# QoE-Aware Container Scheduler for Co-located Cloud Environments

Marcos Carvalho
*Computer Science Department*
*Universidade Federal de Minas Gerais - UFMG*
Belo Horizonte, Brazil
marcoscarvalho@dcc.ufmg.br

Daniel Fernandes Macedo
*Computer Science Department*
*Universidade Federal de Minas Gerais - UFMG*
Belo Horizonte, Brazil
damacedo@dcc.ufmg.br

***Abstract*—Cloud management has traditionally considered Service Level Objectives (SLO) based on QoS metrics. However, QoS-focused metrics have a limited effect on the Quality of Experience (QoE) experienced by the clients. This paper proposes a Kubernetes scheduler extension and resource rescheduling that incorporates QoE metrics into SLOs. As a proof of concept, this work evaluates the architecture using the QoE metric proposed in the ITU P.1203 standard, in the context of video streaming services co-located with other services. Experimental results show that our scheduler improves the average QoE by 50% compared to other schedulers, while resource rescheduling improved the average QoE by 135%. In addition, our architecture eliminated over-provisioning altogether.**

***Index Terms*—Cloud Computing, Containers, Scheduler, QoE, Deep Machine Learning**

## I. Introduction

Cloud computing has gained enormous momentum in the last years with the rapid development of new technologies for data storage, processing, and data transfer over the Internet [1]. There are two agents in the cloud ecosystem: the cloud provider (e.g., AWS, Google Cloud, Microsoft Azure), which provides the resources (CPU, memory, disk, and network); and cloud customers (e.g., Netflix, Dropbox), which use the virtual resources to serve their clients. This is a pay-per-use usage model, known as Infrastructure as a Service (IaaS) [2].

The IaaS model allows customers to use a shared set of compute resources simultaneously via virtual machines (VMs) and, recently, containers. The virtualization allows multiple operating systems and multiple services on the same server [3], in which services are co-located to improve resource utilization [4]. However, there is a trade-off between high resource utilization and interference among services due to co-location. This interference appears when the demand from the services exceeds the resources available on the shared host [5]. Hence, the likelihood of performance degradation increases with the degree of service co-location [6]. In addition, latency-sensitive services are the most affected [7].

Cloud providers guarantee customers a performance level established in a Service Level Agreement (SLA) that specifies a set of Service Level Objectives (SLOs). The SLOs are usually composed by one or more Quality of Service (QoS) measurements [8]. Traditional cloud management uses different QoS metrics for each type of service. While some services require more compute resources (e.g. data analysis, artificial intelligence, data warehousing), others require more network resources (e.g. media streaming, web services) [9]. One way to guarantee the SLO established for each service is to balance the service's workloads between the various cloud servers by scheduling resources [10]. However, as previously mentioned, the co-location of different workloads with different QoS requirements generates interference, especially in latency-sensitive services.

Further, a QoS-based SLO is usually insufficient, because the QoS metrics themselves reflect poorly the end-user experience [11]. This is mostly because it is hard to define the optimal values of QoS metrics to reach the desired QoE [12]. Hence, direct QoE measurement has become a more effective form to analyze end-user engagement [13]. Several works explore QoE management applied to network management [14]. However, these works omit how QoE can be measured and then exploited in cloud computing to improve the End-to-end users QoE. Therefore, it is essential to create QoE-aware cloud management techniques.

This work proposes a QoE-aware scheduler for cloud environments. The proposal is an extension to the Kubernetes scheduler, adding QoE as an SLO metric. A predictor based on machine learning estimates the QoE offered by the cloud. Resource scheduling or rescheduling is based on this estimate. This paper evaluates the architecture in the context of video streaming services, however other services could be supported as well, as long as there is a suitable QoE predictor for that service. The main contributions of this paper are:

- A management system that uses QoE objectives as SLO metrics.
- A novel model to predict video QoE in cloud environments, following the ITU-T Recommendation P.1203 [15]. To the best of our knowledge, this is the first work that uses a set of cloud computing resources (CPU, Mem, Disk, Network) to predict QoE within the cloud.
- A QoE-aware container scheduler and rescheduling for cloud environments based on containers.

978-3-903176-32-4 © 2021 IFIP

Experimental results show that the mean over-provisioning decreased by 75% with the QoE-aware scheduler and 100% with the rescheduler. Further, the video QoE increased by 50% compared to other schedulers. Finally, the video QoE increased by 135% with rescheduling.

This paper is organized as follows. Section II dissects the related work. Section III presents the background information. Section IV details the system architecture. Section V describes the data collection process. Section VI shows our dataset analysis and model building. Section VII explains the experimental setup, followed by the results in Section VIII. Finally, Section IX concludes this work.

## II. Related Work

There have been many studies on scheduling virtual machines on the cloud [16]. In recent years, containers have gained popularity as virtualization technology, due to the lower overhead compared to virtual machines. Hence, container scheduling has become an emerging research topic [17], [18].

In [5] the authors propose a scheduler for Kubernetes based on the service's resource demands. For example, if a service uses more network than CPU resources, it should be allocated on a server with more available bandwidth than processing power. However, it is difficult to predict the service's resource usage beforehand [19]. Like our work, the authors consider the degradation in the services due to service co-location.

In [20] the authors propose a multi-objective container scheduling algorithm in which they consider the servers' CPU and memory usage and the time to transmit the container image over the network. They also take into account the characteristics of the services to associate them with the servers. However, the authors overlooked the effect of co-location. Their proposal does not use rescheduling to mitigate overloads. This work reschedules the containers when there is a QoE degradation caused by co-location.

In [21], the authors improve network QoS by scheduling network-aware containers in a multi-tenant scenario. Specifically, the authors improve bandwidth utilization by decreasing network fragmentation, while minimizing processing delay.

An extension of the Kubernetes scheduler for latency-sensitive services in fog computing was proposed in [22]. The goal is to minimize service response time. In their proposal, the various servers are located in different regions. The scheduler analyzes each location's network status and chooses the server with the lowest Round Trip Time (RTT). However, the authors consider only the network load.

KCSS [23] selects a server based on multiple criteria, using the technical algorithm for the Priority Order for Similarity to the Ideal Solution (TOPSIS) [24]. TOPSIS chooses a solution (server) whose distance from the best solution and the worst solution is minimal using the n-dimensional Euclidean distance. The authors used three criteria related to the work nodes' resource usage: maximize the CPU, disk, and memory usage rate. Thus, the objective is to compact the containers in order to use the maximum resources possible from the servers. KCSS is generic, allowing to use and combine other criteria and resources. Due to this generalization, we use this algorithm to compare with our work.

Table II compares the related works to our proposal. It shows if the works consider co-located services, the maximization objective, where the metrics are collected, and if the proposal employs QoE metrics. Ours stands out for considering other co-located services, and by collecting server and container metrics to maximize the service's QoE, based on the estimation of the end-user's QoE.

TABLE I
Comparison With The Related Work

| Work | Co-location | Max. objective | Source of metrics | QoE? |
|---|---|---|---|---|
| [5] | Yes | Execution time | Server | No |
| [20] | No | Response time | Server | No |
| [21] | No | Network QoS | Server | No |
| [22] | No | Network QoS | Server | No |
| [23] | No | Computing resource usage | Server | No |
| **Our** | Yes | QoE | Server/Container | Yes |

## III. Background

This section discusses concepts related to our proposal. It presents horizontal autoscaling and resource scheduling in cloud computing. It also discusses how to measure the Quality of Experience (QoE), and the Kubernetes platform.

### A. Cloud Horizontal Autoscaling and Resource Scheduling

Cloud environments allow the use of resources on-demand; that is, new instances of a service are spun up when the demand exceeds the initially reserved value [25]. This is known as *horizontal autoscaling*. Different performance metrics can be monitored, such as CPU, memory, and network utilization. Horizontal autoscaling initiates a number of instances I = $\{I_1, I_2, I_3 \ldots I_x\}$ to allocate $m$ resources R = $\{R_1, R_2, R_3 \ldots R_m\}$ on a cloud consisting of S = $\{S_1, S_2, S_3 \ldots S_n\}$ servers. *Resource scheduling* thus chooses in which server $S_i$ to allocate each resource $R_j$.

### B. Estimating QoE for HTTP Adaptive Video Streaming

ITU-T defines QoE assessment as the process of measuring or estimating the QoE for a set of users of a service [26]. In other words, QoE measures the pleasure or discomfort perceived by the users.

Video QoE assessment is a challenging task due to the many networking factors that influence it: different client devices and request patterns, varying network conditions, and significant spatial and temporal variation in cloud computing performance [27]. Furthermore, QoE is affected by human factors, such as users' expectations and perceptions.

The most commonly used video QoE definition is the subjective Mean Opinion Score (MOS). MOS is standardized in ITU-T recommendations [28], and is defined as a numeric value going from 1 (poor) to 5 (excellent). This approach has inherent difficulties: high cost, it is time-consuming, cannot be used in real-time, and lacks repeatability. This motivated the development of objective methods that predict the subjective quality from network characteristics [29].

Some works in the literature use machine learning-based video QoE prediction [30]–[32]. These works use network and service characteristics to infer the MOS. These approaches usually collect data such as delay, jitter, loss, and bandwidth. Some characteristics from the client are also important, such as the number of interruptions in the service, changes in the video bit rate, and playback start time.

Video streaming is migrating to cloud computing. The performance of the cloud computing environment also contributes to the users' QoE. However, to the best of our knowledge, this is overlooked in the state of the art video QoE predictors. QoE forecasting should go beyond the existing methods, which use only network metrics [14]. Cloud-based models should incorporate metrics related to CPU, memory, and disk usage.

### C. *Kubernetes*

Kubernetes[1] is the most popular container-based virtualization solution. It is an open source Docker management tool. Fig 1 shows the Kubernetes architecture. It contains one Master node and multiple Worker nodes. Each node is either a physical or virtual machine over the service layer. The node components are described below:

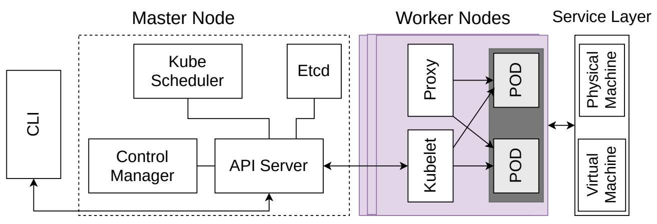

Fig. 1. Kubernetes architecure

**Master Node:** The Master Node manages the cluster and schedules the service deployments. The *API server* receives commands from clients using HTTP and manages the Kubernetes objects. The *kubectl* Command-line interface (CLI) sends commands to the API server. Another option is to use client libraries. The *Controller Manager* monitors the *etcd* storage component, which stores the cluster state. If the cluster state changes, for example, a pod stopped abruptly, the *Controller Manager* makes the necessary changes to restore the previous state. Finally, the *kube scheduler* (KS) schedules each pod on a specific node in the service layer.

**Worker Node:** The Kubernetes cluster is a set of worker nodes running containerized services inside of *pods*. The *pod* is the basic deployment unit in Kubernetes. One or more containers can be created and grouped into a pod. The containers inside a pod share the same IP Address, port space (namespace) and data volumes. Meanwhile, pods are isolated from each other. An end-user request is distributed to worker nodes according to load balancing rules. The *proxy* receives these requests and forwards them to pods.

[1]www.kubernetes.io/

**Kubernetes horizontal autoscaling:** the Horizontal Pod Autoscaling (HPA) system performs horizontal autoscaling based on reactive threshold-based rules for CPU or memory utilization [3]. When the current resource usage is greater than the threshold, 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.

To decide how many containers to scale, the HPA checks the ratio between the current metric and the target value, multiplied by the number of containers already allocated. For example, a new replica will be created if there is only one container with memory utilization of 200MB, and the target is 100MB. After that, the workload is balanced among the replicas through a proxy or an external load balancer [33].

**Kubernetes Resource Scheduling (KS):** KS searches for a suitable worker node to deploy the containers in the waiting queue. First, it verifies which worker nodes can receive the new pod using filters. One example filter checks the amount of free resources (CPU or memory) on a given worker node, or if that node is compatible with the system configuration requested by the container. After filtering, the scheduler employs a round-robin approach to choose the most suitable worker node.

## IV. System Architecture

We propose a video QoE-aware scheduler for cloud computing environments. In the proposed architecture, the cloud computing platform forms the Data Plane, while the Control Plane schedules or reschedules the containers. In addition, the Control Plane monitors if the SLO is being maintained. The architecture is shown in figure 2 and described below.

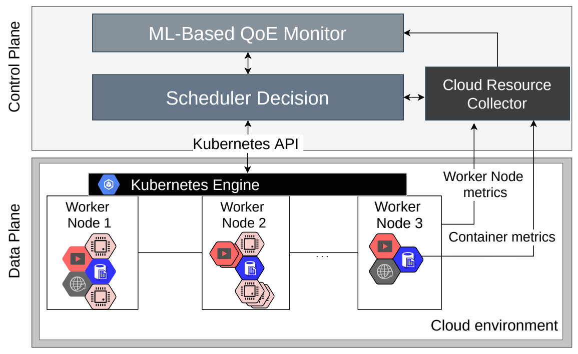

Fig. 2. System Architecture

### A. *Cloud environment*

Our proposal employs the kubernetes engine. The Flannel plugin[2] was used for container communication. Also, each worker node has a cAdvisor[3] instance that stores the resource usage metrics from worker nodes and containers.

### B. *Cloud Resource Monitor*

This module collects resource usage metrics through each of the cAdvisor API. The worker node and container resource

[2]https://github.com/coreos/flannel
[3]https://github.com/google/cadvisor

usage metrics are grouped to be transferred to the ML-based QoE monitor. This grouping is necessary due to the order of features in which the machine learning model was trained.

### *C. ML-Based QoE Monitor*

The ML-based QoE Monitor provides real-time estimates of the QoE that the cloud can offer to a user group, based on resource usage from the worker nodes and containers. This information can be used by a cloud provider to monitor and manage cloud resources. Below we describe how our model was created and its design decisions.

*1) Definition of ML algorithm and model features:* Our machine learning model was modeled using supervised machine learning. We used regression algorithms since the output is a numeric value. The predictor's input includes resource usage metrics from the worker node and the container. Formally, the predictor is a function defined as:

$$f(c, w) \mapsto QoE_{cloud} \in \mathbb{R} \tag{1}$$

In the equation, $c$ stands for the usage metrics of the container, while $w$ stands for the metrics of the worker node. Below, we list each collected resource usage metric. Model training is discussed in section VIII.

**Model inputs:** 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 ($c$). The total number of collected metrics is 70. As a matter of space, we do not detail each metric. However, the cAdvisor[4] and Docker[5] documentation detail each metric.

***CPU data:*** The CPU category contains *CPU usage*, *CPU user*, and *CPU system*. This information is given in terms of CPU time consumed, 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). 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 in bytes.

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

**Model output:** The ouput is the expected QoE for a group of users $U$ ($QoE_{cloud}$), based on the ITU-T Recommendation P.1203[6], which estimates the subjective Mean Opinion Scores (MOS) for a video session. Besides considering the mean value, the output considers the spread of QoE among users. To deal with this, the $QoE_{cloud}$ formula incorporates fairness $F$. Equation 2 shows how to obtain $QoE_{cloud}$.

$$QoE_{cloud} = \frac{(\sum_{i=1}^{U} QoE_i) * F}{U} \tag{2}$$

Fairness F is calculated using equation 3, where $\sigma(X)$ is the standard deviation function, $H$ and $L$ are the highest and lowest allowable QoE values, respectively [34]. As we use MOS to define the video QoE, the values of $H$ and $L$ are 1 and 5, respectively. Since F is normalized between 0 and 1, the QoE also is normalized in the same interval.

$$F = 1 - \frac{2\sigma(QoE)}{H - L} \tag{3}$$

It is important to note that the calculation presented in Equation 2 is used *for training only*, in order to produce the labeled instances. The final model estimates $QoE_{cloud}$ directly. Data collection and model training are discussed in sections V and VIII-A, respectively.

*2) Use of a time series-based model:* Continuous-time QoE prediction is a challenging task that requires accounting for the instantaneous temporal effects of subjective QoE [35]. It has been observed that the QoE is dynamic and continuously time-varying, due to a series of QoE influencing events such as rebuffering and rate adaptation [31]. We analyzed the inputs to decide which machine learning techniques would be more suitable for our dataset. This work considers that the inputs represent a time series.

Our analysis uses the Auto Correlation Function (ACF). ACF plots auto-correlation with its lagged values [36]. Figure 3 shows the auto-correlation plot of MOS for 40 lags. 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 very slow decay in correlation between successive observations, indicating the presence of long range dependence. This phenomenon characterizes strong temporal correlation in a time series [37], and has been characterized as a common phenomenon in cloud computing [38]–[40] and QoE prediction [31], [41].

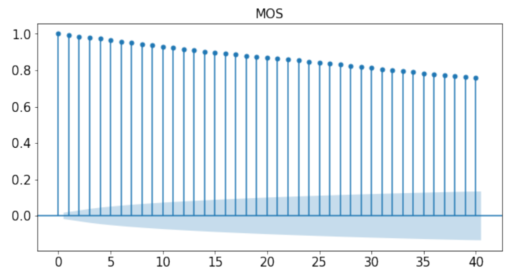

Fig. 3. Autocorrelation plot of MOS at 40 lags

Because of that long term dependence, QoE predictions demand more advanced prediction techniques. Therefore, we use Recurrent Neural Networks (RNNs), specifically, Long Short-Term Memory (LSTM). LSTM was introduced by [42] to solve the vanishing gradient problem. Besides, there are

[4]https://github.com/google/cadvisor/tree/master/metrics
[5]https://docs.docker.com/config/containers/runmetrics/
[6]https://www.itu.int/rec/T-REC-P.1203

studies in the literature [38], [43] showing that LSTM models handle long range dependencies better than GRU.

### *D. Scheduler Decision*

The Scheduler Decision (SD) module performs scheduling and rescheduling. Algorithm 1 describes its algorithm. The *csd* procedure continuously checks for two conditions, being: (i) a non-empty HPA queue; in this case, there is a pending container *p* in the queue to be deployed (line 4). Queue status is queried via the *monitorHPAQueue()* function of the kubernetes API; (ii) QoE degradation in the allocated (containers *C*) (line 6). The *checkBadQoE()* function checks whether the predicted QoE is below a QoE value (defined as SLO) for a specific time interval *T*. Theses checks are done in parallel.

Scheduling works as follows. The function *monitorHPAQueue()* returns the containers to be deployed. Then, the *scheduler* procedure searches for a suitable worker node to deploy $p_i$, taking as parameters the container set (*C*) and the available worker nodes (*W*). This procedure then estimates the QoE for the deployment in worker node $w_j$ (line 12). Finally, in line 13, the algorithm chooses the worker node (*w*) that maximizes the QoE above the SLO. Then, container *p* is deployed to worker node *w* (line 14).

The need for rescheduling is checked on the second part of the *csd* procedure (lines 6-8). The *checkBadQoE* function returns true when there is a QoE degradation in the container $c_i$ due to co-location with other services. This triggers a container reschedule. In the proposed algorithm, rescheduling deletes the container from the current worker node and deploys a new container. For that end, we first call the *rescheduler* procedure (line 7). This procedure searches for a suitable worker node to deploy a new container. Unlike the scheduler algorithm, rescheduling uses as parameters only the current container and the available worker nodes (*W*). After choosing a worker node that maximizes QoE, the container is deleted from the original worker node (line 20). Section VII estimates the time spent on scheduling and rescheduling using real cloud traces.

**Algorithm 1** Container scheduler decision algorithm

```
 1: procedure CSD($C, W, SLO, T$)
 2:     while True do
 3:         $P \leftarrow monitorHPAQueue()$
 4:         for each $p_i \in P$ do
 5:             scheduler($p_i, C, W$)
 6:         for each $c_i \in C$ do                ▷ QoE Monitoring
 7:             if $checkBadQoE(c_i, SLO, T)$ then
 8:                 rescheduler($c_i, W$)
 9: procedure SCHEDULER($p, C, W$)
10:     for each $c_i \in C$ do
11:         for each $w_j \in W$ do
12:             $QoE_{i,j} \leftarrow predictor(c_i, w_j)$
13:     $w \leftarrow \max(X)\ \{\forall X \in QoE \mid X > SLO\}$
14:     deploy container $p$ in $w$
15: procedure RESCHEDULER($container$, $W$)
16:     for each $w_j \in W$ do
17:         $QoE_j \leftarrow predictor(container, w_j)$
18:     $w \leftarrow \max(X)\ \{\forall X \in QoE \mid X > SLO\}$
19:     deploy new container in $w$
20:     delete $container$
```

## V. Data Collection for the predictor

Figure 4 depicts the environment used to collect data from video sessions and train the QoE predictor. The training instances were obtained from clients watching the video on a cloud environment. The collection was done in decreasing rounds from 7 clients to one. Each round has five repetitions. The main goal in this phase was to cover the highest amount of behavioral possibilities from the cloud. This

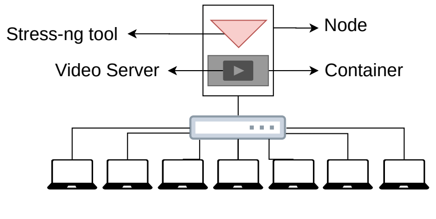

Fig. 4. Data collection environment

setup is composed of seven notebooks as clients connected to the cloud via a switch, one worker node, and a DASH server in a docker container. The DASH server uses NGINX (https://nginx.org/en/) and serves the Big Buck Bunny video (https://peach.blender.org). The encoding used H.264 with segments of 2 seconds. Also, the video streams use resolutions varying between 320x180 and 1920x1080. Finally, the clients watch the video using Firefox.

### *A. Data Collection and Processing*

To collect data from video sessions under different co-located conditions, we use the Linux stress-ng tool to generate random CPU, memory, and disk workloads on the worker node. We also simulated the extra workload on the network interface of the worker node using Iperf. This simulates services such as machine learning training processes, storage systems, and video streaming. CPU stress ranged from 1 to 128 instances of CPU stressors and 30 instances of memory stressors, with 350 MB for each instance. This created constant CPU load between 40% and 95%, and the memory usage increases continuously, reaching up to 90% and decreasing gradually. Disk I/O stress ranged randomly from 1 to 40 processes, which each writing and reading 200MB. These variations on worker node conditions can trigger the DASH client's adaptation algorithm and cause video quality switches. The load generation procedure was also used during the experiments. As discussed in IV-B, we collected worker node and container resource usage metrics and grouped by second.

The client stores the playback stalls time and timestamps for each video session. Also, the video server logged the resolution and bitrate played by the clients at each second. We use this information to calculate the MOS based on the ITU-T Recommendation P.1203 [15] using the software developed by [44]. This software requires devices type, display size, and viewing distance as parameters to calculate MOS. In our case, we use PC, 1920x1080, and 150 cm, respectively.

## VI. Dataset analysis, preparation and Model building

The final dataset is composed of 35.420 instances and 70 features, and one target value (video QoE of a user group). Feature selection and model training are described below.

### A. Feature Selection and scaling

This phase reduces the model training time without affecting the predictor's quality. We use Spearman correlation analysis to conduct feature selection. Our final training dataset consists of the features with a Spearman correlation higher than or equal to 50; therefore, the input data amount was reduced from 70 to 23. Table II shows the feature selection results. This shows a strong negative association between the use of computational resources of the working node and the container with the MOS, indicating that high use can degrade video QoE.

TABLE II
Spearman correlation

| Container metrics | Corr. | Worker node metrics | Corr. |
|---|---|---|---|
| diskio_read_io_service | 0.64 | cpu_usage | -0.71 |
| disk_read_io_serviced | 0.66 | cpu_user | -0.71 |
| mem_usage | -0.54 | diskio_sync_io_service | 0.64 |
| mem_cache | -0.74 | disk_sync_io_serviced | 0.66 |
| rx_bytes | 0.60 | disk_write_io_serviced | 0.66 |
| rx_packets | 0.60 | mem_usage | -0.71 |
| tx_bytes | 0.57 | mem_cache | -0.70 |
| tx_packets | 0.59 | working_set | -0.71 |
| | | mem_container_pgmajfault | 0.70 |
| | | rx_bytes_cni | 0.55 |
| | | tx_bytes_cni | 0.58 |
| | | tx_packets_cni | 0.59 |
| | | rx_bytes_flannel | 0.55 |
| | | rx_packets_flannel | 0.55 |
| | | tx_packets_flannel | 0.54 |

## VII. Experimental Setup

The experimental setup is based on Fig 2. The Kubernetes engine v1.17.3 runs in four physical machines: one master node and three worker nodes. For the master node, we use a i5-3450S CPU@2.80GHz with 4GB of RAM and 1TB disk. The worker nodes have the same compute configuration, a i5-4460 CPU@3.20GHz with 16GB of RAM and 1TB disk.

We created a scenario where each worker node received a different extra workload to simulate co-located services. We use stress-ng and iperf to generate the loads. Table III shows the values of this variation.

The containers were configured with 500MB of RAM *limits* and 200MB of RAM *requests*. The CPU was limited with 4 cores and 1/4 of requests. The configuration also restricts the CPU and memory usage of the container to the pre-configured limits. The Kubernetes Python client API is used to collect information from the cloud. We use seven notebooks as clients playing videos using Firefox version 80. Finally, all clients have the same screen size (information necessary to calculate MOS); however, the hardware varies. These clients are connected to the cloud through a Gigabit Ethernet Switch. We evaluated three HPA policies with different memory thresholds, being 50%, 80%, and 90%. The memory threshold is defined in terms of bytes; in our case these correspond to 250 MB, 400 MB and 450 MB, respectively. We have a small number of clients in our experiment to achieve these values, so these values set as a threshold were based on the average usage of the seven clients connected in the container. We conducted 30 experiments with clients watching the videos in a stress-free environment and then stored the memory used. We take the highest values for each repetition and then calculate the average. On average, clients consume 3.75 MB of memory in the worst case. Out of that value, we calculate the 50% 80%, and 90% memory thresholds, obtaining 1.87MB, 3MB and 3.37MB, respectively. Also, as SLO value, we set the QoE value to 3 and a time threshold of 10 seconds.

TABLE III
Workload extra in each Worker Node

| | CPU (%) | Memory (%) | Disk I/O | Network |
|---|---|---|---|---|
| Worker 1 | 50 - 90 | 10 - 90 | 2GB | 0 |
| Worker 2 | 50 - 55 | 10 - 15 | 0 | 0 |
| Worker 3 | 10 - 90 | 20 - 25 | 0 | 500MB |

## VIII. Results

### A. Model training

The model was trained using Keras version 2.2.4 on a desktop equipped with an Core i5 CPU @ 3.20GHz and 16 GB of RAM. The dataset was divided into three parts: train (70%), validation (20%), and test (10%). The test set was used to obtain the final prediction error. Also, we performed a variation on the LSTM hyperparameters manually. Table IV shows the evaluated value and the chosen configuration.

TABLE IV
LSTM configuration

| | Layers | Neurons | Dropout | Timestamp |
|---|---|---|---|---|
| Evaluated | 1, 2, 3 | 128, 64, 32 | 0.25, 0.50 | 10,20,30,60 |
| Chosen | 3 | 64, 32 | 0.25 | 30 |

Our model was created with three LSTM *layers* followed by a MLP with one hidden layer. Each LSTM *layers* use 64, 32, and 32 *neurons*, respectively. Also, as we deal with time series prediction, the predictor uses 30 time intervals as inputs (thus considering the resource usage of the previous 30 seconds). To reduce overfitting, *dropout* parameters were set with 0.25. We also use the same value to *recurrent dropout*, used in the transformation from one recurrent unit to another. This process is used as a regularizer method in neural networks.

The average value of the RMSE indicates the prediction quality. Table V shows the obtained RMSE values. The results indicate that the model predicts QoE with a low average error.

We evaluate the prediction time to examine if the model can be used in a real environment. Table V shows the average time when using a computer with Intel(R) Core(TM) i5-4460 CPU@3.20GHz and 16 GB of RAM.

The Alibaba cloud is described in [6]. According to the authors, in 2018 the Alibaba cloud had 4.000 servers and 9.000

TABLE V
PREDICTOR PERFORMANCE

| | RMSE | Execution Time (in $\mu$s) |
|---|---|---|
| Average $\pm$ CI (95%) | 0.08 $\pm$ 0.03 | 12.57 $\pm$ 9.3 |
| Median | 0.10 | 15.20 |
| Max | 0.38 | 59.78 |

online services. Each of these services used a maximum of 128 containers. Given this scenario, we estimate the time spent to schedule and reschedule these containers using Algorithm 1. Scheduling would take approximately 6.43 seconds to choose a server for a new container. For rescheduling, it would take 0.05 seconds to search for a new server for a given $C_i$. Considering that the predictor can use the compute resources of the cloud itself, the prediction time seems plausible.

## B. Container Scheduling

This section compares the proposed scheduler with two baselines: the default Kubernetes scheduler and KCSS [23]. Since Kubernetes and KCSS do not perform rescheduling, we evaluate our scheduler and rescheduler in separate experiments. We have named these experiments in the following graphs and tables as *QoE-S* and as *QoE-R* (performing both scheduling and rescheduling), respectively. Also, we implemented two versions of KCSS, considering all worker nodes metrics (Table II). The first one has as criteria the maximization usage of these metrics (KCSS Max), while the second the minimization (KCSS Min). Besides, we considered three scenarios with different percentages of memory usage as HPA threshold, being 50%, 80% and 90%. The following evaluations were made with the original video container deployed on worker node 1. Finally, 30 repetitions were performed, and results consider a confidence interval of 95%.

First, we analyze schedulers and rescheduler's effects on the number of scaled and used containers in each HPA policy. The efficient use of cloud elasticity balances over-provisioning and under-provisioning. The first leads to resource wastage and extra monetary cost, while the latter causes performance degradation [25]. Figure 5 shows the mean number of scaled and used containers for each experiment in each of the HPA policies. It was measured by the amount of container scaled and analyzing those containers that received a request to download the video, respectively.

There is a decrease in the mean amount of scaled containers among HPA policies. Users can achieve 50% of memory usage more quickly with 80% and 90% thresholds, which makes the HPA scale more containers to balance the workload until it is below the threshold. Consequently, this implies an increase in the number of used containers. In the case of HPA with a 50% threshold, seven containers were always used for each experiment, that is, one user per container, which explains the confidence intervals to be zero. It also explains the small variation in the amount of scaled containers. However, in the other policies, the confidence intervals show a greater variation in the number of scaled containers. This is due to fluctuations in the workload in the containers.

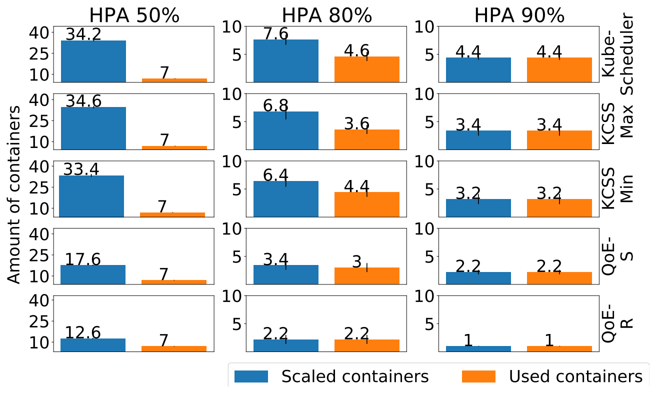

Fig. 5. Mean number of scaled and used containers per HPA policy

Our proposal reduced the number of scaled containers in both policies. Kube-Scheduler and KCSS had a mean number of scaled containers similar to HPA 50%. However, in HPA 80% and 90%, the Kube-scheduler had a higher mean, while the two versions of KCSS had a similar mean. Such improvement can be explained by how each schedule distributed the new video containers among the worker nodes. Kube-Scheduler, KCSS Max, and KCSS Min scheduled several of these new containers for work node 1, where there was greater interference due to high CPU, memory, and disk usage. In contrast, our solution did not schedule any extra containers for worker node 1. Besides, *QoE-R* deleted the original container from worker 1 and scheduled it into a new worker node.

Figure 6 shows the mean number of scaled containers at each worker node for each policy. Kube-scheduler had a better mean distribution among the three nodes because it uses Round-Robin. KCSS Min also had a good mean distribution among the three workers; however, it was higher in workers 1 and 2. This is because KCSS Min tends to choose the worker node with less resource usage. Due to the random use of resources at each node, the CPU and memory utilization of worker node 1 may be less than that of worker node 2 and 3, and vice-versa. However, in worker 3, the network usage is always greater than in other workers, which justifies the lower mean. KCSS Max, on the other hand, had a mean distribution between worker 1 and 3. The maximization criterion aims to compact the containers in those workers with higher resource use. Thus, worker 1 was chosen due to its greater CPU, memory, and disk usage, while node 3 was chosen by network usage. In comparison with these workers, worker 2 had the least variation and resource usage, so it was never chosen.

The way our proposals distributed the containers among the work nodes contributes to the reduction of over-provisioning. Table VI shows the over-provisioning for each experiment and the percentage of decrease using our proposals. We omit HPA 90% because there was no over-provisioning.

## C. Video QoE improvement

We also measured the QoE improvement due to QoE-aware scheduling and rescheduling. We perform this analysis only

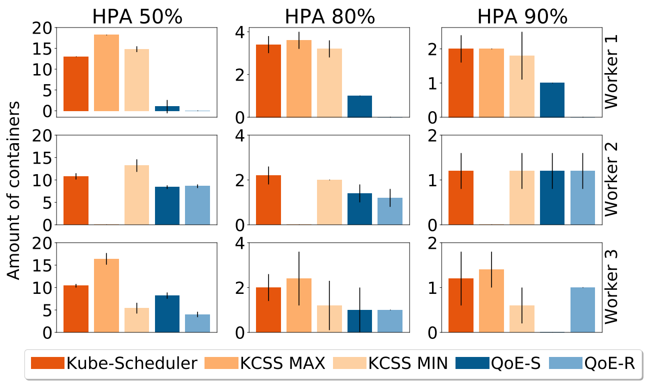

Fig. 6. Mean number of containers scaled per worker node

TABLE VI
MEAN OVER-PROVISIONING REDUCTION

| | Kube-Scheduler | KCSS Max | KCSS Min | *QoE-S* | *QoE-R* |
|---|---|---|---|---|---|
| **HPA 50%** | 79.56% | 79.76% | 79.04% | 60.22% | 44.40% |
| *QoE-S* decrease (%) | 24.30% | 24.49% | 23.81% | | |
| *QoE-R* decrease (%) | 44.19% | 44.28 | 43.77% | | |
| **HPA 80%** | 39.47% | 47.05% | 31.25% | 11.76% | 0% |
| *QoE-S* decrease (%) | 70.20% | 75% | 62.36% | | |
| *QoE-R* decrease (%) | 100% | 100% | 100% | | |

TABLE VII
QUALITY PERCEIVED BY THE USERS USING 80% AS HPA THRESHOLD

| | Mean QoE | Mean playback stalls time | Mean # of resolution changes |
|---|---|---|---|
| Kube-Scheduler | 2.1 ± 0.15 | 19 ± 7 | 21 ± 0.9 |
| KCSS Max | 2 ± 0.14 | 21 ± 8 | 22 ± 8 |
| KCSS Min | 2.2 ± 0.11 | 18 ± 6 | 19.1 ± 0.3 |
| *QoE-S* | 3 ± 0.03 | 8.8 ± 2.3 | 11 ± 0.7 |
| *QoE-R* | 4.7 ± 0.04 | 0.6 ± 0.3 | 0 |

with the 80% and 90% HPA policies since these are more realistic scenarios. Table VII shows that the proposed *QoE-S* improves the QoE by 42.85%, 50% and 36.36% (from 2.1, 2, 2.2 to 3). This improvement occurs because the scheduler chooses other worker nodes that improve the QoE. However, this improvement was limited by the end-users with a degraded QoE in the container allocated in worker 1. Our scheduler kept the SLO at the proposed limit with a variation of 0.03. At the same time, it reduced the mean time of playback stalls by 53.8%, 50% and 51.11%, from 19, 21 and 18 to 8.8. This is because the scheduled containers located at worker nodes 2 and 3 contributed to a better response to user requests. Likewise, the mean number of resolution changes was reduced by 96.84%, 97.14% and 96.66%, from 21, 22 and 19.1 to 11.

On the other hand, using *QoE-R*, we obtained a QoE close to the maximum value (5), and the reduction on the playback stall time was close to half a second. In this case, the stops occur only at the beginning of the video playback, caused by the extra load on worker 1. This contributed to a QoE that is lower than 5. Container rescheduling on the cloud accounted for 56.6% of the QoE's improvement, from 3 to 4.7. Comparing to the Kube-scheduler, KCSS Max and KCSS Min, the increase reached 123.80%, 135%, and 113.63%, respectively. Also, the average number of resolution changes has been reduced to zero because the model predicts a QoE decay in advance, and reschedules the container before the resolution changes.

However, using 90% as the HPA threshold, there was an under-provision, which resulted in similar results for the Kube-scheduler, KCSS Max, KCSS Min and the *QoE-S*, being, respectively; mean QoE value: 1.3±0.12, 1.2±0.11, 1.5±0.18, 1.4±0.4; mean of playback stalls time: 30±5.5, 31±5.1, 29±5, 31±4.4; number of resolution changes: 28±0.6, 29±0.8, 27±0.3, 28±0.8. Although the *QoE-S* schedules the new container in another worker node, this was not enough to improve the end-users' QoE, nor to reduce playback stalls time and resolution changes. This is because there were not enough scaled containers to be scheduled at other worker nodes. However, *QoE-R* for 90% had the same result as for 80%. This was expected since the scheduler works regardless of HPA policies. The scheduling decision depends only on QoE monitoring, that is, as in HPA 80%, the model predicted QoE degradation and rescheduled the container, without taking into account the HPA operation.

## IX. CONCLUSIONS

This work proposed a QoE-aware container scheduler and rescheduler for the cloud. Our system extends the Kubernetes scheduler to use QoE as SLO metrics. We created a predictor based on machine learning that estimates the QoE offered by the cloud, and we proposed an algorithm that schedules or reschedules resources based on this estimate. The proposal was evaluated experimentally in the context of video streaming services co-located with other services.

The results showed that the proposal decreased the average number of scaled containers by up to 55.26% and 71.05%, respectively. Likewise, the scheduler and rescheduler reduced over-provisioning by up to 75% and 100%, while the average QoE was increased by at 50% and 135%, respectively.

Future work will expand this proposal by adding predictors for other services, such as audio transmission and web.

## ACKNOWLEDGMENT

This study was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq (funding agencies from the Brazilian federal government), and FAPEMIG (Minas Gerais state funding agency).

## REFERENCES

[1] T. Chen, R. Bahsoon, and X. Yao, "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *arXiv preprint arXiv:1609.03590*, 2016.

[2] S. H. H. Madni, M. S. Abd Latiff, Y. Coulibaly *et al.*, "Recent advancements in resource allocation techniques for cloud computing environment: a systematic review," *Cluster Computing*, vol. 20, no. 3, pp. 2489–2533, 2017.

[3] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: state of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2017.

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

[5] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana, "Client-side scheduling based on application characterization on kubernetes," in *International Conference on the Economics of Grids, Clouds, Systems, and Services*, 2017, pp. 162–176.

[6] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "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)*, 2019, pp. 1–10.

[7] S. Zhao, S. Xue, Q. Chen, and M. Guo, "Characterizing and balancing the workloads of semi-containerized clouds," in *International Conference on Parallel and Distributed Systems (ICPADS)*, 2019, pp. 145–148.

[8] A. Elhabbash, Y. Elkhatib, G. S. Blair, Y. Lin, A. Barker, and J. Thomson, "Envisioning slo-driven service selection in multi-cloud applications," in *IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2019, pp. 9–14.

[9] Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz *et al.*, "Turbine: Facebook's service management platform for stream processing," in *International Conference on Data Engineering (ICDE)*, 2020, pp. 1591–1602.

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

[11] P. Juluri, V. Tamarapalli, and D. Medhi, "Measurement of quality of experience of video-on-demand services: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 401–418, 2015.

[12] E. Kafetzakis, H. Koumaras, M. A. Kourtis, and V. Koumaras, "Qoe4cloud: A QoE-driven multidimensional framework for cloud environments," in *2012 international conference on telecommunications and multimedia (TEMU)*. IEEE, 2012, pp. 77–82.

[13] L. De Cicco, S. Mascolo, and V. Palmisano, "QoE-driven resource allocation for massive video distribution," *Ad Hoc Networks*, vol. 89, pp. 170–176, 2019.

[14] A. A. Barakabitze, N. Barman, A. Ahmad, S. Zadtootaghaj, L. Sun, M. G. Martini, and L. Atzori, "QoE management of multimedia streaming services in future networks: a tutorial and survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 526–565, 2019.

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

[16] L. Liu and Z. Qiu, "A survey on virtual machine scheduling in cloud computing," in *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 2016, pp. 2717–2721.

[17] M. Adhikari, T. Amgoth, and S. N. Srirama, "A survey on scheduling strategies for workflows in cloud environment and emerging trends," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–36, 2019.

[18] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck, "Resource management in a containerized cloud: Status and challenges," *Journal of Network and Systems Management*, vol. 28, no. 2, pp. 197–246, 2020.

[19] M. Masdari and A. Khoshnevis, "A survey and classification of the workload forecasting methods in cloud computing," *Cluster Computing*, pp. 1–26, 2019.

[20] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, "A new container scheduling algorithm based on multi-objective optimization," *Soft Computing*, vol. 22, no. 23, pp. 7741–7752, 2018.

[21] L. R. Rodrigues, M. Pasin, O. C. Alves, C. C. Miers, M. A. Pillon, P. Felber, and G. P. Koslovski, "Network-aware container scheduling in multi-tenant data center," in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–6.

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

[23] T. Menouer, "Kcss: Kubernetes container scheduling strategy," *The Journal of Supercomputing*, pp. 1–27, 2020.

[24] Y.-J. Lai, T.-Y. Liu, and C.-L. Hwang, "Topsis for modm," *European journal of operational research*, vol. 76, no. 3, pp. 486–500, 1994.

[25] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[26] N. Barman and M. G. Martini, "QoE modeling for http adaptive video streaming–a survey and open challenges," *IEEE Access*, vol. 7, pp. 30 831–30 859, 2019.

[27] T. Hobfeld, R. Schatz, M. Varela, and C. Timmerer, "Challenges of QoE management for cloud applications," *IEEE Communications Magazine*, vol. 50, no. 4, pp. 28–36, 2012.

[28] P. Recommendation, "800.1–mean opinion score terminology," *ITU-T, Geneva, Switzerland*, 2003.

[29] M. Alreshoodi and J. Woods, "Survey on QoE\qos correlation models for multimedia services," *arXiv preprint arXiv:1306.0221*, 2013.

[30] M. Lopez-Martin, B. Carro, J. Lloret, S. Egea, and A. Sanchez-Esguevillas, "Deep learning model for multimedia quality of experience prediction based on network flow packets," *IEEE Communications Magazine*, vol. 56, no. 9, pp. 110–117, 2018.

[31] N. Eswara, S. Ashique, A. Panchbhai, S. Chakraborty, H. P. Sethuram, K. Kuchi, A. Kumar, and S. S. Channappayya, "Streaming video QoE modeling and prediction: A long short-term memory approach," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 3, pp. 661–673, 2019.

[32] T. Begluk, J. B. Husić, and S. Baraković, "Machine learning-based qoe prediction for video streaming over lte network," in *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE, 2018, pp. 1–5.

[33] K. Takahashi, K. Aida, T. Tanjo, and J. Sun, "A portable load balancer for kubernetes cluster," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2018, pp. 222–231.

[34] T. Hoßfeld, L. Skorin-Kapov, P. E. Heegaard, and M. Varela, "A new QoE fairness index for QoE management," *Quality and User Experience*, vol. 3, no. 1, p. 4, 2018.

[35] C. G. Bampis, Z. Li, and A. C. Bovik, "Continuous prediction of streaming video QoE using dynamic networks," *IEEE Signal Processing Letters*, vol. 24, no. 7, pp. 1083–1087, 2017.

[36] R. Agrawal and R. Adhikari, "An introductory study on time series modeling and forecasting," *Nova York: CoRR*, 2013.

[37] S. Gupta and A. D. Dileep, "Long range dependence in cloud servers: a statistical analysis based on google workload trace," *Computing*, pp. 1–19, 2020.

[38] S. Gupta and D. A. Dinesh, "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)*. IEEE, 2017, pp. 1–6.

[39] S. Gupta, A. D. Dileep, and T. A. Gonsalves, "A joint feature selection framework for multivariate resource usage prediction in cloud servers using stability and prediction performance," *The Journal of Supercomputing*, vol. 74, no. 11, pp. 6033–6068, 2018.

[40] B. Song, Y. Yu, Y. Zhou, Z. Wang, and S. Du, "Host load prediction with long short-term memory in cloud computing," *The Journal of Supercomputing*, vol. 74, no. 12, pp. 6554–6568, 2018.

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

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[43] D. Zhang, G. Lindholm, and H. Ratnaweera, "Use long short-term memory to enhance internet of things for combined sewer overflow monitoring," *Journal of hydrology*, vol. 556, pp. 409–418, 2018.

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