Journal of Network and Computer Applications 224 (2024) 103839

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Research paper

Detection of quality of service degradation on multi-tenant containerized services[☆]

Pedro Horchulhack[a], Eduardo K. Viegas[a,*], Altair O. Santin[a], Felipe V. Ramos[a], Pietro Tedeschi[b]

[a] Graduate Program in Computer Science (Programa de Pós-Graduação em Informática - PPGIa) at the Pontifical Catholic University of Parana (Pontifícia Universidade Católica do Paraná), Brazil
[b] Autonomous Robotics Research Center, Technology Innovation Institute, Abu Dhabi, United Arab Emirates

ARTICLE INFO

Keywords:
QoS
Container
Docker
Kubernetes
Machine learning
Multi-tenancy

ABSTRACT

Computational services are progressively migrating to container-based solutions due to their faster provision time and lower resource allocation overhead. Service providers rely on container multi-tenancy to share their computing infrastructure and pave their way to profit. Yet, the Quality of Service (QoS) impact due to container multi-tenancy on deployed services is still widely overlooked in the literature. This paper proposes a new service provider monitoring model for detecting container multi-tenancy that can lead to QoS degradation, split into two phases. First, we monitor the container resource usage through the container engine and the associated process namespace, thus, keeping container isolation. The main assumption is that the assessment of resource utilization, as measured by the container engine, should be complemented by the concurrent reporting of resource utilization from the container processes. Second, we detect QoS degradation of distributed containerized services through a time-series classification approach coped with a feature reduction technique. The feature reduction selects the prominent performance features from the deployed distributed service, whereas the time-series classifier ensures the evaluation of the deployed service over time, improving the system detection accuracy. Thanks to an extensive experimental assessment considering a containerized distributed Big Data processing platform, we show that container multi-tenancy can affect the processing performance and the QoS of deployed services. In addition, we show that the proposed model can detect QoS degradation at the service provider domain with a true-positive rate up to 90%, a false-positive rate of 8%, and an average F1-Score up to 0.94.

1. Introduction

Over the last years, computational services have increasingly relied on publicly shared computational infrastructures (Arunarani et al., 2019). Public service providers use virtualized hardware resources provided through a *pay-as-you-go* model, where clients only pay for the hardware they use, leaving idle resources to be allocated to other tenants (Shen and Chen, 2022). The conventional service model performs the hardware sharing through a hypervisor software, in which Virtual Machines (VMs) are executed on top of a virtualized abstraction layer of the physical hardware provisioned to tenants (Arunarani et al., 2019). As a result, the resource provision task typically introduces significant allocation overheads, considering that an additional Operating System (OS) must be instantiated for every newly provisioned VM (Mavridis and Karatza, 2019).

In this context, computational services have been progressively migrating in recent years to container-based deployments due to their reduced resource footprint and lower virtualization overheads (Zhang and Zhou, 2023). Containers are usually executed as an isolated host OS process, sharing the host libraries and resources (Xavier et al., 2013). On the one hand, it demands significantly less computational resources and provision time (Kozhirbayev and Sinnott, 2017). On the other hand, they are managed by the host OS kernel, which does not adequately address the resource sharing fairness in a multi-tenant setting (Truyen et al., 2016).

The traditional VM-based deployment relies on the hypervisor software to fairly manage the physical hardware (Zolfaghari et al., 2021). The hypervisor addresses the hardware provision, concurrent access,

☆ This work was partially sponsored by Brazilian National Council for Scientific and Technological Development (CNPq) grants n. 304990/2021-3 and 407879/2023-4.

* Corresponding author.

E-mail addresses: pedro.horchulhack@ppgia.pucpr.br (P. Horchulhack), eduardo.viegas@ppgia.pucpr.br (E.K. Viegas), santin@ppgia.pucpr.br (A.O. Santin), felipe.ramos@ppgia.pucpr.br (F.V. Ramos), pietro.tedeschi@tii.ae (P. Tedeschi).

https://doi.org/10.1016/j.jnca.2024.103839
Received 5 September 2023; Received in revised form 17 November 2023; Accepted 27 January 2024
Available online 1 February 2024

and sharing, ensuring proper management when several tenants compete for limited hardware resources (Parast et al., 2022). In contrast, the resource provision task of containers is managed by the OS, which often addresses the hardware resource sharing as a traditional OS process (Joy, 2015). For example, Linux-hosted containers are controlled through namespaces and control groups (`cgroups`), which only address the container isolation and resource provision (Zeng et al., 2023). Consequently, containers cannot reasonably address the tenants' dispute of resources while regulated to the same level of guarantees as a traditional hypervisor does (Morabito et al., 2015). In practice, multi-tenancy may significantly affect the QoS of provisioned containerized services and remains frequently overlooked in the literature. In general, current solutions, in their vast majority, assume that containers are subject to the same level of performance degradation experienced in traditional VM-based service deployments.

Service providers rely on their infrastructure multi-tenancy to pave their way for profit and reduce costs (Yao et al., 2019). A common practice even involves the overallocation of virtual resources to tenants to optimize the utilization of the service provider physical hardware. This situation can lead to a degradation in service QoS given the limited physical hardware availability, which can ultimately restrict the tenants' complete utilization of their paid virtual hardware. Taking it into account, hypervisors have been continuously improved to ensure fair resource sharing between hosted tenants (Kontodimas et al., 2015; Zhong et al., 2012; Yang et al., 2021). For instance, based on the tenant's lack of CPU usage over time, the VMWare CPU scheduler grants higher priorities to tenants with more CPU shares (Ali et al., 2019). As a result, even in a shortage of physical hardware resources, the hypervisor can ensure fair hardware access, decreasing the impact on the tenant service QoS. In contrast, container engines rely only on the host's kernel scheduler, which may defer the container access to the physical hardware to pave the way for a higher privileged host OS process (Cinque et al., 2022). Surprisingly, prior works overlook such a challenge by assuming that the impact of hardware overallocation in container settings is similar to those experienced in traditional VM-based deployments (Felter et al., 2015).

In practical terms, identifying a multi-tenant configuration that may lead to a QoS degradation poses a significant challenge for service providers (Fayos-Jordan et al., 2020). The resource utilization of containerized services exhibits substantial variability and evolves according to the deployed service. Additionally, isolating containers from service providers is crucial to address security and data privacy concerns (Bélair et al., 2019). A situation that leaves designed QoS degradation detection schemes without empirical evidence of the current client service performance. Consequently, identifying a multi-tenant configuration that has the potential to impact the client's QoS remains an ongoing challenge for service providers.

Contribution. In this paper, we propose a new monitoring model for service providers to detect multi-tenant interferences that can impact the client container QoS. The proposal is split into three phases. First, to maintain the container integrity, we periodically collect the container resource usage through the container engine and the associated container namespace. Our primary assumption is that container QoS degradation can be identified within the service provider domain by evaluating the resource usage of the container processes and the associated resource usage provided by the container engine. The main insight is that the assessment of resource utilization, as measured by the container engine, should be complemented by the concurrent reporting of resource utilization from the container processes. Second, the collected set of features is used as input to a feature reduction module that selects the most prominent performance features collected from the client containers. The objective is to evaluate the service performance even if the client executes a distributed service (e.g., Big Data processing). Third, the output of the feature reduction module is used by a time series classifier implemented through a Recurrent Neural Network (RNN) architecture. The rationale of such a technique is that the time-series classification can classify performance degradation while considering the historical container performance.

In summary, the main contributions of this paper are:

- An evaluation of the QoS degradation of distributed containerized services in a multi-tenant scenario. The performed evaluations, considering a distributed Big Data processing service as a use-case, have shown that the performance of container-based services is significantly affected in a multi-tenant setting. In detail, multi-tenancy can increase by up to 84% the processing time of containerized Apache Spark jobs.
- A monitoring model for service providers aiming the identification of multi-tenant configurations that can lead to QoS degradation in distributed containerized services settings. The proposed model is executed within the service domain without violating client isolation. It can detect multi-tenancy interferences to deployed containers with up to 90% of true-positive rates at the container level and up to 0.88 of F1-Score at the service level.

Roadmap. The structure of this paper is outlined as follows. Section 2 provides a detailed description of containers and multi-tenant environments. Section 3 presents a comprehensive review of related works on QoS degradation detection. Section 4 assesses the impact of container multi-tenancy on the QoS of containerized services. Section 5 describes our proposed model, followed by Section 6, which presents its prototype. Our proposed scheme is evaluated in Section 7, and Section 8 concludes our work.

2. Background

This section provides an in-depth description of the microservice orchestration, encompassing the virtualization and containerization service provision approaches. Furthermore, it introduces the key elements of Deep Learning (DL) that will be used in our contribution.

2.1. Container

Containers have been increasingly used due to their lightweight sharing of physical hardware resources (Henkel et al., 2020). In this scenario, the client deploys the needed containerized application service, sharing the host OS and its libraries with other tenants. The OS resources are abstracted, while the container is isolated from the host OS and other containers (Casalicchio and Perciballi, 2017). Unlike traditional VM-based approaches, where the hypervisor handles physical resource access and fair sharing among other tenants, a container relies on the host OS to manage the resource allocation. On the one hand, the host OS must ensure the fairness of resource distribution among its tenants, which is generally executed and operated as an isolated host OS process. On the other hand, the host OS kernel controls the isolation and resource allocation (Zhong and Buyya, 2020).

In practice, Docker-based containers in Linux OS are managed through namespaces and `cgroups` (Sultan et al., 2019). The namespaces handle and isolate the container resources such as Process ID (PID), User IDs, filesystems, and network interfaces. The client can only access their associated namespace's isolated resources in the container space. The `cgroups` manage and partition the hardware resources between the processes by hierarchically organizing them, wherein each container has a predefined resource allocation limit. Therefore, process containerization can avoid the significant performance overhead on the host, unlike the commonly associated performance burden incurred by traditional VMs-based deployments. Al-Dhuraibi et al. (2018).

Fig. 1 depicts the architectural deployment differences between VMs and containerized deployment approaches. In such a case, we illustrate a traditional setting, where VM virtualization is conducted through a type 2 hypervisor, which runs on top of a OS. Each VM deployment demands the instantiation of a new OS, resulting in considerable resource overhead. In contrast, containers are managed as a host OS process, thus not incurring significant resource allocation overheads but competing for the same physical resources without proper hypervisor management.

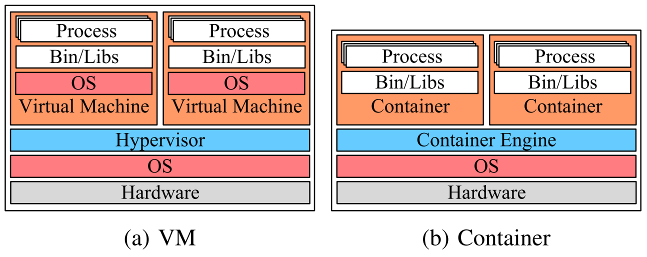


Fig. 1. Comparison between VM and container deployment strategies.

2.2. Orchestrating microservices

Containers are strongly associated with microservices deployment, as they ease their provision while ensuring fault-tolerance, elasticity, and resource management (Alshuqayran et al., 2016). Microservice architectures are based on Service Oriented Architecture (SOA), which decouples the application into loosely-coupled services independently deployed by a fully automated deployment system (Li et al., 2023).

A microservice-based system comprises single units, each responsible for a single task. For example, a monolithic Web Application can be loosely decoupled into a front-end, back-end, and a database. The communication between each microservice is then achieved through an Application Programming Interface (API). In contrast to a traditional monolithic approach, those services are usually implemented by leveraging containers (Section 2.1), which brings a lightweight feature for deployed applications (Kang et al., 2016).

As the number of components increases, similar to the traditional cloud computing approach, it is important to guarantee efficient maintainability and flexibility. In such a case, Kubernetes, also known as K8s, can be used as an open-source container orchestration service to deploy, maintain, and scale containerized applications (Vayghan et al., 2018). Fig. 2 illustrates the Kubernetes architecture, based on the master-worker paradigm (Chen et al., 2018), in which each node is responsible for creating a Pod (Zhang et al., 2021). The main Kubernetes components are described as follows:

- **Container Engine.** Daemon used to manage and control the container deployment, e.g. Docker.
- **Container.** Isolated process that encapsulates the service application code as a OS process. The container engine instantiates the process according to a predefined image.
- **Pod.** Set of containers running on a cluster, the smallest Kubernetes logical resource. Pods are used to deploy containerized services on a Kubernetes architecture.
- **API Server.** Core Kubernetes component. It provides an API for end-users to communicate with the tool components, such as deployed Pods, and DaemonSets.
- **Kube Scheduler.** It schedules the deployment of Pods to a Kubernetes node following previously defined rules.
- **Controller Manager.** Manages the shared state of the cluster provided by the API Server. It has several controllers, such as Node, Job, and Namespace. Those components supervise the received requests from each Kubernetes object and update the system state accordingly.
- **etcd.** A distributed key–value data store used to coordinate and share resources and configuration data between the system components.
- **CNI Network Plugins.** Container Network Interface (CNI) plugins that implement virtual network interfaces for container communication. They are responsible for allocating IP addresses to pods and exposing them to other nodes within the cluster.

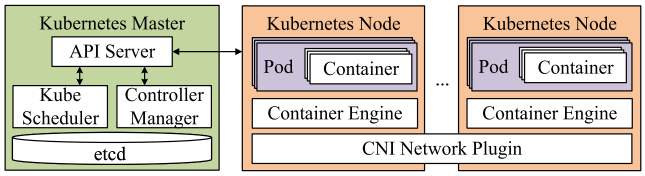


Fig. 2. Kubernetes architecture.

2.3. Deep learning

Over the past years, DL techniques have been increasingly and largely used for classification tasks (Zhu et al., 2017). To accomplish this task, the DL model is built via a computationally intensive training process. The objective of the training task is to derive a behavioral DL model from a comprehensive dataset comprising a significant number of input samples of all evaluated classes. The samples are represented by a feature vector that captures the event behavior under the given problem scenario. Further, the acquired model is evaluated using a test dataset that assesses the expected classification accuracy rates of the deployed model in production environments.

DL techniques can be implemented through several architectures, with each configuration tailored for a specific application (Zhou et al., 2019). In this work, we focus on two main DL-based strategies, namely Deep AutoEncoder (Harush et al., 2021) and RNN (Spooren et al., 2019). Deep AutoEncoders have been typically used for dimensionality reduction and image denoising through a two-step process named *encoder* and *decoder*. The *encoder*'s goal is to compress the deep neural network input by learning a set of representative feature relations. In contrast, the *decoder* aims to decompress the *encoder* output. As a result, the Deep AutoEncoder output is a learned representation of the most representative relations between its input features. In contrast, RNN architectures employ loops to enable pattern recognition tasks that consider information from previous inputs, thereby facilitating time-series classification. The RNN stores input values based on a specified time interval and is commonly used for classifying temporal series or scenarios influenced by historical inputs.

3. Related work

Identifying service QoS degradation in multi-tenant environments poses challenges that have been addressed through diverse mechanisms in recent literature. As an example, in a conventional VM-based scenario, Mi et al. (2011) introduced a technique for pinpointing the underlying cause of performance degradation by analyzing the response latency of client application requests. While their proposed scheme successfully detects performance degradation in deployed cloud services, it lacks explicit identification of the responsible node, either the client or the service provider. Shea et al. (2014) compared the service performance between private and public clouds based on average resource usage over time. They demonstrate that public clouds can experience unstable performance due to more service clients and intensive resource utilization by other tenants. The authors did not attempt to detect performance-degraded VMs or investigate the underlying causes of performance degradation.

Studies on multi-tenancy interferences that can lead to QoS degradation in the literature are still in the early stages. Ruan et al. (2016) examined the performance variation in containerized applications running on VMs in comparison to applications directly deployed on a bare-metal infrastructure. By surveying different hardware metrics, including CPU usage, memory bandwidth, disk I/O performance, and network latency, the authors demonstrated that bare-metal deployed applications exhibit better performance with lower hardware usage. However, the authors did not use a classification technique for detecting multi-tenancy interference. In Wang et al. (2018a), W. Wang

Table 1
Workload profiles executed in our testbed (Fig. 3, *Tenant Pod*).

Workload	Description	Profile	Workload configuration	Node resource commitment ratio (α)
httperf	A docker container running a HTTP benchmark that continuously performs HTTP requests to a predefined website	Low Average High	≈20% of network link ≈50% of network link ≈100% of network link	0.20 0.50 1.00
sysbench	A Docker container running a CPU benchmark that computes prime numbers	Low Average High	≈1 CPU core ≈2 CPU cores ≈4 CPU cores	0.25 0.50 1.00
HiBench (Huang et al., 2022) (RandomForest)	A set of Docker containers (1 master, 3 workers) running an Apache Spark job that builds an ensemble of decision trees on a given input dataset randomly generated	Low Average High	≈20% of node CPU cores ≈50% of node CPU cores ≈100% of node CPU cores	0.20 0.50 1.00
HiBench (Huang et al., 2022) (Terasort)	A set of Docker containers (1 master, 3 workers) running an Apache Spark job that performs input sorting according to a standard benchmark	Low Average High	≈20% of node CPU cores ≈50% of node CPU cores ≈100% of node CPU cores	0.20 0.50 1.00

et al. presented a technique for identifying cloud performance, wherein multiple benchmarks were used to map each cloud performance profile. The authors evaluated service deployment suitability across multiple cloud providers and simultaneously identified performance degradation using their proposed scheme, which involved periodic benchmark executions. However, it introduces a significant trade-off in hardware resource usage as time passes.

Vicentini et al. (2018) proposed an Machine Learning (ML)-based model to detect hardware resource over-commitment in traditional VM-based clouds, with a specific focus on multi-tenancy interferences in VM environments. The authors reported significant QoS degradation of monitored applications even in a VM-based deployment. Another technique was proposed by Grohmann et al. (2019), which measures key performance indicators provided by the application, including CPU usage and memory when compared to resource usage reported by the cloud providers through the application of five shallow classifiers and one DL classifier. The authors identified performance degradation with high accuracy while assuming that public cloud providers enable the collection of their underlying hardware usage. Masouros et al. (2021) proposed a RNN technique for the prediction of performance degradation of container-based cloud providers under interference conditions. The authors considered collecting low-level system metrics (e.g., instructions per cycle and last-level cache misses) as input for their proposed scheme. To represent an interfering tenant, the authors executed resource-specific benchmarks. Further, they identified performance interference with high accuracies when assuming the availability of public cloud providers' low-level system metrics.

Apart from DL-based techniques, Wang et al. (2018b) proposed a self-adaptive cloud monitoring technique based on correlation analysis coped with a feature reduction technique to predict performance degradation. On the one hand, correlation analysis was used to reduce the monitoring overhead when selecting the most significant features to represent a system fault. On the other hand, the feature reduction generates eigenvectors representing the metrics' principal components using cosine similarity between two eigenvectors to detect if the QoS is being affected. The authors induced a performance degradation scenario considering CPU, disk, memory, and network. Their proposal can identify performance degradation with feature engineering techniques. However, the authors do not consider labeling the node where the interference is generated. Another monitoring scheme was proposed by El-Kassabi et al. (2019). The proposed approach identified performance degradation by computing a trust score based on system metrics. The authors addressed a container orchestration environment in the cloud provider domain, while none of ML techniques are considered.

4. Problem statement

This section analyzes the QoS impact of multi-tenancy on containerized applications. We deployed a Big Data processing architecture in a distributed and containerized manner by leveraging Kubernetes. Specifically, we introduce the deployed testbed and investigate the performance issues experienced by container resource sharing in a controlled multi-tenant environment.

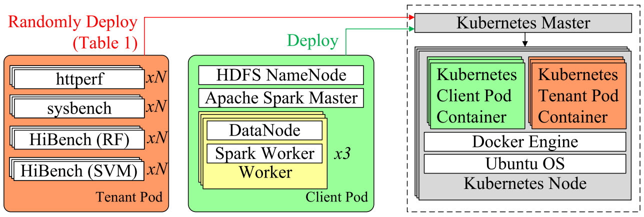


Fig. 3. Client Pods with Apache Spark jobs are deployed in a containerized environment, exposing them to multi-tenancy interferences resulting from the concurrent execution of Tenant Pods workloads (Table 1).

4.1. Testbed

To realistically assess the QoS impact of multi-tenancy on containerized applications, we examine a typical processing demanding distributed application executed as a set of containers. Specifically, we consider a distributed Big Data processing application wherein the operator deploys the job as a set of distributed containers executed on a multi-tenant container orchestration environment. Such a situation is common nowadays due to the widespread adoption of container-based solutions given their isolation with minimal performance overhead (Tang et al., 2022; Ahmed et al., 2020). In addition, container orchestration such as Kubernetes (see Fig. 2) is widely used to share physical infrastructure with multiple tenants in a container environment (Ceesay et al., 2017; Cañete et al., 2024).

The testbed is depicted in Fig. 3, we consider a client that executes its Big Data processing task as a container-based solution in a distributed manner. During this process, the workers deployed by the client experience performance degradation due to resource sharing enforced by a service provider leveraging container multi-tenancy.

The distributed environment is deployed through Kubernetes 1.27 and Docker 24.0.2. Kubernetes is used as a container orchestration framework representing the service provider in our testbed, while Docker is used as the container engine at each node (see Fig. 2). The testbed comprises 4 physical machines, with 3 machines serving as Kubernetes workers and 1 machine functioning as the Kubernetes master. Each node is equipped with an 8-core Intel i7 CPU, 16GB of memory, interconnected through a gigabit network, running an Ubuntu OS 20.04. For the client application, we adopt a containerized distributed Apache Spark 3.4.1 integrated with a Hadoop Distributed File System (HDFS) 3.3.5. The client pod consists of 1 Apache Spark master container, 1 HDFS NameNode container, and 3 worker containers that function as both Apache Spark workers and HDFS DataNodes. Each worker container is configured to leverage 2 CPU cores through the Docker limit configuration. The application container is deployed through a Kubernetes Pod, ensuring consistent execution across each evaluation round.

We evaluate two Apache Spark jobs related to ML in our testbed, considering them as a use case for the client pod (Fig. 3, *Client Pod*), as described below:

- **HiBench** (Huang et al., 2022) Random Forest (RF) **Job**: Distributed containerized Apache Spark job that builds an ensemble of decision trees on a given input dataset randomly generated. The job was implemented in four steps as follows: (*i*) a dataset, with 7.7 GB and 1,000 columns, is randomly generated and stored in HDFS, (*ii*) the stored dataset is loaded to a Spark RDD, (*iii*) a RF classifier with 100 decision trees is trained, and (*iv*) the built classifier is evaluated in a test dataset;
- **HiBench** (Huang et al., 2022) Support Vector Machine (SVM) **Job**: Distributed containerized Apache Spark job that builds an SVM model on a given input dataset randomly generated. The job was implemented in four steps as follows: (*i*) a dataset, with 6.7 GB and 10,000 columns, is randomly generated and stored in HDFS, (*ii*) the stored dataset is loaded to a Spark RDD, (*iii*) a SVM classifier is trained, and (*iv*) the built classifier is evaluated in a test dataset;

To reproduce a real use-case scenario with a multi-tenant configuration that can potentially result in QoS degradation, we concurrently deploy a varying number of additional Kubernetes Pods during each execution of the client Pod (Fig. 3, *Tenant Pod*). More specifically, we consider four tenant pod workloads, namely *httperf*, *sysbench*, *HiBench (RandomForest)* and *HiBench (TeraSort)*, as described in Table 1. Each tenant pod workload varies the configuration of the generated workload, thereby increasing or decreasing resource usage accordingly (Table 1, *Workload Configuration*). During the testbed execution, the number of deployed tenants in the tenant Kubernetes Pod varies, creating, on purpose, a controlled multi-tenancy interference setting for evaluation.

Each testbed execution round procedure is executed as follows:

(1) *Tenant Pod Building*. A tenant pod configuration profile is established to generate a concurrent workload that induces a potential multi-tenant inference (Table 1). To achieve this goal, a random selection of workloads and associated profiles is chosen (ranging 0 to 10 workloads, at *Low*, *Average*, or *High* configuration).
(2) *Tenant Pod Submission*. The tenant pod, randomly defined, is sent to the Kubernetes Master for deployment on the Kubernetes Nodes.
(3) *Client Pod Submission*. The client pod is submitted to the Kubernetes Master.

For each execution round, we compute the cluster level of multi-tenant interference based on the deployed concurrent workloads. Each deployed workload has an associated node resource commitment ratio based on the average usage of resources (Table 1, *Node Resource Commitment Ratio*). The node resource commitment ratio was determined empirically by averaging the measured resource usage required by each workload across multiple execution rounds of the testbed. As a result, the cluster commitment ratio is computed according to the following equation:

$$Cluster\,Commitment = \frac{\sum_{i=0}^{N}\sum_{j=0}^{M}\alpha_j}{N}, \tag{1}$$

where N denotes the number of Kubernetes nodes, M the number of workloads j deployed on a node i, and α the node resource commitment ratio for the workload execution (Table 1, *Node Resource Commitment Ratio*). Our assumption is that the node resource commitment can be measured based on the average workload configurations deployed, as shown in Table 1. Recalling that we vary the workload configuration at every testbed deployment, as shown in Fig. 3. Therefore, considering the network and CPU allocation, it serves as a measure of node resource utilization. Indeed, the cluster commitment can then be computed by aggregating the average resource commitment of each node in the cluster.

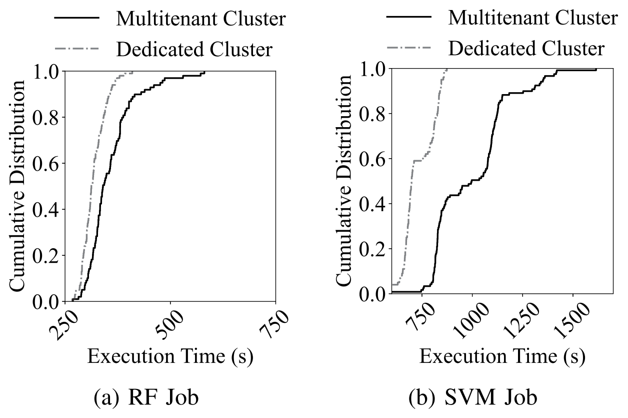


Fig. 4. Performance impact on a distributed containerized Apache Spark job in a multi-tenancy interference setting.

The testbed was executed for a total of 200 execution rounds, wherein 100 rounds accounted for the monitoring of the *HiBench (Huang et al., 2022) RF Job*, and 100 remaining rounds accounted for the monitoring of the *HiBench (Huang et al., 2022) SVM Job*. Both monitored jobs were executed with the same tenant pod configuration profile in our testbed, allowing a proper performance degradation evaluation.

4.2. Assessing the QoS impact due to multi-tenancy in containerized distributed applications

The conducted experiments aimed to address the following research questions (RQ):

- (***RQ1***) *How does multi-tenancy affect the QoS as measured by processing time of containerized services?*
- (***RQ2***) *Does the impact of multi-tenancy vary based on the type of job application?*

The first experiment focuses on addressing *RQ1* by evaluating the job processing time of the containerized Apache Spark job when executed in a multi-tenant environment. To achieve this objective (as described in Section 4.1), we execute the selected jobs while systematically varying the number of concurrently deployed containers running the workload configuration outlined in Table 1. We use the benchmark container to generate the multi-tenancy interference setting in our testbed that can potentially result in a QoS degradation (see Fig. 3). Specifically, the cluster can either be free of multi-tenant interferences or under a varying impact of multi-tenant workloads. In particular, we evaluate the client QoS in terms of the processing time of the deployed jobs.

Fig. 4 shows the distribution of job processing times of the evaluated Apache Spark jobs for our deployed testbed. Indeed, we observe a significant increase in job processing time when the client application is exposed to multi-tenant interferences. For example, in a dedicated cluster without concurrent workloads, approximately 80% of SVM job executions are completed within 770 s. However, in a multi-tenant cluster, the same job requires 1150 s on average, resulting in a 49% increase in processing time. Therefore, the container engine demonstrates an inability to effectively manage the sharing of physical resources among

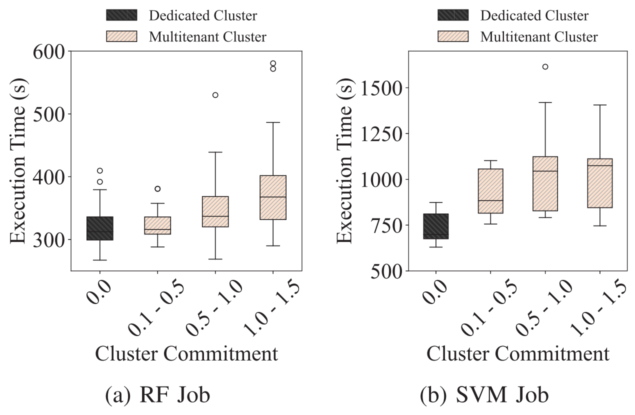


Fig. 5. Performance impact distribution of a distributed containerized Apache Spark job according to the cluster resource commitment.

tenants, even when hardware resources are available. Multi-tenancy interference substantially impacts the QoS of containerized services as measured by their processing time.

To answer *RQ2*, we conduct a detailed analysis of the distribution of the processing impact of the selected jobs concerning the cluster resource commitment ratio. To this aim, we assess the resulting cluster commitment (as defined in Eq. (1)) concerning the job execution time. The objective is to evaluate the impact of cluster resource commitment in a multi-tenant scenario on the processing time of the containerized Apache Spark job.

Fig. 5 shows the distribution in job processing time for each selected job according to the level of cluster commitment. The performance impact varies depending on the cluster commitment ratio and the specific job being executed. For example, at a cluster commitment ratio of 1.0, the average job processing time is increased by 16% for the RF job and 84% for the SVM job. The discrepancy in performance impact can be attributed to the resource-bound nature of the selected Apache Spark jobs. In particular, the RF job exhibits a CPU-bound behavior, as each decision tree is built in a distributed manner. In contrast, the SVM job relies on CPU resources and network utilization to optimize the weights of the support vectors throughout the execution rounds. Regardless of the specific characteristics of the deployed service, multi-tenancy interference can significantly impact the QoS, here measured by the processing performance of containerized applications.

4.3. Discussion

In this section, we discuss the multi-tenancy impact on containerized services' performance. The experimental campaign has demonstrated that multi-tenancy interferences significantly influence the processing performance of deployed containerized services, with a particular focus on a Big Data processing framework as the application use case. Moreover, existing approaches in the literature for detecting multi-tenant QoS degradation (as discussed in Section 3) commonly assume a traditional VM-based deployment, where the hypervisor software plays a crucial role in facilitating the monitoring process. In contrast, detecting performance degradation caused by multi-tenancy in containerized services presents a notable challenge for service providers. This is because the designed detection approaches must operate without breaching client isolation, meaning they cannot access the applications running within the client's domain. Consequently, the service provider cannot determine when a higher level of multi-tenancy might impact the QoS of its tenants. This phenomenon forces service providers to either unintentionally degrade the QoS of their clients or reduce their profits by accommodating fewer tenants on their available hardware infrastructure. Hence, service providers must adopt QoS degradation detection approaches designed explicitly for container environments, ensuring that client isolation is preserved when conducting such a task.

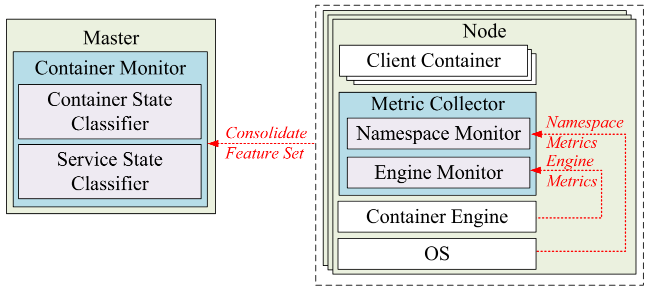


Fig. 6. Proposed model for detecting multi-tenant interference in distributed containerized services within the service domain.

5. Detection of QoS degradation on container-based services

To tackle the challenge of QoS degradation due to the multi-tenancy interferences, we introduce a novel model that operates within the service provider domain while strictly adhering to client isolation constraints. The proposed approach aims to empower service providers by identifying tenants where the existing multi-tenancy configuration might impact the QoS of deployed containers, all while respecting client isolation and without necessitating access to the running applications.

To this aim, the proposed scheme focuses on a distributed containerized service deployed in a multi-tenant environment, such as a containerized Apache Spark application running on Kubernetes. The application service is executed within a containerized tenant, provided and managed by the service provider. The service provider can access the resource utilization of the container as a traditional OS process but cannot access the container's internal components, such as the running service APIs and the performance of the containerized OS. The proposed scheme consists of two primary phases, namely the *Metric Collector* and the *Container Monitor*, as depicted in Fig. 6.

The goal of the *Metric Collector* is to continuously gather metrics on the deployed container and its associated processes. Our proposal operates on the assumption that container performance degradation can be identified by analyzing container resource usage and the running processes associated with it. Our key insight is to monitor the container's resource utilization, as provided by the container engine, and track the associated container process tree through the container namespace. As a result, the service provider can effectively monitor deployed containers while maintaining the integrity of client isolation.

The objective of the *Container Monitor* is to consolidate the metrics obtained from the deployed containers, as collected by the *Metric Collector* module. This module assesses whether the deployed distributed service encounters performance issues caused by multi-tenancy interferences that can be attributed to the service provider. It assesses the QoS of the provider tenants at both container and service granularity. On the one hand, the container QoS is evaluated based on the metrics associated with the container itself and its corresponding set of processes. On the other hand, the service QoS is assessed by analyzing the metrics collected from all containers associated with the evaluated service. To achieve this goal, our proposed scheme employs a time-series classification approach for the monitoring task. Indeed, we evaluate the QoS of deployed containers within the service provider domain by considering the container-related metrics obtained from the container engine and the associated processes retrieved from the host OS namespace. Doing so allows us to assess the container QoS without compromising the client isolation.

The subsequent subsections provide a detailed description of our proposed model, delineating the various modules that constitute its implementation.

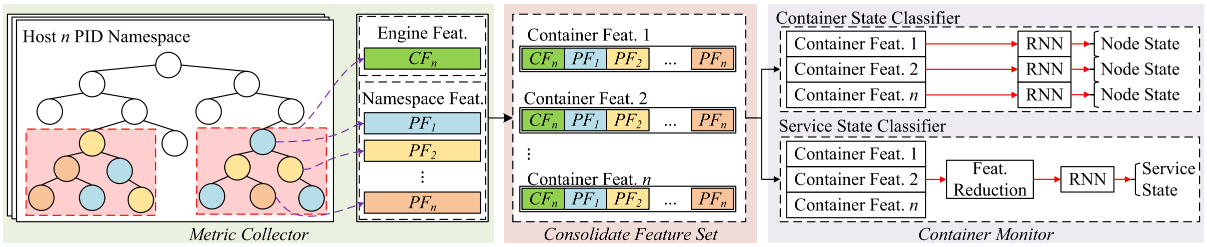


Fig. 7. Proposed model execution flow for detecting multi-tenancy interferences on provider domain that can lead to QoS degradation on containerized services. Proposal detects QoS degradation at container and service levels.

5.1. Metric collector

Monitoring the QoS of containerized services in multi-tenant environments poses a significant challenge for service providers. This challenge arises primarily from the inability to access the client's deployed service, which typically operates within an isolated container and follows a loosely decoupled component architecture based on the principles of microservices. The service provider can only access the metrics provided by the container engine and the running processes of the container through the host OS namespace. To overcome this limitation, our proposal entails continuously collecting metrics from the *Container Engine* and the *Container Namespace*. Our main assumption is that QoS degradation resulting from the current multi-tenancy configuration will manifest as adverse effects on the container process metrics, while the metrics associated with the container engine will indicate high resource usage. For example, this could manifest as low network usage in a specific container process, as measured by the *Container Namespace*, alongside high CPU utilization, as measured by the *Container Engine*.

The operation of the *Metric Collector* module is depicted in Fig. 6, and it is executed at regular intervals on every service provider node, e.g., every 5 s. In this scenario, the monitor module collects two distinct sets of metrics: *Engine* and *Namespace*. The *Engine* metrics provide information about the container's resource usage, as measured by the container engine, such as CPU utilization measured by the container engine (e.g., Docker) On the other hand, the *Namespace* metrics encompass the complete container process tree along with the associated PID resource utilization, such as the network utilization for each process executed within the container. The extracted set of metrics serves a dual purpose. Firstly, it is used as input to the *Container State Classifier*, which detects QoS degradation at the container level (as depicted in Fig. 6). Secondly, the metrics are forwarded to the *Service State Classifier* module, where they are used to classify the level of service degradation in a distributed manner.

As a result, the *Metric Collector* module can provide pertinent metrics to help in the detection of the container's current QoS. It is worth noticing that this is accomplished without compromising client isolation, as the metrics are collected based on the host OS namespace tree and the employed container engine. The subsequent subsection elucidates how the collected metrics are adopted for QoS monitoring.

5.2. Container monitor

Our proposed model considers that container multi-tenancy can lead to QoS degradation at both the container and service levels. The prior focus is on detecting the performance of a specific client container, as this can have a cascading effect on the overall performance of the distributed client service. The latter assessment focuses on evaluating the performance of the client service in relation to the QoS degradation observed across multiple client containers. This is particularly relevant in scenarios with a distributed operating environment, such as microservice architectures. Fig. 7 depicts the operational flow of our proposed *Container Monitor* module. It is split into two detection approaches, namely *Container State Classifier* and *Service State Classifier*.

Algorithm 1 Consolidate Feature Extraction

```
Require:
  Service $S$                                   ▷ Monitored containerized service
  Number of Process $k$                         ▷ Number of process to filter
  Resource for Filter $r$                       ▷ Resource to use for filter
  procedure FEATUREEXTRACTION($s$, $M$, $r$)
      $\mathcal{X} \leftarrow \emptyset$
      for $cont_i \in S$ do
          $CF_i \leftarrow getContainerEngineMetrics(cont_i)$
          $tree_i \leftarrow getContainerNamespaceTree(cont_i)$
          $PF_i = \underset{p \in tree_i}{\text{argtop k}}\ getResourceUtilization(r, p)$
          $X \leftarrow X + \{CF_i, PF_i\}$
      end for
      return $\mathcal{X}$
  end procedure
```

The operation of our proposed monitor initiates with the metric collection as conducted by the *Metric Collector* (see Fig. 7). In the case of deploying a distributed service S using N containers, the *Metric Collector* module extracts two subsets of features, namely the *Engine* and *Namespace* feature sets, for each container $cont_i$. The *Engine* feature set, denoted as CF, encompasses the metrics associated with the container as provided by the container engine (Fig. 7, CF). Conversely, the *Namespace* feature set, denoted as PF, holds the metrics associated with the entire container process namespace $tree$ (Fig. 7, PF). In this regard, for each process p within container $cont_i$, there exists a corresponding PF feature set that describes the process's resource usage.

Since the number of processes can vary depending on the running container service, the *Container Monitor* module ranks and filters the processes based on a pre-defined metric related to resource usage, resulting in a set of k selected processes. For example, the module may select the top 5 container processes with the highest CPU usage over a given time period. The process ranking criteria should be defined based on the operator's specific requirements. For example, CPU-bound services can be ranked based on CPU utilization, while network-bound services should be ranked based on network usage. The objective is to identify the processes associated with the deployed container service without requiring access to the isolated container space. Consequently, each container $cont_i$ is associated with a *Engine* feature set CF and a set of k *Namespace* feature sets PF. The resulting feature set $\mathcal{X}$ serves two purposes: (*i*) identifying container QoS degradation, and (*ii*) identifying service QoS degradation, as illustrated in Fig. 7 (*Container State Classifier* and *Service State Classifier*).

The feature extraction process is outlined in Algorithm 1. The input to Algorithm 1 consists of a containerized service S to be monitored, a resource r that is used for ranking the container processes, and a number k for process filtering. For every container $cont_i$ associated with the service S, the algorithm extracts the *Engine* feature set CF_i. The *Namespace* metrics are extracted for every process from the container $cont_i$. Furthermore, it is filtered based on the specified resource r, which, if ranked as the top k resource usage process, it is added to

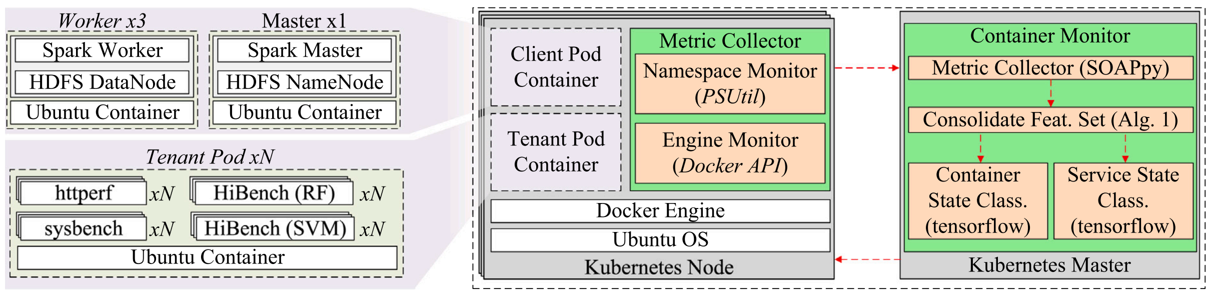


Fig. 8. Prototype architecture of our proposed model.

the vector PF_i. The resulting tuple CF_i and PF_i are then added to the service feature vector $\mathcal{X}$, i.e., the algorithm output.

5.2.1. *Container state classifier*

The *Container State Classifier* uses the resulting feature set $\mathcal{X}$ as input to a RNN model for the identification of the *Node State* (Fig. 7, *Container State Classifier*). The module determines the current configuration of the service provider node, classifying it as either *normal* or *multi-tenant interference*. The *normal* state refers to a container state where the service provider's current multi-tenant configuration does not impact the QoS of the deployed containers. In other words, it represents a container environment without any performance interference caused by multi-tenancy. In contrast, the *multi-tenant interference* state refers to a scenario in which the current multi-tenant configuration is indeed affecting the performance of the running container service, thereby impacting the QoS of the service provider's clients.

Given a RNN model $f(x) : x \rightarrow y$ that outputs the *Node State* y on given a container feature set x, such that $x \in \mathbb{R}$. The goal of the module is to apply the model on the input x to find the *Node State* y. Our proposed scheme defines the detection of the *Node State* as a time-series classification task. This approach is justified by the fact that container resource usage can vary over time, either due to the behavior of the deployed service or the occurrence of multi-tenant interferences. By treating it as a time-series task, the module can capture and analyze these changes over time, thereby improving the system's accuracy.

5.2.2. *Service state classifier*

The objective of the *Service State Classifier* module is to assess the containerized service QoS. Similarly to the *Container State Classifier*, the module receives as input the extracted feature set $\mathcal{X}$ from the monitored containerized service S (Fig. 7, *Service State*). In this case, the goal of the *Service State* detection module is to identify service-level QoS degradation based on the metrics extracted from all service containers. However, this task can be challenging due to the variable number of deployed containers and their diverse performance characteristics.

Further, the module uses a feature reduction technique to address such a shortcoming. The main goal is to capture the correlation among the extracted features from all service containers in a distributed manner. This correlation representation enables the module to effectively utilize the feature set as input for the RNN model. The flow of the *Service State Classifier* module is shown in Fig. 7. It assesses the service state by leveraging the consolidated extracted features from all associated containers (Alg. 1). The module employs a feature reduction technique to establish correlations among the input features within a reduced feature space. Furthermore, this reduced feature space is fed into the RNN model, which generates the output for the *Service State*. Similarly, the *Service State* represents the service state as either *normal* or *multi-tenant interference*.

6. Prototype

Our proposed scheme is implemented, deployed, and evaluated on the testbed previously described in this paper (see Section 4). Fig. 8 shows the used APIs and the deployed architecture. We remark that we deployed a Kubernetes cluster on 4 physical machines. Out of these, 3 machines are configured as Kubernetes Nodes responsible for container deployment, while the remaining machine serves as the Kubernetes Master.

The *Metric Collector* module (see Section 5.1) is implemented on top of the Kubernetes Nodes. The *Engine* metrics are collected through the Docker Engine API. The *Namespace* metrics are collected through the PSUtil API 5.7.2. The proposed prototype evaluates the host OS namespace and applies a filtering process to identify and analyze the container-related processes at each monitoring interval. For evaluation purposes, we collect the metrics of the containers at regular intervals of every 5 s. A total of 37 features are collected for the *Engine Monitor* metrics, and 20 metrics for each container process by the *Namespace Monitor*, resulting in a total of 57 metrics as described in Table 2.

The Container Monitor module, as described in Section 5.2, is executed at intervals of 5 s. In this case, the module receives the collected features from the Container Monitor module through a REST web service implemented in Python 3.11. We filter the top 5 most CPU-demanding container processes for our evaluation based on CPU resource utilization (Algorithm 1, Resource for Filter). The resulting features are used by both the *Container State Classifier* and *Service State Classifier* modules (see Fig. 7). The first module employs an RNN model to identify the Node State. Conversely, the second module applies the feature reduction technique before using it for classification purposes (see Section 5.2).

We implement our *Feature Reduction* module (Fig. 7, *Feature Reduction*) through a Deep AutoEncoder architecture, executed on top of Keras API 2.4.0, and TensorFlow API 2.4.1. The *Feature Reduction* deep autoencoder is implemented with the following architecture:

- *Input*. The input is composed of 20 features for each container process. Each container is represented by the top 5 most CPU-demanding processes, resulting in 100 *Namespace* features. These features are combined with 37 *Engine* features, for a total of 137 features. The total features are combined for all 3 service-related containers, resulting in an input size of 411. This input of 411 features is adopted as input to the Deep AutoEncoder (Fig. 7, Service State).
- *Encoder*. Three dense layers were implemented with a *relu* activation function, with 512, 256, and 128 units, respectively.
- *Encoder Output*. An encoder output layer is implemented with a sigmoid activation function and 20 units.
- *Decoder*. Three layers with a *relu* activation function, with 128, 256, and 512 units respectively.
- *Decoder Output*. An output layer with 411 units and a *sigmoid* activation function.

Table 2
Extracted features by the containerized monitoring module in a 5-s time interval periodicity.

Feature set	Device	#	Extracted features	Description
Container Engine (Docker API)	CPU	1	Context Switches	Number of context switches made
		2	Idle	CPU time spent idling
		3	Interrupts	Number of interrupts made
		4	Soft Interrupts	Number of soft interrupts made
		5	I/O Wait	Number of jiffies spent waiting for I/O completion
		6	SoftIRQ	Number of software interrupts made
		7	System	Time spent by the kernel
		8	User	Time spent by the user
	Disk	9	In Flight	Number of I/O requests in flight
		10	I/O Ticks	Time active
		11	Read I/O	Number of read requests processed
		12	Read Merge	Number of merged I/O read requests with in-queue I/O
		13	Read Sectors	Number of sectors read
		14	Read Ticks	Total time spent in read requests
		15	Write I/O	Number of write requests processed
		16	Write Merge	Number of merged I/O write requests with in-queue I/O
		17	Write Sectors	Number of sectors written
		18	Write Ticks	Total time spent in write requests
		19	Time In Queue	Total wait time for all requests
	Memory	20	Active Pages	Number of active pages
		21	Active File	Number of active file cache memory
		22	Inactive Pages	Number of inactive pages
		23	Inactive	Number of inactive file cache memory
		24	Mapped	Total amount of KB mapped
		25	Page Faults	Number of page faults
		26	Major Page Faults	Number of major page faults
		27	Page In	Number of KB the system had paged from disk
		28	Page Out	Number of KB the system had paged to disk
		29	Page Reuse	Number of KB the system reused pages
		30	Page Free	Number of KB the system free memory
	Network	31	Recv. Bytes	Number of bytes received to the host
		32	Sent Bytes	Number of bytes sent from the host
		33	Recv. Packets	Number of packets received to the host
		34	Sent Packets	Number of packets sent from the host
		35	Num. Connections	Number of connections active
		36	Drop In	Number of incoming packets dropped
		37	Drop Out	Number of outgoing packets dropped
Container Namespace (PSUtil)	CPU	38	Children System	Process' children time spent in kernel
		39	Children User	Process' children time spent in the user space
		40	System	Time spent by the kernel
		41	User	Time spent by the user
	Disk	42	Read Chars	Total read chars from the disk
		43	Read Bytes	Total read bytes from the disk
		44	Read Count	Total read operations made
		45	Written Chars	Total written chars from the disk
		46	Written Chars	Total written bytes from the disk
		47	Written Chars	Total write operations made
	Memory	48	Data	Amount of memory committed to the executable code
		49	Size	Total program size
		50	Resident	Resident set size
		51	Shared	Number of resident shared pages
	Network	52	Recv. Bytes	Number of bytes received to the process
		53	Sent Bytes	Number of bytes sent from the process
		54	Recv. Packets	Number of packets received to the process
		55	Sent Packets	Number of packets sent from the process
		56	Drop In	Number of incoming packets dropped
		57	Drop Out	Number of outgoing packets dropped

The implemented Deep AutoEncoder architecture used the $rmse$ formula as $loss$ using $adam$ optimizer with 0,0001 as the learning rate. For the model building procedure, 1,000 epochs are executed with a batch size of 512. Also, the architecture considered an early stopping approach with the patience of 10 epochs, monitoring the validation $loss$.

The *Container State Classifier* and *Service State Classifier* modules (Fig. 7) rely on a RNN as implemented through a Long Short-Term Memory (LSTM). The model is implemented through the following architecture:

- *Input*. The 20 features generated by the Encoder layer are fed as an input to the LSTM (*Encoder Output* layer of the Deep AutoEncoder)
- *LSTM*. Two LSTM layers with 1,024 and 512 units respectively
- Two $dense$ layers are implemented with a $relu$ activation function, with 512, and 256 units respectively
- *Output*. A $dense$ layer is implemented with the $softmax$ activation function, with 2 units

The *Container State Classifier* module also uses the aforementioned LSTM architecture. However, it receives as input 137 feature values generated by each individual monitored container for the classification task, as it does not make use of our *Feature Reduction* module, considering that it is implemented within each deployed container (*Container State Classifier*, Fig. 7).

The implemented LSTM architecture used the *sparse categorical crossentropy* as $loss$ using $adam$ optimizer with 0,0001 as the learning rate. For the model building procedure, 1,000 epochs are executed with a batch size of 512.

7. Evaluation

This section evaluates the proposed model for the detection of QoS degradation on containers caused by multi-tenant interference by addressing four main research questions:

- (***RQ3***) *Can the proposed container state classifier effectively detect multi-tenant interferences at the container level?*
- (***RQ4***) *Can the proposed service state classifier accurately detect multi-tenant interferences at the service level?*
- (***RQ5***) *What is the impact of the multi-tenant interference ratio on the classification accuracy of our proposed technique?*
- (***RQ6***) *What is the performance of our proposed model when the multi-tenancy interference ratio is varied over time?*

The following subsections outline the construction of the used models in the evaluation and describe their performance in the testbed.

7.1. Model building

The performance of the proposed model, as depicted in Fig. 6, is evaluated using the testbed described before (see Section 4.1). In order to evaluate the performance, we executed the deployed testbed while varying the deployed Apache Spark job and the number of concurrently deployed processing load tenants (Fig. 3, *Kubernetes Tenant Pod Container*). Similarly, we collected 300 measurements for each configuration, while the implemented prototype fetch the used feature values from deployed tenants. Each testbed execution is labeled as either *normal* or *multi-tenant interference* according to the underlying physical hardware usage (see Section 4.1). Indeed, we label the events collected from a job execution as *multi-tenant interference* when its execution time surpasses the time observed at the fourth quartile in a cluster commitment ratio at 0.0 (Fig. 5, 375 and 950 s for the *RF* and *SVM* jobs respectively). Further, we consider *multi-tenant interference* when the job execution time significantly surpasses those observed in a multi-tenant-free setting.

The collected data are split into three main datasets, namely training, validation, and testing, each composed of 40%, 30%, and 30% of the execution rounds in each configuration. The training dataset is used for the model building of the *Container State*, *Feature Reduction*, and *Service State* modules (see Fig. 6). We also applied a robust feature scaling implemented by *scikit-learn* to standardize the data in a −1 to +1 range for the proposed architecture. The validation dataset is used to evaluate the model generalization during the training task. The testing dataset is used to report the final model accuracies.

Finally, the proposed model is evaluated with respect to its True-Positive (TP), True-Negative (TN), and F1 scores by considering the following classification performance metrics:

- *True-Positive* (TP): rate of *service degradation* samples correctly classified as *service degradation*.
- *True-Positive* (TN): rate of *normal* samples correctly classified as *normal*.
- *False-Positive* (FP): rate of *normal* samples incorrectly classified as *service degradation*.
- *False-Negative* (FN): rate of *service degradation* samples incorrectly classified as *normal*.

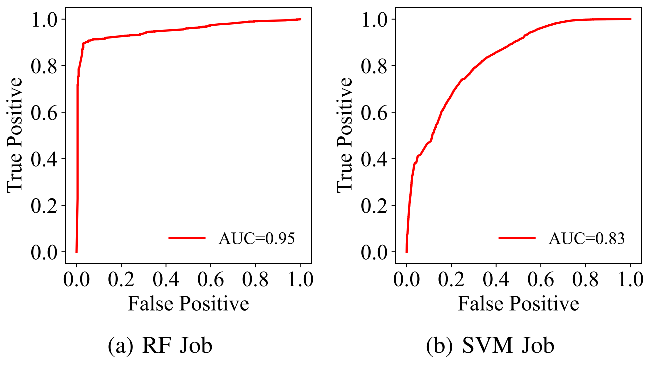


Fig. 9. ROC curve of our proposed *Container State Classifier* module for detecting QoS degradation at the container level.

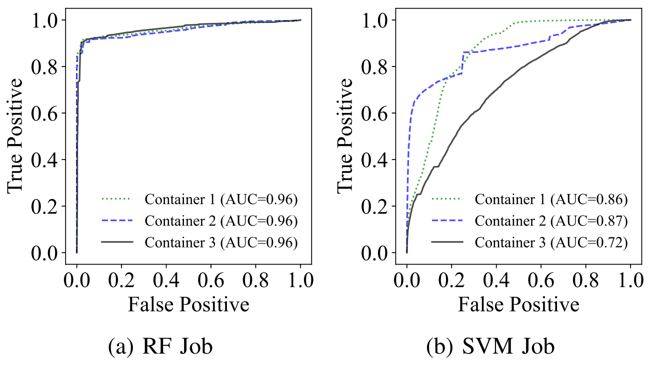


Fig. 10. ROC curve in a container granularity of our proposed *Container State Classifier* module for detecting multi-tenancy interference.

The F1 score is computed as the harmonic mean of precision and recall values while considering *multi-tenant interference* as positive samples and *normal* as negative samples, as shown in Eq. (4).

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

7.2. Detecting multi-tenant interferences

The first experiment addresses *RQ3* and evaluates the classification accuracy of our proposed *Container State Classifier* module for detecting multi-tenancy interferences. To achieve this goal, we build the *Container State Classifier* module using the previously described LSTM architecture. It is important to note that the module is applied independently to each deployed container and does not utilize the *Feature Reduction* technique (see Fig. 7).

Fig. 9 shows the Receiver Operating Characteristic (ROC) curve for detecting multi-tenancy interferences for each executed Apache Spark job and monitored container in our evaluation. The results indicate that our proposed model achieved high detection accuracies across different Apache Spark jobs and monitored containers. Our proposed scheme demonstrated high performance in detecting multi-tenancy interferences for both the RF and SVM jobs. Specifically, the Area Under the Curve (AUC) value for the RF job was measured to be 0.95, while the average AUC for the SVM job was 0.83, indicating a noticeable difference of 0.12 between the two. As observed in the previous evaluation (see Section 4.2), the variations in accuracy can be attributed to the distinct characteristics and behaviors of the evaluated jobs. Each

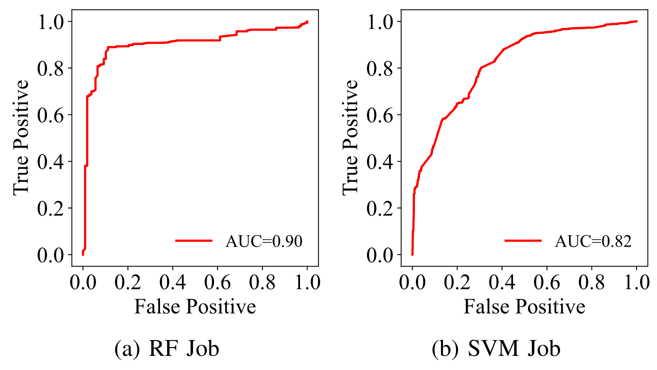


Fig. 11. ROC curve of our proposed *Service State Classifier* module for detecting multi-tenancy interference at the service level.

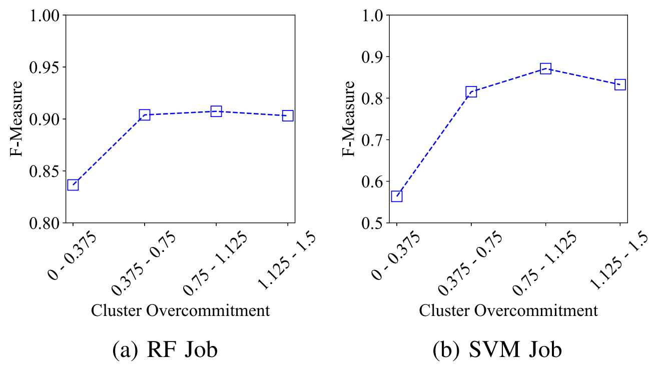


Fig. 12. Proposed model detection performance of cluster performance issues according to degradation level.

Table 3
Proposed model accuracies according to each evaluated job on a service-related granularity.

Apache Spark	Evaluation metric			
job	*TP*	*TN*	*AUC*	*F-Measure*
RF	0.84	0.89	0.90	0.90
SVM	0.87	0.59	0.82	0.78

job has different resource requirements and workload patterns, leading to variations in performance degradation and, consequently, different detection accuracies. Fig. 10 further investigates the AUC values for each container job. The measured AUC values for each container vary based on the deployed job, with the RF job achieving an average AUC value of 0.96, while the SVM job ranges from 0.72 to 0.87 (Fig. 10(a) *vs*.10(b)).

Our proposed model achieves high accuracy performance across different containers, as evidenced by the consistently high AUC scores obtained for each evaluated job. Further, investigating the generated ROC curve shows that our model can improve detection accuracy when a higher False-Positive (FP) rate is tolerated. For instance, one can provide a 90% of TP rate for the RF job if a 5% of FP is tolerated (Fig. 9(a)). Indeed, adjusting the trade-off between detection sensitivity and FP rate is beneficial for service providers, especially in performance-critical scenarios. By setting a higher FP rate, the provider can detect potential multi-tenancy interferences more effectively, ensuring timely actions to maintain service quality. Such a threshold can be used in critical services, wherein performance degradation cannot be tolerated. The proposed model effectively detects container-level multi-tenancy interferences, providing indications for countermeasures, such as migrating the client container to a different host, to avoid QoS degradation.

The second experiment aims to answer *RQ4* and evaluates our proposed *Service State Classifier* module for detecting service-level performance degradation due to multi-tenant interferences. To achieve this goal, we evaluate the detection accuracy of the *Service State Classifier* module, when implemented using an LSTM model. Thus, it utilizes the data from the *Feature Reduction* module, implemented using a Deep AutoEncoder. Recalling that the model input is generated by the three deployed containers to detect service-level performance degradation.

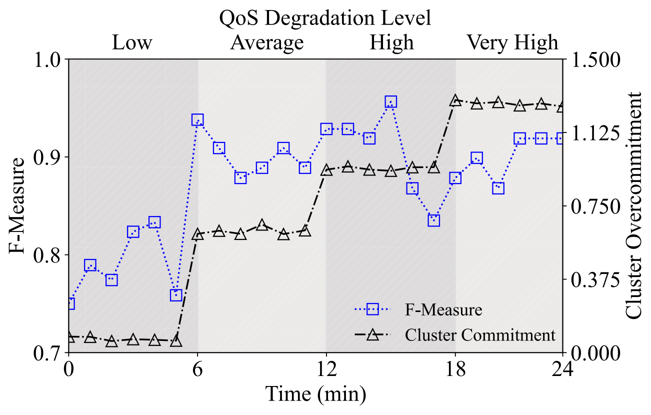
Fig. 11 shows the ROC curve for our *Service State Classifier* module. Our proposed scheme provided similar accuracies when compared to the container-level counterpart (Fig. 9 vs. Fig. 11). Our model achieved AUC values of 0.90 and 0.82 for the RF and SVM jobs, respectively, indicating its effectiveness in detecting service-level performance degradation. The *Feature Reduction* module selects the most meaningful metrics from the distributed containers, ensuring that only relevant data is used for the classification task.

Table 3 summarizes the obtained accuracy for the evaluated Apache Spark jobs, demonstrating the high detection accuracies achieved by our proposed model at the service-level scenarios. Further investigation of the ROC curve (Fig. 11) reveals that our model can achieve a high TP rate of up to 80% while maintaining low FP rates of only 7% and 23% for the RF and SVM jobs, respectively. This demonstrates the effectiveness of our model in detecting service-level degradation due to multi-tenant interferences.

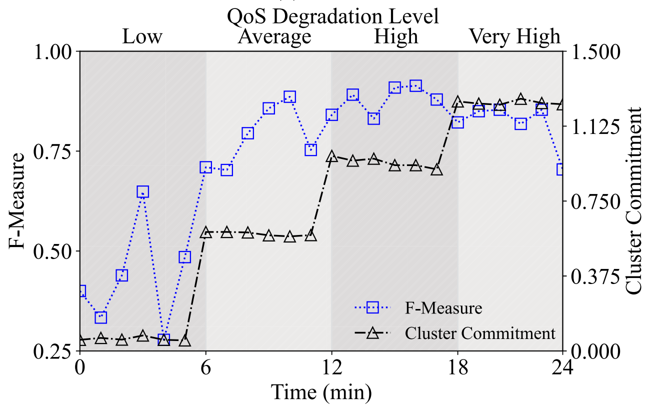
To answer *RQ5*, we further investigate the relationship between the cluster resource commitment ratio and the accuracy of our model in detecting QoS degradation under different levels of interference. Fig. 12 shows the F-Measure obtained by our *Service State Classifier* module according to the cluster commitment intervals. In summary, we observe that as the level of cluster commitment increases, leading to more severe QoS degradation, the accuracy of our proposal also improves. However, the classification accuracies are noticeably affected when the multi-tenant interference ratio is close to the chosen operation point. For instance, the F-Measure increases from 0.57 to 0.83 when there is a cluster commitment ranging from 0.375 to 0.75 during the SVM job execution (Fig. 12(b)). It is important to note that the choice of the operation point may vary depending on the specific characteristics of the deployed containerized service and its sensitivity to multi-tenant interferences from the service provider. As a consequence, our proposed model demonstrates increased accuracy as the service QoS experiences more severe degradation.

To answer *RQ6*, we conducted an experimental campaign to evaluate the performance of our proposed model under a realistic setting, where the multi-tenancy interference level changes over time during the execution of the client containerized service. This dynamic allocation and reallocation of tenants over the same physical hardware, as time passes, is a common scenario in real-world deployments. To assess the performance of our proposed model under varying multi-tenancy interference levels, we conducted an experiment in which we executed our testbed for a duration of 24 minutes. During this time, we varied the number of concurrently executed workloads (Table 1), simulating different levels of multi-tenancy interference.

Fig. 13 depicts the classification accuracy of our *Service State Classifier* module over time, with measurements taken every minute. The results prove that our proposed scheme achieves high classification accuracies when deployed in a realistic setting that simulates real-time applications. Indeed, we obtained an average F1-Score of 0.88 for the RF job and 0.72 for the SVM job, indicating its effectiveness in detecting multi-tenancy interferences within the service provider domain. The classification accuracy is high, independent of the interference level (high or low). This latter reinforces the capability of our model to adapt to different interference scenarios and accurately detect performance degradation in containerized services.



(a) RF Job



(b) SVM Job

Fig. 13. Proposed model detection performance of service performance issues while service provider varies the degradation level. Accuracy is measured in one-minute time intervals.

8. Conclusion

Container-based deployments have gained popularity in recent years due to their advantages in terms of fast provisioning and low-performance overhead. However, container-based services often do not consider the multi-tenancy aspects that have been recognized as impacting the performance of traditional VM-based services. In our contribution, we conducted experiments demonstrating that containerized services experience significant performance degradation when deployed in multi-tenant environments, significantly impacting the QoS of deployed services. Further, we focused on tackling the detection of multi-tenancy interferences in containerized distributed services within the service provider domain, enabling service providers to effectively utilize multi-tenancy while ensuring that it does not compromise the QoS experienced by their clients. Our proposed scheme leverages the metrics the container engine provides and the container-associated namespace. By analyzing these metrics, our scheme enables the detection of both container-level and service-level performance degradation. This comprehensive approach allows service providers to identify and address multi-tenancy interferences at different levels. Consequently, when performance degradation is detected, appropriate countermeasures can be taken, such as migrating affected containers to less overloaded service provider nodes.

CRediT authorship contribution statement

Pedro Horchulhack: Data curation, Investigation, Methodology, Software, Writing – original draft. **Eduardo K. Viegas:** Investigation, Methodology, Project administration, Resources, Supervision, Writing – original draft, Writing – review & editing. **Altair O. Santin:** Supervision, Writing – review & editing. **Felipe V. Ramos:** Conceptualization, Data curation, Software. **Pietro Tedeschi:** Methodology, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Ahmed, N., Barczak, A.L.C., Susnjak, T., Rashid, M.A., 2020. A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench. J. Big Data 7 (1), http://dx.doi.org/10.1186/s40537-020-00388-5.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P., 2018. Elasticity in cloud computing: State of the art and research challenges. IEEE Trans. Serv. Comput. 11 (2), 430–447.

Ali, Q., Dunn, D., Garg, R., Lu, X., Muirhead, T., Taheri, R., Zubb, J., 2019. Performance of vSphere 6.7 Scheduling Options. White Paper, VMWare Inc., [Online]. Available: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/performance/scheduler-options-vsphere67u2-perf.pdf.

Alshuqayran, N., Ali, N., Evans, R., 2016. A systematic mapping study in microservice architecture. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications. SOCA, IEEE, http://dx.doi.org/10.1109/soca.2016.15.

Arunarani, A., Manjula, D., Sugumaran, V., 2019. Task scheduling techniques in cloud computing: A literature survey. Future Gener. Comput. Syst. 91, 407–415.

Bélair, M., Laniepce, S., Menaud, J.-M., 2019. Leveraging kernel security mechanisms to improve container security. In: Proceedings of the 14th International Conference on Availability, Reliability and Security. ACM, http://dx.doi.org/10.1145/3339252.3340502.

Cañete, A., Amor, M., Fuentes, L., 2024. HADES: An NFV solution for energy-efficient placement and resource allocation in heterogeneous infrastructures. J. Netw. Comput. Appl. 221, 103764. http://dx.doi.org/10.1016/j.jnca.2023.103764.

Casalicchio, E., Perciballi, V., 2017. Measuring docker performance. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. ACM, http://dx.doi.org/10.1145/3053600.3053605.

Ceesay, S., Barker, A., Varghese, B., 2017. Plug and play bench: Simplifying big data benchmarking using containers. In: 2017 IEEE International Conference on Big Data. Big Data, IEEE, http://dx.doi.org/10.1109/bigdata.2017.8258249.

Chen, X., Ji, J., Luo, C., Liao, W., Li, P., 2018. When machine learning meets blockchain: A decentralized, privacy-preserving and secure design. In: 2018 IEEE International Conference on Big Data. Big Data, IEEE, http://dx.doi.org/10.1109/bigdata.2018.8622598.

Cinque, M., Corte, R.D., Pecchia, A., 2022. Micro2vec: Anomaly detection in microservices systems by mining numeric representations of computer logs. J. Netw. Comput. Appl. 208, 103515. http://dx.doi.org/10.1016/j.jnca.2022.103515.

El-Kassabi, H.T., Adel Serhani, M., Dssouli, R., Navaz, A.N., 2019. Trust enforcement through self-adapting cloud workflow orchestration. Future Gener. Comput. Syst. 97, 462–481, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18313529.

Fayos-Jordan, R., Felici-Castell, S., Segura-Garcia, J., Lopez-Ballester, J., Cobos, M., 2020. Performance comparison of container orchestration platforms with low cost devices in the fog, assisting internet of things applications. J. Netw. Comput. Appl. 169, 102788. http://dx.doi.org/10.1016/j.jnca.2020.102788.

Felter, W., Ferreira, A., Rajamony, R., Rubio, J., 2015. An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS, pp. 171–172.

Grohmann, J., Nicholson, P.K., Iglesias, J.O., Kounev, S., Lugones, D., 2019. Monitorless: Predicting performance degradation in cloud applications with machine learning. In: Proceedings of the 20th International Middleware Conference. pp. 149–162.

Harush, S., Meidan, Y., Shabtai, A., 2021. DeepStream: Autoencoder-based stream temporal clustering. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. SAC '21, Association for Computing Machinery, New York, NY, USA, pp. 445–448, [Online]. Available: https://doi-org.ez433.periodicos.capes.gov.br/10.1145/3412841.3442083.

Henkel, J., Bird, C., Lahiri, S.K., Reps, T., 2020. Learning from, understanding, and supporting DevOps artifacts for docker. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ACM.

Huang, S., Huang, J., Dai, J., Xie, T., Huang, B., 2022. Hibench suite - the big data micro benchmark suite. [Online]. Available: https://github.com/Intel-bigdata/HiBench.

Joy, A.M., 2015. Performance comparison between linux containers and virtual machines. In: 2015 International Conference on Advances in Computer Engineering and Applications. pp. 342–346.

Kang, H., Le, M., Tao, S., 2016. Container and microservice driven design for cloud infrastructure DevOps. In: 2016 IEEE International Conference on Cloud Engineering. IC2E, IEEE, http://dx.doi.org/10.1109/ic2e.2016.26.

Kontodimas, K., Kokkinos, P., Kuperman, Y., Varvarigos, E., 2015. Analysis and evaluation of I/O hypervisor scheduling. In: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing. UCC, IEEE, http://dx.doi.org/10.1109/ucc.2015.19.

Kozhirbayev, Z., Sinnott, R.O., 2017. A performance comparison of container-based technologies for the cloud. Future Gener. Comput. Syst. 68, 175–182. http://dx.doi.org/10.1016/j.future.2016.08.025.

Li, X., Pan, L., Liu, S., 2023. An online service provisioning strategy for container-based cloud brokers. J. Netw. Comput. Appl. 214, 103618. http://dx.doi.org/10.1016/j.jnca.2023.103618.

Masouros, D., Xydis, S., Soudris, D., 2021. Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems. IEEE Trans. Parallel Distrib. Syst. 32 (1), 184–198, [Online]. Available: https://ieeexplore.ieee.org/document/9158547/.

Mavridis, I., Karatza, H., 2019. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. Future Gener. Comput. Syst. 94, 674–696, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18305764.

Mi, H., Wang, H., Yin, G., Cai, H., Zhou, Q., Sun, T., Zhou, Y., 2011. Magnifier: Online detection of performance problems in large-scale cloud computing systems. In: 2011 IEEE International Conference on Services Computing. pp. 418–425.

Morabito, R., Kjallman, J., Komu, M., 2015. Hypervisors vs. Lightweight virtualization: A performance comparison. In: 2015 IEEE International Conference on Cloud Engineering. IEEE, http://dx.doi.org/10.1109/ic2e.2015.74.

Parast, F.K., Sindhav, C., Nikam, S., Yekta, H.I., Kent, K.B., Hakak, S., 2022. Cloud computing security: A survey of service-based models. Comput. Secur. 114, 102580. http://dx.doi.org/10.1016/j.cose.2021.102580.

Ruan, B., Huang, H., Wu, S., Jin, H., 2016. A performance study of containers in cloud environment. In: Wang, G., Han, Y., Martínez Pérez, G. (Eds.), Advances in Services Computing. Springer International Publishing, Cham, pp. 343–356.

Shea, R., Wang, F., Wang, H., Liu, J., 2014. A deep investigation into network performance in virtual machine based cloud environments. In: IEEE INFOCOM 2014 - IEEE Conference on Computer Communications. pp. 1285–1293.

Shen, H., Chen, L., 2022. A resource-efficient predictive resource provisioning system in cloud systems. IEEE Trans. Parallel Distrib. Syst. 33 (12), 3886–3900. http://dx.doi.org/10.1109/tpds.2022.3172493.

Spooren, J., Preuveneers, D., Desmet, L., Janssen, P., Joosen, W., 2019. Detection of algorithmically generated domain names used by botnets: A dual arms race. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. SAC '19, Association for Computing Machinery, New York, NY, USA, pp. 1916–1923, [Online]. Available: https://doi-org.ez433.periodicos.capes.gov.br/10.1145/3297280.3297467.

Sultan, S., Ahmad, I., Dimitriou, T., 2019. Container security: Issues, challenges, and the road ahead. IEEE Access 7, 52976–52996.

Tang, W., Ke, Y., Fu, S., Jiang, H., Wu, J., Peng, Q., Gao, F., 2022. Demeter. In: Proceedings of the 13th Symposium on Cloud Computing. ACM, http://dx.doi.org/10.1145/3542929.3563476.

Truyen, E., Landuyt, D.V., Reniers, V., Rafique, A., Lagaisse, B., Joosen, W., 2016. 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.

Vayghan, L.A., Saied, M.A., Toeroe, M., Khendek, F., 2018. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In: 2018 IEEE 11th International Conference on Cloud Computing. CLOUD, IEEE, http://dx.doi.org/10.1109/cloud.2018.00148.

Vicentini, C., Santin, A., Viegas, E., Abreu, V., 2018. 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, pp. 543–552.

Wang, W., Tian, N., Huang, S., He, S., Srivastava, A., Soffa, M.L., Pollock, L., 2018a. Testing cloud applications under cloud-uncertainty performance effects. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 81–92.

Wang, T., Xu, J., Zhang, W., Gu, Z., Zhong, H., 2018b. Self-adaptive cloud monitoring with online anomaly detection. Future Gener. Comput. Syst. 80, 89–101, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X1730376X.

Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C.A.F., 2013. Performance evaluation of container-based virtualization for high performance computing environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 233–240.

Yang, Y., Jiang, H., Zhang, G., Wang, X., Lv, Y., Li, X., Fdida, S., Xie, G., 2021. S2H: Hypervisor as a setter within virtualized network I/O for VM isolation on cloud platform. Comput. Netw. 201, 108577. http://dx.doi.org/10.1016/j.comnet.2021.108577.

Yao, M., Chen, D., Shang, J., 2019. Optimal overbooking policy for cloud service providers: Profit and service quality. IEEE Access 7, 96132–96147.

Zeng, Q., Kavousi, M., Luo, Y., Jin, L., Chen, Y., 2023. Full-stack vulnerability analysis of the cloud-native platform. Comput. Secur. 129, 103173, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404823000834.

Zhang, X., Li, L., Wang, Y., Chen, E., Shou, L., 2021. Zeus: Improving resource efficiency via workload colocation for massive Kubernetes clusters. IEEE Access 9, 105192–105204. http://dx.doi.org/10.1109/access.2021.3100082.

Zhang, R., Zhou, R., 2023. An efficient online auction for placing and pricing cloud container clusters. IEEE Trans. Netw. Sci. Eng. 1–12.

Zhong, Z., Buyya, R., 2020. A cost-efficient container orchestration strategy in Kubernetes-based cloud computing infrastructures with heterogeneous resources. ACM Trans. Internet Technol. 20 (2), 1–24.

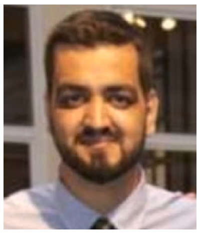
Zhong, A., Jin, H., Wu, S., Shi, X., Gen, W., 2012. Optimizing xen hypervisor by using lock-aware scheduling. In: 2012 Second International Conference on Cloud and Green Computing. IEEE, http://dx.doi.org/10.1109/cgc.2012.115.

Zhou, Z., Tam, V., Lui, K., Lam, E., Yuen, A., Hu, X., Law, N., 2019. Applying deep learning and wearable devices for educational data analytics. In: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence. ICTAI, IEEE, http://dx.doi.org/10.1109/ictai.2019.00124.

Zhu, D., Jin, H., Yang, Y., Wu, D., Chen, W., 2017. DeepFlow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data. In: 2017 IEEE Symposium on Computers and Communications. ISCC, IEEE, http://dx.doi.org/10.1109/iscc.2017.8024568.

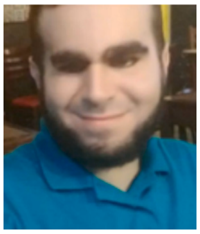
Zolfaghari, R., Sahafi, A., Rahmani, A.M., Rezaei, R., 2021. Application of virtual machine consolidation in cloud computing systems. Sustain. Comput.: Inform. Syst. 30, 100524. http://dx.doi.org/10.1016/j.suscom.2021.100524.

Pedro Horchulhack earned his BS degree in Computer Science in 2022 and his MSC degree in Computer Science in 2023, both from PUCPR. Currently, he is actively pursuing his Ph.D. degree in Computer Science at PUCPR, with research interests encompassing machine learning, distributed systems, big data, and computer security.



Eduardo K. Viegas received the BS degree in computer science in 2013, the MSC degree in computer science in 2016 from PUCPR, and the Ph.D. degree from PUCPR in 2018. He is an associate professor of Graduate Program in Computer Science (PPGIa). His research interests include machine learning, network analytics and computer security.

Altair O. Santin received the BS degree in Computer Engineering from the PUCPR in 1992, the M.Sc. degree from UTFPR in 1996, and the Ph.D. degree from UFSC in 2004. He is a full professor of Graduate Program in Computer Science (PPGIa) and head of Security & Privacy Lab (SecPLab) at PUCPR. He is a member of the IEEE, ACM, and the Brazilian Computer Society.



Felipe Veiga Ramos is currently working towards his MsC degree in computer science at PUCPR. His research interests include machine learning, distributed systems, and computer security.

Pietro Tedeschi is currently Senior Security Researcher at Technology Innovation Institute, Autonomous Robotics Research Center, Abu Dhabi, United Arab Emirates, from January 2022. He obtained his Ph.D. in Computer Science and Engineering from Hamad Bin Khalifa University (HBKU), College of Science and Engineering (CSE), Doha, Qatar in December 2021. He received his Master's degree with honors in Computer Engineering at Politecnico di Bari, Italy. From 2017 to 2018, he worked as Security Researcher at CNIT (Consorzio Nazionale Interuniversitario per le Telecomunicazioni), Italy, for the EU H2020 SymbIoTe project. He is also serving in the TPC of several conferences. His major research interests include security and privacy in Unmanned Aerial Vehicles, Wireless, Internet of Things, Cyber-physical Systems, and Applied Cryptography.