Swarm[2], and more. However, this work employs a container-based infrastructure based on the Kubernetes Engine[3] for container orchestration[4] As mentioned in the Kubernetes network model (see section 2.2), the connectivity between containers relies on an external plugin, and this work used the Flannel Plugin for that. Furthermore, the containers are deployed on physical machines (worker nodes), where network communication among them is made through a physical switch. Now, we briefly explain the Control Plane modules, and in the following sections, we detail the two main modules proposed in this work. These modules are highlighted in black in figure 4.2.

The **Cloud Resource Monitor** module performs data requests (container metrics and worker node metrics), preprocesses them, and then sends the data in bulk to the *ML-Based QoE Monitor*. This preprocessing consists of grouping the data as the

[1]https://www.openstack.org/

[2]https://docs.docker.com/engine/swarm/

[3]https://kubernetes.io/

[4]Although this work uses Kubernetes, section 4.5 will discuss how to expand the Control Plane to other platforms.

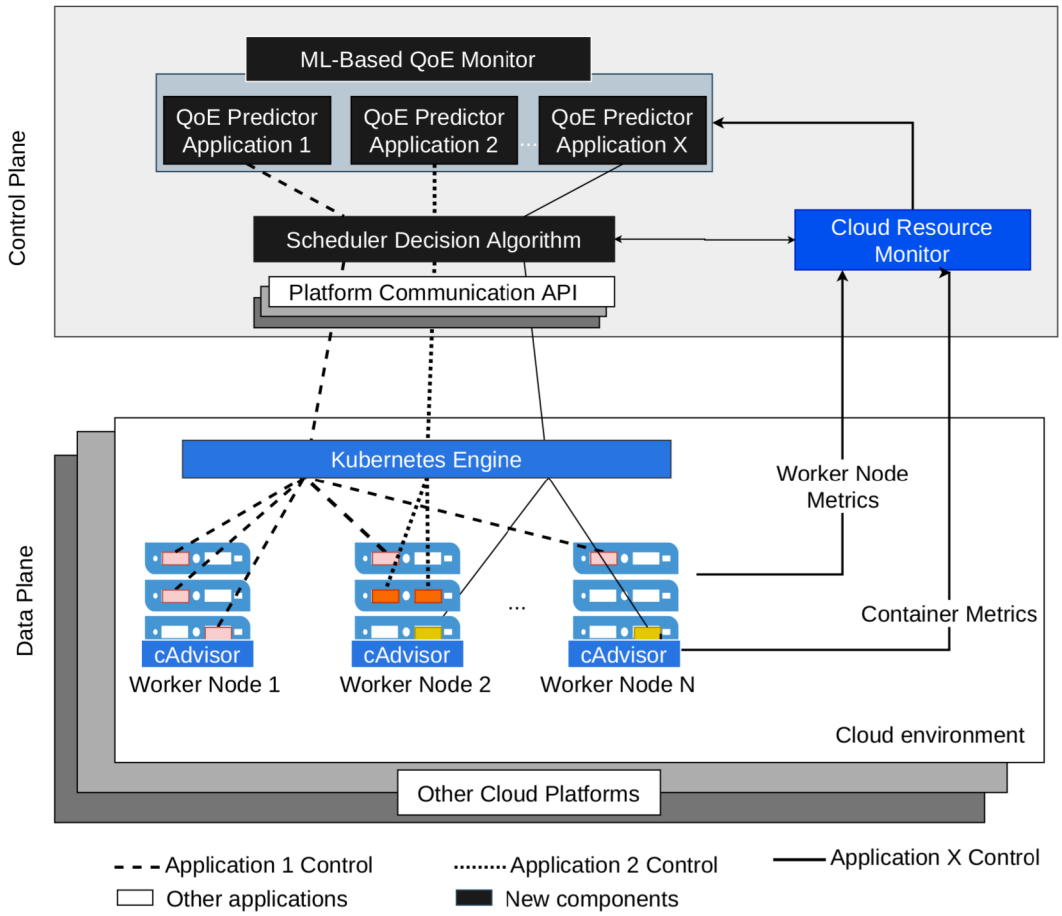

Figure 4.2. System Architecture

ML model's was trained, respecting the order of features and timestamp, and calculates the moving average of a predefined resource (for example, CPU) for all containers over time. This average will be used by the *Scheduler Decision Algorithm*, as described in section 4.4.

The Cloud Resource Monitor employs pre-existing tools to collect metrics. Several tools are proposed for resource monitoring in containers, such as [Zou et al., 2019], *docker stats*[5], Nagios[6], and Google's cAdvisor[7]. This work employs cAdvisor to collect relative resource usage metrics from worker nodes and containers. We decided to use cAdvisor due to the ease of use and wide use in the Kubernetes environment [Peinl et al., 2016]. Further, we consider the compatibility of the Kubernetes and cAdvisor solutions due to their development being from the same company.

The **ML-Based QoE Monitor** consists of a repository that can store a set of QoE predictors for different applications. Each predictor provides real-time estimates of

[5]https://docs.docker.com/engine/reference/commandline/stats/
[6]https://www.nagios.org/
[7]https://github.com/google/cadvisor

the QoE that the cloud can offer to a container's users for their respective applications, based on resource usage metrics from the worker node and container.

Finally, the **Scheduler Decision Algorithm** implements QoE-aware scheduling and rescheduling.

4.3 ML-Based QoE Monitor

The proposed system architecture supports QoE predictors for several applications, and the *Cloud Resource Monitor* module offers metrics that can be used to create those predictors, which will be stored in the *ML-Based QoE Monitor*. Also, the cloud provider can use the predictors' information to monitor the QoE delivered to the users and manage the cloud resources to support co-located applications.

This work created two QoE predictors for different applications to evaluate the architecture's genericity. Section 4.3.1 defines the model input used for both QoE predictors. Next, Section 4.3.2 defines the generic model output used by both applications, then each model output is detailed in 4.3.3 and 4.3.4, respectively.

4.3.1 Definition of ML Model and Features

Our system architecture does not define how the predictors should be modeled. This means that cloud providers can model each application with different machine learning algorithms. However, the model must consider that the estimated QoE value is used in the *Scheduler Decision Algorithm* as a numeric value. This limits the use of regression algorithms or a transformation from a class to real numbers. The ML models should map cloud resource usage into an expected users' QoE. The predictors' input includes resource usage metrics from the worker node and the container. The next chapter will evaluate which ML algorithm to be used and whether to consider data inputs as a time series. Formally, the predictors are a function defined as:

$$f(c_k, w_m) \mapsto QoE_{j,k,m} \in R \tag{4.1}$$

In the equation, c_k stands for the usage metrics of the container k, while w_m stands for the metrics of the worker node m. $QoE_{j,k,m}$ is the model output, which refers to the QoE value predicted for a container's users for application j into the container c_k deployed in the worker node w_m.

The resource usage metrics were separated into CPU, Disk, Memory, File system, and Network categories. We collect all metrics equally for the worker node (w) and

the container (c). We list below each resource usage metric used as feature input. The total number of metrics is 72 and is detailed in Appendix A.

CPU data: The CPU category contains *CPU usage*, *user CPU time*, and *system CPU time*. This information is given per CPU, in nanoseconds.

Disk I/O data: The Disk I/O category contains the number of bytes *read*, and *written*, and the number of *async*, *sync*, and *total* (sum of reads and writes) I/O operations. Each of these metrics is collected from the I/O service and the serviced daemon.

Memory data: This category contains *memory usage*, *max usage*, *cache*, *RSS*, *mapped file*, *working set*, *failcnt*, *pgfault*, *pgmajfault* and *swap*. All values are expressed in bytes.

Filesystem data: The filesystem category contains *filesystem capacity*, *filesystem usage*, *filesystem base usage*, and *filesysem inodes*. All values except inodes are expressed in bytes.

Network data: The network category contains *tx bytes*, *rx bytes*, *tx packets* and *rx packets*. This information is collected for each network interface. In the containers, there is only one interface (eth0). There are three interfaces in the worker nodes, being: *cni* (Container Network Interface), that provides network connectivity of containers; Flannel, that allocates a subnet to each worker node; and the server's physical interface.

4.3.2 Generic Definition of Model Output

In this work, the model output value is the expected QoE for the container's users k deployed into worker node w for application j, represented by $QoE_{j,k,m}$. In this case, we will create estimators to approximate the values obtained from Equation 4.2, which shows the generic formula to obtain the average users' QoE. In this equation, the mean QoE is multiplied by a dispersion factor in order to ensure higher fairness between the users.

$$QoE_{j,k,m} = \frac{(\sum_{i=1}^{|U|} QoE_{user_i}) \times F}{U} \tag{4.2}$$

Each user can have a different QoE value, which creates a challenge to fairly summarize the overall QoE of the container's users. To deal with this, the QoE incorporates the fairness coefficient F.

$$F = 1 - \frac{2\sigma(QoE)}{H - L} \tag{4.3}$$

Fairness F is calculated using equation 4.3, as proposed by [Hoßfeld et al., 2018], where $\sigma(X)$ is a function that calculates the standard deviation of the QoE set and

$0 \leq F \leq 1$. Equation 4.3 penalizes containers according to the amount of dispersion of the QoE: a higher variance will lead to a lower F. H and L are variables that assume the highest and lowest allowable QoE values, respectively. Therefore, those variables can assume different values that depend on the method used to calculate the MOS in each application. For example, the MOS can be measured between 1 and 5 as well as 1 and 10. As the QoE value will be used in the SLO metric, it is important to note that the SLO threshold must be defined between that interval (H-L), since the *Scheduler Decision Algorithm* will use it in the scheduler/rescheduler procedure. In other words, the cloud provider must configure the SLO threshold value in the *Scheduler Decision Algorithm* as the QoE interval is measured in each application.

4.3.3 Training Output for the VoD Model

As mentioned previously, the model output is created considering the QoE estimated for the container's users. In the case of VoD, to estimate the QoE, we consider combining two main aspects of video-on-demand transmission as parameters in the ITU-P P.1203 Recommendation. The first one is the *video resolution* playing in each client, which can vary due to cloud conditions and is associated with the *bitrate*, which is the second aspect considered. Therefore, the QoE_{user_i} value in Equation 4.2 is measured as per-second video session quality scores.

Table 4.1. Example of QoE_{cloud} calculation for VoD

QoE Client 1	QoE Client 2	QoE Client 3	QoE Client 4	QoE Client 5	QoE Client 6	QoE Client 7	Mean QoE [0,1]	F	QoE_{cloud}
5.0 **1.0**	1.68 **0.17**	5.0 **1.0**	2.42 **0.35**	1.91 **0.22**	5.0 **1.0**	5.0 **1.0**	0.67	0,25	1.68
5.0 **1.0**	3.68 **0.67**	5.0 **1.0**	2.42 **0.35**	5.0 **1.0**	5.0 **1.0**	5.0 **1.0**	0.86	0.52	2.82
5.0 **1.0**	5.0 **1.0**	5.0 **1.0**	5.0 **1.0**	5.0 **1.0**	5.0 **1.0**	5.0 **1.0**	1.0	1.0	5.0

Table 4.1 shows a real example from a container's users. Each line presents the QoE of 7 clients, totaling three seconds of a video session. Each line results from users' QoE, where each one has their QoE value measured between 1 and 5 (in bold are the normalized QoE values). Besides that, the normalized mean QoE and the F value also are presented. Finally, after employing equation 4.2, the expected result is the container's users' QoE used as the output value, transformed to values between 1 and 5, as shown in last column.

4.3.4 Training Output for the Live Virtual Classroom Model

The model output used for the Live Virtual Classroom Model is different from the VoD model due to the nature of the Live Virtual Classroom application. Unlike VoD, Live Virtual Classroom needs to re-encode the original video resolution in real-time into multiple resolutions before transmitting it to users. When the server does not have sufficient computing resources to do it, the transcode process can be harmed, and video data will not be sent to the users even if the network conditions allow it, which could cause stall events. Hence, we incorporate stall events in the QoE metric for Live Virtual Classroom to penalize the user's QoE. Furthermore, we observed in preliminary experiments that a stall event was also occurring when the client was watching the video in high resolution and bitrate. Applying the same method as in VoD, the user's QoE would be considered high; however, the experience perceived by the users is poor because of the interruption caused by stalling, which is more frustrating for higher video qualities [Duanmu et al., 2016].

Equation 4.4 shows how QoE is calculated for each user i. In addition to estimating QoE based on ITU-T Recommendation P.1203 (considering video resolution and bitrate), during n seconds of the video session, the QoE is penalized by stall length, calculated as described in equation 4.4.

$$QoE_{user_i} = \sum_{t=1}^{n} (QoE_t - StallLength_t) + 1 \tag{4.4}$$

The $StallLenght_t$ is calculated based on equation 4.5, which uses the formula proposed in [Ghadiyaram et al., 2018].

$$StallLength_t = \begin{cases} e^{\alpha * s_t} - 1 & \text{if stall event occurred} \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

The author's proposed alpha value is 0.2, which is also used in this work, and s_t denotes the length of a stall at a discrete-time instance.

Table 4.2 shows a real example of how QoE_{user_i} was calculated for a user i watching the video and considering a stall event occurs during six seconds. The first column QoE_t refers to the QoE in each time t calculated using ITU-T Recommendation P.1203 (considering resolution and bitrate). The second column is the stall event duration time t. The next column is the $StallLength_t$ calculated using Equation 4.5. As QoE is calculated based on MOS, which varies between 1 and 5, we also normalized the $StallLengh$ in this interval, represented in the fourth column. Finally, in the last

Table 4.2. Example of QoE_{user_i} calculation for Live Classroom

QoE_t	Stall Event Time t (s)	$StallLength_t$	normalized $StallLength_t$ [1-5]	QoE_{user_i}
5.0	1	0.221402	1.0	5.0
5.0	2	0.491824	1.515404	4.4
5.0	3	0.822118	2.144922	3.8
5.0	4	1.225540	2.913815	3.0
5.0	5	1.718281	3.852945	2.1
5.0	6	2.320116	5.0	1.0

column, the QoE_{user_i} is calculated based on equation 4.4 (considering the normalized $StallLength_t$). Note that there is a relationship between stall length and users' QoE degradation captured by an exponential function. Finally, to produce the final model output for a container's users, equation 4.2 is applied, penalizing the average QoE due to unfairness.

4.4 Scheduler Decision Algorithm

This section describes the greedy-based approach proposed to control the scheduling and rescheduling process. The Scheduler Decision (SD) module performs the scheduling and rescheduling process taking into account the QoE. Each procedure uses the predictors as an oracle to predict the container's users' QoE if instantiated on a given worker node. This allows the container placement to be QoE-aware, which means that the scheduler and rescheduler procedures presented below have the same objective: maximize the QoE above an SLO value. We detail the algorithm below. Before that, Table 4.3 describes the symbols used in the algorithms.

Algorithm 1 describes our proposed algorithms. The main procedure, named *CSD (Container Scheduler Decision)*, continuously checks for two conditions, being: (i) a non-empty HPA queue; in this case, there is a pending container p in the Horizontal Pod Autoscaler (HPA) queue to be deployed (line 3). Queue status is queried via the *monitorHPAQueue()* function of the Kubernetes API; (ii) QoE degradation for each container c_i already deployed (lines 8-10). The *checkBadQoE()* function checks whether the predicted QoE is below a QoE value (defined as SLO) for a specific time interval T. This means that this function is using the predictors to estimate the container's users' QoE into all containers, taking as input the containers and worker node metrics. These checks are done in parallel. Besides that, each application has its own SLO threshold configuration value and time interval T, which is defined by the

Table 4.3. Scheduler Decision Algorithm Notations

Symbol	Definition
P	Set of new containers to be deployed.
p_i	New container instance i.
$p_i.App_j$	Application type of container p_i (e.g., [VoDApp,LiveApp]).
SLO	Service Level Objective threshold specified by the cloud customer. (Value between the highest and lowest QoE measured in application j)
r	Container resource monitored to obtain the moving average of its usage.
$cMax$	Container metric that has the highest moving average r resource.
W	Set of metrics for all worker nodes available.
w_i	Worker node metrics for instance $i \in$W.
C	Set of metrics for already deployed containers.
c_i	Container metrics for instance $i \in$C.
T	Time interval threshold for monitoring QoE.
QoE_j	A list of QoE predicted for every worker node $j \in$W.

application provider in the container configuration file. These values are queried via *getContainerConfig(x)*, which is called before scheduling a new container (line 5) and checking for QoE degradation (line 9). It is worth noting that the SLO value should be within the highest and lowest QoE ranges defined in equation 4.3.

Scheduling works as follows. The function *monitorHPAQueue()* returns the new containers to be deployed. Then, the scheduler procedure (lines 12-16) searches for a suitable worker node to deploy a new container p_i, referring to the application $p_i.App_j$, taking as parameters a container c and the available worker nodes W. However, in this stage, we face a problem choosing which container c to be considered. The scheduler procedure uses the predictors (line 14), which require container and worker node metrics as input to predict the container's users' QoE. We already have all worker node metrics (W), and we need to choose an existing container to serve as a baseline for the new container p_i. Thus, we estimate the usage of the new container as the worst case scenario of the existing containers of the same application. This is implemented in the *ContainerResourceUsage()* procedure. This procedure constantly calculates the moving average of each of the container's resources r (e.g.,CPU). We employ a pessimistic approach, choosing the container with the highest moving average of resources r (considering a *window*) at the very moment before calling the scheduler function. Note that $p_i.App$ (see Table 4.3) is passed as a parameter so that the function searches only within the cluster to which p_i belongs. This procedure then estimates the QoE for deploying the new container in each worker node w_j (line 14). Finally, in line 15, the algorithm chooses the worker node (w) that maximizes the QoE above the SLO.

Then, container p_i is deployed to worker node w (line 16). The predictors were modeled considering that the container and work node metrics are aggregated in timestamp (as described in Section 5.2.2). Therefore, the first container scheduling/rescheduling needs to wait to complete each predictor's timestamp.

Algorithm 2 Container scheduler decision algorithm

```
1:  procedure CSD(C, W)
2:      while True do
3:          P ← monitorHPAQueue()                          ▷ Non-empty HPA Queue
4:          for each p_i ∈ P do
5:              SLO ← getContainerConfig(p_i.App)
6:              cMax ← max(containersResourceUsage(p_i.App, r, windows))
7:              scheduler(p_i, cMax, W, SLO)
8:          for each c_i ∈ C do                            ▷ QoE Monitoring
9:              SLO, T ← getContainerConfig(c_i.App)
10:             if checkBadQoE(c_i, SLO, T) then
11:                 rescheduler(c_i, W, SLO)
12: procedure SCHEDULER(p, c, W, SLO)
13:     for each w_j ∈ W do
14:         QoE_i ← predictor(c, w_j)                      ▷ Scoring step
15:     w ← max(X) {∀X ∈ QoE | X > SLO}                    ▷ Select step
16:     deploy container p in w
17: procedure RESCHEDULER(container, W, SLO)
18:     for each w_j ∈ W do
19:         QoE_j ← predictor(container, w_j)
20:     w ← max(X) {∀X ∈ QoE | X > SLO}
21:     deploy new container in w
22:     delete container
```

The need for rescheduling is checked on the second part of the csd procedure (lines 8-10). The *checkBadQoE()* function returns true when there is a QoE degradation in the container c_i due to co-location with other applications/services. This triggers a container to reschedule. In the proposed algorithm, rescheduling deletes the container from the current worker node and deploys a new container in a more favorable worker node. For that end, we first call the rescheduler procedure (line 11). This procedure searches for the most suitable worker node to deploy the new container. Unlike the scheduler procedure, rescheduling uses as parameters only the current container and the available worker nodes (W). This because we already have historical data from the current container to serve as a baseline. After choosing a worker node that maximizes QoE, the container is deleted from the original worker node (line 22).

Figure 4.3 highlights how our solution works on top of Kubernetes. For both the scheduler and rescheduler process, our solution uses the default Kubernetes Filter, which lists those work nodes with the minimum resources available to deploy the containers. Then, our solution scores each work node based on the predicted QoE (using the predictors) in the scoring step. Finally, in the select step, our solution chooses that work node that maximizes the QoE above the SLO.

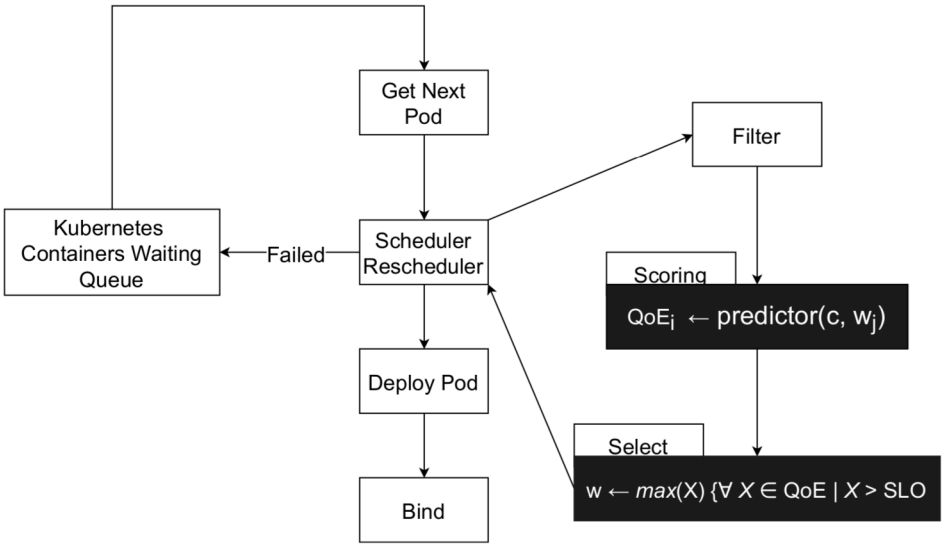

Figure 4.3. Our Scheduler/Rescheduler process vs Kubernetes Scheduler

4.5 Discussion

This section discusses some limitations in this work and how they can be addressed.

Limitation #1: Generalization of the System Architecture : There are two forms to port our proposal to other platforms: i) use in another **docker container orchestrator platform**, such as Docker Swarm[8] and SaltStack[9]. To do that, we have to consider using the same tools to collect metrics from containers and server, in this case, the cAdvisor, as well as some modification in the *Scheduler Decision Algorithm* (discussed below). In this case, our proposal is limited by using docker and cAdvisor, whereas there are other container virtualization technologies (e.g., LXD[10]) and tools to collect metrics from containers and servers (e.g., *mpstat* or *sar*, *docker stats*). To apply these three modules directly in another docker platform, considering the use of cAdvisor, the *Scheduler Decision Algorithm* API needs some modifications

[8]https://docs.docker.com/engine/swarm/
[9]https://github.com/saltstack/salt
[10]https://linuxcontainers.org/lxd/introduction/

to interact with the docker platform to perform container scheduling/rescheduling. ii) Another approach is to port our methodology to any cloud platform, even those that also support virtual machines, such as OpenStack[11]. This means that all three modules in the control plane must be adapted to be used over a different data plane.

Limitation #2: Multi-tier Architecture: Both applications in this work use a one-tier architecture. However, nowadays, many applications are modeled using multi-tier architectures, which means that an application is developed and distributed among more than one layer executed in separate resources (containers or VMs). For example, an e-commerce website where the front-end is running in one container and the database system (back-end) is on another container. In this case, there is internal communication between those layers that is not measured in this work. Consequently, if this communication has a problem, resulting in a degradation in the user's QoE, the QoE predicted in the front-end container will not reflect this problem. However, one solution is adding the network link metrics between the back-end and front-end containers as input to the ML model. In the same way, back-end container metrics (CPU, memory, disk, and network) could also be added.

Limitation #3: Measuring the QoE of a group of users instead of individual QoE: This work schedules and reschedules containers based on the user's QoE. For that reason, the quality of the predictor is essential for good system performance. However, QoE assessment is challenging due to the many networking factors and compute resource variations in cloud computing systems [Hobfeld et al., 2012]. Besides that, there are technical difficulties in measuring each user's QoE separately on the cloud, mainly in monitoring the resource usage independently for each user. These difficulties drove our work to consider QoE estimates for all the container's users instead of estimating QoE separately for each user. Limitation #6 discusses how QoE measurements reported by the user's devices could be used in our proposal.

Limitation #4: Applications without a well-defined QoE: Our proposal does not deal with applications that do not have a well-defined QoE, which means that these applications need a separate scheduling and rescheduling algorithm. Although we can achieve good results without analyzing/managing these applications, future work could investigate a single scheduler/resheduler algorithm for both QoE-aware and non QoE-aware applications.

Limitation #5: Container scheduling considering QoE monitored only in the cloud: This work estimates the container's users' QoE within the cloud environment, which means that we do not consider how each technology in the end-to-end

[11]https://www.openstack.org/

path between users and cloud can affect the users' QoE. For example, considering the Internet Service Provider (ISP) and the home network.

In this context, several works measure users' QoE in each part of the network topology between users and the cloud ([Moura et al., 2020; Miranda et al., 2020; de Oliveira and Macedo, 2021; Carvalho et al., 2019]). Although this is outside the scope of this work, our proposal could use these approaches to refine the container scheduling/rescheduling.

However, it is essential to note that only receiving the QoE measured outside the cloud (or the end-to-end QoE) is insufficient to ensure good performance in container placement within the cloud. This is because the QoE received from outside the cloud does reflects the QoE offered by the cloud *combined* with other technologies. Considering only the QoE measured outside the cloud could erroneously start the rescheduling process. This occurs when the cloud may have good conditions to provide a high QoE, while technical problems outside the cloud are degrading the QoE. Thus, the *Scheduler Decision Algorithm* should compare the QoE predicted within the cloud with the QoE on the path, which would allow the cloud administrator to know how much the current cloud configuration is degrading the overall user's QoE.

Limitation #6: QoE measured directly in the user's devices. In addition to measuring QoE in various technologies into the user's path to the cloud, another option is to measure users' QoE directly in their devices by the client's software. Some application providers do it, such as Netflix and Facebook video calls. This approach aggregates end-to-end information about the user's QoE. However, as in Limitation #5, receiving only this information would not be essential for container management. On the other hand, end-to-end QoE information would be helpful for the *Scheduler Decision Algorithm*, considering that we may be optimizing the cloud's QoE at a given time – e.g., rescheduling a container – but the user's QoE would not change because of some technical problem in the user's network path, where we can not act.

Further, user-provided QoE may have a limited use in public clouds, where the application provider would prefer not to share the QoE measured from their clients with the cloud provider. On the other hand, on the private cloud, where the cloud and application provider are the same, the QoE information on the cloud and clients can be shared.

Limitation #7: Transfer Learning to another cloud environment: As mentioned before, the methodology proposed in this work can be used in several cloud environments, and the control plane can be used directly in other cloud platforms. Generalization capability to new domains is crucial for machine learning models when deploying to real-world conditions [Dou et al., 2019]. However, transferring the model

trained from one cloud to another can be a theoretical limitation, considering the model generalization capacity. The difference in the cloud environments and clients' patterns is the main reason for limiting an already trained model to be transferred to another environment. This also brings another limitation that will be discussed next (#8).

However, in practice, as we will show in Section 5.3.2, we have had good results in transferring the trained models from one cloud to another with the same platform but different hardware configurations.

Limitation #8: Need to retrain the model: Maintaining the same accuracy as in the trained model in the production environment is considered a challenge because we expect the model to continue to make good predictions on unseen data. However, we cannot assume that the new data has the same distribution as the training dataset to hold good predictions. This distribution can vary as the worker node and the container settings change, such as when a hardware or system update occurs. Likewise, the distribution alters when workload patterns change. In this case, the model needs to be retrained to maintain its accuracy, which could be a limitation, considering the cost and time to retrain the model. Besides that, there is the difficulty of deciding when to retrain a model. One option for future work is to use the Concept Drift technique, which is widely used to identify when to retrain a model as the distribution of data changes [Lu et al., 2018].

Limitation #9: Container migration: Our proposal employs container rescheduling when QoE degradation occurs. However, another alternative would be container migration. Unlike container rescheduling, the migration process requires transferring the container's files from the original server to another [Torre et al., 2019]. Therefore, the time needed to start the application on the new server depends on the connection's quality between the servers.

However, it is essential to note that, as in the container rescheduling process, the container migration process needs to decide when and which worker node the container will be transferred. With this, the migration process can consider the QoE degradation and use QoE estimators to choose the best worker node.

4.6 Summary

This chapter shows the problem statement from theoretical point of view applied to the container scheduling for co-located cloud applications. This chapter also shows how we can use machine learning techniques to performs the QoE-aware containers scheduling and rescheduling. Also, we presented the proposed architecture for handling QoE-

aware container scheduling or rescheduling, where we divided into two layers, Data Plane and Control Plane.

Besides that, this chapter shown two machine learning model employed in the control plane. We defined the model input for both models, where we describe the model features collected, and we defined the model output for each application, being Video On-Demand and Live Virtual Classroom.

Finally, we defined the scheduler decision algorithm that performs the scheduling and rescheduling. The next chapter will present the data collection process for building the two models used in this work.

Chapter 5

Data Collection, Model Building, and Model Results

This chapter presents the complete process to train the models used to predict the users' QoE in each application adopted in this work. For this, two distinct models were created. Section 5.1 presents the environment setup and the data collection processes to create the training datasets. Next, Section 5.2 justifies the use of Deep RNN algorithms and how the models were trained. Then, Section 5.3 exhibits the models' results in terms of performance, generalization and gives a brief explanation of the models. Finally, section 5.4 summarizes this chapter.

5.1 Data Collection

Figure 5.1 summarizes the data collection process, which includes the environment setup overview and each process adopted. These processes are separated into subsections that will be better described next.

First, Section 5.1.1 describes the environment setup, including hardware and software used, and introduces how the data collection was conducted. Next, Subsection 5.1.2 explains the interference generation approach to simulate the co-located cloud applications. Then, Subsection 5.1.3 describes which information was collected from the user's side to label the training instances. Also, the parameters used to calculate the MOS using the software proposed by Robitza et al. [2018] that implements ITU P.1203 Recommendation. Finally, Section 5.1.4 characterizes the amount of metrics collected from the work node and the container to generate the initial training instances. With that, we discuss the method applied to reduce the amount of features and conclude by summarizing the final training datasets.

5.1.1 Data Collection Environment Setup

As shown in Figure 5.1, the experiment setup contains seven notebooks as clients connected to the cloud via a Gigabit Ethernet switch and one worker node. The worker node is an Intel(R) Core(TM) i5-4460 CPU@3.20GHz, 16 GB of RAM, and an HD with 250GB with Ubuntu Server 16.04. The clients have different hardware, but all run Ubuntu Desktop 18.04.

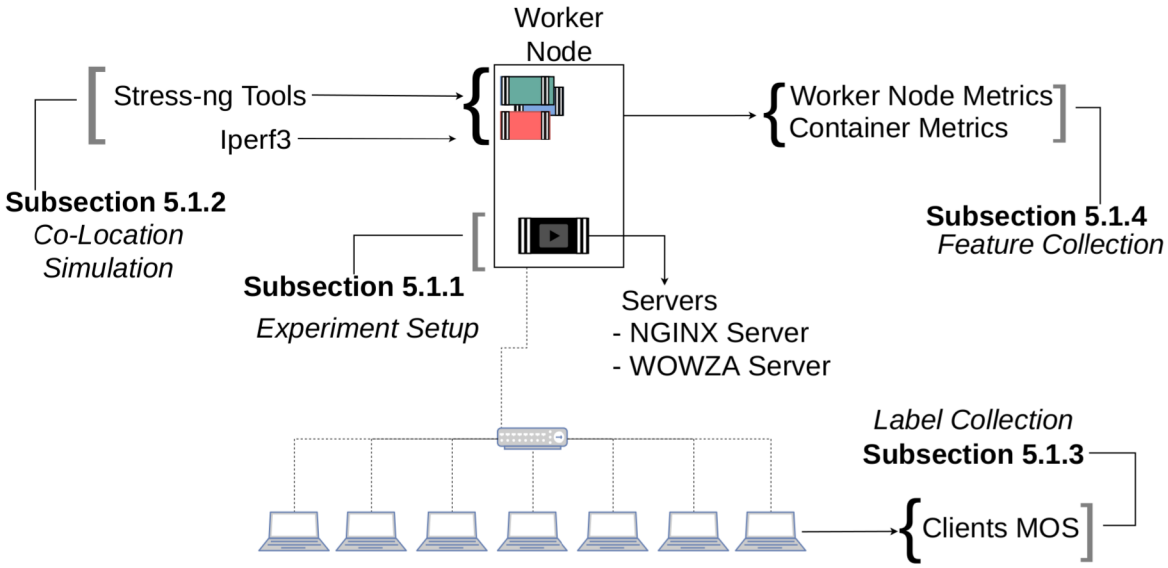

Figure 5.1. Training Instances Collection - Experiment Setup

For the VoD application, we used a DASH server into a docker container based on the NGINX[1] server. For this application, we used the Big Buck Bunny video[2] video with 9 minutes and 10 seconds of duration. The video was encoded using H.264 standard with chunks of 2 seconds. Also, the video has resolutions varying between 320X240 and 1920X1080 with different bitrates, as described in Table 5.1.

Table 5.1. VoD and Live Virtual Classroom adaptive streaming configuration

VoD		Live Virtual Classroom	
Resolution	**Bitrate (Kbps)**	**Resolution**	**Bitrate (Kbps)**
320X240	[46, 89, 131]	426X240	300
480X240	[178,222,263,334,396]	640X360	400
854X480	[522,595]	854X480	500
1280X720	[791,1000,1200,1500]	1208X720	1500
1920X1080	[2100,2500,3100,3500,3800,4200]	1366X768	6900

[1]https://nginx.org/en/
[2]https://peach.blender.or

For the Live Virtual Classroom application, we employed the Wowza Streaming Engine (https://www.wowza.com/). This platform is not an open platform but has a complete trial version (90 days). Also, this is a live streaming solution used by companies such as Facebook, Vimeo, and SpaceX[3].

This work does not consider the network path between where the original video is recorded and the cloud. For this case, we used a video class pre-recorded with 1 hour and 40 minutes of duration, using H.264 codec with 1366X768 dimensions, and deployed it into the Wowza Server. Hence, the video class will be transcoded in real-time to different resolutions and bitrates (Table 5.1, Live Class) and packaged into the HLS format, and then it will be consumed by the clients (students). The transcoding process creates video resolutions and bitrates based on Google Youtube Live Streaming examples[4].

In terms of the data collection process, training instances were obtained while clients watched the video under interference (co-located environment) and without interference. This process was repeated in decreasing rounds from 7 clients to one. The main goal in this phase was to cover the highest amount of behavioral possibilities from the cloud and simulate different quantities of users connected to the container. While the experiment runs, we logged the video metadata information on the client's side to calculate the MOS, which is used to label each training instance (output). At the same time, we also logged the container and worker node resource usage from the worker node's side to compose the features (input). Finally, we grouped input and output to create the final training datasets. Next, we describe interference generation, how the client's information was collected and how the final training datasets were created.

5.1.2 Interference Generation to simulate Co-Located Applications

Cloud providers usually co-locate different applications on the same worker node to improve resource utilization [Chen et al., 2019b]. However, these co-located applications generate interference among them, which occurs when the demand from the application exceeds the resources available on the shared worker node [Medel et al., 2017].

Furthermore, each cloud application has different workloads. For example, while applications such as big data analysis, machine learning training, and video transcoder

[3]Note: Information available on the Wowza website.
[4]https://support.google.com/youtube/answer/2853702?hl=enzippy=%2Cp

consume more CPU, memory, and disk, other applications use more network resources, such as VoD streaming and e-commerce.

To collect cloud resource usage and video sessions under different conditions, we use *Stress-ng* (https://wiki.ubuntu.com/Kernel/Reference/stress-ng) and *iperf* (https://iperf.fr/) to simulate different workloads. *Stress-ng* is a tool that can generate workload on the CPU, memory, and disk I/O, while *iperf* produces workload on the network interface. As illustrated in Figure 5.1, we use these tools to simulate other co-located applications with the video servers[5].

We determine the same extra computing and network workload to run co-located with our video streaming applications. *stress-ng* creates a continuous CPU load that varies between 40% and 95% of usage, and the memory usage increases continuously, reaching up to 90% and decreasing gradually. Disk I/O stress ranged randomly from 1 to 40 processes, each writing and reading 500MB. At the same time, we perform a TCP downlink and uplink transmission with *iperf* to create network load. Transmissions change from 0 bytes transmitted to 1024MB transmitted every 10 seconds.

This extra computing and networking workload on the worker node creates interference on the video streaming applications, which triggers the client's adaptation algorithm and, therefore, causes video quality switches and the user's QoE variation. Video transmission information is collected from the user's side, which we describe below.

5.1.3 Video Quality Measured at the Clients

As discussed in section 2.4, VoD and Live Virtual Classroom usually have different HAS implementations. In this work, for the VoD application, we used the DASH.js (https://github.com/Dash-Industry-Forum/dash.js) client, and for the Live Virtual Classroom, we use the HLS.js client (https://hls-js.netlify.app/demo/). Besides that, VoD and Live Virtual Classroom have different behaviors and, consequently, different forms to calculate the MOS, as discussed in section 4.3.3. and 4.3.4.

However, to produce the label value for the training instances – in this case, the MOS value – for both applications, the clients logged the same video metadata information. This information is: *playback stall time* and *timestamps* for each video session, and the *codec*, *frames per second (fps)*, *resolution*, and *bitrate* played by the clients at each second.

[5]We have tested these tools within the container, and we did not see a significant difference in results.

This information is used to calculate the MOS based on ITU-T Recommendation P.1203 using the software developed by [Robitza et al., 2018]. This software also requires device information, display size and viewer distance as parameters. In our case, we use a PC, 1920x1080 screen resolution, and 15 cm distance from the monitor, respectively. P.1203 can be run in four models, each one requiring different information, from metadata to full bitstream data. Since we collect only metadata information, we employ Model 0. Besides that, this software requires a JSON file input containing the metadata and returns a JSON file containing the MOS values for every second. Appendix B shows an example input JSON used in this work and Appendix B shows the JSON output. After generating each JSON file (one per client), we use it to calculate the final MOS for each application, as explained in sections 4.3.3 for VoD, and 4.3.4 for Live Virtual Classroom. It is noteworthy that the client information collection is only done to generate the training instances and training the models. After that, in a production environment, the model itself will estimate the user's MOS.

5.1.4 Feature Collection, Selection, and Final Training

As mentioned, the input instances consist of metrics from the worker node and the container. We use the cAdvisor API to collect these metrics and group them with the MOS. Table 5.2 describes the initial datasets. We have 32 metrics collected from the container and 40 from the worker node for both applications, which sum 72 metrics. These metrics are aggregated by 1 second with the MOS value (label) to create each training instance. VoD had 99.370 training instances while the Live Virtual Classroom had 123.596 instances.

Table 5.2. Initial Dataset Description

# Container Metrics	# Worker Node Metrics	Total
32	40	72

VoD training instances	Live Class training instances
99.370	123.596

The initial amount of metrics collected (72) was reduced to decrease the model training time without affecting the predictor's quality. For this, we employed feature selection techniques. Given that the collected metrics have nonlinear relationships, we use *Spearman Correlation* analysis. *Spearman Correlation* is defined as a non-parametric test based on ranks. This test indicates the relation between two variables by a monotonic function. With that, we transform our initial dataset into our final

training dataset, considering only those metrics with a *Spearman Correlation* higher than or equal to 0.50 (positive or negative correlation). Note that this phase only reduces the number of metrics (features), which means the number of instances has not changed.

In terms of reduction, Table 5.3 summarizes the final amount of metrics used. For the VoD application, the amount of input metrics was reduced from 72 to 18, representing a reduction of 75%. In terms of container metrics, the reduction was 93.75%, from 32 to 2. For the worker node, the reduction was 60%, from 40 to 16. Therefore, the final training dataset for VoD contains 2 metrics from the container (C) and 16 metrics from the worker node (W), totalizing 18 metrics as features.

On the other hand, for Live Virtual Classroom, the input metrics were reduced from 72 to 27, signifying a 62.5% decrease. In addition, there was a decrease from 32 to 17 for container metrics, representing 46.87% of the reduction. Moreover, the worker node metrics were reduced from 40 to 10, representing a 75% of reduction.

Table 5.3. Final Training Datasets Description

	VoD (99.370 instances)	Live Virtual Classroom (123.596 instances)
# Container Metrics	2	17
# Worker Metrics	16	10
Total	**18**	**27**

Table 5.4 shows the selected features to create both training datasets (for VoD and Live Virtual Classroom). As can be seen, there are some container (C) and worker node (W) metrics with a strong negative association with the MOS, indicating that high use can degrade the user's QoE. On the other hand, some metrics showed a strong positive association, indicating that high use can improve the user's QoE.

Table 5.4. Spearman Correlation – VoD and Live Virtual Classroom – (C = Container, W = Worker Node)

VoD			Live Virtual Classroom		
Metrics	Correlation	Source	Metrics	Correlation	Source
mem_cache	-0.80	C	cpu_usage	0.77	C
tx_bytes	0,53	C	cpu_user	0.77	C
cpu_usage	-0.75	W	cpu_system	0.78	C
cpu_user	-0.75	W	diskio_sync io_service	-0.51	C
cpu_system	-0.55	W	disk_sync io_serviced	-0.56	C
diskio_sync io_service	-0.72	W	disk_write io_serviced	-0.56	C
disk_sync io_serviced	-0.74	W	mem_usage	0.62	C
disk_write io_serviced	-0.74	W	mem_max_usage	-0.71	C
mem_usage	-0.76	W	mem_cache	0.59	C
mem_cache	-0.66	W	mem_rss	0.64	C
working_set	-0.77	W	working_set	0.64	C
rx_bytes_cni	0.55	W	filesystem_usage	-0.76	C
tx_packets_cni	0.50	W	filesystem_inodes	-0.72	C
rx_bytes flannel	0.55	W	rx_bytes	0.54	C
rx_packets flannel	0.50	W	rx_packets	0.57	C
tx_bytes flannel	0.56	W	tx_bytes	0.58	C
rx_bytes enp3s0	0.62	W	tx_packets	0.54	C
rx_packets enp3s0	0.65	W	diskio_sync_ io_service	-0.55	W
			mem_cache	0.57	W
			mem_container pgfault	0.68	W
			mem_container pgmajfault	0.68	W
			rx_bytes_cni	0.55	W
			rx_packets_cni	0.57	W
			tx_bytes_cni	0.55	W
			rx_bytes_flannel	0.54	W
			rx_packets_flannel	0.53	W
			tx_bytes_flannel	0.59	W

5.2 Model Building

This section shows the models' building process. First, subsection 5.2.1, justifies the use of Deep RNN algorithms. Then, subsection 5.2.2 shows the methodology employed to train the models. We used a search method to find the hyperparameters as well as implemented a technique to prevent overfitting.

5.2.1 Use of a time series-based model

As mentioned, each training instance was collected sequentially over periods of 1 second each. Therefore, we consider that the inputs represent a time series, and because of that, we analyzed our final datasets to decide which machine learning techniques would be more suitable for a time series.

Our analysis uses Autocorrelation Function (ACF) plots. ACF plots auto-correlation with its lagged values [Agrawal and Adhikari, 2013]. Figures 5.2 and 5.3 show the autocorrelation plot of MOS for 30 lags for VoD application and Live Virtual Classroom application, respectively. In the x-axis, we have the lag(k), and the y-axis gives the auto-correlation (r_k) at each lag. The results show a slow decay in the correlation between successive observations, indicating long-range dependence for both applications, although the decay is slower for VoD. This slow decay phenomenon is known as a strong temporal correlation in a time series [Gupta and Dileep, 2020] and has been characterized as a common phenomenon in cloud computing [Gupta and Dinesh, 2017; Gupta et al., 2018; Song et al., 2018] and QoE prediction[Ye et al., 2014; Eswara et al., 2019].

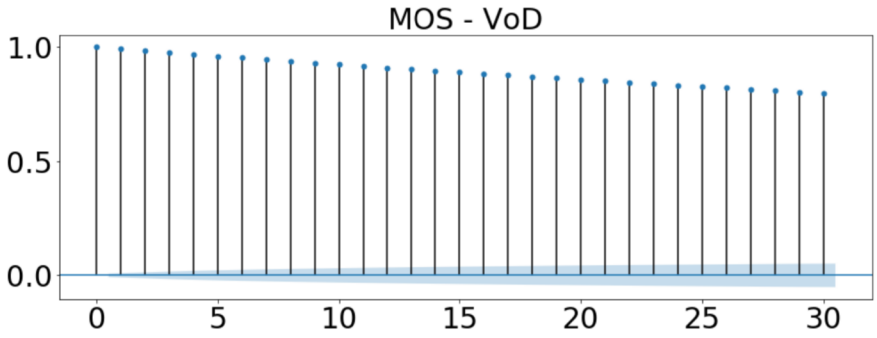

Figure 5.2. Autocorrelation plot of MOS at 30 lags

This long-term dependence requires more advanced prediction techniques. In this case, we use Recurrent Neural Networks (RNNs); specifically, we implemented and compared two variations of RNN (i) Long Short-Term Memory (LSTM) and (ii) Gated Recurrent Unit (GRU).

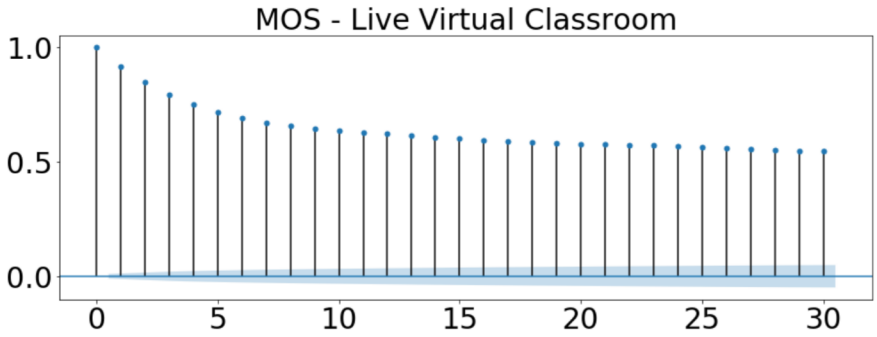

Figure 5.3. Autocorrelation plot of MOS at 30 lags

The following section discusses how these RNNs methods were implemented and the methodology applied in training the models.

5.2.2 Model Training Methodology

The methodology is the same for processing both datasets (VoD and Live Virtual Classroom). Also, both algorithms (LSTM and GRU) follow the same implementation method. Below we describe the datasets processing and algorithms implementation.

In the data processing phase, we prepared the training datasets. First, we separated all datasets into training (70%), validation (20%), and test (10%) datasets. This means that the model will fit using the training dataset, and while the training process occurs, the fitted model uses the validation dataset to predict the values and tune the model's parameters. Finally, the test dataset is used to provide a final evaluation of the model. Note that the test dataset is not used in the training process, which allows for an unbiased final evaluation. Also, we analyzed each feature's distribution and, given that all of them had a uniform distribution, we chose to use feature scaling between the [0,1] range.

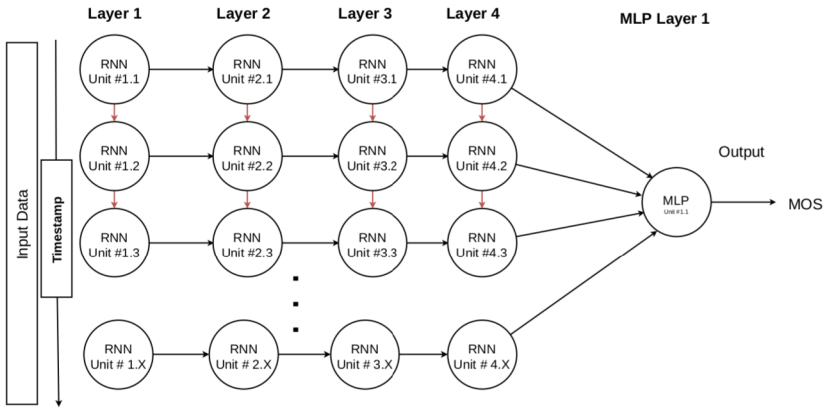

Figure 5.4. General Architecture of RNNs Implementation

In terms of algorithm implementation, the models were built using Keras version 2.2.2 on a computer with Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz and 16GB of RAM. Figure 5.4 shows the general models' architecture, composed of up to 4 RNNs layers followed by 1 Multilayer Perceptron (MLP) layer. The amount of units vertically depends on the timesteps (input).

The final models' architecture was created by tuning the model's hyper-parameters. Hyperparameters are parameters that are not directly learned in the training model process, however, these parameters control the performance of the model. In this work, we employ the Grid Search technique, which is a widely-used method to compute the optimum values of the model's hyperparameters [Reimers and Gurevych, 2017]. Grid Search performs an exhaustive search on a manually defined subset of hyperparameters and their respective configuration – Table 5.5 shows which hyperparameters were considered and the configurations used for each one, as well as a brief description.

Table 5.5. Evaluated Hyperparameters and Configurations

Hyperparameters	Values	Description
# Layers	[1, 2, 3, 4]	Amount of LSTM/GRU Layers
Batch Size	[128, 256, 512]	# of training examples used in each epoch.
# RNN units	[16, 32, 64, 128]	# of LSTM/GRU cells.
# Epochs	[300, 500, 1000]	# of times that the learning algorithm will work through.
Recurrent Dropout	[0.25, 0.50, 0.75]	Regularization method: used in the transformation of the recurrent state.
Dropout	[0.25, 0.50, 0.75]	Regularization method: in which units are probabilistically excluded.
Timestamp	[10, 20, 30]	Amount of time steps running in LSTM.

We define the models to have a maximum of 4 layers. Besides that, the training was made in a batch. In this case, we consider three batch sizes as described in the table. Also, the amount of units in each layer varies from 16 to 128. Moreover, the models were trained considering three different numbers of epochs.

The Recurrent Dropout and Dropout values represent the probability of excluding recurrent inputs and units. Deep learning models are more likely to be overfitting, which means that they may fit the training data exactly, making it impossible to generalize to other environments or data. We use these hyperparameters as regularization methods to reduce overfitting and improve model performance.

Besides that, the timestamp was divided into three options (10, 20, 30). These values represent the time interval (s) in which the input data will be aggregated and passed on to the algorithms. In practice, this means how far back the model must "look" to predict the next second. For example, the algorithm takes 10 seconds of data (from the past) as input to predict what will happen in the 11th second.

Note that the Grid Search will compare all these hyperparameter configurations. However, for the Activation Function, Loss Function, and Optimizer hyperparameters, were fixed. For the Activation Function, we used a **sigmoid** function. For Optimizer, we use **Adam** [Kingma and Ba, 2015] algorithm, and for the Loss Function, we used the **Mean Squared Error (MSE)** metric. Also, for the MLP layer, we set the activation function to use the **sigmoid** function.

Although we had defined the number of epoch to train the models, we used the Early Stopping technique. This technique is also considered a regularization method, which can prevent overfitting and improve model generalization. In short, Early Stopping monitors the model's performance at each epoch and stops the training process[6] when it notices a decay in performance (either decreases the accuracy or increases the error). In our Early Stopping configuration, we set the 30 as the number of epochs with no improvement and considered the validation loss performance to be monitored. This means that, after 30 epoch without improving the validation loss, the training will be stopped and moves on to the next hyperparameter configuration. This value can be considered high compared with other works of literature [Reimers and Gurevych, 2017]. However, we also consider larger batch sizes and train the models at more epochs.

Finally, the model will be saved during the training phase and updated as the model improves. Hence, we will have the best possible model at the end of the training process, considering the training dataset and the evaluated hyperparameters and configurations. The next section discusses the model's results.

5.3 Results

This section shows the models' results. First, Section 5.3.1 shows the final models obtained using the feature described in Table 5.4 and performing the training methodology described in Section 5.2.2. Also, we selected for each application a model to be implemented in the experimental evaluation (Chapter 6). Finally, Section 5.3.2 shows the capacity of the selected model to generalize to another environment. Furthermore, we analyze each feature's importance to the model's prediction.

[6]Note: Stops the training process in the current state of Grid Search and moves on to the next hyperparameter configuration to be tested in the grid.

5.3.1 Model Selection and Evaluation

Grid Search and Early Stopping techniques automate the model training process, leaving only the developer to choose which hyperparameters and configurations to use. As mentioned, at the end of the model's training, we will have the best possible model considering the evaluated hyperparameters and configurations.

Table 5.6 shows the best models' configurations for both applications using GRU and LSTM. In general, both RNN algorithms had similar configurations. For example, for both RNN algorithms and applications, the amount of layers is the same. This means that models were configured to have depth also in space, stacking multiple recurrent hidden layers on top of each other (in this case, 3 hidden layers). Also, the best results considered the maximum batch size possible and the minimum value set to dropout. Besides that, the best models were trained in 1000 epochs.

Table 5.6. LSTM and GRU Configuration

	LSTM		**GRU**	
	VoD	**Live Classroom**	**VoD**	**Live Classroom**
# of Layers	3	3	3	3
RNN units	64,64,32	32,32,64	64,64,64	16,16,32
Batch Size	512	512	512	512
Epochs	1000	1000	1000	1000
Recurrent Dropout	0.50	0.25	0.25	0.25
Dropout	0.25	0.25	0.25	0.25
Timestamp	30	10	10	10

However, the models' configurations differed in the amount of RNN units in each layer. For example, for VoD applications, the best model using LSTM converges in the last layer, while for Live Virtual Classroom, LSTM units expand in the last layer. On the other hand, using GRU, for VoD, the model keeps the amount of GRU units linear through the layers and, for Live Virtual Classroom, the amount of GRU units also expands in the last layer.

Another difference is the recurrent dropout and timestamp used in the VoD model. Comparing these hyperparameters between RNN algorithms, LSTM performed more recurrent dropouts as well as required more timestamps. This difference in timestamp between the RNN algorithm in VoD application can be explained by the LSTM capacity to observe/learn the longer-range better than GRU. Besides that, the timestamp difference between VoD and Live Virtual Classroom trained by LSTM can be explained by the autocorrelation plot. As observed in Figures 5.2 and 5.3, VoD and Live Virtual Classroom have different behaviors in the MOS decay during successive

observations. VoD has a more continuous decay than Live Virtual Classroom, which suggests that what happened in the more distant past is correlated to the present. On the other hand, Live Virtual Classroom has a fast decay in the first 10 stages, suggesting that what happens in the present has a more significant correlation to the closest past.

Despite these differences between the architectures, both RNN algorithms achieve satisfactory prediction quality in both applications. We use the Root Mean Squared Error (RMSE) metric to measure this prediction quality. RMSE represents the mean of the differences between the actual value (ground truth) and the predicted value (generated by the prediction model), squared.

Table 5.7 shows the RMSE achieved by GRU and LSTM, measured in each application's validation and test datasets. In addition to the satisfactory average error on the validation dataset, both models performed better in the training dataset. This characteristic is present in models that did not overfit the training dataset, which means that the models perform well in new data (unseen data).

Table 5.7. GRU and LSTM RMSE

	LSTM		**GRU**	
	VoD	Live Classroom	VoD	Live Classroom
Validation (20%)	0.044	0.039	0.084	0.099
Test (10%)	0.032	0.035	0.072	0.085

Although the RMSE for both RNN algorithms achieved a satisfactory result, LSTM performed better than GRU in both applications. In Chapter 6, we show our implemented proposal in an experimental environment. Since the LSTM models stood out, we chose them to be used in the ML-Based QoE Monitor Module (see Figure 4.2). Before that, in the following subsection, we evaluate other results considering only the LSTM models.

5.3.2 Model Generalization and Feature Importance

To evaluate our proposal in a more realistic cloud environment, we implemented our solution in another context different from where the models were created. This new environment is similar to where the models were trained regarding the number of clients and software used, but differs in computing resources (more computing resource capacity). Chapter 6 details this environment.

Since computing capacity is different, we performed an experiment to see if our models would generalize in this new environment, eliminating the need to conduct the

entire data collection and model building processes (or even retraining the models). In this case, we consider the model's capacity to generalize by measuring the model's error in this new environment. Therefore, we collect data for both applications using the software described in section 5.1.1; the interference generation described in section 5.1.2, and we collected all the container and worker node metrics described in Table 5.4. In addition, we also collect video metrics as described in section 5.1.3 Finally, we tested the performance of LSTM models on this data.

Table 5.8 shows the average RMSE for both models. The experiment was done with 10 video sections. As can be seen, on average, both models keep the RMSE lower than the RMSE obtained in the validation dataset. This result can be attributed to the use of techniques to improve the model's generalizability and avoid overfitting.

Table 5.8. Evaluation of Models' Generalization in Another Environment.

Model (LSTM)	RMSE Average ± CI (95%)	# Video Session
VoD	0.043 ± 0.015	10
Live Virtual Classroom	0.033 ± 0.016	10

In addition to creating good models, the literature has shown the importance of trying to explain them [Holzinger, 2018]. To better understand our models, we quantify the impact of each feature on the model's predictions. This concept is related to the interpretability of models, and we employ the permutation importance technique [Fisher and Dominici, 2018]. Permutation importance (PI) randomly shuffles a single column of the validation dataset, while keeping the output value and other columns unchanged. Besides that, PI is calculated after the model is trained; thus, it does not affect the model. The main idea is to measure the error in the shuffled column. As a result, an important feature will cause a significant change in the error, while less important ones will make a minor change.

Figures 5.5 and 5.6 show the PI results, considering the five most important features for the VoD and Live Virtual Classroom models, respectively. Although the container memory cache metric had a higher Spearman Correlation with MOS, the most important features of the VoD model came only from the worker node computing metrics. This result suggests that the model learned the relationship between interference on the worker node and the user's QoE.

On the other hand, the most important feature of the Live Virtual Classroom model came from container network metrics. Although container computing metrics are in the set of features (such as CPU, Mem, and Disk), this is an expected result, considering that when there is more container network transmission capacity, the HLS.js

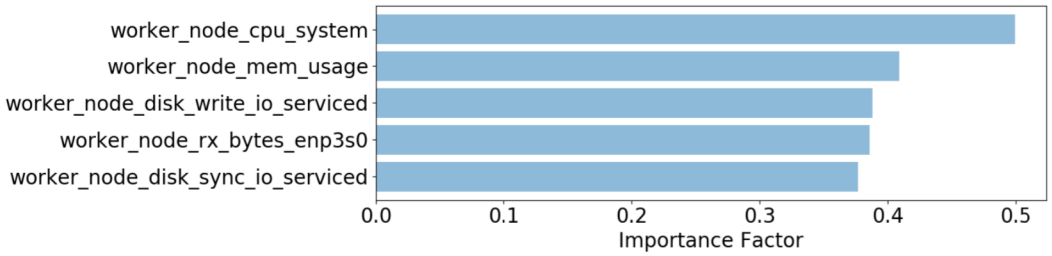

Figure 5.5. Feature importance permutation - VoD Model

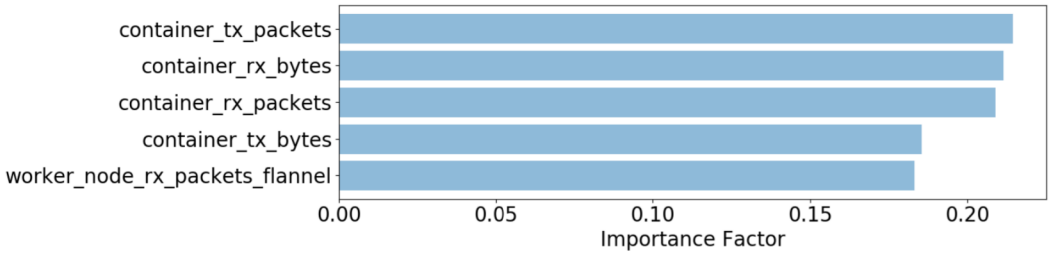

Figure 5.6. Feature importance permutation - Live Virtual Classroom Model

client can request higher quality chunks (which are larger in size). Furthermore, even though the CPU metrics of the worker node were not considered in the model's input, we know that the unavailability of this resource can affect the transmission capacity in the container network, for example, interfering in bandwidth and delay [Cai et al., 2021]. Also, the worker node's high CPU and memory usage can cause stall events, implying a decrease in transmissions. Thus, this result suggests that the model is responsive to interference in the work node, which becomes noticeable in the container's network utilization.

5.4 Summary

This chapter showed how the training datasets for both applications were created, which included the data collection process and feature selection to decrease the model training time without affecting the predictors' quality. In addition, we explained the use of a time series-based model and how the models were trained using two RNN variations (LSTM and GRU). Furthermore, we present which hyperparameters were evaluated and those used in the final models. Finally, we discussed the models' results in terms of performance and generalization. Also, we use an interpretability method to identify which features are most important to prediction.

The next chapter will employ both models on our system architecture, specifically on the ML-Based QoE Monitor component, and evaluate our proposal in an experimental environment.

Chapter 6

Experimental Evaluation

This chapter presents an experimental evaluation of our proposal presented in Chapter 4. Section 6.1 shows the experimental setup, which included the testbed used and the environment configuration. Then, Section 6.2 shows the results. Finally Section 6.3 summarizes this chapter.

6.1 Experimental Setup

The objective of the experimental evaluation is to assess how QoE-aware scheduling can improve the users' QoE. This section presents the experimental setup in which we implement the system architecture presented in Figure 4.2 including 14 clients on a remote testbed called Virtual Wall[1].

The Virtual Wall testbed is hosted at imec IDLab.t[2] and has a complete ecosystem to create a cloud computing environment, including servers to be used as physical machines or as virtual resources [Municio et al., 2021]. Figure 6.1 shows the Testbed.

Table 6.1 shows the machine's configuration used in our experiment. We allocated five physical machines for the Kubernetes Engine (version V1.17.3) installation: One to the master node and four to the worker nodes. Also, we allocated one physical machine to implement our control plane, which runs Cloud Resource Monitor, ML-Based QoE Monitor, and Scheduler Decision Algorithm. Besides that, we allocated 14 virtual machines as clients, seven for VoD and seven for the Live Virtual Classroom application. These virtual machines were allocated to different physical machines from those already mentioned. Each virtual machine runs Firefox Version 79.

[1]https://doc.ilabt.imec.be/ilabt/virtualwall/
[2]https://idlab.technology/
[3]Note: One for Control Plane and another to Data Plane

Figure 6.1. Virtual Wall testbed - Figure from virtual wall website

Table 6.1. Servers and clients configuration

Servers (6)	Clients (14)
2x Hexacore 2.4 GHz	2x Dual core AMD opteron 2GHz
24GB RAM	4GB RAM
1x 250GB HD	1x 80GB HD
2 X 1 gigabit nic[3]	1 gigabit nic
Ubuntu Server	Ubuntu Desktop 18.04

In terms of software used to implement our control plane, we use the cAdvisor version 0.35 into a docker container (version 20.10.2) and Python 3 to collect the container and worker node metrics on the Cloud Resource Monitor. Besides that, the ML-Based QoE monitor uses Keras 2.2.2 to read the VoD and Virtual Classroom models. Finally, the Scheduler Decision Algorithm was implemented using Kubernetes Python Client Version 17.17.0 (https://github.com/kubernetes-client/python). This means that our algorithm was implemented in Python 3.

We evaluated three HPA policies using different memory usage thresholds on HPA configuration for both applications, being 50%, 80%, and 90%. The threshold values were set based on the average memory usage of the seven clients connected in the container. For each application, we conducted 20 experiments with clients watching the video without extra workload and then stored the container memory usage (using cAdvisor). Then, we calculate the average usage over the highest values obtained from each repetition.There is a difference in the average memory usage between applications,

where the Live Virtual Classroom application consumes more memory than VoD. This is because the Wowza server performs the video transcoding in real-time, even without any connected user requesting the chunks. On average, in the worst case, the memory consumption was 5 MB in the VoD application and 2.5GB in the Live Virtual Classroom. Then, we calculate the 50%, 80%, and 90% memory thresholds out of that value, obtaining 2,5MB, 4MB, and 4,5MB for the VoD application. The threshold used for the Live Virtual Classroom application was 1.25GB, 2GB, and 2.25GB, respectively. This means that new containers will be scaled (Kubernetes horizontal autoscaling) when resource usage achieves these values, as explained in Section 2.2.

As described, the Container Scheduler Decision Algorithm (Algorithm 1) performs container scheduling considering an SLO threshold value and a time interval T. In our experiment, we configure the SLO value as QoE value to 3 and the time T of 10 seconds. Also, the moving average was calculated in the *ContainerResourceUsage()* function based on the memory usage in a 30-second window for the VoD Application. For Live Virtual Classroom, we consider the CPU usage over a 10-second window. Window values were based on the models' configuration. Finally, we created a scenario where each worker node received a different extra workload to simulate different co-located application conditions. This scenario follows the methodology explained in section 5.1.2. Table 6.2 shows the extra workload in each worker node. In Disk I/O, the stress ranged randomly from 1 to 40 processes, each writing and reading 1GB and 2GB.

Table 6.2. Extra workload in each worker node

	CPU(%)	Memory (%)	Disk I/O	Network
Worker 1	50 - 90	5 - 90	0	0
Worker 2	50 - 55	10 - 90	1GB	1024MB
Worker 3	0	10 - 25	0	0
Worker 4	10 - 15	15 - 20	2GB	1024MB

The following evaluations were made with VoD and Live Virtual Classroom original video containers deployed on worker node 1 simultaneously. Finally, 20 repetitions, each with 9 minutes and 10 seconds, were performed, and results consider a confidence interval of 95%.

6.2 Results

This section compares our proposal container scheduler with two baselines: the default Kubernetes-Scheduler (KS) and KCSS [Menouer, 2021]. We implemented two versions

of the KCSS algorithm. The first one has as criteria the maximization of resource usage (KCSS Max), while the second the minimization (KCSS Min). As mentioned in related work, the KCSS algorithm allows using multiple metrics and objectives. To become more comparable with our solution, for both KCSS versions, we consider the same worker node metrics used in our models (Table 5.4).

However, Kubernetes and KCSS do not perform container rescheduling. Therefore, to be fair in the evaluation, we evaluated our scheduler and rescheduler in separate experiments. We have named these experiments in the following graphs and tables as QoE-Scheduler and QoE-Rescheduler (respectively, performing both scheduling and rescheduling).

6.2.1 Container Scheduling

The first evaluation considers the schedulers and rescheduler's effects on the number of scaled and used containers in each HPA policy. Figures 6.2 and 6.3 show the mean number of scaled and used containers, respectively, for each experiment in each HPA policy as well as for each application. It was measured by the amount of containers scaled, and we consider as used containers only those containers that received a request to download the video.

As Figure 6.2 shows, there is a decrease in the mean amount of scaled containers among HPA policies for both applications. This occurs because users can achieve 50% of memory usage more quickly than 80% and 90%, which demands more containers to balance the workload until it is below the threshold. The highest availability of containers implies an increase in the number of used containers. As Figure 6.3 shows, with HPA 50% threshold, seven containers were always used for each experiment in each application. This means one user per container, which explains the confidence intervals to be zero.

Our QoE-Scheduler reduced the number of scaled containers in HPA 50% and 80% for both applications and in HPA 90% for the VoD application. In particular, for the Live Virtual Classroom application in HPA 90%, the Kubernetes-Scheduler achieves the same amount of scaled containers as our QoE-Scheduler. However, in general, our solution achieves a more significant reduction of scaled containers when compared to Kubernetes-Scheduler and KCSS Max; meanwhile, KCSS min kept the amount of scaled containers closer to our solutions as in VoD HPA 50% and Live Virtual Classroom HPA 80%. Such improvement can be explained by how each scheduler distributed the new video containers among the worker nodes. The Kubernetes-Scheduler, KCSS Max, and KCSS Min scheduled several of these new containers for work node 1 and work

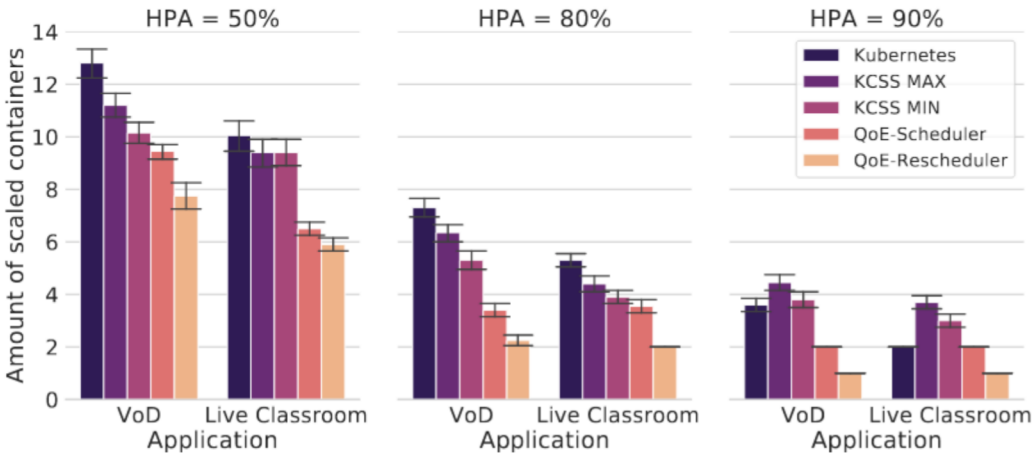

Figure 6.2. Amount of scaled containers

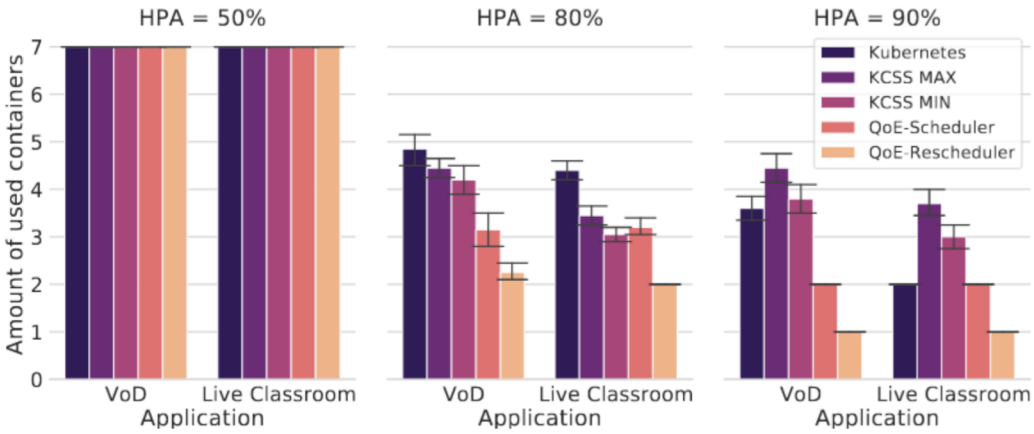

Figure 6.3. Amount of used containers

node 2, where there was more interference due to high resource usage. In contrast, our solutions did not schedule any extra containers for worker node 1 or 2. Besides, QoE-rescheduler deleted the original container from worker node 1 and scheduled it among workers 3 and 4.

Figure 6.4 shows the mean number of scaled containers per worker node, considering both applications in each HPA policy. Kubernetes-Scheduler uses the Round-Robin algorithm, and because of that, the KS has a better mean distribution among the four worker nodes. On the other hand, KCSS Min distributed more containers among workers 1, 2, and 3. This is because KCSS Min tends to choose the worker node with less resource usage. Due to the random extra workload at each worker node, the resource utilization of worker nodes 1, 2, and 3 may be less than that of worker node 4 and vice-versa.

In contrast, the KCSS Max compacts the containers in those worker nodes with

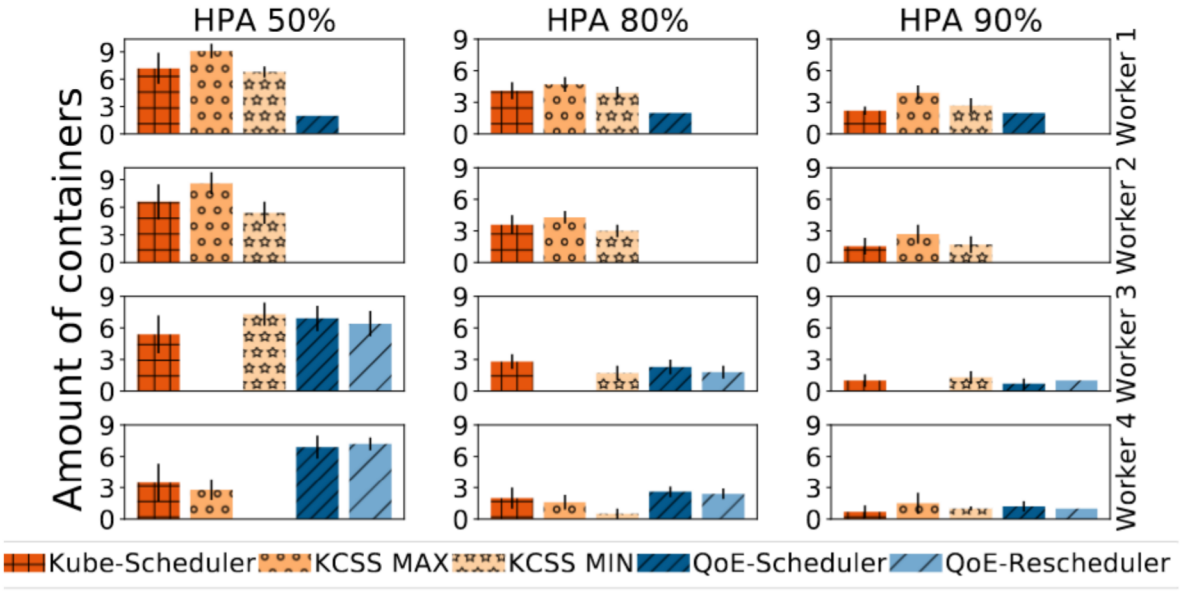

Figure 6.4. Mean number of containers scaled per worker node

higher resource utilization. Therefore, as worker nodes 1, 2, and 4 have more resource usage, the KCSS Max scheduler distributed the container among these worker nodes. Finally, our solutions distributed the containers among worker nodes 3 and 4. This is because the VoD and Live Virtual Classroom models predicted a higher QoE on these worker nodes.

Table 6.3. Mean Over-Provisioning Reduction

	Kubernetes Scheduler	KCSS Max	KCSS Min	QoE-Scheduler	QoE-Rescheduler	
HPA 50%	45.3%	37,5%	31,3%	26,3%	10,2%	VoD
HPA 80%	32,8%	29,6%	20,7%	5,8%	0%	
HPA 50%	30,6%	25,5%	25,5%	7,6%	10,1%	Live Virtual Classroom
HPA 80%	16,9%	20,4%	20,5%	11,1%	0%	

The way our proposal distributed the containers among the work nodes contributes to reducing over-provisioning, which is characterized by wasted resources and extra monetary costs [Qu et al., 2018]. This work considers over-provisioning as the difference between scaled containers (Figure 6.2) and used containers (Figure 6.3). Table 6.3 shows the over-provisioning (percentage) for each experiment. We omit HPA 90% because there was no over-provisioning. As can be seen, there was over-provisioning in all schedulers; however, our QoE-Scheduler decreased it in both HPA policies. Furthermore, QoE-Rescheduler in HPA 80% eliminates the over-provisioning.

6.2.2 QoE improvement

Although the Kubernetes-Scheduler, KCSS Max, and KCSS Min do not perform QoE-aware container scheduling, we measure the effects of these schedulers on the user's QoE and compare it with our proposal. In other words, we compare the scheduler's impact on the user's QoE when it is QoE-aware with those non-QoE-aware.

Figure 6.5 shows the mean of the user's QoE obtained in each experiment for both applications. We perform this analysis only with the 80% and 90% HPA policies since these are more realistic scenarios (with more than one client per container). As can be seen, in both HPA policies, our QoE-Scheduler improves the user's QoE in each application. For example, in the HPA 80% for VoD application, QoE-Scheduler improves the QoE by 95.2%, 86.3%, and 78.2%, from 2.1, 2.2, and 2.3 to 4.1, compared with Kubernetes-Scheduler, KCSS Max, and KCSS Min, respectively. Furthermore, for the Live Virtual Classroom app also in HPA 80%, QoE-Scheduler improves the QoE by 61.5%, 82.6%, and 75%, from 2.6, 2.3, and 2.4 to 4.2. This improvement occurs because the QoE-Scheduler chooses other worker nodes that increase the QoE (with less interference). Besides that, as a significant result, QoE-Scheduler kept the SLO at the proposed limit on the scheduler decision algorithm, which means that in both applications, the mean of the user's QoE was higher than 3. It is important to note that a good QoE predictor performance is essential to achieve this result. If the predictor erroneously estimates the container's user's QoE for a worker node with interference to the point of degrading the QoE, it can generate an SLO violation. However, this improvement was limited by the end-users with a degraded QoE in the container allocated in worker node one. Another observation is about a lower observed QoE in HPA 90% compared to HPA 80%. This is an expected result, considering that there was an under-provision in each experiment. Under-provision is associated with degraded performance [Qu et al., 2018].

On the other hand, we obtained a QoE close to the maximum value (5) using the QoE-Rescheduler. This is because the QoE-Rescheduler deleted the original containers from worker 1 and scheduled them on other worker nodes that can improve the user's QoE. Thus, although there is a cost in terms of time to perform QoE-Rescheduler, which means deleting the original container from worker one and scaling it into another worker node, the rescheduling process was needed to improve the QoE. Also, QoE-Rescheduler achieved a similar mean QoE in both HPA policies. This was expected since the scheduler works regardless of HPA policies. The container scheduling decision depends only on QoE models, which predict QoE degradation and reschedule the container without considering the HPA operation.

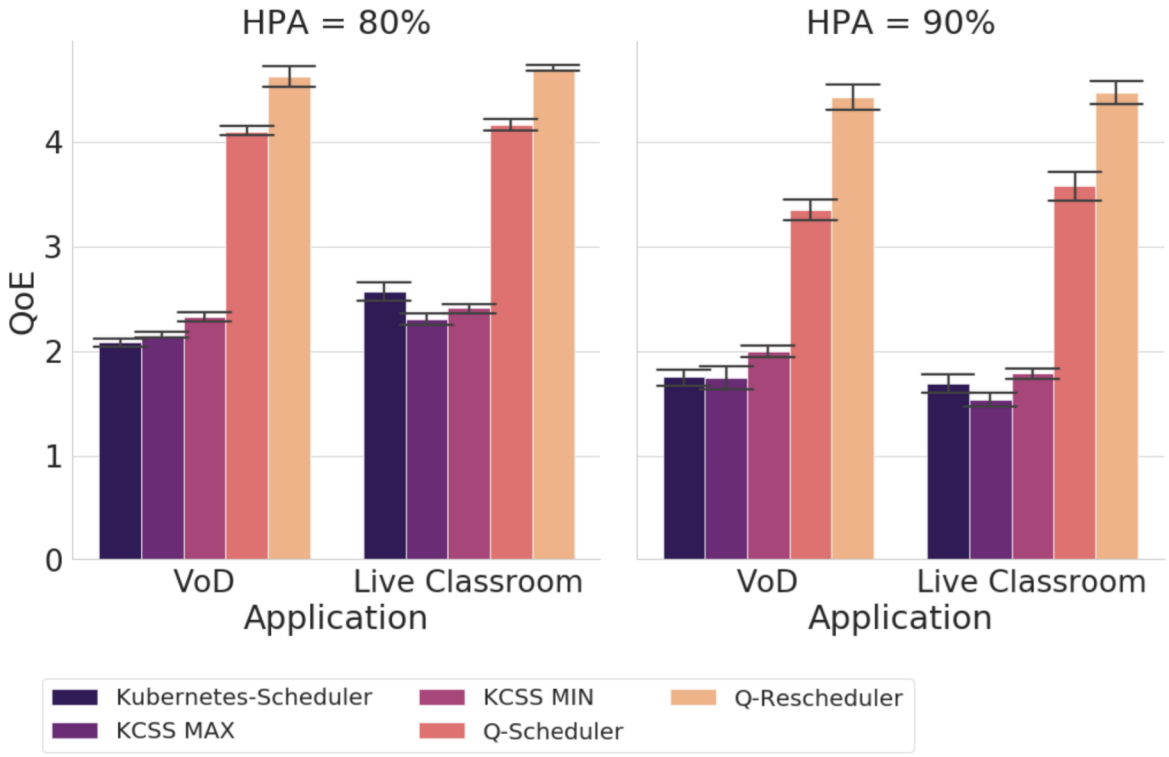

Figure 6.5. Quality of experience perceived by the clients

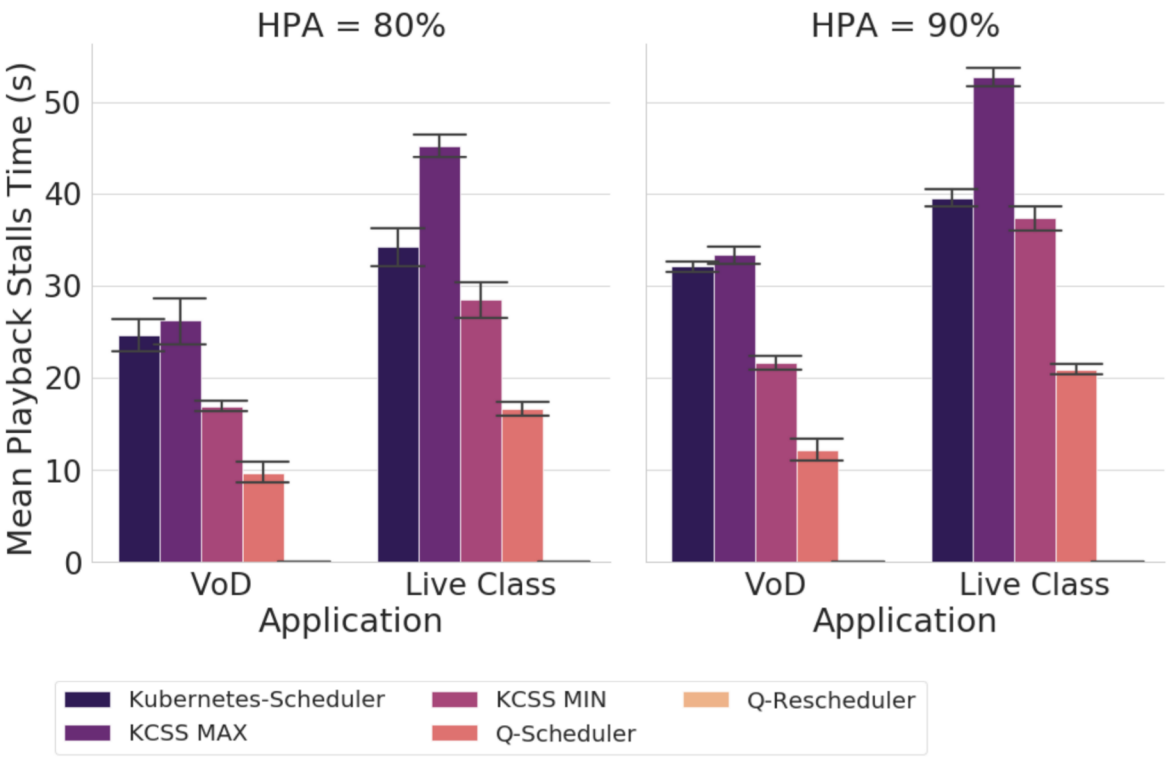

Figure 6.6. Mean playback stall time during video session

Another QoE aspect evaluated was the mean playback stall time (in seconds) during video sessions. Figure 6.6 shows the comparison between our solution and the baselines. As can be seen, our QoE-Scheduler reduced the mean playback stall time in both HPA and applications. For example, in the VoD application on HPA 80%, our solution reduced by 60.9%, 63.3%, and 43,1%, from 24.6, 26.2, and 16.9 to 9.6 seconds, compared with Kubernetes-Scheduler, KCSS Max, and KCSS min, respectively. Also, for the Live Virtual Classroom application on the HPA 80%, the QoE-Scheduler decreased the mean playback stall time by 51.1%, 63.0%, and 41.4% from 34.2, 45.2, and 28.5 to 16.7. This reduction can be attributed to the scheduled containers located at worker nodes 3 and 4 contributing to a better response to user requests. For the Live Virtual Classroom application, this means more computing resources to real-time transcode the video. Finally, our QoE-Rescheduler eliminates the stall event. Again, the QoE-Rescheduler deleted the original video from worker 1 before the QoE degradation. Despite that, in both cases (rescheduling, migration), the application's connection can be interrupted with the server and, consequently, cause stall events. However, it did not occur during container rescheduling. One explanation for this is the replay of chunks in the application buffer while the rescheduling process took place.

When comparing the mean playback stall time between applications, it is possible to notice that the stall time is higher in the Live Virtual Classroom application. This is an expected result considering that the Live Virtual Classroom was more likely to have a stall event due to the lack of computing resources (caused by the co-located application and interference) to perform the real-time video transcoding. On the other hand, in VoD applications, the same lack of resources can be compensated for by transmitting smaller chunks. In addition, the mean playback stall time was higher in the HPA 90%, which we also considered an expected result due to the higher needed time to achieve 90% of memory usage and then perform container scheduling. This means that the container spends more time under interference at worker node 1.

Finally, we measured the mean number of resolution changes during the video transmission in both applications. This was calculated as the sum of each user's video resolution changes, including the resolution change and the bitrate. In fact, some users may experience resolution changes more often than others; however, we are interested in analyzing this result based on the group's perception. Table 6.4 shows the result. As can be seen, our proposal also reduced the mean number of resolution changes in both experiments. Again, this improvement is a consequence of the choice of work nodes with better conditions to deploy the containers. Also, the mean number of resolution changes has been reduced to zero on the QoE-Rescheduler because the model predicts

a QoE decay in advance.

In addition, in the HPA 90%, there were more resolution changes than in HPA 80%. Again, this can be explained by the time it takes to achieve 90% of memory usage. Furthermore, in contrast to the result of the mean playback stall time, where there was more stall event in the Live Virtual Classroom application than in the VoD application, in this case, resolution changes occur more often in the VoD application. This can be explained by differences in adaptive streaming configuration between applications. As Table 5.1 shows, VoD has a set of more bitrate settings options than Live Virtual Classroom, allowing DASH to switch more often.

Table 6.4. Mean number of resolution changes

	VoD		Live Virtual Classroom	
	HPA 80%	HPA 90%	HPA 80%	HPA 90%
Kubernetes-Scheduler	43 ± 1.1	56 ± 3	23 ± 2.1	33 ± 3
KCSS Max	46 ± 2.3	62 ± 2	20 ± 1.4	28 ± 2.1
KCSS Min	39 ± 0.6	68 ± 3.1	36 ± 0.9	46 ± 1.8
QoE-Scheduler	26 ± 0.9	42 ± 2	21 ± 0.3	28 ± 2
QoE-Rescheduler	0	0	0	0

6.3 Summary

This chapter evaluates the performance of our QoE-aware scheduler and rescheduler in a real cloud environment. The first sessions present the methodology used to deploy the cloud environment, the interference generation (to simulate application co-location), and our algorithm's parameters.

Experimental results suggest that QoE-aware container scheduling can improve users' QoE by 95.2% compared to the default Kubernetes-Scheduler, and by up to 86.3% between the two KCSS versions. In addition, the mean QoE value remained above the pre-established value in the SLO configuration. In addition, other important factors in the users' QoE were improved, such as the stall time and the change of resolutions. Our solution also proved to be more efficient than the analyzed baselines in terms of scaled container amount. We attribute this improvement to the ability of application models to rank work nodes according to their capacity to provide better QoE to users. In other words, the way the containers were distributed in the cloud contributes to the reduction of scaled and used containers. With that, our proposal reduced the over-fitting in the best case by 5.8% using the QoE-Scheduler. Although

our proposal outperformed the analyzed baselines, the results showed that the KCSS Min was the approach that came closest to our results.

Chapter 7

Conclusion and Future Work

This work proposes a QoE-aware container for co-located cloud environment. First, two models were built using deep machine learning based on work node and container metrics. Then, the models were used to rank the work nodes according to the expected QoE in each one of them. With that, our scheduling algorithm chooses the work node that maximizes and keeps the users' QoE above a pre-established SLO.

Unlike other works found in the literature, our proposal estimates the QoE considering only the cloud state. In other words, estimates the QoE the cloud can offer to content provider clients. Also, we consider the degradation/interference caused by application co-location, which many cloud providers often practice [Cheng et al., 2018]. Furthermore, we carry out the container rescheduling process when the proposed models identify a QoE degradation.

The main contributions of this dissertation are the proposal of a QoE-aware container scheduler/rescheduler and the methodology for creating machine learning models to estimate QoE within the cloud. Furthermore, the implementation and evaluation in a real environment, considering one of the most used cloud systems nowadays (Kubernetes), contributes more effectively to the evaluation of the proposal.

It was verified, through experimental evaluation, that QoE-aware scheduling increased the quality of experience of users compared to methods that do not consider. Furthermore, other user experience factors were evaluated, such as the average stall event time and resolution changes. The results showed that choosing the most suitable working node in terms of QoE decreases the average number of stall events during video transmission as well as the number of changes of resolution in the applications. In addition, our proposal reduced the number of scheduled containers and, consequently, the number of used containers, thus reducing over-provisioning.

7.1 Future Work

As mentioned, to the best of our knowledge, this is the first work that considers users' QoE in the container scheduling process in a cloud environment with co-located applications. Given this, we found some future research opportunities derived from our work.

The first proposal for future work is related to the complexity of the evaluated scenario. The experiments performed had few clients for each application, and therefore the interaction and concurrency of resources inside the container were not evaluated. Furthermore, our evaluation was also restricted to a few work nodes.

Our current proposal considers applications with only one tier, in which the application's front-end and back-end are running in the same container. A second proposal as future work is to evaluate our proposal in a scenario with multiple tiers in which there is a dependency between containers. This future work proposal refers to limitation #2 in section 4.5.

A third proposal is to consider the user's QoE measured from another technology (such as in Internet Service Providers (ISPs), Wi-Fi and 4/5G network) in the end-to-end path between users and cloud to be combined with our solution. This means, for example, considering the QoE measured in a Wi-Fi network (or even in the end devices) to improve the container scheduling/rescheduling decision algorithm. On the other hand, cloud QoE measurement can be combined with other solutions that act on other technologies, such as the links of the Internet Service Providers (ISPs).

7.2 Publications

As a preliminary result during the development of this dissertation, we published a scientific article (list below), and we are preparing an extended version for a journal submission.

- Carvalho, Marcos and Macedo, Daniel Fernandes. **QoE-Aware Container Scheduler for Co-located Cloud Environments**. IFIP/IEEE International Symposium on Integrated Network Management (IM), 2021.

Appendix A

Metrics Description

Table A.1. Metris Description

Metrics	Description
cpu_usage	Total of time CPU usage.
cpu_user	Amount of time a process has direct control of the CPU.
cpu_system	Amount of time the kernel is executing system calls.
diskio_async_io_service	Indicates the number of bytes async operation.
diskio_read_io_service	Indicates the number of bytes read operation.
diskio_sync_io_service	Indicates the number of bytes sync operation.
diskio_total_io_service	Indicates the number of bytes total I/O operations.
diskio_write_io_service	Indicates the number of bytes write operation.
disk_async_io_serviced	The number of async operations performed.
disk_discard_io_serviced	The number of discard block (not in use by file system).
disk_read_io_serviced	The number of read operations performed.
disk_sync_io_serviced	The number of sync operations performed
disk_total_io_serviced	The total of I/O operations performed.
disk_write_io_serviced	The number of write operations performed.
mem_usage	Total of memory usage (in bytes)
mem_max_usage	Show max memory usage recorded (in bytes)
mem_cache	The amount of memory used by the processes of
mem_rss	The amount of memory that does not correspond to anything on disk (in bytes)
mem_swap	The amount of swap currently used by the processes (in bytes)
mapped_file	Indicates the amount of memory mapped by the processes (in bytes).
working_set	The amount of working set memory (in bytes).
mem_failcnt	The number of times that the requested memory failed.
mem_container_pgfault	Indicates the number of times of page faults.
mem_container_pgmajfault	Indicates the number of times of major fault.
filesystem_capacity	The capacity on the file system
filesystem_usage	The bytes used on the filesystem.
filesystem_base_usage	Base usage consumed by the container's writable layer.
filesystem_inodes	The number of used inodes on a file system.
rx_bytes	The count of bytes received.
rx_packets	The count of packets received.
tx_bytes	The count of bytes transmitted.
tx_packets	The count of packets transmitted.

As mentioned in section 4.3.1, the number of metrics collected was 72, being 32

from the container and 40 from the worker node. However, both the container and the worker node have the same metrics collected; the difference is that the worker node has more network interface. Table A.1 describes the metrics collected from container and work node. These metrics are named as in the Kubernetes API (column 1), and we presented a brief description[1,2].

[1]https://docs.docker.com/config/containers/runmetrics/

[2]https://github.com/splunk/fluent-plugin-kubernetes-metrics/blob/develop/metrics-information.md

Appendix B

JSON Input and Output Format

Appendix B shows an example of the JSON input format for Model 0, used in this work, and the respective output format. In short, the input content is video segment information (bitrate, codec, fps, resolution), aggregated by start and duration ("I13"). Also, the stall event is recorded as a list of pairs of timestamps and how long the stall lasted (duration) ("I23"). Finally, the type of device configuration is also required. This field required device type (pc, handheld, or mobile), display size, and viewing distance ("IGen").

The output JSON format is composed of per-second audiovisual measured as MOS value (1-5) ("034"). An overall stalling quality ("023"), audiovisual quality score ("035") and quality score ("046"). Notice that, as mentioned, this works does not consider audio on the user's QoE (will assume constant high-quality audio), which means that the 034 output refers to video quality only. Also, the stall event does not impact the MOS value. It is calculated separately on overall stalling quality ("023").

Finally, MOS value "034" is used as the user's QoE (QoE_{User_i}) on our ML model output (discussed in Section 4.3.2). Besides that, as the stall event is not considered on MOS, we do not consider it on VoD applications. However, we incorporate the stall event measured on JSON input ("I23") in the Live Virtual Classroom application (also discussed in Section 4.3.4).

Listing 1 P.1203 JSON Input Format

```
{
    "I11": {
    "segments" : [],
    "streamId" : 42
    }
    "I13": {
    "segments" : [
            {   "bitrate": 4200.0,
                "codec": "h264", //only "h264" supported in standard
                "duration": 3, //duration in s
                "fps": 24.0,
                "resolution": "1920x1080",
                "start": 0  //media start timestamp in s

            },
            {
                "bitrate": 791.0,
                "codec": "h264",
                "duration": 4,
                "fps": 24.0,
                "resolution": "1280x720",
                "start": 3
            },...
        ],
    "streamId" : 42
    },
    "I23": {
        "stalling": [[0,2],[4,1]] //[start timestamp, duration]
        "streamId": 42 //unique identifier for the stream
    },
    "IGen": {
        "device": "pc", //pc, handheld, or mobile
        "displaySize": "1920x1080",
        "viewingDistance": "150cm"
    }
    }
}
```

Listing 2 P.1203 JSON Output Format

```
{
    "O23": 4.219206291281412, //stalling quality
  "O34": [
   5.0, //per-second audiovisual quality scores
   5.0,
   4.67953852907711,
   4.67953852907711,
   4.67953852907711,
   4.67953852907711,
   4.67953852907711,
   5.0,
   5.0,
   5.0,
   ...,
   ]
  "O35": 4.997692214848531, //audiovisual quality score
  "O46": 4.3135308869955145, // overall quality score
  "date": "2020-11-27T14:24:21.356849",
  "mode": 0, // used mode
  "streamId": 42
}
```

Bibliography

Abdallah, M., Griwodz, C., Chen, K.-T., Simon, G., Wang, P.-C., and Hsu, C.-H. (2018). Delay-sensitive video computing in the cloud: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 14(3s):1--29.

Adhikari, M., Amgoth, T., and Srirama, S. N. (2019). A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Computing Surveys (CSUR)*, 52(4):1--36.

Agrawal, R. and Adhikari, R. (2013). An introductory study on time series modeling and forecasting. *Nova York: CoRR*.

Ahmad, I., AlFailakawi, M. G., AlMutawa, A., and Alsalman, L. (2021). Container scheduling techniques: A survey and assessment. *Journal of King Saud University-Computer and Information Sciences*.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., and Merle, P. (2017). Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430--447.

Alreshoodi, M. and Woods, J. (2013). Survey on qoe\qos correlation models for multimedia services. *arXiv preprint arXiv:1306.0221*.

Aral, A., Brandic, I., Uriarte, R. B., De Nicola, R., and Scoca, V. (2019). Addressing application latency requirements through edge scheduling. *Journal of Grid Computing*, 17(4):677--698.

Bampis, C. G., Li, Z., Moorthy, A. K., Katsavounidis, I., Aaron, A., and Bovik, A. C. (2017). Study of temporal effects on subjective video quality of experience. *IEEE Transactions on Image Processing*, 26(11):5217--5231.

Barakabitze, A. A., Barman, N., Ahmad, A., Zadtootaghaj, S., Sun, L., Martini, M. G., and Atzori, L. (2019). QoE management of multimedia streaming services in future networks: a tutorial and survey. *IEEE Communications Surveys & Tutorials*, 22(1):526--565.

Barman, N. and Martini, M. G. (2019). Qoe modeling for http adaptive video streaming–a survey and open challenges. *Ieee Access*, 7:30831--30859.

Cai, Q., Chaudhary, S., Vuppalapati, M., Hwang, J., and Agarwal, R. (2021). Understanding host network stack overheads.

Carvalho, M., Silva, V. F., e Silva, E. d. B., Macedo, D. F., de Resende, H. C., Marquez-Barja, J. M., Both, C. B., Bardini, A. Z., and Wickboldt, J. (2019). Qoe-based video orchestration for 4g networks. In *2019 IEEE 30th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, pages 1--6. IEEE.

Chakraborty, P., Dev, S., and Naganur, R. H. (2015). Dynamic http live streaming method for live feeds. In *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 1394--1398. IEEE.

Chalapathy, R. and Chawla, S. (2019). Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407*.

Chen, M., Challita, U., Saad, W., Yin, C., and Debbah, M. (2019a). Artificial neural networks-based machine learning for wireless networks: A tutorial. *IEEE Communications Surveys & Tutorials*, 21(4):3039--3071.

Chen, W., Ye, K., and Xu, C.-Z. (2019b). Co-locating online workload and offline workload in the cloud: An interference analysis. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 2278--2283. IEEE.

Cheng, Y., Chai, Z., and Anwar, A. (2018). Characterizing co-located datacenter workloads: An alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1--3.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Cisco (2017). Cisco visual networking index: Global mobile data traffic forecast update, 2017-2022 white paper.

da Rosa Righi, R., Lehmann, M., Gomes, M. M., Nobre, J. C., da Costa, C. A., Rigo, S. J., Lena, M., Mohr, R. F., and de Oliveira, L. R. B. (2019). A survey on global management view: toward combining system monitoring, resource management, and load prediction. *Journal of Grid Computing*, 17(3):473--502.

De Cicco, L., Mascolo, S., and Palmisano, V. (2019). QoE-driven resource allocation for massive video distribution. *Ad Hoc Networks*, 89:170--176.

de Oliveira, M. and Macedo, D. F. (2021). Slicing wi-fi links based on qoe video streaming fairness. *International Journal of Network Management*, page e2155.

Dou, Q., Castro, D. C., Kamnitsas, K., and Glocker, B. (2019). Domain generalization via model-agnostic learning of semantic features. *arXiv preprint arXiv:1910.13580*.

Duanmu, Z., Zeng, K., Ma, K., Rehman, A., and Wang, Z. (2016). A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154--166.

Dutta, S., Taleb, T., and Ksentini, A. (2016). Qoe-aware elasticity support in cloud-native 5g systems. In *2016 IEEE International Conference on Communications (ICC)*, pages 1--6. IEEE.

Elhabbash, A., Elkhatib, Y., Blair, G. S., Lin, Y., Barker, A., and Thomson, J. (2019). Envisioning slo-driven service selection in multi-cloud applications. In *IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pages 9--14.

Erfani, S. M., Rajasegarar, S., Karunasekera, S., and Leckie, C. (2016). High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning. *Pattern Recognition*, 58:121--134.

Eswara, N., Ashique, S., Panchbhai, A., Chakraborty, S., Sethuram, H. P., Kuchi, K., Kumar, A., and Channappayya, S. S. (2019). Streaming video qoe modeling and prediction: A long short-term memory approach. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(3):661--673.

Ferdaus, M. H., Murshed, M., Calheiros, R. N., and Buyya, R. (2017). An algorithm for network and data-aware placement of multi-tier applications in cloud data centers. *Journal of Network and Computer Applications*, 98:65--83.

Fisher, A., R. C. and Dominici, F. (2018). Model class reliance: Variable importance measures for any machine learning model class, from the ârashomonâ perspective.

Fu, R., Zhang, Z., and Li, L. (2016). Using lstm and gru neural network methods for traffic flow prediction. In *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, pages 324--328. IEEE.

Gao, C., Yan, J., Zhou, S., Varshney, P. K., and Liu, H. (2019). Long short-term memory-based deep recurrent neural networks for target tracking. *Information Sciences*, 502:279--296.

Ghadiyaram, D., Pan, J., and Bovik, A. C. (2017). A subjective and objective study of stalling events in mobile streaming videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 29(1):183--197.

Ghadiyaram, D., Pan, J., and Bovik, A. C. (2018). Learning a continuous-time streaming video qoe model. *IEEE Transactions on Image Processing*, 27(5):2257--2271.

Guarnieri, T., Drago, I., Vieira, A. B., Cunha, I., and Almeida, J. (2017). Characterizing qoe in large-scale live streaming. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1--7. IEEE.

Guo, J., Chang, Z., Wang, S., Ding, H., Feng, Y., Mao, L., and Bao, Y. (2019). Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1--10.

Guo, Y. and Yao, W. (2018). A container scheduling strategy based on neighborhood division in micro service. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1--6. IEEE.

Gupta, S. and Dileep, A. D. (2020). Long range dependence in cloud servers: a statistical analysis based on google workload trace. *Computing*, pages 1--19.

Gupta, S., Dileep, A. D., and Gonsalves, T. A. (2018). A joint feature selection framework for multivariate resource usage prediction in cloud servers using stability and prediction performance. *The Journal of Supercomputing*, 74(11):6033--6068.

Gupta, S. and Dinesh, D. A. (2017). Resource usage prediction of cloud workloads using deep bidirectional long short term memory networks. In *2017 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, pages 1--6. IEEE.

Haouari, F., Baccour, E., Erbad, A., Mohamed, A., and Guizani, M. (2019). Qoe-aware resource allocation for crowdsourced live streaming: A machine learning approach. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1--6. IEEE.

Hobfeld, T., Schatz, R., Varela, M., and Timmerer, C. (2012). Challenges of qoe management for cloud applications. *IEEE Communications Magazine*, 50(4):28--36.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735--1780.

Holzinger, A. (2018). From machine learning to explainable ai. In *2018 world symposium on digital intelligence for systems and machines (DISA)*, pages 55--66. IEEE.

Hora, D. N. d., Teixeira, R., Van Doorselaer, K., and Van Oost, K. (2016). Predicting the effect of home wi-fi quality on web qoe. In *Proceedings of the 2016 workshop on QoE-based Analysis and Management of Data Communication Networks*, pages 13--18.

Hoßfeld, T., Skorin-Kapov, L., Heegaard, P. E., and Varela, M. (2018). A new qoe fairness index for qoe management. *Quality and User Experience*, 3(1):4.

Imdoukh, M., Ahmad, I., and Alfailakawi, M. G. (2020). Machine learning-based auto-scaling for containerized applications. *Neural Computing and Applications*, 32(13):9745--9760.

ITU Telecommunication Standardization Sector (2017). ITU-T Rec P.1203: Parametric bitstream-based quality assessment of progressive download and adaptive audiovisual streaming services over reliable transport.

Juluri, P., Tamarapalli, V., and Medhi, D. (2015). Measurement of quality of experience of video-on-demand services: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):401--418.

Kafetzakis, E., Koumaras, H., Kourtis, M. A., and Koumaras, V. (2012). Qoe4cloud: A QoE-driven multidimensional framework for cloud environments. In *2012 international conference on telecommunications and multimedia (TEMU)*, pages 77--82. IEEE.

Kapočius, N. (2020). Performance studies of kubernetes network solutions. In *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1--6. IEEE.

Kavitha, B. and Varalakshmi, P. (2017). Performance analysis of virtual machines and docker containers. In *International Conference on Data Science Analytics and Applications*, pages 99--113. Springer.

Khan, M. A. and Salah, K. (2020). Cloud adoption for e-learning: Survey and future challenges. *Education and Information Technologies*, 25(2):1417--1438.

Kingma, D. and Ba, J. (2015). Adam: A method for stochastic optimization in: Proceedings of the 3rd international conference for learning representations (iclrâ15). *San Diego*.

Krogfoss, B., Duran, J., Perez, P., and Bouwen, J. (2020). Quantifying the value of 5g and edge cloud on qoe for ar/vr. In *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1--4. IEEE.

La, H.-L., Tran, A.-T. N., Le, Q.-T., Yoshimi, M., Nakajima, T., and Thoai, N. (2020). A use case of content delivery network raw log file analysis. In *2020 International Conference on Advanced Computing and Applications (ACOMP)*, pages 71--78. IEEE.

Lai, Y.-J., Liu, T.-Y., and Hwang, C.-L. (1994). Topsis for modm. *European journal of operational research*, 76(3):486--500.

Li, X., Darwich, M., Bayoumi, M., and Salehi, M. A. (2020). Cloud-based video streaming services: A survey. *arXiv preprint arXiv:2011.14976*.

Li, X. and Wu, X. (2015). Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4520--4524. IEEE.

Liu, B., Li, P., Lin, W., Shu, N., Li, Y., and Chang, V. (2018). A new container scheduling algorithm based on multi-objective optimization. *Soft Computing*, 22(23):7741--7752.

Liu, L. and Qiu, Z. (2016). A survey on virtual machine scheduling in cloud computing. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 2717--2721. IEEE.

Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., and Zhang, G. (2018). Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346--2363.

Madni, S. H. H., Abd Latiff, M. S., Coulibaly, Y., et al. (2017). Recent advancements in resource allocation techniques for cloud computing environment: a systematic review. *Cluster Computing*, 20(3):2489--2533.

Maenhaut, P.-J., Volckaert, B., Ongenae, V., and De Turck, F. (2020). Resource management in a containerized cloud: status and challenges. *Journal of Network and Systems Management*, 28(2):197--246.

Malakar, S., Goswami, S., Ganguli, B., Chakrabarti, A., Roy, S. S., Boopathi, K., and Rangaraj, A. (2021). Designing a long short-term network for short-term forecasting of global horizontal irradiance. *SN Applied Sciences*, 3(4):1--15.

Mao, Y., Oak, J., Pompili, A., Beer, D., Han, T., and Hu, P. (2017). Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1--8. IEEE.

Masdari, M. and Khoshnevis, A. (2020). A survey and classification of the workload forecasting methods in cloud computing. *Cluster Computing*, 23(4):2399--2424.

Medel, V., Tolón, C., Arronategui, U., Tolosana-Calasanz, R., Bañares, J. Á., and Rana, O. F. (2017). Client-side scheduling based on application characterization on kubernetes. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 162--176. Springer.

Mei, Y., Cheng, L., Talwar, V., Levin, M. Y., Jacques-Silva, G., Simha, N., Banerjee, A., Smith, B., Williamson, T., Yilmaz, S., et al. (2020). Turbine: Facebookâs service management platform for stream processing. In *International Conference on Data Engineering (ICDE)*, pages 1591--1602.

Menouer, T. (2021). Kcss: Kubernetes container scheduling strategy. *The Journal of Supercomputing*, 77(5):4267--4293.

Miranda, G., Macedo, D. F., and Marquez-Barja, J. M. (2020). A qoe inference method for dash video using icmp probing. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1--5. IEEE.

Moura, H. D., Macedo, D. F., and Vieira, M. A. (2020). Wireless control using reinforcement learning for practical web qoe. *Computer Communications*, 154:331--346.

Moysen, J. and Giupponi, L. (2018). From 4g to 5g: Self-organized network management meets machine learning. *Computer Communications*, 129:248--268.

Municio, E., Cevik, M., Ruth, P., and Marquez-Barja, J. M. (2021). Experimenting in a global multi-domain testbed. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1--2. IEEE.

Park, Y., Yang, H., and Kim, Y. (2018). Performance analysis of cni (container networking interface) based container network. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 248--250. IEEE.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2013). How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.

Patrick Le Callet, S. M. and Andrew Perkis, eds., L. S. (2012). Qualinet white paper on definitions of quality of experience. *European Network on Quality of Experience in Multimedia Systems and Services (COST Action IC 1003)*, 4(5):2.

Peinl, R., Holzschuher, F., and Pfitzer, F. (2016). Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265--282.

Priya, V., Kumar, C. S., and Kannan, R. (2019). Resource scheduling algorithm with load balancing for cloud service provisioning. *Applied Soft Computing*, 76:416--424.

Qi, S., Kulkarni, S. G., and Ramakrishnan, K. (2020). Understanding container network interface plugins: design considerations and performance. In *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN*, pages 1--6. IEEE.

Qu, C., Calheiros, R. N., and Buyya, R. (2018). Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):1--33.

Quinlan, J. J., Zahran, A. H., and Sreenan, C. J. (2016). Datasets for avc (h. 264) and hevc (h. 265) evaluation of dynamic adaptive streaming over http (dash). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1--6.

Qureshi, K. N., Din, S., Jeon, G., and Piccialli, F. (2020). Internet of vehicles: Key technologies, network model, solutions and challenges with future aspects. *IEEE Transactions on Intelligent Transportation Systems*, 22(3):1777--1786.

Reimers, N. and Gurevych, I. (2017). Optimal hyperparameters for deep lstm-networks for sequence labeling tasks. *arXiv preprint arXiv:1707.06799*.

Ren, R., Li, J., Wang, L., Zhan, J., and Cao, Z. (2018). Anomaly analysis for co-located datacenter workloads in the alibaba cluster. *arXiv preprint arXiv:1811.06901*.

Robitza, W., Göring, S., Raake, A., Lindegren, D., Heikkilä, G., Gustafsson, J., List, P., Feiten, B., Wüstenhagen, U., Garcia, M.-N., et al. (2018). Http adaptive streaming qoe estimation with itu-t rec. p. 1203: open databases and software. In *Proceedings of the 9th ACM Multimedia Systems Conference*, pages 466--471.

Santos, G., Paulino, H., and Vardasca, T. (2020). Qoe-aware auto-scaling of heterogeneous containerized services (and its application to health services). In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 242--249.

Santos, J., Wauters, T., Volckaert, B., and De Turck, F. (2019). Towards network-aware resource provisioning in kubernetes for fog computing applications. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 351--359. IEEE.

Seufert, M., Egger, S., Slanina, M., Zinner, T., Hoßfeld, T., and Tran-Gia, P. (2014). A survey on quality of experience of http adaptive streaming. *IEEE Communications Surveys & Tutorials*, 17(1):469--492.

Shrestha, A. and Mahmood, A. (2019). Review of deep learning algorithms and architectures. *IEEE Access*, 7:53040--53065.

Singh, S. and Chana, I. (2016a). Cloud resource provisioning: survey, status and future research directions. *Knowledge and Information Systems*, 49(3):1005--1069.

Singh, S. and Chana, I. (2016b). Resource provisioning and scheduling in clouds: Qos perspective. *The Journal of Supercomputing*, 72(3):926--960.

Skorin-Kapov, L., Varela, M., Hoßfeld, T., and Chen, K.-T. (2018). A survey of emerging concepts and challenges for qoe management of multimedia services. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 14(2s):1--29.

Slivar, I., Skorin-Kapov, L., and Suznjevic, M. (2016). Cloud gaming qoe models for deriving video encoding adaptation strategies. In *Proceedings of the 7th international conference on multimedia systems*, pages 1--12.

Slivar, I., Skorin-Kapov, L., and Suznjevic, M. (2019). Qoe-aware resource allocation for multiple cloud gaming users sharing a bottleneck link. In *2019 22nd conference on innovation in clouds, internet and networks and workshops (ICIN)*, pages 118--123. IEEE.

Soldani, J., Tamburri, D. A., and Van Den Heuvel, W.-J. (2018). The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215--232.

Song, B., Yu, Y., Zhou, Y., Wang, Z., and Du, S. (2018). Host load prediction with long short-term memory in cloud computing. *The Journal of Supercomputing*, 74(12):6554--6568.

Stockhammer, T. (2011). Dynamic adaptive streaming over http– standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 133--144.

Takahashi, K., Aida, K., Tanjo, T., and Sun, J. (2018). A portable load balancer for kubernetes cluster. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 222--231.

Torre, R., Urbano, E., Salah, H., Nguyen, G. T., and Fitzek, F. H. (2019). Towards a better understanding of live migration performance with docker containers. In *European Wireless 2019; 25th European Wireless Conference*, pages 1--6. VDE.

Tosatto, A., Ruiu, P., and Attanasio, A. (2015). Container-based orchestration in cloud: state of the art and challenges. In *2015 Ninth international conference on complex, intelligent, and software intensive systems*, pages 70--75. IEEE.

Wang, M., Cui, Y., Wang, X., Xiao, S., and Jiang, J. (2017). Machine learning for networking: Workflow, advances and opportunities. *IEEE Network*, 32(2):92--99.

Wei, L., Cai, J., Foh, C. H., and He, B. (2016). Qos-aware resource allocation for video transcoding in clouds. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(1):49--61.

Wu, J., Cheng, B., Yang, Y., Wang, M., and Chen, J. (2017). Delay-aware quality optimization in cloud-assisted video streaming system. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 14(1):1--25.

Xiao Yuan, A. C. (2019). A brief analysis on the implementation of the kubernetes scheduler.

Xie, J., Yu, F. R., Huang, T., Xie, R., Liu, J., Wang, C., and Liu, Y. (2018). A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges. *IEEE Communications Surveys & Tutorials*, 21(1):393--430.

Xu, C., Rajamani, K., and Felter, W. (2018). Nbwguard: Realizing network qos for kubernetes. In *Proceedings of the 19th International Middleware Conference Industry*, pages 32--38.

Yang, R., Ouyang, X., Chen, Y., Townend, P., and Xu, J. (2018). Intelligent resource scheduling at scale: a machine learning perspective. In *2018 IEEE symposium on service-oriented system engineering (SOSE)*, pages 132--141. IEEE.

Yao, G., Ding, Y., Ren, L., Hao, K., and Chen, L. (2016). An immune system-inspired rescheduling algorithm for workflow in cloud systems. *Knowledge-Based Systems*, 99:39--50.

Ye, Z., EL-Azouzi, R., Jimenez, T., and Xu, Y. (2014). Computing quality of experience of video streaming in network with long-range-dependent traffic. *arXiv preprint arXiv:1412.2600*.

Yue, T., Wang, H., Cheng, S., and Shao, J. (2020). Deep learning based qoe evaluation for internet video. *Neurocomputing*, 386:179--190.

Zeng, H., Wang, B., Deng, W., and Zhang, W. (2017). Measurement and evaluation for docker container networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 105--108. IEEE.

Zhan, Z.-H., Liu, X.-F., Gong, Y.-J., Zhang, J., Chung, H. S.-H., and Li, Y. (2015). Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys (CSUR)*, 47(4):1--33.

Zhang, J., Huang, H., and Wang, X. (2016). Resource provision algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, 64:23--42.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7--18.

Zhao, S., Xue, S., Chen, Q., and Guo, M. (2019). Characterizing and balancing the workloads of semi-containerized clouds. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 145--148.

Zhong, Z., He, J., Rodriguez, M. A., Erfani, S., Kotagiri, R., and Buyya, R. (2020). Heterogeneous task co-location in containerized cloud computing environments. In *International Symposium on Real-Time Distributed Computing (ISORC)*, pages 79--88.

Zou, Z., Xie, Y., Huang, K., Xu, G., Feng, D., and Long, D. (2019). A docker container anomaly monitoring system based on optimized isolation forest. *IEEE Transactions on Cloud Computing*.