A Machine Learning Model for Detection of Docker-based APP Overbooking on Kubernetes

Felipe Ramos*, Eduardo Viegas*, Altair Santin*, Pedro Horchulhack*, Roger R. dos Santos*, Allan Espindola*
*Graduate Program in Computer Science
Pontifical Catholic University of Parana, Brazil
{felipe.ramos, eduardo.viegas, santin, pedro.horchulhack, robson.roger, allan.espindola}@ppgia.pucpr.br

ICC 2021 - IEEE International Conference on Communications | 978-1-7281-7122-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICC42927.2021.9500259

***Abstract*—Resource allocation overbooking is an approach used by cloud providers that allocates more virtual resources than available on physical hardware, which may imply service quality degradation. Docker in cloud computing environments is being increasingly used due to their fast provisioning and deployment, while the impact of overbooking of resources allocation due to multi-tenancy remains overlooked. This paper proposes a machine learning model to detect overbooking in Kubernetes environments within the docker container. The proposed model continuously monitors distributed container OS usage and application performance metrics. The collected metrics are used as input to a machine learning model that identifies multi-tenancy interference incurring in application performance degradation. Experiments performed on a Kubernetes cluster with a Docker-based Big Data processing application showed that our proposed model could detect resource overbooking with up to 98% accuracy. This implies an overbooking on a resource of up to 1.2 in the client's domain.**

***Index Terms*—Container Orchestration, Docker, Kubernetes, Machine Learning.**

I. Introduction

Public cloud computing services have become widely used due to their pay-as-you-go premise [1]. A popular service model in cloud computing is known as *Infrastructure-as-a-Service* (IaaS), wherein the cloud client acquires a configurable virtual machine (VM) according to its service needs. In IaaS model, software components are not shared between tenants, which introduces a processing burden. Recently a new lightweight multi-tenancy structure has been introduced to replace traditional hypervisors, namely service containerization [2]. In such a case, instead of hosting several VMs executing their OS, containers can share their host OS libraries by using a container engine (e.g., Docker) to create isolated spaces to each container, performed as a traditional process in the host OS [3].

To increase profits and optimize resource allocation, public cloud providers often allocate more virtual resources than their physical hardware counterparts can handle [4]. This situation, known as overbooking, leverages the fact that the virtual resources requested by cloud tenants will be in an idle state most of the time. Therefore, the physical hardware can be provided to other cloud tenants concurrently. For instance, a cloud provider may allocate 20 virtual CPU cores to 10 cloud tenants in a physical machine containing only eight physical CPU cores.

Resources overbooking may cause quality-of-service (QoS) degradation to cloud tenants if other clients request their virtual resources concurrently, as there will not be a proper physical hardware counterpart to handle the processing needs [5]. The hypervisors often employ fairness algorithms to deal with overbooking, wherein VMs in an idle state receive a higher hardware usage priority when they request it. However, overbooking in container orchestration scenario is a very tricky problem, as multi-tenancy between containers is handled by the host OS, which treats them as a traditional user-space process [6]. Despite being a known and widely studied challenge in VM-based clouds, overbooking of resources in service containers remain overlooked in the literature [7].

Traditionally, to ensure QoS, public cloud providers establish *Service Level Agreements* (SLA) that defines a minimum expected service quality level. SLA represents a contract, evaluated by a *Service Level Indicator* (SLI) that measures if the established SLA has been fulfilled [8] [9]. However, SLAs are often related to service uptime, despite hardware provision guarantees. It is up to the cloud client to ensure that the cloud provider is provisioning the contracted hardware [10].

The client cannot access the provider host information, but only its dockerized version within its domain. As a result, the client does not have access to information such as the number of concurrent containers being executed in the same physical hardware, neither their physical hardware usage [4].

This paper proposes a machine learning model to identify the resource overbooking within a client domain in Kubernetes orchestration environments. The proposed model was implemented in a twofold manner. First, it periodically and continuously monitors the service container application performance and containerized OS usage metrics. Our approach assumes that we can detect overbooking-related issues by analyzing both application performance and its dockerized OS resource usage counterpart. Second, we treat resource overbooking detection as a classification task through machine learning techniques. Therefore, our proposed model can detect resource overbooking issues in service containers within the client domain, without host-related data and possible conflict of interest with the cloud provider.

In summary, the main paper contributions are:

- The evaluation of multi-tenancy impact in dockers service. The performed evaluation, first in the literature according to our best knowledge, with a big data pro-

cessing application as a use-case, shows that overbooking significantly affects service containers performance.
- The proposition and evaluation of a machine learning model to detect resource issues overbooking in service containers within the client domain. Our proposed model can detect overbooking of resources with up to 98% of accuracy.

The remainder of this paper is organized as follows. Section II discusses dockerization and multi-tenancy aspects. Section III presents related works on multi-tenancy identification. Section IV shows an evaluation of multi-tenancy impact on docker services. Section V introduces our proposed model, while Section VI describes the implemented prototype. Section VII evaluates, and Section VIII draws conclusions.

II. Preliminaries

A. Containers

Containers are significantly more lightweight than traditional VMs, as they share their host OS libraries [3]. They are executed in a user-space process in the host OS, wherein the container engine must ensure isolation between both containers and host. For instance, in Linux OSes, a popular approach to containerization relies on the Docker engine, which provides container isolation through namespaces [11].

Each namespace creates an abstraction to the running process (container). The processes inside a namespace have a different and isolated host resource view. Namespaces are often implemented through kernel APIs, enabling the container engine to create resources ”clone”. For instance, mounting a virtual file system for the container storage, create a process ID (PID) for a container execution, and a network bridge for the container network traffic [12]. As a result, containers do not rely on hypervisors for multi-tenancy but rather are executed as a user-space process in the host OS. However, process scheduling in OS does not take into account the multi-tenancy aspects inherent in containers.

Traditional hypervisors often employ fairness algorithms for VMs resource scheduling, such as disk, network, and CPU utilization [4]. Consequently, the container performance may significantly degrade in a multi-tenancy setting, as other tenants (containers) may exhaust the physical hardware. It is essential to take into account that scheduling algorithms were not designed to handle multi-tenancy. Besides, as executed as a user-space process, a single container may spawn other child processes, which are scheduled as another process by the host OS, further degrading performance for other containers [12].

B. Container Orchestration

Container engines such as Docker are responsible for deploying and managing containers in a single host. Thus a container orchestration framework must be used to provide scalability through several nodes. Kubernetes is one of the most used orchestration engines, which manages the scheduling and deployment of containers in a physical cluster [12]. Kubernetes cluster comprises several workers, which are used to deploy containers, and a master node, which performs the proper cluster management. The containers' deployment is accomplished through a Pod configuration file, which describes the service cluster configuration, e.g., a Pod file containing front-end, back-end, and database containers. The Pod file is used as input to the Kubernetes master, which handles the proper container deployment between its available Kubernetes workers.

Multi-tenancy aspects within a single Kubernetes worker are handled by the container engine itself, e.g., Docker. Kubernetes provides performance configuration for container deployment, which is achieved by setting lower and upper bounds for CPU and memory usage, which are then set as a container property when it is deployed, then handled by the host OS. Notably, proper overbooking of resources is not managed in Kubernetes, as the orchestration engine is only responsible for scheduling container deployment among several workers. The orchestration engine may deploy a heavily resource-demanding container in a Kubernetes worker without considering their resource usage, which may significantly degrade the performance of other already deployed containers in the selected worker.

III. Related Works

In general, research effort is conducted on providing approaches for performing overbooking of resources without the cloud client QoS degradation at the cloud computing provider perspective.

For instance, F. Caglar *et al.* [13] proposes a machine learning technique to predict future cloud tenant resource usage for better overbooking of resources. The authors can increase the physical hardware usage in IaaS clouds, hence, increase the cloud provider profits without incurring client QoS issues. On the other hand, D. Hoeflin *et al.* [5] proposes an analytical model to improve physical hardware resource usage on IaaS.

L. Tomás *et al.* [14] proposes a new cloud provider configuration to enable the mapping of critical client services to physical CPU cores for performance improvements. Ideally, identifying overbooking of resources should be made at the client-side due to conflict of interests. For instance, a machine learning technique for identifying overbooking was proposed by C. Vicentini *et al.* [4]. The author's approach performs periodic benchmarks within the client VM to identify performance deviations with a classification technique.

An opportunistic offloading technique was proposed by X. Sun *et al.* [15] wherein a set of client services were migrated to the cloud if response time is not critical.

S. Venkateswaran2019 *et al.* [16] proposes a bare-metal VM placement SLA to provide performance guarantees. On the other hand, E. Truyen *et al.* [7] proposes a new SLA model for service containers, which does not take into account the overbooking of resources.

To the best of our knowledge, we are the first to address in the service containerization context, identifying overbooking of resources that may incur QoS degradation from the client perspective.

IV. Problem Statement

In recent years, several traditionally deployed services in IaaS clouds have migrated to dockerized solutions. Simultaneously, the performance impact due to overbooking of resources in container-based environments remains overlooked in the literature.

This section investigates the impact of multi-tenancy and resource overbooking on dockerized applications' performance. We deployed a Big Data processing task in a distributed and dockerized manner through Kubernetes for the analyses. Also, it is described the deployed testbed and the performance issues experienced due to overbooking.

A. Testbed

To evaluate the performance degradation due to resources overbooking, we deploy a distributed container orchestration environment through Kubernetes $v.1.19$ and Docker $v.19.03.13$. Four physical machines compose the testbed, wherein one node is used for the Kubernetes master, and three nodes as Kubernetes workers. The nodes are equipped with an 8-core Intel i7 CPU, 16GB of memory, interconnected through a gigabit network, with an Ubuntu OS v. 18.04.

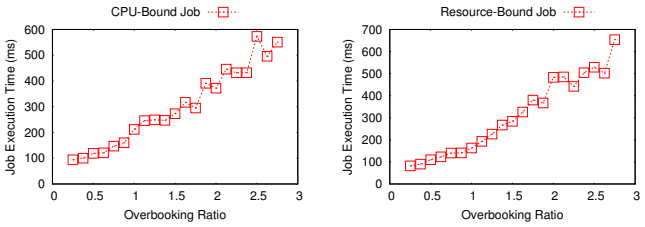
As an application use-case, we consider a containerized distributed Apache Spark $v.3.0.0$ cluster [17], composed of 1 Apache master container and 3 Apache spark worker containers, the workers are configured to use 2 CPU cores. The application container cluster is deployed through a Kubernetes Pod. Two Apache Spark jobs [18] were evaluated, as described below:

- ***CPU-bound Job***: Distributed containerized Apache Spark job that computes PI number up to a 10,000nd digit precision. Processing demand is strictly CPU-bound.
- ***Resource-bound Job***: Distributed containerized Apache Spark job that continuously computes word occurrences in a 500MB file. Processing demand is both CPU, disk, and network bounded.

To create an overbooking of resource situation for each job execution, we also vary the number of concurrent Kubernetes tenants (containers). To achieve such a goal, before the evaluated Apache Spark jobs are deployed and executed, we also submit a Kubernetes Pod that instantiates containers to run single-threaded CPU benchmarks through the $sysbench$ tool. The number of deployed tenants in the concurrent Kubernetes Pod varies throughout the testbed execution, creating a controlled resource overbooking throughout the evaluation.

B. The impact of multi-tenancy in containers

Figure 1 shows the job processing time for each evaluated Apache Spark job according to the overbooking ratio. It is possible to note the job processing time degradation, i.e., performance, on both evaluated jobs when a 0.75 overbooking ratio is surpassed, with further impact if the provider performs a 1.0 rate of overbooking. For instance, in a 2.0 overbooking ratio, the CPU-bound job increases its job processing time by 75%, while the Resource-bound job increases it by 196%. Noteworthy, only a 0.1 increase in the overbooking ratio incurs, on average, on 11%, and 14% of job processing time increase for both CPU-bound and Resource-bound jobs, respectively.



(a) CPU-bound Job. (b) Resource-bound Job.

Fig. 1: Performance impact on evaluated containerized Apache Spark jobs under overbooking of resources settings. The overbooking ratio is computed as the relation between the number of deployed containers and the available physical CPU cores in each Kubernetes node.

As a result, the overbooking of resources in containerized applications significantly degrades application performance. However, from a client perspective, s/he cannot establish the issue root-cause, as in its dockerized domain, the resource overbooking is not visible.

V. ML modelling for detecting Docker-based App overbooking onto Kubernets

This section presents the novel machine learning model to detect resource overbooking within the client domain in container orchestration environments. It is composed of two main steps, namely *Containerized Monitoring* and *SLI Deviation Detection*, as shown in Figure 2. The model's implementation in a classifier aims to monitor both application and container OS (docker) metrics to identify multi-tenancy issues caused due to the overbooking of resources.

The proposal considers a container orchestration service wherein a provider tenant (client) wishes to monitor her application for provider resources overbooking in real-time. In such a context, the client cannot access the provider infrastructure settings, including the Docker or the container host OS. The monitoring can only be performed within the client container domain by monitoring its applications and OS. The monitoring task aims to find SLI deviations in real-time, for instance, a service provider that cannot provide the proper CPU time for the container client. Our proposal considers the identification of SLI deviations as a supervised machine learning task. Therefore, a set of dockerized metrics, such as OS resources usage and dockerized application performance metrics, is used as input for a machine learning classifier that detects SLI deviations.

The next subsections further describes the proposed model, including the *Containerized Monitoring* and *SLI Deviation Detection* modules.

A. Containerized Monitoring

Monitoring performance issues within containerized applications is a challenging task. The docker does not have access

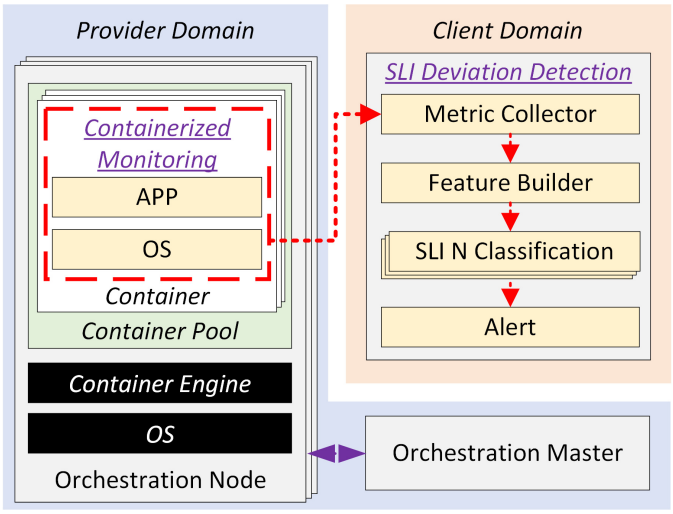


Fig. 2: The proposed machine learning model for detection of overbooking of resources within the client domain.

to host-related metrics, e.g., CPU steal-time in Linux, that measures the amount of requested CPU time the hypervisor could not handle to the VM. In contrast, a dockerized app only has access to its OS-related (containerized OS) and application-related (containerized processes) metrics.

The *Containerized Monitoring* module goal is to continuously monitor both containerized OS and app performance metrics. Our approach's rationale is to use both containerized OS and application performance metrics to detect performance issues within the docker domain. The assumption is that a machine learning classifier can be applied to detect SLI deviations by analyzing the containerized OS usage metrics and application performance metrics. In a multi-tenant interference scenario, the OS usage metrics will be high, while the application will perform poorly. Therefore, it can be detected through the analysis of its features by a machine learning algorithm.

The *Containerized Monitoring* module is composed by two entities, namely *APP Collector* and *OS Collector*. The prior periodically and continuously extracts application performance metrics within the docker environment, e.g., extract features related to the number of clients that the application has successfully processed within a time interval. On the other hand, the *OS Collector* periodically and continuously extracts the containerized OS usage metrics, e.g., the usage percentage of the dockerized CPU. As a result, the *Containerized Monitoring* extracts two sets of features regarding the application performance and OS usage, both within the tenant domain.

B. SLI Deviation Detection

The detection of SLI deviations caused by resource overbooking interference within the containerized (client) environment is challenging. Our proposal considers the identification of multi-tenancy issues as a supervised machine learning classification task to address such a challenge.

The classification procedure is executed periodically and continuously in a client domain environment (*SLI Deviation Detection*, Figure 2). The client domain is executed outside the provider domain. Therefore, it does not seem prone to conflict of interest with the provider. The classification starts with a *Metric Collector* module that collects both *APP Metrics* and *OS Metrics* from the *Containerized Monitoring* module (Section V-A) from a set of containers being executed in the provider domain. The metrics are collected periodically at each predefined time interval, e.g., every 5-seconds. The set of collected metrics from each docker is forwarded to a *Feature Builder* module, whose goal is to compound a feature vector for classification. A set of feature vectors are built, in which each vector comprises both *APP Metrics* and *OS Metrics* for a single container.

The set of built feature vectors are forwarded to a *SLI Classification* module, which executes a machine learning classifier to classify each given feature vector as an overbook or normal scenario. Overbook classified scenarios are containers currently experiencing multi-tenancy interference due to provider overbooking of resources. As a result, the proposed model can identify multi-tenancy interference within the container environment.

C. Discussion

Identifying resources overbooking interference within the client domain is challenging, especially in containerized environments that have been overlooked in the literature. Our proposal treats resource overbooking detection as a classification problem through machine learning techniques to address such a challenge. The classification task receives as input both containerized application performance and containerized OS usage metrics. As a result, it can detect, for instance, an application performing poorly. Simultaneously, the containerized OS is heavily used, an indication of a possible multi-tenancy interference in a resource overbooking scenario. The proposal can also be implemented within the client domain, thus addressing possible conflict of interests with the cloud provider.

VI. Prototype

A proposal prototype was implemented as a distributed processing application executed in a container orchestration environment, as shown in Figure 3. The prototype was implemented in the same testbed evaluated previously (Section IV), through Kubernetes and a client application executing an Apache Spark cluster, composed of three Apache Spark workers and one Apache Spark master. Each client container executes both *APP Metrics Collector* and *OS Metrics Collector*. The prior collects five application-related performance features from the Apache Spark monitoring API, while the former collects eight containerized OS-related features with the PSutil $v.5.7.2$ python API.

The collected set of features from both collectors are listed in Table I. The features are collected from each container in a

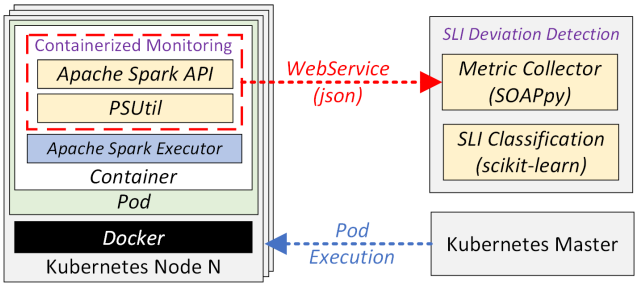


Fig. 3: Implemented prototype of the proposed machine learning model for detection of overbooking of resources within client domain.

TABLE I: Extracted set of features from the containerized monitoring module every 5-second time interval.

Feature Set	*Extracted Features*
APP Metrics (Apache Spark API)	JVM CPU Usage, Shuffle Writes, Shuffle Reads, Exec. Memory Usage, JVM GC Time
OS Metrics (PSUtil)	CPU Usage, Memory Usage, Read Disk Sectors, Written Disk Sectors, Uploaded Bytes, Downloaded Bytes, Uploaded Packets, Downloaded Packets

5-second time interval. At each time interval, the *Metric Collector* module receives the collected features through a SOAP (Simple Object Access Protocol) web-service, as implemented through the SOAPpy API $v.0.12.22$. A machine learning classifier classifies each container's built feature vector through the scikit-learn API $v.0.23$.

VII. Evaluation

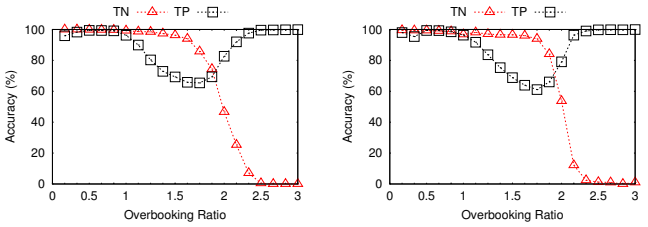
The present evaluation aims at answering the following research questions: ($Q1$) *Is the proposal able to detect resource overbooking within the client domain?* ($Q2$) *What is the minimal resource overbooking needed for accurate detection?* ($Q3$) *What is the detection delay for the identification of resource overbooking?*

The next items describe our testbed and how our proposal performs when detecting multi-tenancy interference within the client domain.

A. Overbooking of Resource Detection

To evaluate the proposed model was used the same set of experiments performed in Section IV-A to implement our proposal prototype. Each overbooking degree setting, from 0.25 to 3.0 ratio, was evaluated, wherein the containerized Apache Spark jobs are monitored for 10 minutes of execution in each scenario. Therefore, for each evaluated overbooking setting, an average of 120 (5 per second) feature vectors are collected for each Apache Worker container.

The first experiment aims to answer question $Q1$ and evaluate the proposed model accuracy for identifying the resources overbooking within the client domain. We evaluate four commonly used machine learning classifiers [19], (i) the Decision Tree (DT) classifier with a confidence factor of



(a) CPU-bound Job. (b) Resource-bound Job.

Fig. 4: Gradient Boosting classification accuracy with all features while varying the overbooking label threshold. The proposed approach can reach high detection accuracies with an overbooking ratio threshold as small as 1.2.

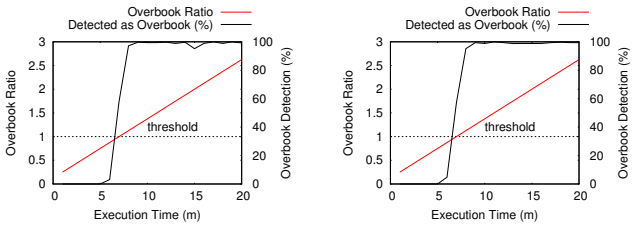
TABLE II: Proposed approach accuracies for overbooking detection within client domain considering scenarios with less than 1.0 of overbooking ratio as normal.

Feat. Set	Classifier	Accuracies (%)			
		CPU-bound Job		Resource-bound Job	
		TP	*TN*	*TP*	*TN*
All Feat.	DT	94.45	89.01	94.74	88.84
	RF	98.42	90.43	97.51	92.01
	GBT	98.35	90.43	98.15	91.87
	kNN	96.03	68.60	88.34	51.37
Feat. Sel.	DT	94.59	88.88	95.09	89.39
	RF	98.56	89.92	97.72	91.73
	GBT	98.63	89.92	***98.08***	***91.73***
	kNN	96.03	68.21	90.40	58.95

0.25, (ii) the Random Forest (RF), and (iii) Gradient Boosting (GBT) classifiers with 100 decision trees as their base-learners, and the (iv) k-Nearest Neighbor (kNN) classifier with five neighbors. Each classifier was evaluated with and without feature selection being made. To this end, the feature selection applies a filter-based information gain technique, wherein only features with an information gain over 0.2 are used for the model building procedure. Two classification classes were used, *normal* and *overbooking*.

For the model building process, the collected data from 2 Apache workers (containers) are used for training, while the data from the remaining Apache worker is used for testing. The classifiers were evaluated with respect to their True-Negative (TN) and True-Positive (TP) accuracies. The TN rate denotes the ratio of *normal* (non overbooking) measurements correctly classified as *normal*. The TP rate denotes the ratio of *overbooking* measurements correctly classified as *overbooking*.

Table II shows the proposal classification accuracy for the evaluated classifier according to each Apache Spark job to detect resources overbooking ratio higher than a 1.0 threshold. It is possible to note that our proposal was able to provide high detection accuracies, up to 98.15% of TP, and 91.87% of TN for the resource-bound job with the GBT classifier with all features, while a 98.35% of TP, and 90.43% of TN for the CPU-bound job with the same classifier. The proposal provided similar classification accuracy regardless of the evaluated Apache Spark job, showing its applicability. Therefore,



(a) CPU-bound Job. (b) Resource-bound Job.

Fig. 5: The proposed model overbooking detection rate using GBT classifier with all features, through several overbooking of resources configurations. The classifier is trained with a 1.0 overbooking detection ratio threshold

the proposal can detect resources overbooking within the client domain with high detection accuracy.

To answer question $Q2$, we evaluate the proposal sensitivity to overbooking of resources. We vary each scenario label as *normal* or *overbooking* according to its overbooking ratio, as the operator may define the detection sensitivity for each need. Figure 4 shows the proposal accuracies for the overbooking ratio label threshold with the GBT classifier will all features (most accurate classifier, Table II). It is possible to note that the proposal can identify with high detection accuracy overbooking interferences that will cause significant processing time increase (when the overbooking ratio surpasses the threshold of 1.2, as evaluated in Section IV). If demanded by the operator, the proposal can reliably detect overbooking up to 1.25 ratio while reaching up to 95% in both TN and TP detection rates, regardless of the processed Apache Spark job.

Finally, to answer question $Q3$, we deploy our proposed model with a 1.0 overbooking ratio label threshold and investigate the overbooking detection sensitivity while varying the concurrent multi-tenancy interference. Figure 5 shows the proposal detection accuracy over time and the detection delay at each overbooking ratio change. It is possible to note that our proposal can detect multi-tenancy interference as soon as the current Apache job in execution begins to degrade its performance after a 1.0 overbooking threshold is surpassed. Besides, one can further decrease the detection speed by reducing the feature extraction collection period (5-second in our prototype).

VIII. Conclusion

Resources overbooking is a known challenge in traditional cloud computing environments, which remains overlooked in current containerized environments. This paper has proposed a novel approach for identifying resources overbooking within the client domain, which incurs the quality of service degradation.

Through the application of machine learning techniques over both containerized OS and application performance metrics, the proposed model was able to identify with high detection accuracy. Such a situation happens when the cloud provider performs resources overbooking that will affect containerized services' processing performance.

As future works, we will evaluate the proposed model accuracies on further containerized applications and under more dynamic environments.

Acknowledgment

This work was partially sponsored by Brazilian National Council for Scientific and Technological Development (CNPq), grant nº 430972/2018-0).

References

[1] S. Goyal, "Public vs private vs hybrid vs community - cloud computing: A critical review," *International Journal of Computer Network and Information Security*, vol. 6, no. 3, pp. 20–29, Feb. 2014.

[2] V. Medel, O. Rana, J. ángel Bañares, and U. Arronategui, "Modelling performance & resource management in kubernetes," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*. ACM, Dec. 2016.

[3] T. Combe, A. Martin, and R. D. Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sep. 2016.

[4] C. Vicentini, A. Santin, E. Viegas, and V. Abreu, "A machine learning auditing model for detection of multi-tenancy issues within tenant domain," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 543–552.

[5] D. Hoeflin and P. Reeser, "Quantifying the performance impact of overbooking virtualized resources," in *2012 IEEE International Conference on Communications (ICC)*. IEEE, Jun. 2012.

[6] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Transactions on Internet Technology*, vol. 20, no. 2, pp. 1–24, May 2020.

[7] E. Truyen, D. V. Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen, "Towards a container-based architecture for multi-tenant SaaS applications," in *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware - ARM 2016*. ACM Press, 2016.

[8] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "SmartVM: a SLA-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, May 2018.

[9] E. Viegas, A. Santin, J. Bachtold, D. Segalin, M. Stihler, A. Marcon, and C. Maziero, "Enhancing service maintainability by monitoring and auditing SLA in cloud computing," *Cluster Computing*, Nov. 2020.

[10] V. Abreu, A. O. Santin, E. K. Viegas, and V. V. Cogo, "Identity and access management for IoT in smart grid," in *Advanced Information Networking and Applications*. Springer International Publishing, 2020, pp. 1215–1226.

[11] P. C. V. Varma, V. K. C. K., V. V. Kumari, and S. V. Raju, "Analysis of a network IO bottleneck in big data environments based on docker containers," *Big Data Research*, vol. 3, pp. 24–28, Apr. 2016.

[12] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[13] F. Caglar and A. Gokhale, "ioverbook: Intelligent resource-overbooking to support soft real-time applications in the cloud," in *2014 IEEE 7th International Conference on Cloud Computing*, 2014, pp. 538–545.

[14] L. Tomás and J. Tordsson, "Cloud service differentiation in overbooked data centers," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 2014, pp. 541–546.

[15] X. Sun and N. Ansari, "Latency aware workload offloading in the cloudlet network," *IEEE Communications Letters*, vol. 21, no. 7, pp. 1481–1484, Jul. 2017.

[16] S. Venkateswaran and S. Sarkar, "Time-sensitive provisioning of bare metal compute as a cloud service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, Jul. 2019.

[17] E. K. Viegas, A. O. Santin, and V. Abreu, "Machine learning intrusion detection in big data era: A multi-objective approach for longer model lifespans," *IEEE Trans. on Network Science and Engineering*, 2020.

[18] C. Vicentini, A. Santin, E. Viegas, and V. Abreu, "SDN-based and multitenant-aware resource provisioning mechanism for cloud-based big data streaming," *Journal of Network and Computer Applications*, vol. 126, pp. 133–149, Jan. 2019.

[19] E. Kugler, A. O. Santin, V. V. Cogo, and V. Abreu, "A reliable semi-supervised intrusion detection model: One year of network traffic anomalies," in *Int. Conf. on Comm. (IEEE ICC)*, 2020.