IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 2 August 2023, accepted 30 August 2023, date of publication 5 September 2023, date of current version 12 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3312218

APPLIED RESEARCH

A 2-Phase Strategy for Intelligent Cloud Operations

GIACOMO LANCIANO[1,2], (Member, IEEE), REMO ANDREOLI[2], (Member, IEEE), TOMMASO CUCINOTTA[2], (Senior Member, IEEE), DAVIDE BACCIU[3], AND ANDREA PASSARELLA[4]

[1]Faculty of Sciences, Scuola Normale Superiore, 56126 Pisa, Italy
[2]Real-Time Systems Laboratory (RETIS), TECIP Institute, Scuola Superiore Sant'Anna, 56124 Pisa, Italy
[3]Department of Computer Science, University of Pisa, 56127 Pisa, Italy
[4]Institute of Informatics and Telematics, National Research Council of Italy, 56124 Pisa, Italy

Corresponding author: Tommaso Cucinotta (tommaso.cucinotta@santannapisa.it)

This work was supported in part by the European Union's Horizon 2020 Research and Innovation Programme under Grant 957337 and Project MARVEL.

ABSTRACT When operating large cloud computing infrastructures, ensuring healthiness of physical resources and software components is of paramount importance to meet the demanding service levels expected by customers. This is only possible using automations that can detect anomalies and alert the on-call personnel, or trigger healing procedures. In production-grade deployments, such automations are generally based on static thresholds or predefined pattern-matching rules, checked against relevant metrics and logs. Defining and maintaining them is cumbersome and, as the infrastructure grows, they need continuous adjustments. To tackle this problem, we propose an *intelligent* automation system for cloud operations that learns, from what operators have done in the past, what actions should be applied in response to the observed anomalies. Such system is designed to operate elastic groups of cloud instances realizing typical (replicated) cloud services. The mechanism is based on a 2-phase machine learning pipeline, composed of: a first, lighter, model that automatically detects anomalous patterns, based on past observations of the *normal* behavior, causing activation of the second, more involved, model; this is a model that recommends specific corrective actions, based on historical operational data reporting the actions applied to heal the faulty components. The approach was validated on an OpenStack deployment, where we deployed both a synthetic application and a multi-node Cassandra NoSQL data-store, and injected different types of anomalies while these systems were exercised using synthetic workloads. For both applications, we obtained a remarkable accuracy (mostly beyond 90%, and also going beyond 95% in some cases), for the anomaly detection and corrective action recommendation tasks, by applying the models on the respective test sets. This allows us to conclude that the presented mechanism constitutes an efficient and effective technique to help operating cloud services in presence of a number of faults, albeit the types and heterogeneity of faulty conditions might be expanded in future evolutions of the framework. The implementation and the material needed to reproduce our results are available under an open-source license.

INDEX TERMS Cloud operations, fault management, machine learning, monitoring, OpenStack.

I. INTRODUCTION

Cloud computing has become an essential technology in the modern distributed computing landscape [1]. Many diverse application domains [2], [3] leverage on services deployed in either public or private clouds, like smart cities, industrial factories, healthcare, e-Commerce, or even telecommunications, with the increasing adoption of Network Function Virtualization (NFV) [4]. Cloud infrastructures and services have rapidly evolved [5] from the initial

The associate editor coordinating the review of this manuscript and approving it for publication was Somchart Fugkeaw.

infrastructure-as-a-service (IaaS) provisioning model to the platform-as-a-service (PaaS) one, that is the most widespread nowadays. PaaS enables development and deployment of modern cloud-native applications [6], deeply integrated with a plethora of APIs and services, such as: reliable relational and NoSQL databases, advanced and secure networking, load-balancing (LB) and auto-scaling, serverless computing, integrated machine learning (ML) frameworks for training and operating large models, etc. Correspondingly, cloud infrastructures have significantly grown in size and complexity, having to deal with an ever-growing software stack on top of which such a wide variety of services can be made available. Guaranteeing high reliability and availability is only possible thanks to effective operations teams, that work 24/7 to keep such systems up, running and responsive.

The problem of operating big and complex distributed infrastructures is far from trivial. Industrial practices rely on monitoring metrics collected from physical and virtual elements of the infrastructure, e.g.: physical hosts, virtual machines (VMs), containers, networking appliances, and others. Metrics are persisted such that they can be visually, or analytically [7], inspected by operators. Also, they are typically automatically checked against a number of predefined, usually threshold-based, rules that possibly identify problems and trigger appropriate corrective actions. A classical example is a LB with self-healing capabilities, that adds new instances to an elastic compute group whenever the number of healthy instances goes below a configured amount. Other mechanisms are instead based on predefined pattern-matching rules to be checked against logs [8]. In a large-scale cloud operations scenario, there are thousands of active automation rules. However, as new scenarios occur, such rules need continuous adjustments to keep on being effective. Also, in response to an issue being identified, an operator typically starts a (non-trivial) root-cause analysis [9], [10], to understand what caused it, and ultimately what is the right fix. One of the greatest challenge for cloud providers is to *sustainably* deal with the ever-increasing size of the physical infrastructure. Ideally, without having to correspondingly increase the number of operators that continuously watch dashboards, and troubleshoot and fix infrastructure problems. In other words, they should aim at that ''rapid provisioning with nearly zero human interaction'', originally predicated by NIST [11].

Therefore, cloud providers are increasingly investing in developing intelligent techniques to support humans operators in their tasks, such as anomaly detection (AD) [12], resource allocation and capacity planning. Given the abundance of operational data, it is natural to seek for data-driven approaches like ML, that can augment the capabilities of operators to ''navigate'' the zillions of available time-series and logs. For instance, considering the AD problem, many works in the literature leverage on either supervised [13], [14] or unsupervised [15], [16], [17] ML algorithms to detect early symptoms of anomalous conditions at different levels of a cloud infrastructure. Similarly, many recent works [18], [19], [20] propose time-series forecasting techniques to anticipate the evolution of workloads and scale compute resources (e.g., VMs or containers) accordingly. However, effectively using such approaches in production is not straightforward. Indeed, there are many characteristics of the monitored system to take into account, like the topology of the physical infrastructure [21], the design patterns used to realize the individual applications [22], [23], or the in-place QoS requirements [24].

A. CONTRIBUTIONS

In this paper, we tackle the problem of automated operations for elastically deployed cloud services, proposing a strategy for *intelligent* anomaly detection and correction suggestion that consists of two phases: *(i)* detecting anomalous operational conditions of an application made of an elastic group of cloud instances; *(ii)* identifying the faulty component within the group, and proposing the best corrective action to restore it. Both phases rely on ML models to *learn* from the appropriate operational data to detect early symptoms of anomalous conditions and to identify the proper corrective actions to apply, without explicitly coding static rules. This work aims at closing the cloud operations loop in a totally automated fashion, envisioning a system with the ability to learn from the corrective actions applied by operators in similar previous cases. We validated the proposed approach by deploying a synthetic application and a Cassandra NoSQL cluster on an OpenStack testbed. We trained and tested the ML models on system-level monitoring data gathered while injecting different types of anomalies on the mentioned applications, including exogenous workload interferences, sudden failure of a cluster member, and saturation of CPU capacity and disk I/O bandwidth. For *(i)*, we trained an AD model such that it could generalize to variable-sized groups of instances. On the respective test sets, for the synthetic application, we obtained a ROC-AUC of 97% and an accuracy of 90.34%, while, for Cassandra, we obtained a ROC-AUC of 94% and an accuracy of 87.50%. For *(ii)*, we trained a supervised multi-label classification model, such that it could associate corrective actions to instances individually. On the respective test sets, for the synthetic application, we obtained an accuracy of 96.15%, while, for Cassandra, we obtained an accuracy of 98.75%. See Section IV for more details on the performed experimentation. The implementation of our approach and the material needed to reproduce our results are available under an open-source license.

B. PAPER OVERVIEW

After a brief recall of related research in Section II, in Section III we present our architecture for intelligent cloud operations. Our approach was prototyped and validated on an OpenStack test-bed, as shown by the results reported in Section IV. Our results allow for drawing a few conclusions, reported in Section V, alongside possible ideas for future research on the topic.

II. RELATED WORK

Recently, ML-based approaches have been increasingly proposed as effective solutions to a diverse set of resource management tasks [25] for both public and private cloud infrastructures. In this section, we provide an overview of related research works on the topic.

In [19], the authors propose a time-series forecasting framework that enhances the monitoring capability of the Monasca service for OpenStack. Such framework enables proactive operations approaches, like defining predictive auto-scaling policies that are able to anticipate load peaks. Similarly, in [20], the authors propose an analogous proactive auto-scaling approach tailored for edge computing applications on Kubernetes. In addition, they also provide an automatic model retrain mechanism to counteract concept drift. In [23], the authors propose the RScale framework, based on Gaussian Process (GP) regression, to predict the tail-latency of distributed DAG-alike topologies of microservices. In [21], the authors propose a time-series forecasting approach that boosts its accuracy by also incorporating *topology* information, leveraging on graph neural networks (GNNs). In [18], the authors describe a simple predictive scaling strategy that exploits the estimation of a percentile of the resource demand, and a probabilistic cost function for over-/under-provisioning the cluster. Remarkably, the authors evaluate their technique on data coming from ∼40K real AWS auto-scaling groups. In [22], the authors use supervised learning methods to implement proactive auto-scaling policies for multi-tier elastic applications, taking into account unstable performance of individual components. Specifically, they use linear regression to predict the short-term request arrival rates and the evolution of the response times. The predictions are then mapped to the appropriate elasticity configurations with a Random Forest (RF). In [26], the authors propose a neural network-based model to predict the resources utilization and execution time of continuous integration tasks by analyzing open data from a Travis system. In [24], the authors propose a successful Reinforcement Learning-based approach to service-chains deployment in NFV, that jointly minimizes operation costs and maximizes requests throughput, also taking into account different QoS requirements.

In [14], the authors evaluate several supervised learning approaches for Anomaly Detection (AD) by injecting faults in a Kubernetes cluster. Similarly, in [13], the authors also evaluate supervised learning techniques for off-line AD in an NFV environment. The authors train their models on host monitoring data collected while injecting anomalies in a test-based running the ClearWater IMS system on top of OpenStack. Also, the authors of [12] provide a thorough survey where they discuss the risks, in terms of anomalous behaviors, correlated to switching to a NFV/cloud model. For instance, incurring in temporal interferences generated by virtualization and resource over-commitment. In [15], the authors propose a real-time unsupervised AD technique based on Hierarchical Temporal Memory (HTM). In [16], the authors propose a mechanism based on Self-Organizing Maps (SOMs) to address AD in the context of NFV data centers. They use a multi-variate clustering method to group similar daily patterns of VMs in one or more service components, such that group changes are regarded as a possible anomaly. In [10], the authors describe a root-cause analysis (RCA) approach for NFV anomalies, based on a digital twin. They frame the problem as a dynamic set-covering, and propose a scalable solution based on hidden Markov models. In [27], a variational autoencoder based on RNNs is proposed for AD in cloud scenarios, where the autoencoder trained on normal/healthy conditions, is expected to produce larger errors under anomalous/unhealthy conditions. This is followed by a one-dimensional CNN classifier used to identify the anomaly as being either a case of process death, CPU stress, network delay or packet loss.

Compared to the above research, the work proposed in the present paper tries to bridge the gap between detecting a possible issue within a cloud system or component, identifying the exact affected element within the infrastructure, and deciding what corrective action to apply in order to return the system to a normal behavior. This work aims at closing this loop in a totally automated fashion, and with the ability to learn from the corrective actions applied by humans in similar previous cases. Most ML-based approaches focus on specific operations aspects, like auto-scaling. Instead, our scope includes a wider range of operations problems. Unlike most related works, we framed the problem of deciding a corrective action as a multi-label classification task. Typically, operations teams cater collections of procedures known to be effective at recovering their systems from (recurrent) error conditions. Also, when responding to an issue, the same teams are required to log their actions, in a ticketing system. Such information can be correlated with system-/app-level data, to learn *"intelligent"* operations models. For instance, the approach described in [27] brings an interesting resemblance with our approach, in that both include an unsupervised layer for AD, followed by an anomaly classifier. However, the previous work analyzes metrics from a single instance at a time only, and it does not consider the common case of horizontally-scalable elastic clusters. Furthermore, in our work we aim at letting the system learn what corrective action to apply to the anomaly being analyzed, imitating what was made with prior manual interventions.

III. PROPOSED APPROACH

In this section, we present an overview of the proposed architecture, discussing some important implementation details.

A. GENERAL ARCHITECTURE

Traditional approaches to cloud operations ensure healthiness of applications through (often complex) automations that are, hopefully, able to detect possible abnormal conditions, send appropriate alerts and possibly trigger recovery actions.

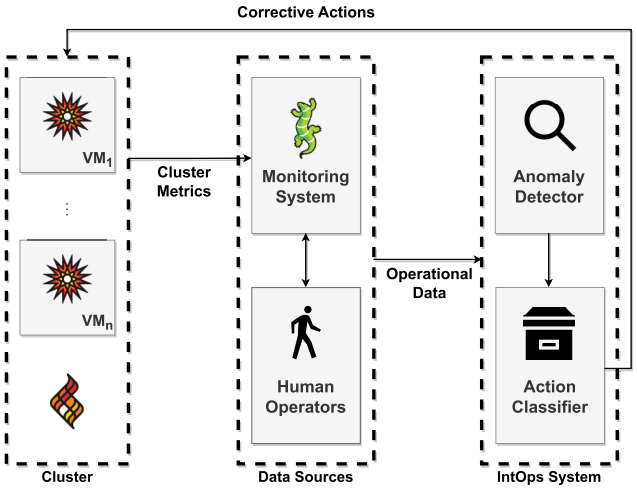


FIGURE 1. Architectural diagram of the proposed intelligent operations approach.

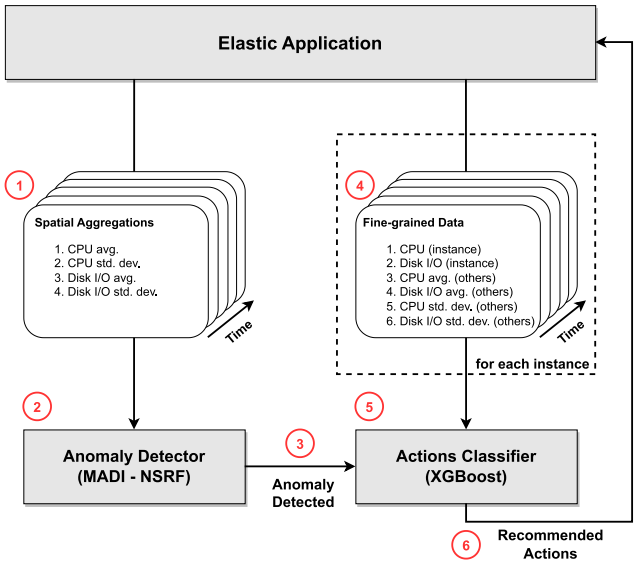


FIGURE 2. Implementation details of the proposed intelligent operations approach.

However, such mechanisms are still typically based on static rules and thresholds, that are often very easy to interpret, but quickly become cumbersome to maintain as the scale of the system grows. Using ML to solve the kind of problems mentioned above is strongly supported by the abundance of (very diverse) operational data that are produced in cloud environments. Either the infrastructure components themselves, or the on-call personnel that work around the clock to make sure that everything runs smoothly, continuously generate useful information. Such information can be leveraged upon to devise intelligent automations, that adapt as they observe more diverse operational conditions.

Figure 1 shows how our approach enhances the control loop of a cloud infrastructure, by: ingesting the operational data coming from two important data sources, i.e., the monitoring system providing metrics related to the monitored instances, and the information on actions applied by human operators to the system; analyzing these data using our proposed Intelligent Operations method (IntOps, in the figure), capable of identifying anomalies, and outputting recommendations of the action needed to restore a healthy service. In such settings, it is possible, e.g., to anticipate the occurrence of system outages, by analyzing the historical data describing the relevant system- and application-level metrics during past outages. Or, e.g., to identify their likely root-cause, and the most effective corrective actions to be applied, by looking for similarities in logs and reports associated with past incidents. At the moment, our approach allows for: detecting performance degradation due to workload co-located on the same physical hosts, recommending a relocation on healthier host; detecting faulty members of load-balanced groups of instances, that stop taking their share of the load, recommending to reboot the offending instance; detecting the shortage of allocated resources due to dynamic workload changes and its expected evolution in the short-term, recommending an elastic scaling action, to prevent serious performance degradation.

A number of other anomalous scenarios are planned to be integrated into the framework, including transient failures and hardware degradation (i.e., not entire failures, but faults impairing seriously the performance of disks, memory modules, network interface cards, etc.). Remarkably, when the set of observed anomalous scenarios to consider grows, ML-based approaches like ours scale significantly better than traditional *static* rules and thresholds. Indeed, such approaches require continuous manual tuning to capture new, unforeseen, anomalous behavior, and possibly to develop a separate criterion for each possible case. Instead, for ML models, it is often sufficient to add the new observed behavior to the training set, and restart the training procedure, without explicitly coding new rules. Furthermore, provided that the resulting ML model exhibits a satisfactory generalization power, re-training might not even be necessary.

B. IMPLEMENTATION DETAILS

To demonstrate the effectiveness of our approach, we implemented it to work with data exported from OpenStack, one of the most used open-source cloud orchestration frameworks [28]. For simplicity, in our validation (see Section IV) we only considered the CPU and disk I/O activity measurements generated by the runs of our test application. However, our approach can handle a variable number of system- or application-level metrics. Then, we manually labelled such raw data to distinguish among different operational conditions, also taking into account the related response times measurements, collected client-side, as a general indication of the QoS. In other words, we emulated the information that is typically produced by operations teams after a system outage occur (e.g., *post-mortem* documents). Notice that, while the test application was running on a horizontally-scalable group of VMs, our approach is *agnostic*

with respect to the used virtualization technology. For both steps, we preferred to use ML models that are sufficiently lightweight to train and use, yet they achieve the right levels of accuracy in AD tasks. This allows us to possibly scale our proposed solution to the analysis of several elastic groups composed of many instances. Furthermore, we seeked models that guarantee a sufficient level of interpretability, as cloud automations should be highly dependable and auditable, thus we prefer not to use heavyweight models based on DNNs. However, explainability analysis goes beyond the scope of this paper and will be addressed in future works.

Figure 2 summarizes the main implementation details of our approach. The AD model (i.e., step *(i)*) continuously monitors the status of a specific elastic cloud application by considering coarse-grained (aggregated) data. Then, as soon as an anomalous condition is detected, the classification model (i.e., step *(ii)*) is triggered to analyze fine-grained data, considering the underlying instances individually, to possibly recommend a corrective action for each of them.

For step *(i)*, we trained an AD model on spatial aggregates of such data, so that the model can generalize to elastic groups of instances. Also, as it is continuously executed, this step acts as a *low-cost* filter that prevents the system from running the (more expensive) step *(ii)* on higher-resolution data when it is not necessary. In these settings, it is impractical to assume the availability of large amounts of *labelled* data. We opted for MADI [17], an *unsupervised* AD approach that leverages on *negative sampling* to cope with labelled data shortage. MADI works spectacularly well with high-dimensional data that capture complex multi-modal behaviors, assuming that the presence of anomalous behavior is *scarce*. It assumes all provided training data to be positive examples, and computes a *negative space* to sample from, assuming that every behavior that significantly differ from the positive examples is to be considered anomalous. As there are now two distinct classes of examples, it is possible to use any supervised classification algorithm. We used the Negative Sampling Random Forest (NSRF) provided by MADI, setting the hyper-parameters as specified in the paper. We trained NSRF on a dataset containing multiple traces of positive-only examples of expected CPU and disk I/O activity patterns. Such data were preprocessed by calculating spatial aggregations, i.e., mean (μ) and standard deviation (σ), to make the model agnostic to the actual number of instances in the cluster. After that, we applied standard scaling (i.e., subtracting to a signal its μ and dividing by its σ) and built the set of training samples by applying a rolling window of 5 observations, shifted by 1 observation at a time. Each sample consisted in a 2D vector with dimensions 5×4, i.e., a 5-minutes time-frame, partially overlapping with adjacent samples, containing 2 spatial aggregations of 2 distinct metrics. However, given that NSRF is not designed to natively work with multi-variate time-series, we reshaped each training sample to a 1D vector with dimension 20, such that the rows of the original 2D vector are stacked horizontally, and the contributions of the different signals are interleaved.

For step *(ii)*, we developed ourselves a simple, yet effective, supervised multi-label classification model using XGBoost [29], a powerful framework that offers performant implementations of gradient-boosted trees (GBT) [30]. The job of this model is to learn to distinguish among different classes of anomalous conditions patterns, such that they can be associated with the appropriate corrective actions. This model is designed to run only when triggered by the AD model, that continuously analyze new observations as soon as they become available, and flags them if the corresponding application operates outside the expected conditions. Therefore, as it is supposed to run infrequently, we designed the model to work on instances' raw data, in an attempt to enhance its classification capabilities. Indeed, such model is designed to compare the behavior of an individual instance with the rest of the group (i.e., the other instances that implement the same application), by taking as inputs a combination of spatially-aggregated and raw data, under the assumption that all instances in the same group behave consistently. Given a flagged group of instances, that could even be fairly large, the classifier is applied to each one separately, to output a (possible) recommended corrective action for each of them. Notice that this strategy potentially allows for identifying both the root-cause and the appropriate counter-measure even when an anomaly is caused by multiple instances at once. We trained the model on data collected while injecting anomalies during runs of our test application (see Section IV). We applied a preprocessing similar to the one used for AD, such that each training sample consisted in a 2D vector representing a 5-minutes window on the raw data. However, for each window, we generated a number of training samples equal to the number of instances in the group. Each sample consisted in a 5×6 vector, where the columns contain the following information: 1) CPU utilization of the specific instance; 2) Disk I/O activity of the specific instance; 3) μ of the CPU utilization of the other instances; 4) μ of the disk I/O activity of the other instances; 5) σ of the CPU utilization of the other instances; 6) σ of the disk I/O activity of the other instances. Also in this case, given that XGBoost is not designed to natively work with multi-variate time-series, we reshaped each training sample to a 1D vector with dimension 30, such that the rows of the original 2D vector are stacked horizontally, and the contributions of the different signals are interleaved. We used the metadata of our runs to *label* each sample according to the corresponding type of behavior that, in turn, is associated with a specific corrective action. If a given sample was related to an injected instance, and at least 3 observations had been collected during the injection, then the sample was labelled accordingly: 1 for *stress*, 2 for *fault*, and 3 for *saturation* (0 otherwise).

IV. EXPERIMENTS

This section presents the results of an empirical validation of the approach described in Section III, conducted by deploying

both a synthetic application and the Cassandra NoSQL data store on OpenStack. We used data from such deployments to train the underlying ML models, and assess their accuracy.

A. EXPERIMENTAL SET-UP

We carried out our experiments on an OpenStack installation (*Yoga* release), that was deployed using Kolla [31], a tool for automated deployment of OpenStack services using Docker containers. OpenStack was hosted on 3 physical hosts:

1) A Dell R630 server, equipped with: 2 Intel Xeon E5-2640 v4 CPUs (20 hyper-threads each) running at 2.40 GHz; 64 GB of RAM; a 3.3 TB Dell PERC H330 Mini hard disk; Ubuntu 22.04 LTS; Linux kernel 5.15.0. This host was used as *controller* and *compute* node.
2) A Dell R740xd server, equipped with: 2 Intel Xeon Gold 6238R CPUs (56 hyper-threads each) running at 2.20 GHz; 128 GB of RAM; a 2.2 TB Dell PERC H740P Mini hard disk; Ubuntu 20.04 LTS; Linux kernel 5.4.0. This host was used as *compute* node.
3) A workstation, equipped with: an Intel Core i7-4790K quad-core CPU (8 hyper-threads) running at 4.00 GHz; 16GB of RAM; a 500 GB Samsung 850 SSD; Ubuntu 22.04 LTS; Linux kernel 5.15.0. This host was used as *compute* node.

These were all connected to the same switch using a 1 Gb link cable. We deployed a test application leveraging on the following services: *(i)* Heat [32], to orchestrate a horizontally-scalable cluster of Nova [33] instances; *(ii)* Octavia [34], for load-balancing; *(iii)* Monasca [35], for telemetry. The application cluster was configured to have 3 instances, each one deployed on a different physical host, such that we could better control the experiments that involved monitoring the disk I/O activity, by reducing interferences. Each instance was provided with 1 vCPU and 2 GB of RAM, and with Ubuntu 20.04 server cloud image. To better control our experiments, we disabled both the elasticity and the self-healing capabilities of the cluster, and we made sure that each instance was pinned to a different physical CPU core, that remained unchanged for the entire duration of the experiments. The instances were reachable through an Octavia load-balancer (LB), that was configured with a *least-connections* strategy. Monasca was configured to collect new CPU and disk I/O activity measurements every minute.

B. SYNTHETIC WORKLOAD GENERATOR

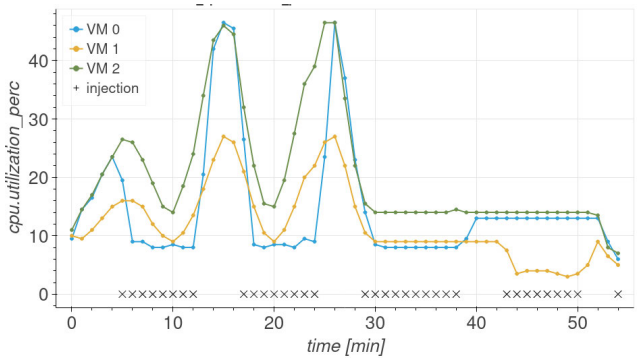
We used the open-source *distwalk* [36] tool to generate traffic on our deployment, consisting of: a server component, that accepts connections from clients via TCP/IP; a client component, sending requests to the server, asking to perform different kinds of tasks (e.g., stressing the CPU, moving data to/from the disk, networking activities, etc.). One can tune the amount of resources to be consumed in a given time frame, by specifying the way in which requests are submitted, with distributed inter-arrival times, payload sizes, or I/O transfer sizes. A client can also be configured to spawn multiple threads (submitting traffic in parallel) and/or to break their execution in multiple sessions, by closing and re-establishing their TCP/IP connections with the server. We set the client such that the CPU and disk I/O activity of the instances followed a set of dynamic workload profiles. The client was configured to spawn 2 threads per instance, each one provided with a trace specifying the operation rates (i.e., requests per second), to be maintained for 1 minute each. Each thread was also configured to create a total of 5000 sessions over each run, such that a new target instance could be selected by the load-balancer at each new session establishment.

C. APACHE CASSANDRA

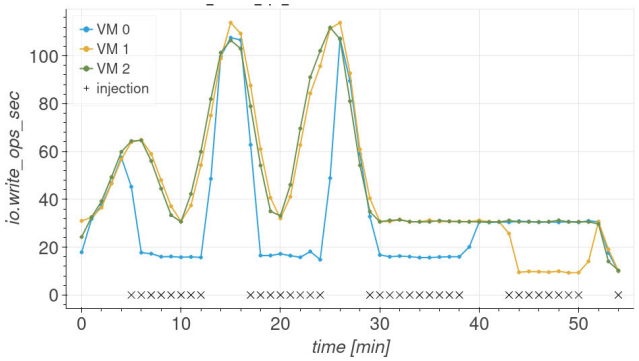
Beside the aforementioned synthetic application, we also used Apache Cassandra, a widely known open-source NoSQL data store, to also test our approach in more realistic scenarios. Based on the design principles of Dynamo [37], Cassandra is a distributed data store characterized by a scalable and fault-tolerant peer-to-peer architecture, able to handle large amounts of data by spreading the load across the cluster. In practice, this is done by partitioning the key-space of a table primary key, spreading its shares over the peers. Cassandra offers the possibility to tune the level of write/read consistency, and the replication strategy. Such features make it a great cloud storage solution for critical big-data applications that require high scalability and availability, or for high-throughput use cases with less stringent consistency requirements. We deployed Cassandra on our OpenStack test-bed, with each peer hosted in a VM on a different physical host, and the keyspace replicated across the whole cluster to avoid data loss in case of anomalies. The traffic is generated using YCSB [38], a well-known open-source benchmarking tool for NoSQL data stores, which allows for configuring: the probability distribution of requests across the key-space; the number of pre-inserted records; the proportion of read, update, scan and insert operations to issue; and other performance-related parameters. In our case, to avoid saturating the available disk space, YCSB was configured to load into the cluster a pre-fixed amount of records (1 million, 1 KB each). Also, the traffic throughout each run included update operations only (3 millions in total, at a rate of 1000 ops/sec), such that the cluster could still perform write operations without increasing the total number of records. The cluster was also set with a *replication* level equal to 3, and a *consistency* level varying between 1 and 2.

D. ANOMALY INJECTION

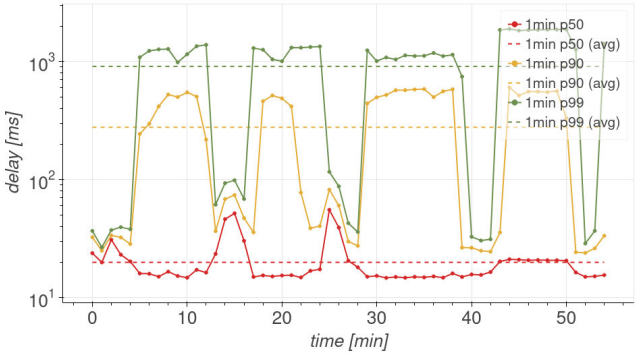
To train and evaluate the ML models underlying our approach, we needed examples of anomalous conditions to associate with the typical corrective actions described in Section III. For simplicity, we considered only three of the most common anomaly types in cloud environments: *(i)* interferences generated by external load co-located on the same physical hosts; *(ii)* faulty members of load-balanced



(a) CPU usage

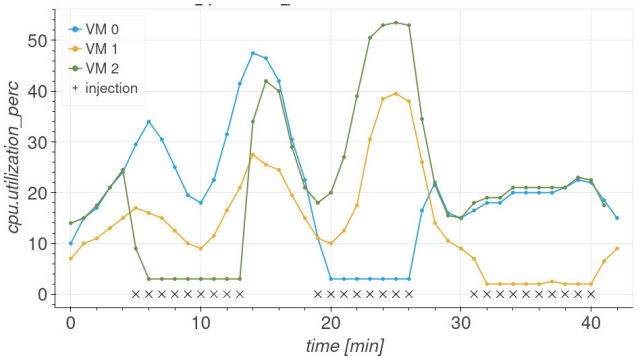


(b) Disk I/O activity

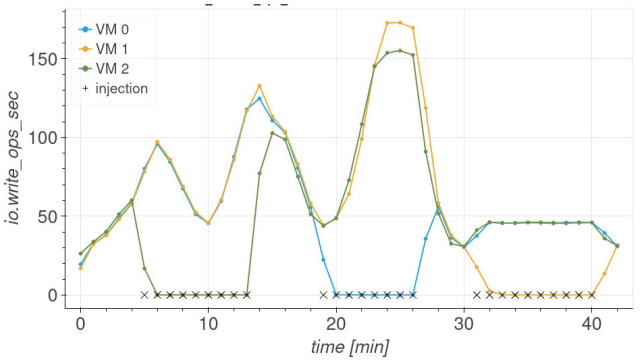


(c) Client-side response times

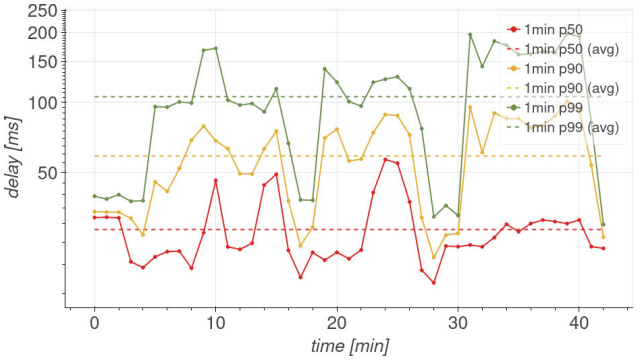
FIGURE 3. Interferences generated by *stress-ng* on *distwalk*.



(a) CPU usage



(b) Disk I/O activity

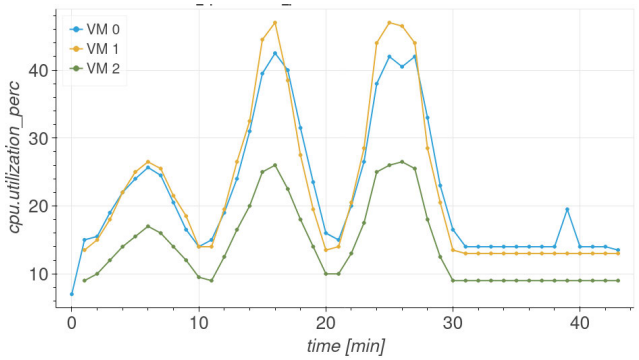


(c) Client-side response times

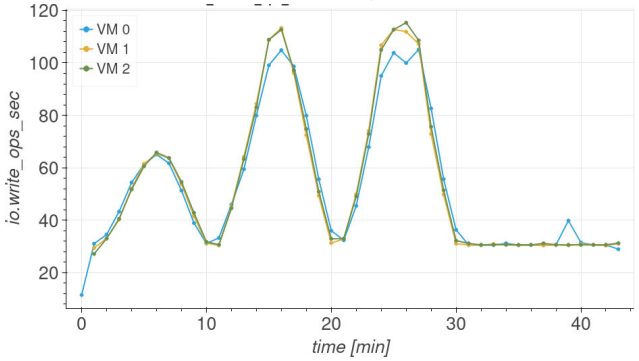
FIGURE 4. Faults due to killing an instance of *distwalk*.

groups of instances that stop picking their traffic share; and *(iii)* saturation of the current resource capacity. To generate data describing such anomalous conditions, we artificially injected them during the execution of our runs. Specifically, for *(i)*, we used *stress-ng* [39] to simulate the interference of external processes that end up being scheduled on the same physical host of an instance. For *(ii)*, it was sufficient to kill the application process running on a specific instance to make it stop responding to requests. Whereas, for *(iii)*, we just made sure to send a workload that could not be properly handled by the currently allocated resources. We also augmented the diversity of the anomalous behaviors to be observed by our ML models by generating and enforcing schedules of randomly distributed anomalies. However, to better control our experiments, we made sure we had only one, randomly selected, unhealthy instance at a time. Also, once an instance was injected with an anomaly, we made sure it remained unhealthy for an extended period (e.g., 5-10 minutes), automatically recovering afterwards.

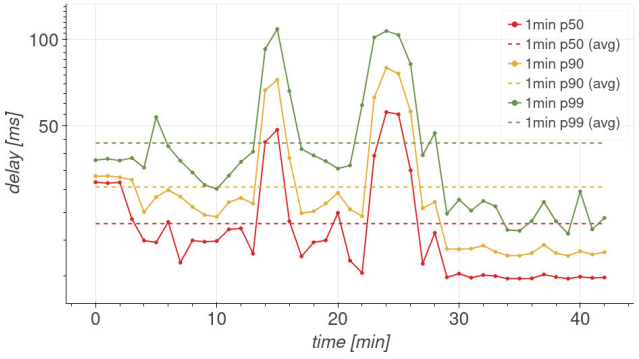
A few examples of the data extracted from our experimental runs with the *distwalk* application are depicted in Figures 3 to 5. These report the observed resource-consumption levels (on the Y axis for subfigures (a) and (b)) and experienced client-side response times (on the Y axis for subfigures (c)), over time (ont the X axis), during runs of *distwalk* using the same workload profile as input. The system-level metrics `cpu.utilization_perc` and `io.write_ops_sec` were collected by using the Monasca monitoring system, while the client-side response times were extracted from the *distwalk* client logs. Figures 3 to 5



(a) CPU usage



(b) Disk I/O activity



(c) Client-side response times

FIGURE 5. **Saturation of the disk bandwidth for *distwalk*.**

present results in the case of performance degradation due to co-located stress workload, and saturation of the available resources, respectively, while Figure 4 refers to a scenario with an instance completely failing.

Note that, due to how the system components and the *distwalk* client are configured, if anomalies are *not* injected, then the LB continues to equally distribute the load among the available instances. In such case, the disk I/O activity level should be more or less the same for all instances. However, due to the different processors the available physical hosts are equipped with, we can observe differences in terms of CPU utilization levels, even though the workloads follow the same profile during the run. A clear example of this scenario is depicted in Figures 5a and 5b where, during the first 10 minutes of the run, the available resources were sufficient to handle the workload. In this case, instance 2 was (randomly) scheduled on the physical host equipped with the most powerful processor (see Section IV-A), and exhibited a lower CPU utilization with respect to the other instances, while the disk I/O activity was more or less equivalent. Furthermore, in such *normal* cases, we can also observe particularly low client-side response times. In Figure 5c (and similar) we can indeed appreciate how the distribution of the response times evolve during a run, in terms of 50th, 90th and 99th percentiles. Each point in the plot refers to a specific statistic calculated over a 1-minute interval. For instance, a point at 0 refers to all the response times registered during the first minute of the run, and so on. Whenever the system did not saturate (e.g., during the first 10 minutes of the run), we generally observed a p90 *below* 35 ms. Therefore, we took this value as a rough indication of a *good* QoS.

1) WORKLOAD DEGRADATION

When using *stress-ng* to simulate interferences from co-located, I/O-intensive, external workloads, we observed the CPU and disk I/O activity of the affected instances significantly dropping and staying around relatively low values. For instance, in Figure 3, when the stress was injected around minutes 7-12 on instance 0, such instance exhibited a CPU utilization around ∼10% (see Figure 3a) and a disk I/O activity around ∼20 ops/sec (see Figure 3b), compared to the reference values, ∼28% and ∼62 ops/sec, respectively, exhibited by instance 2 during the first peak of the workload. The effect of the stress injection is even more significant around minutes 18-24, where instance 0, randomly picked again, exhibited similar resource utilization values, but this time with reference values being ∼46% and ∼108 ops/sec, respectively, during the third peak of the workload. The stress injection also significantly affects the response latency perceived by the client. Indeed, in Figure 3c, we can observe peaks of ∼600 ms in the p90 curve, corresponding to the injection intervals. Due to how *distwalk* is designed, while an instance experiences interferences, but is still barely able to send responses, the client accumulates delay by waiting for responses, before triggering the subsequent requests. This is the reason why, during stress-injected runs, we typically observe longer ''tails'' of delayed requests that keep on being sent at the last rate specified in the workload schedule (e.g., around minutes 30-50).

2) COMPLETE INSTANCE FAILURE

We simulated an instance complete failure killing one of the *distwalk* servers processing the requests. When doing so, we observed the disk I/O activity of the affected instance dropping to 0, and its CPU usage stabilizing around 2-3% (i.e., the standard load generated by the OS background processes). On the other hand, the activity on the other instances increased accordingly, due to the LB redirecting the extra load on them. For instance, in Figure 4, when the fault was injected around minutes 6-13 on instance 2, we can see that the disk I/O activity of the other instances reached ∼100 ops/sec during the first peak of the workload (Figure 4b). As we know

a-priori that the workload should have closely followed an *ideal* sinusoidal pattern, we can definitely tell that it increased significantly with respect to the expectations. The effect of the fault injection, this time on instance 0, is even more significant around minutes 20-26, with instance 1 reaching ∼170 ops/sec and instance 2 reaching ∼150 ops/sec. Similar behaviors can be observed for the CPU usage, although it is less evident for the instances scheduled on the most powerful physical processor (see Figure 4a). Obviously, the fault injection also significantly affects the response latency perceived by the client. Indeed, in Figure 4c, we can observe peaks of ∼100 ms in the p90 curve, corresponding to the injection intervals. However, overall, the impact on the QoS is significantly smaller than what we observed for the stress injection. This happens because, eventually, the LB detects the injected instance to be unhealthy and interrupts the current connections to redirect the load on the others. When the connection is closed, the *distwalk* client ignores the remaining requests planned for the corresponding session and moves to the next, partially compensating the accumulated delay.

3) DISK BANDWIDTH SATURATION

The disk bandwidth constitutes one of the critical bottlenecks in resource saturation scenarios. When an instance receives an unexpectedly high volume of requests, the *distwalk* server process starts falling behind the expected schedules, the requests pile-up in the queue, and the response times start increasing. For instance, in Figure 5, we can see such a phenomenon occurring around minutes 14-16 and 23-26, during the second and third peaks of the workload, respectively. By looking at the system-level metrics in Figures 5a and 5b, we can typically observe that, in such cases, the workload profiles of the different instances do not closely follow the expected sinusoidal pattern, and start diverging. However, considering only such metrics, and assuming not to have any a-priori knowledge of the expected workload, we cannot exclude that such deviations are just noise. Furthermore, it is even trickier to infer that a saturation phenomenon is occurring when the workload is mainly I/O-intensive, rather than CPU-intensive, since the actual bandwidth of traditional, rotational, hard drives depends on multiple factors, and it is not guaranteed to be always consistent. Therefore, the most effective way to detect that a saturation phenomenon is occurring, is by looking at the IOWait metrics if available, or just the client-side response times. Indeed, in Figure 5c, we can observe peaks of ∼80 ms in the p90 curve, during the aforementioned workload peaks. Similarly to the fault injection scenario, the impact of the disk saturation on the QoS is significantly smaller than what we observed for the stress injection.

Similarly to the *distwalk* application, also the Cassandra cluster was injected with anomalies during our experimental runs. For instance, Figure 6 shows the measurements recorded during one of such runs, where we used *stress-ng* to generate interference, while the cluster, with replication level set to 3 and consistency level set to 2, was serving the

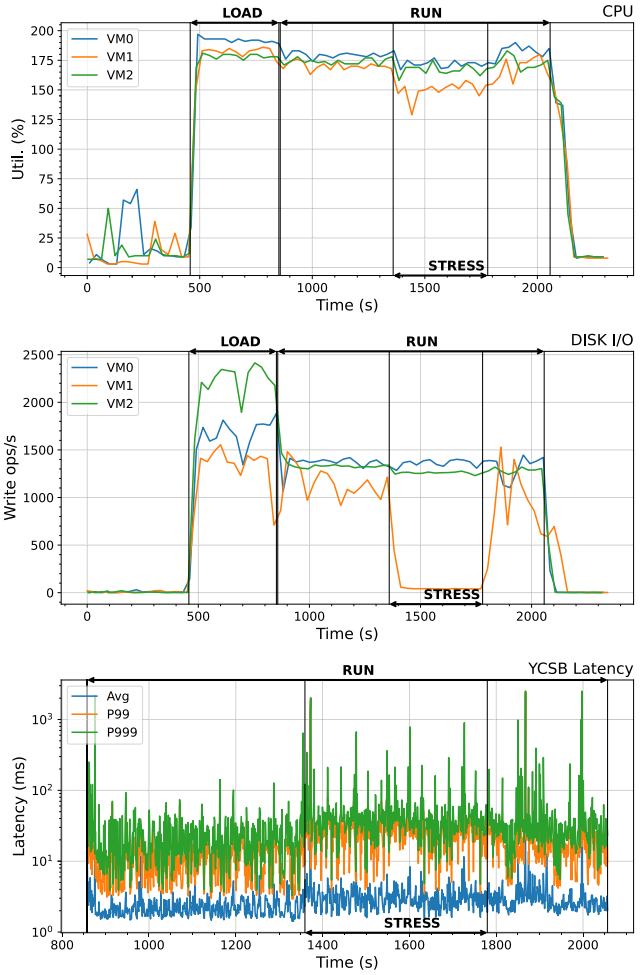
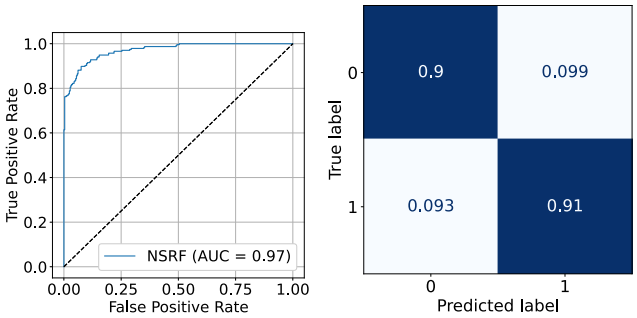


FIGURE 6. Interferences caused by *stress-ng* on Cassandra.

load generated by YCSB, as explained in Section IV-C. In the figure, we highlighted: the LOAD phase, when YCSB loads the cluster with 1 million keys and their associated 1 KB values; the RUN phase, when YCSB imposes a constant target update throughput of 1000 ops/s; and the STRESS phase, when one of the Cassandra replicas undergoes heavy disk I/O interference from *stress-ng*. As in Figure 3, we can appreciate that, during the STRESS phase, the disk I/O activity of the affected instance drops significantly, with respect to the other members of the cluster. However, the effect of the interference on the CPU utilization is less evident. During the same phase, we can also observe the latency perceived by the YCSB client increasing consistently.

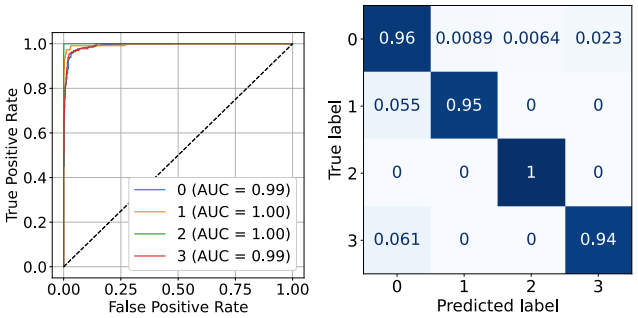
E. RESULTS

After conducting several runs under different conditions, both with *distwalk* and Cassandra, we collected the corresponding CPU and disk I/O activity measurements and trained our models for the AD and classification steps. For both applications' datasets, separately, we held out the same portions of data to be used as training and test sets for the models. However, the two models were trained on different



(a) ROC-AUC (b) Confusion matrix

FIGURE 7. AD performance, computed on the *distwalk* test set (0 = anomalous; 1 = normal).



(a) "One-vs-rest" ROC-AUC (b) Confusion matrix

FIGURE 8. Classifier performance, computed on the *distwalk* test set (0 = normal; 1 = stress; 2 = fault; 3 = saturation).

TABLE 1. Performance metrics of the AD model, for each class of samples, computed on the *distwalk* test set (0 = anomalous; 1 = normal).

Class	Precision	Recall	F1 score
0	0.923	0.901	0.912
1	0.881	0.907	0.894

TABLE 2. Performance metrics of the classifier, for each class of samples, computed on the *distwalk* test set (0 = normal; 1 = stress; 2 = fault; 3 = saturation).

Class	Accuracy	Precision	Recall	F1 score
0	0.961	0.988	0.962	0.975
1	0.989	0.904	0.945	0.924
2	0.995	0.926	1.000	0.962
3	0.977	0.816	0.939	0.873

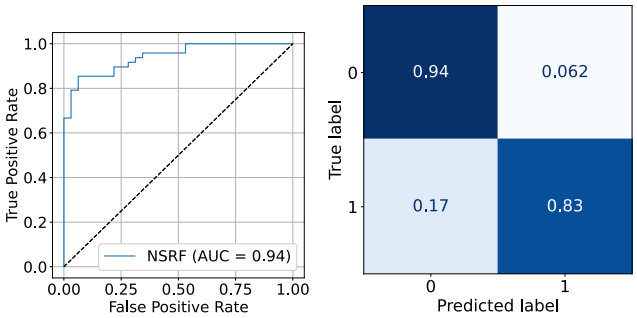
views of the same information (details in Section III). This way, the AD step can act as a filter and let the system trigger the (more costly) classification step only when it is deemed useful. This work is accompanied by an open-source repository [40] including all the material required to reproduce the presented results.

1) SYNTHETIC APPLICATION - ANOMALY DETECTION

As explained in Section III, we decided to implement this step with MADI [17], using the NSRF variant. By preprocessing the collected data, we obtained a training set of 1087 and a test set of 528 input vectors, with shape 5×4 (as explained in Section III-B). Such training and test sets contain a fraction of positive (normal) examples equal to 42.59% and 44.70%, respectively, while the rest is constituted by anomalous examples. Therefore, since NSRF is an unsupervised approach that assumes the input data to consist in mainly positive behavior, we trained it only on the 463 positive examples from the training set. After training NSRF, that typically takes just a few seconds on the CPU of our first physical host (see Section IV-A), without any specific acceleration settings, we ran the obtained model on the test set, this time using both positive and negative examples. Thanks to its negative sampling strategy, NSRF solves a *binary* classification task, and outputs the probability of an input belonging to the positive class. Such feature allowed us to produce the Receiver Operating Characteristic (ROC) curve [41] shown in Figure 7a, corresponding to an Area Under Curve (AUC) of 97%. The ROC curve is a technique to visualize the evolutions of the True-Positive Rate (TPR) and False-Positive Rate (FPR) of the model, considering a variable classification threshold over the output probability of the model. This way, we could select a sensible value for the threshold t_p, to be applied on the output to determine the class of a given input, such that the FPR was below 10%, and the TPR was above 85% (i.e., corresponding to the upper-left region of Figure 7a). By setting $t_p = 0.594$, we obtained an accuracy of 90.34%, corresponding to the confusion matrix [41] shown in Figure 7b. In Table 1, we also report other per-class performance measures.

2) SYNTHETIC APPLICATION - CLASSIFICATION

As explained in Section III, we decided to address this step, consisting in a multi-label classification task, by implemented our model using XGBoost [29]. By preprocessing the collected data, we obtained a training set of 3261 and a test set of 1584 input vectors, with shape 5×6 (as explained in Section III-B). Note that the preprocessing employed for this model produces a dataset 3 times bigger than the one used for the AD model. This is due to the fact that, for each 5-minutes window on the raw data, such preprocessing produces a number of input samples equal to the number of active instances in the considered application (3, for our runs). Also, such preprocessing produces an inherently imbalanced dataset, due to the fact that, for each 5-minutes window, only one sample is marked as anomalous, given that we made sure not to inject multiple anomalies at once. Indeed, the training set is composed for the 78.44% by normal, for the 8.40% by stress-injected, for the 5.24% by fault-injected, and for the 7.91% by saturation examples. Similarly, the test set is composed for the 78.41% by normal, for the 6.94% by stress-injected, for the 6.31% by fault-injected, and for the 8.33% by saturation examples. However, XGBoost offers the capability to easily specify weights for each class, such that each



(a) ROC-AUC (b) Confusion matrix

FIGURE 9. AD performance, computed on the Cassandra test set (0 = anomalous; 1 = normal).

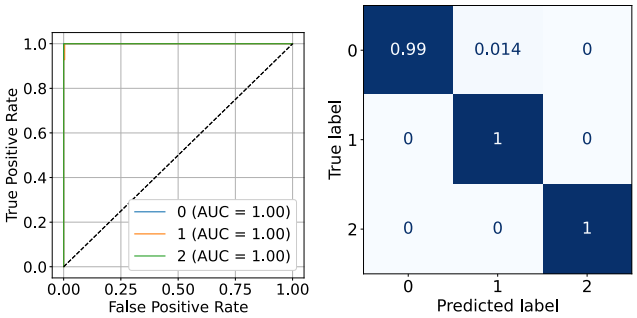
TABLE 3. Performance metrics of the AD model, for each class of samples, computed on the Cassandra test set (0 = anomalous; 1 = normal).

Class	Precision	Recall	F1 score
0	0.789	0.938	0.857
1	0.952	0.833	0.889

one proportionally contributes to the gradient updates. After training the classifier, that typically takes less than 10 seconds on the CPU of our first physical host (see Section IV-A), without any specific acceleration settings, we used the test set to evaluate its performance. Given that XGBoost can be set to output the distribution of the probability of an input to belong to each of the available classes, also in this case we were able to produce a ROC curve, shown in Figure 8a. However, for the multi-label classification task, ROC curves can only be produced in a "one-vs-rest" fashion, i.e., each time considering a specific class against all the others (as they were a single one). Remarkably, all the generated ROC curves correspond to AUC values of nearly 100%. Then, we applied the model on the test set and obtained an accuracy of 96.15%, corresponding to the confusion matrix in Figure 8b. Table 2 reports other per-class performance measures. Note that, in this case, also the accuracy can be computed in a "one-vs-rest" fashion.

3) CASSANDRA - ANOMALY DETECTION

Similarly to the synthetic application, by preprocessing the data collected during Cassandra runs, we obtained a training set of 224 and a test set of 80 input vectors. Such training and test sets contain a fraction of positive (normal) examples equal to 57.14% and 60%, respectively, while the rest is constituted by anomalous examples. After training NSRF on positive samples only, we validated the model on the test set, and obtained the ROC curve shown in Figure 9a, corresponding to an AUC of 94%. By setting $t_p = 0.616$, we obtained an accuracy of 87.50%, corresponding to the confusion matrix shown in Figure 9b. In Table 3, we also report other per-class performance measures.



(a) "One-vs-rest" ROC-AUC (b) Confusion matrix

FIGURE 10. Classifier performance, computed on the Cassandra test set (0 = normal; 1 = stress; 2 = fault).

TABLE 4. Performance metrics of the classifier, for each class of samples, computed on the Cassandra test set (0 = normal; 1 = stress; 2 = fault).

Class	Accuracy	Precision	Recall	F1 score
0	0.988	1.000	0.986	0.993
1	0.988	0.824	1.000	0.903
2	1.000	1.000	1.000	1.000

4) CASSANDRA - CLASSIFICATION

Similarly to the synthetic application, by preprocessing the data collected during Cassandra runs, we obtained a training set of 672 and a test set of 240 input vectors. The training set is composed for the 85.71% by normal, for the 6.25% by stress-injected, and for the 8.04% by fault-injected examples. Instead, the test set is composed for the 86.67% by normal, for the 5.83% by stress-injected, and for the 6.31% by fault-injected examples. After training the classifier, we validated it on the test set, and obtained the "one-vs-rest" ROC curve shown in Figure 10a. We also obtained an accuracy of 98.75%, corresponding to the confusion matrix shown in Figure 10b. Table 4 reports other per-class performance measures, computed in a "one-vs-rest" fashion.

V. CONCLUSION

We proposed an ML-based strategy for *intelligent* cloud operations that consists of: *(i)* detecting anomalous conditions of a cloud application and *(ii)* identifying the corrective actions to be applied to faulty components. Both steps rely on ML models trained on operational data. Step *(i)* acts as a filter that allows the system to run the more expensive step *(ii)* on higher-resolution data only when needed. Our approach was validated using data exported from an OpenStack deployment. We used a workload generator sending traffic to a load-balanced group of Nova instances, resulting in CPU and disk I/O activity on the instances, and injected different types of anomalies that we could recover from, by applying precise corrective actions. For *(i)*, we trained an anomaly detection model (specifically, MADI [17]) on aggregated cluster data, such that it could even generalize to variable-sized groups of instances. On the respective test sets, for the synthetic application, we obtained a ROC-AUC of 97% and an accuracy of 90.34%, while, for Cassandra,

we obtained a ROC-AUC of 94% and an accuracy of 87.50%. For *(ii)*, we trained a supervised classification model, based on XGBoost [29], on a combination of spatially-aggregated and raw instances data, such that it could better compare the behavior of an individual instance with respect to its group, and associate a corrective action to instances separately. On the respective test sets, for the synthetic application, we obtained an accuracy of 96.15%, while, for Cassandra, we obtained an accuracy of 98.75%. To implement our approach, we decided to use rather simple, yet very effective, ML models. Such a design choice allows for our approach to be highly dependable, especially because it is relatively easy to interpret and troubleshoot the output of this kind of models, possibly leveraging on automated explainability techniques. We plan to conduct a more thorough study to better (quantitatively) compare our approach to existing alternatives. In this research area, such an activity is rendered particularly difficult by the general lack of open-source implementations readily integrated within a framework like OpenStack. For the classification, we opted for supervised-learning. However, it would be interesting to apply unsupervised or weakly-supervised approaches to our problem, to possibly weaken our dependency on labelled data. On a related note, even though we have already shown that our approach is able to deal with multivariate data, it would be interesting to also extend our experimental validation by considering additional metrics, to properly assess how the approach scales with respect to the number of considered features. Also, we plan to integrate automatic model retraining, to counteract the disastrous effects of *distribution drifts*, and model explainability techniques in the end-to-end pipeline. Indeed, guaranteeing a sufficient level of robustness to fluctuations and interpretability is of utmost importance, as cloud operations are a scenario where any type of automation should be highly dependable and auditable. Finally, we plan to properly package our approach as an OpenStack service, to offer a reliable, open solution for intelligent cloud operations to a wide audience.

REFERENCES

[1] B. Marr. (2021). *The 5 Biggest Cloud Computing Trends in 2022*. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2021/10/25/the-5-biggest-cloud-computing-trends-in-2022/

[2] A. Kannan, J. LaRiviere, and R. P. McAfee, "Characterizing the usage intensity of public cloud," *ACM Trans. Econ. Comput.*, vol. 9, no. 3, pp. 1–18, Sep. 2021, doi: 10.1145/3456760.

[3] M. Attaran and J. Woods, "Cloud computing technology: Improving small business performance using the internet," *J. Small Bus. Entrepreneurship*, vol. 31, no. 6, pp. 495–519, Nov. 2019, doi: 10.1080/08276331.2018.1466850.

[4] M. Chiosi et al., "Network functions virtualisation—Introductory white paper," SDN OpenFlow World Congr., Darmstadt, Germany, Oct. 2012. [Online]. Available: http://portal.etsi.org/NFV/NFV_White_Paper.pdf

[5] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing Principles and Paradigms*. Hoboken, NJ, USA: Wiley, 2011.

[6] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proc. 6th ACM Symp. Cloud Comput.*, Aug. 2015, p. 167, doi: 10.1145/2806777.2809955.

[7] R. Buyya, K. Ramamohanarao, C. Leckie, R. N. Calheiros, A. V. Dastjerdi, and S. Versteeg, "Big data analytics-enhanced cloud computing: Challenges, architectural elements, and future directions," in *Proc. IEEE 21st Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2015, pp. 75–84.

[8] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy, "Metric selection and anomaly detection for cloud operations using log and metric correlation analysis," *J. Syst. Softw.*, vol. 137, pp. 531–549, Mar. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217300596

[9] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (Micro) service-based cloud applications: A survey," *ACM Comput. Surv.*, vol. 55, no. 3, pp. 1–39, Mar. 2023, doi: 10.1145/3501297.

[10] W. Wang, L. Tang, C. Wang, and Q. Chen, "Real-time analysis of multiple root causes for anomalies assisted by digital twin in NFV environment," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 2, pp. 905–921, Jun. 2022.

[11] P. Mell and T. Grance. (2011). *The NIST Definition of Cloud Computing. SP 800–145*. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-145/final

[12] M. Zoure, T. Ahmed, and L. Réveillère, "Network services anomalies in NFV: Survey, taxonomy, and verification methods," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 2, pp. 1567–1584, Jun. 2022.

[13] A. Gulenko, M. Wallschläger, F. Schmidt, O. Kao, and F. Liu, "Evaluating machine learning algorithms for anomaly detection in clouds," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 2716–2721.

[14] Q. Du, Y. He, T. Xie, K. Yin, and J. Qiu, "An approach of collecting performance anomaly dataset for NFV infrastructure," in *Algorithms and Architectures for Parallel Processing*, J. Vaidya and J. Li, Eds. Berlin, Germany: Springer, 2018, pp. 59–71.

[15] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, Nov. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231217309864

[16] T. Cucinotta, G. Lanciano, A. Ritacco, M. Vannucci, A. Artale, J. Barata, E. Sposato, and L. Basili, "Behavioral analysis for virtualized network functions: A SOM-based approach," in *Proc. 10th Int. Conf. Cloud Comput. Services Sci.*, 2020, pp. 150–160.

[17] J. Sipple, "Interpretable, multidimensional, multimodal anomaly detection with negative sampling for detection of device failure," in *Proc. 37th Int. Conf. Mach. Learn.*, 2020, pp. 9016–9025. [Online]. Available: https://proceedings.mlr.press/v119/sipple20a.html

[18] Q. Rebjock, V. Flunkert, T. Januschowski, L. Callot, and J. Castellon, "A simple and effective predictive resource scaling heuristic for large-scale cloud applications," in *Proc. 2nd Int. Workshop Appl. AI Database Syst. Appl.*, 2020, pp. 1–5.

[19] G. Lanciano, F. Galli, T. Cucinotta, D. Bacciu, and A. Passarella, "Predictive auto-scaling with OpenStack Monasca," in *Proc. 14th IEEE/ACM Int. Conf. Utility Cloud Comput.*, Dec. 2021, pp. 1–10.

[20] L. Ju, P. Singh, and S. Toor, "Proactive autoscaling for edge computing systems with kubernetes," 2021, *arXiv:2112.10127*.

[21] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba, "Topology-aware prediction of virtual network function resource requirements," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 1, pp. 106–120, Mar. 2017.

[22] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood, "Predictive auto-scaling of multi-tier applications using performance varying cloud resources," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 595–607, Jan. 2022.

[23] P. Kang and P. Lama, "Robust resource scaling of containerized microservices with probabilistic machine learning," in *Proc. IEEE/ACM 13th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2020, pp. 122–131.

[24] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, "NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning," in *Proc. IEEE/ACM 27th Int. Symp. Quality Service (IWQoS)*, Jun. 2019, pp. 1–10, doi: 10.1145/3326285.3329056.

[25] S. Ilager, R. Muralidhar, and R. Buyya, "Artificial intelligence (AI)-centric management of resources in modern distributed computing systems," in *Proc. IEEE Cloud Summit*, Oct. 2020, pp. 1–10.

[26] M. Borkowski, S. Schulte, and C. Hochreiner, "Predicting cloud resource utilization," in *Proc. IEEE/ACM 9th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2016, pp. 37–42.

[27] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection and classification using distributed tracing and deep learning," in *Proc. 19th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2019, pp. 241–250.

[28] *2022 User Survey Report—OpenStack is More Alive Than Ever With 40 Million Cores in Production*. Accessed: Jul. 2023. [Online]. Available: https://www.openstack.org/user-survey/2022-user-survey-report

[29] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 785–794, doi: 10.1145/2939672.2939785.
[30] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, Oct. 2001. [Online]. Available: https://www.jstor.org/stable/2699986
[31] OpenStack. (2023). *Kolla Documentation*. [Online]. Available: https://docs.openstack.org/kolla
[32] OpenStack. (2023). *Heat Documentation*. [Online]. Available: https://docs.openstack.org/heat
[33] OpenStack. (2023). *Nova Documentation*. [Online]. Available: https://docs.openstack.org/nova
[34] OpenStack. (2023). *Octavia Documentation*. [Online]. Available: https://docs.openstack.org/octavia
[35] OpenStack. (2023). *Monasca Documentation*. [Online]. Available: https://docs.openstack.org/monasca
[36] (2023). *Distwalk—Distributed Processing Emulation Tool for Linux*. [Online]. Available: https://github.com/tomcucinotta/distwalk
[37] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.
[38] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, Jun. 2010, pp. 143–154.
[39] C. I. King. (2023). *Stress-NG*. [Online]. Available: https://github.com/ColinIanKing/stress-ng
[40] G. Lanciano, R. Andreoli, T. Cucinotta, D. Bacciu, and A. Passarella. (2023). *Companion REPO of the Paper, A 2-Phase Strategy for Intelligent Cloud Operations*. [Online]. Available: https://github.com/giacomolanciano/intelligent-cloud-operations
[41] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognit. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016786550500303X

GIACOMO LANCIANO (Member, IEEE) received the M.Sc. degree in engineering in computer science from the Sapienza University of Rome, Italy, and the Ph.D. degree in data science from Scuola Normale Superiore, Pisa, Italy. He was a Research Intern with Nokia Bell Labs, Stuttgart, Germany, working on large language models for deployment code analysis. His research interests include the intersection of cloud computing and data science, with a focus on data-driven methods for data center operations support.

REMO ANDREOLI (Member, IEEE) received the M.Sc. degree (Hons.) in computer science from the University of Pisa. He is currently pursuing the Ph.D. degree with Scuola Superiore Sant'Anna, and he's part of an industrial collaboration between Ericsson Research and Scuola Sant'Anna. His research focuses on resource management optimization techniques for cloud infrastructures. He received a Best Student Paper Award from CLOSER 2021.

TOMMASO CUCINOTTA (Senior Member, IEEE) received the M.Sc. degree in computer engineering from the University of Pisa, Italy, and the Ph.D. degree in computer engineering from Scuola Superiore Sant'Anna (SSSA), Pisa, where he has been investigating on real-time scheduling for soft real-time and multimedia applications, and predictability in infrastructures for cloud computing and NFV. He has been MTS with Bell Labs, Dublin, Ireland, investigating on security and real-time performance of cloud services. He has been a Software Engineer with Amazon Web Services, Dublin, where he worked on improving the performance and scalability of DynamoDB. He has been an Associate Professor with SSSA, since 2016, and the Head of the Real-Time Systems Laboratory (RETIS), since 2019. He has coauthored more than 120 research papers on international conferences and journals, and holds eight international patent grants.

DAVIDE BACCIU received the Ph.D. degree in computer science and engineering from IMT Lucca. He is currently an Associate Professor with the University of Pisa, where he heads the Pervasive AI Laboratory. He is the coordinator of the H2020 TEACHING and Horizon EiC EMERGE projects. He has coauthored over 180 research works on (deep) neural networks, generative learning, Bayesian models, learning for graphs, continual learning, and distributed and embedded learning systems. He is the chair of the IEEE CIS Neural Network Technical Committee, and the Vice-President of the Italian Association for AI. He is Senior Editor of the IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS.

ANDREA PASSARELLA received the Ph.D. degree, in 2005. He is currently a Research Director with the Institute for Informatics and Telematics (IIT), National Research Council of Italy (CNR). Prior to join IIT, he was with the Computer Laboratory, University of Cambridge, U.K. He is the PI of the EU CHIST-ERA SAI (Social Explainable AI) Project. He has published more than 170 articles on human-centric data management for self-organizing networks, decentralized AI, next generation internet, online and mobile social networks, opportunistic, and ad hoc and sensor networks. He is the coauthor of the book *Online Social Networks: Human Cognitive Constraints in Facebook and Twitter Personal Graphs* (Elsevier, 2015). He received four best paper awards, including the IFIP Networking 2011 and IEEE WoWMoM 2013. He is the General Chair of the IEEE PerCom 2022. He is the founding Associate Editor-in-Chief of the *Online Social Networks and Media* (OSNEM) (Elsevier).

Open Access funding provided by 'Scuola Superiore "S.Anna" di Studi Universitari e di Perfezionamento' within the CRUI CARE Agreement