Disambiguating Performance Anomalies from Workload Changes in Cloud-Native Applications

Alexandru Baluta
York University
Toronto, Ontario, Canada
balutaal@yorku.ca

Yar Rouf
York University
Toronto, Ontario, Canada
yarrouf@my.yorku.ca

Joydeep Mukherjee
California Polytechnic State University
San Luis Obispo, California, USA
jmukherj@calpoly.edu

Zhen Ming Jiang
York University
Toronto, Ontario, Canada
zmjiang@cse.yorku.ca

Marin Litoiu
York University
Toronto, Ontario, Canada
mlitoiu@yorku.ca

ABSTRACT

Modern cloud-native applications are adopting the microservice architecture in which applications are deployed in lightweight containers that run inside a virtual machine (VM). Containers running different services are often co-located inside the same virtual machine. While this enables better resource optimization, it can cause interference among applications. This can lead to performance degradation. Detecting the cause of performance degradation at runtime is crucial to decide the correct remediation action such as, but not limited to, scaling or migrating. We propose a non-intrusive detection technique that differentiates between degradation caused by load and by interference. First, we define an operational zone for the application. Then we define a disambiguation method that uses models to classify interference and normal load. In contrast to previous work, our proposed detection technique does not require intrusive application instrumentation and incurs minimal performance overhead. We demonstrate how we can design effective Machine Learning models that can be generalized to detect interference from different types of applications. We evaluate our technique using realistic microservice benchmarks on AWS EC2. The results show that our approach outperforms existing interference detection techniques in F_1 score by at least 2.75% and at most 53.86%.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **System administration**.

KEYWORDS

Microservice; Interference; Cloud Computing; Machine Learning

ACM Reference Format:
Alexandru Baluta, Yar Rouf, Joydeep Mukherjee, Zhen Ming Jiang, and Marin Litoiu. 2024. Disambiguating Performance Anomalies from Workload Changes in Cloud-Native Applications. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645046

ICPE '24, May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645046

1 INTRODUCTION

DevOps practice encourages the use of microservice architecture, where a traditional monolithic application is broken down into a collection of easier-to-manage smaller services. Benefits of microservice architecture include better scalability, easier continuous delivery support, data decentralization, and improved fault isolation. Due to these advantages, more application developers are adopting the microservice architecture recently. Microservices are deployed as cloud-native applications on public cloud platforms with each service encapsulated within a container running inside a virtual machine (VM). Containers have risen in popularity for their faster deployment speeds and their ability to allow applications to run in complete isolation from one another without incurring extra overhead. Container orchestration platforms such as Docker [25] and Kubernetes [7] are increasingly gaining popularity and are commonly used in public cloud platforms such as Amazon Web Services (AWS) [3] and Google Cloud Platform [4].

Multiple microservices are generally consolidated on a single VM. This is done to improve resource consumption levels inside a VM and thereby optimize cloud costs. When multiple microservices are deployed on the same VM, they can often compete with each other for shared host-level resources. Such shared resource contention can alter application behavior and make it deviate from the development time specifications and performance. We refer to application performance degradation due to shared resource contention as *performance interference*. Performance interference has been observed before in applications running on public cloud environments [17, 20, 21, 23, 28–30, 36].

Detecting and disambiguating performance interference anomalies from other causes of performance degradation is very important for application runtime self-management and for avoiding outages. For example, a performance degradation due to legitimate workload surge can be automatically mitigated by horizontal scaling whereas interference might be mitigated though application redeployment and relocation. Another use case where detection interference anomalies is important is in reducing the number of

alerts the IT operators have to deal with. It is estimated that the an IT operator faces *80-100* alerts per system [35]. We need to detect if there is interference, and disambiguate if the performance degradation is from interference or from normal behavior. Therefore, it is important that an application detects when its execution environment is perturbed significantly by other applications.

Automatically detecting runtime interference in cloud-native applications is a challenging task. Application workload variability and unpredictability can produce the same effects as interference, and hence it is difficult to distinguish between them. Researchers have looked, with limited success, into instrumenting applications [11], recording their mean request response times, and comparing them against their baseline response times to detect the presence of interference. Continuously instrumenting application code and monitoring the response times of running services [19] can significantly increase the cost of development and maintenance. In addition, continuous monitoring of service metrics can incur a prohibitive overhead when the service is facing a heavy workload [10, 18]. Furthermore, interference detection techniques that model the baseline behavior may not generalize well to scenarios where the interfering application changes at runtime.

To address some of the above challenges, we propose a lightweight method that uses a small number of deployment and runtime performance metrics to automatically disambiguate interference and workload effects on performance. The method involves building or training a model such as queuing, regression, or machine learning models prior to deployment. By using interference for training when using machine learning models, we are no longer dependent on costly response time instrumentation. At runtime, we can use the model and readily available performance data to detect the interference. The method is non-intrusive and does not require any application performance profiling. We use resource utilization metrics that can be easily collected at runtime from within each application container along with simple application level metrics such as request throughput or response time. The assumption is that an autonomic manager is application specific, has access only to managed application metrics and does not see the other application metrics, only their effect on the managed application metrics. The basis of this assumption is that the environment is dynamic and collocated applications might be deployed after the target application. This is in line with the current practice when we develop autoscalers for each application. We also make the assumption that the ML model can be trained as a DevOps process activity and then used at runtime within an AIOps (Artificial Intelligence for IT Operations) platform for interference detection and mitigation.

The paper addresses the following research questions:

RQ-1: How severe is the impact of performance interference in cloud-native applications deployed on public cloud? To address this, we performed experiments to characterize the impact of interference on a microservice benchmark, Acme [1], hosted on the AWS EC2 cloud platform. The results indicate that the response time of Acme can be severely impacted by interference by at least a factor of 39% and at most a factor of 6955% at moderate CPU utilization levels.

RQ-2: How well does a disambiguation method perform when the interfering workloads used for model training and deployment are similar? To answer RQ-2, we develop and train models to detect interference caused by applications with similar performance characteristics. Results show that ML models outperform state-of-the-art regression based and threshold based interference detection techniques by at least 0.67% and at most 23.29%. Furthermore, our method incurs minimal overhead of only 1% to 2% on the service response time at runtime.

RQ-3: How well does our disambiguation method perform when the interfering workloads used for model training and deployment are not similar? This question covers a common case when we do not know a priori what applications might share the infrastructure with our managed application. To address RQ-3, we show that we can develop and train models to detect interference caused by an arbitrary application that stresses the same resources as used by the target application. In addition, this model generalizes well for detecting interference from different containerized interfering applications. Lastly, we evaluate scalability by using our model to detect interference against a large multi-tired microservice application. As presented in our results, our technique outperforms state-of-the-art regression based and threshold based interference detection techniques by at least 2.75% and at most 53.86%.

The paper makes the following contributions:

(1) We introduce a formal definition of cloud-native application interference disambiguation, rooted in queuing theory.
(2) We introduce an interference detection method that generalizes across interference scenarios and outperforms the state-of-the-art.
(3) We evaluate the method using several model types, including queuing and machine learning models.
(4) We validate the method on four industrial strength applications and show that it scales to large deployments and works in public clouds.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 details the motivation and methodology. Section 4 describes experimental setup. Section 5 presents the results of RQ-1, RQ-2, and RQ-3. Section 6 details threats to validity. Section 7 is the conclusion and discussion of future work.

2 RELATED WORK

2.1 Interference Definitions

Koh et al. [23] investigate the impact of VM-level performance interference from co-located workloads and characterize interference impact on low level system metrics. They cluster workloads by interference type and construct performance prediction models for co-located workloads using weighted means and regression analysis. Paul et al. [30] similarly characterize performance interference in co-located VMs and further measure the application performance degradation and impact on low-level system metrics. The authors highlight types of workloads that perform well or not when co-located. In contrast to these work, we characterize the impact of interference among containerized microservices.

2.2 Interference Measurement and Classification

Jha et al. [20] measured intra-microservice and inter-microservice interference using four benchmark microservices. The results of their experiments showed significant container interference present in both intra-container and inter-container cases. Garg, and Lakshmi. [17] ran experiments using well-known containerized benchmarks to detect microservice interference and accordingly identified the shared resources subject to contention in these scenarios. Novaković et al. [29] propose DeepDive to mitigate VM-level performance interference. Their approach detects interference by comparing low-level system metrics against a workload's baseline values and migrates VMs to other physical machines as needed. Wang et al. [36] present Vmon, a system that detects and quantifies VM-level performance interference. Vmon profiles an application running on a VM to observe how Hardware Performance Counters correlate with application performance. DeepDive and Vmons detect VM-level interference through use of low-level system metrics whereas our work detects microservice interference using metrics that are readily available and do not pose prohibitive overhead in collecting. Additionally, our work is differentiated in that we evaluate the effectiveness of model re-use across different scenarios.

2.3 Interference Regression

Joshi et al. [21] propose Sherlock, a method for detecting long-lived performance interference in containerized services caused by co-located VMs. The authors employ a regression detection technique to model performance interference using VM and application-level metrics at runtime. Kang and Lama [22] predict microservice interference using Gaussian Process models trained at runtime using a sliding window technique. Baluta et al. [12] predict microservice interference using AutoML models trained at runtime using a sliding window technique. Our work differentiates from these as we explore the reuse of pre-trained interference detection models in varying environments.

3 DEFINITIONS, MODELS AND METHODOLOGY

Assume that at deployment time, we profile the performance of an application and the interference effects in the space (U, λ, R), where U is the utilization, R is the response time and λ is the workload (arrival rate of requests) of the application and infrastructure. That profile is captured in a model and the profile is defined over a large workload space and interference levels.

In Figure 1, we illustrate interference in a simplified two-dimensional space, U and R. The blue area is the baseline profile for one workload ([λ=λ_1]). The response time of application, with no interference, falls within 95 percentile Confidence interval, denoted [$minR_{base}$ $maxR_{base}$]. With the help of tunable slack variable, α, we define a *operational zone* around the operational area (gray area). This parameter is application-dependent and defined by the application owner and can be derived from Service Level Agreements. Interference zones are those areas in the space (U, λ, R) where response time is outside the interval [$minR_{base}$-α, $maxR_{base}$+α] but cannot be explained through the change of the workload. For example, red areas in the figure are interference anomalies, whereas

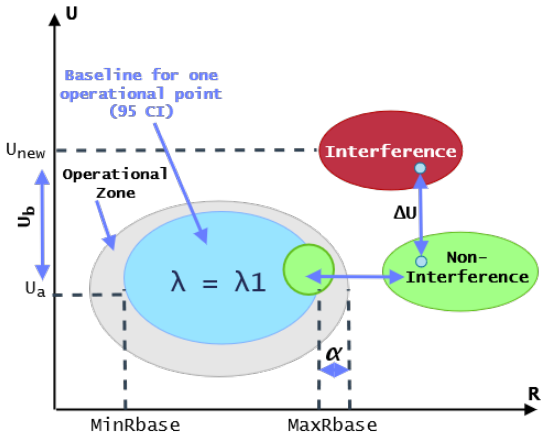

Figure 1: Interference Range

green areas depict normal behaviour, non-interference, caused by workload. Although the anomaly depicted in red can be explained by other causes (bugs, misconfigurations, etc.), in this paper we focus on interference anomalies. ΔU is the utilization step size between non-interference and what is classified as interference. Although both are outside the operational zone, it is important to know what causes the shift in the operation region. Different root causes of anomalies can be mitigated through different runtime actions.

In Section 3.1, we formulate interference as a Queuing Network problem. In Section 3.2, we motivate the use of Machine Learning to address interference. In Section 3.3, we describe our methodology including data collection and pre-processing, as well as model training and deployment.

3.1 Queuing Network Models and Interference Modeling

In this section, we explain how interference can be formulated as Queuing Network Models (QNM). To begin, we consider a containerized application a running inside a VM without interference, as seen in Figure 2. We assume that the application a stresses a single resource k in the VM so as to incur an utilization of $U_{a,k}$ on the resource. We assume a request arrival rate of λ_a to application a. If, at runtime, we had a way to measure service demand $D_{a,k}$ of application a at resource k, we could then predict the no-interference mean request response time R_a of application a as:

$$R_a = \frac{D_{a,k}}{1 - U_k} \tag{1}$$

where U_k is the total utilization incurred at resource k in the VM. Since in a no-interference scenario, the only application stressing resource k is our monitored application a, $U_k = U_{a,k}$. Finally, we substitute U_k with $U_{a,k}$ in eqn. 1 to obtain the response mean time R_a of application a in a no-interference environment as given by:

$$R_a = \frac{D_{a,k}}{1 - U_{a,k}} \tag{2}$$

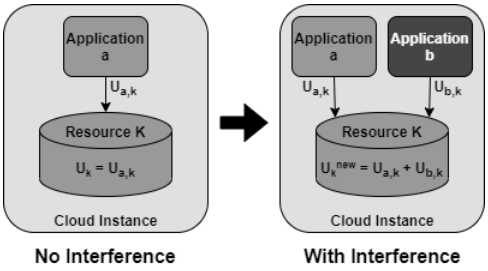

Figure 2: Effect of Interference on Resource Utilization

By considering the forced and utilization laws as well as steady state, we have

$$R_a = \frac{D_{a,k}}{1 - D_{a,k} * \lambda} \tag{3}$$

and this allows us to extrapolate the response time outside of an operational point and for any workload λ (cf. green areas in Figure 1).

To illustrate interference, consider another application b running inside the same VM, as seen in Figure 2. Application b incurs an utilization of $U_{b,k}$ on resource k inside the VM. Accordingly, from eqn. 1, the new response time R_a^{new} of application a when it faces interference from application b is:

$$R_a^{new} = \frac{D_{a,k}}{1 - U_k^{new}} \tag{4}$$

where U_k^{new} represents the total utilization of resource k, i.e. the sum total of the utilization incurred by both applications a and b at resource k given by:

$$U_k^{new} = U_{a,k} + U_{b,k} \tag{5}$$

Equations 2 and 5 explain interference (cf. Figure 1, red zones).

3.1.1 QNM Methodology. We tune QNM following the approaches in [14, 38]. Since we work under the assumption that we do not have access to the metrics of other applications and we consider the system is a product form network [13], we can use Eq. 3 to estimate the response time R and compare it with the measured response time. If the measured response time is outside the bounds [R-α, R+α] then we label it as affected by interference, otherwise we consider the deviation as caused by application load.

3.2 Machine Learning Classification and Interference Modeling

Based on eqn. 2 and eqn. 4, we observe that a point $[\lambda_a, R_a, U_k]$ is transposed to $[\lambda_a, R_a^{new}, U_k^{new}]$ in the same tri-dimensional space under the impact of interference. Based on this observation, we aim to use resource and application metrics to train ML models that can classify two distinct classes of response time values, one class denoting the no-interference scenario represented by R_a, and the other denoting the interference scenario represented by R_a^{new} (corresponding to red and green zones in Figure 1). We note that it appear more practical to use ML models instead of QNMs for interference detection since it is infeasible to accurately measure resource service demands for all containers belonging to a monitored microservice application serving different kinds of workloads. Furthermore, multiple levels of virtualization involved in a cloud-based microservice container can involve estimating service demands for unknown virtualized resources [27], which can be difficult. In contrast, ML models use resource utilization and throughput metrics that are easy to collect at the container and VM levels.

We use our understanding of QNMs as motivation for applying machine learning to detect interference. From eqn. 4, since the new response time R_a^{new} of application a is only impacted by the total utilization U_k incurred at resource k, it follows that R_a^{new} remains unchanged even if U_k is incurred by different types of applications as long as the levels of total utilization incurred at resource k are similar. Consequently, we aim to use this logic as motivation for training our ML models to predict interference classes with one type of benchmark application which can then be used at runtime to classify similar interference from other types of applications.

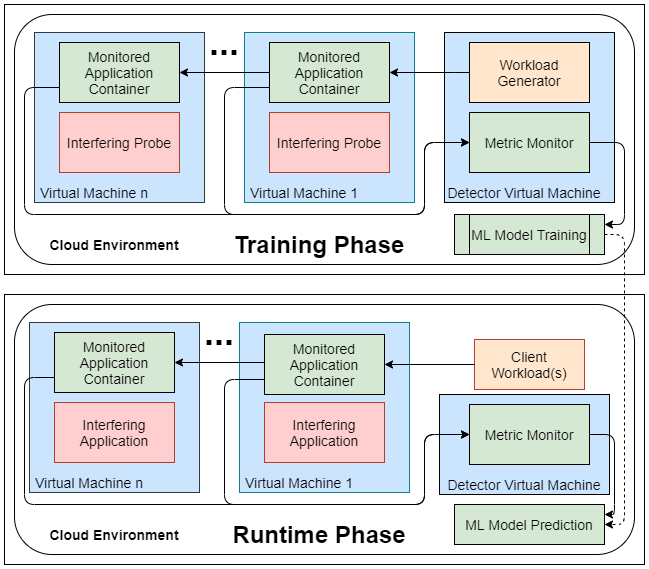

Figure 3: Overview of Interference Detection Technique

3.3 Machine Learning Methodology

In this section, we present our methodology focused on ML models since they are central to our work. Section 3.3.1 defines the assumptions of our technique. Section 3.3.2 details data generation and pre-processing. Section 3.3.3 describes the methodology to train an ML model for interference detection. Section 3.3.4 details the use of an ML model for interference detection at runtime.

3.3.1 General Methodology. We refer to the in-production application as the *target application*. The application owner deploys their target application on managed cloud services. In doing so, the application owner delegates management of physical servers or even VMs to the cloud provider. Consequently, the application owner only has guaranteed visibility into the target application's containers and access to VM level metrics. Our target application consists of multi-tiered microservices, each bundled inside a container. It is typical for microservice applications to be distributed over several VMs, with each VM hosting one or more microservice containers.

In our study, we consider an *interfering application* as one that runs on the same VMs as the target application and competes for shared VM resources.

Our detection technique monitors resource utilization data from inside the VM and the application containers metrics that are easy to collect. We record container utilization metrics at each target application container along with VM utilization metrics collected at a *sampling interval* of t seconds. We also record the overall application throughput and the response time, at each sampling interval. The sampling interval needs to be tuned for each target application so as to incur minimal levels of performance overhead on the system.

3.3.2 Data Collection and Pre-Processing for ML. Figure 3 presents the high level overview of our interference detection technique. As seen in the figure, our technique runs in two phases. The first phase is the *training phase* where we run controlled experiments to train our ML model in an offline model training environment. In the training phase, we run our target microservice application in a controlled cloud environment where its response time is not impacted by performance interference. This is done by running the target application in isolation on the n VMs hosting the microservice without any interfering application present. The objective is to obtain baseline metrics when no interference is present. To this end, we increase the arrival rate of the application workload to the target application in steps to incur a wide range of resource utilization observed at each application container. This is done through a *Workload Generator* tool running inside a Detector VM which resides alongside the application VMs, as seen in Figure 3. We ensure that our workload generation setup is free from internal software bottlenecks and network latency issues by following the approach detailed in past work [28]. Since application owners can only access metrics from the VM and their own application containers, we omit including metrics of any other kind in our data set for ML model training. To this end, a *Metric Monitor* tool from inside our Detector VM, as shown in Figure 3, collects measurable metrics from the target application and its environment such as application response time and throughput, and container and VM resource usage. Although we choose the average request response time of a target application as our performance metric, other metrics such as the mean throughput values can also be used. Using only the application throughput and response time along with VM and application resource utilization metrics as input allows for a smaller feature set that is easy to monitor, collect and does not incur prohibitive monitoring overhead at runtime.

We repeat each step of workload generation N times to obtain N estimates of the average application request response time R_{base} at each step. We refer to N as the *number of workload repetitions*. We use this data to construct the 95% Confidence Interval (CI) of the baseline application response time at each step. The upper and lower limits of this CI are denoted as $maxR_{base}$ and $minR_{base}$, respectively. To mitigate against performance variability inherent in public cloud platforms, we repeat each step, i.e. set the value of N in our training phase such that the width of our CI is within 5% of its sample mean at each step. Alternatively, if the cloud platform has significant performance variability, the application owner may consider deploying the application containers inside *dedicated VMs* which have same specification as general VMs and are offered by public cloud platforms to provide stable performance. Although dedicated VMs are more expensive than general VMs, the application owners only need to use them for a short period of time to collect training data for ML models.

Once we obtain the 95% CI of the application's baseline response time, we next introduce interference into the system. For this purpose, we run a controllable *interfering probe* along with the target application on the n VMs hosting the target application as shown in Figure 3. Doing so imposes additional stress on shared resources which is expected to increase response time as depicted in Equation 4. Similar to before, we send the same step-wise increasing workload to our target application from our workload generator tool from inside the Detector VM. We also simultaneously vary load on the interfering probe to diversify its degree of shared resource contention and introduce different levels of performance interference on our target application at each step. We record the average request response time R_t of the target application along with VM and container utilization metrics collected by the Metric Monitor tool when the interfering probe is also running. This is done at our sampling interval of t seconds continuously for a duration of x seconds to obtain the training data set to be used in the ML model. The application throughput, response times, and container and VM utilization metrics for each sampling interval represent a single record in the training data set of our ML model.

We use the baseline response time data obtained in the training phase to label the ML data set in our offline model training. As mentioned before, our ML model outputs binary classification with 2 labels, where label 1 indicates interference and label 0 indicates no-interference. To this end, we compare the value of R_t at each record in this set with its corresponding value of $maxR_{base}$ obtained at the same step of workload. As depicted in Figure 1, if R_t exceeds $maxR_{base}$, we infer that the application response time suffers from performance interference and accordingly assign the label 1 to the record. Otherwise, the record is assigned with label 0. Once the training data set is labeled, we remove the associated response time pairing from each record to construct the final input data set to the ML model.

3.3.3 ML Model Training. AutoML [15, 24, 32] has gained popularity as an effective solution for training well-performing ML models. The AutoML framework evaluates several ML models trained on the same data set against one another and outputs the best performing model. We leverage the H2O AutoML framework [24] in our ML model construction. This framework can be easily integrated with the DevOps feedback loop to automate runtime interference detection.

The framework trains multiple models including but not limited to XGBoost and Stacked Ensemble Models. An ensemble model is composition of several ML models and their predictions. The framework automatically tunes ML algorithm hyper-parameters using random grid search over a predefined range of possible hyper-parameter values. H2O AutoML iteratively evaluates models under these varying hyper-parameter configurations and outputs the best performing model. In addition, the H2O AutoML framework employs 5-Fold cross validation by default to promote models generalizing well to unseen data. We employ an 80-20 stratified train-test

set split while preserving class ratios. We evaluate our models by *Precision*, *Recall*, and F_1 *score*.

When a model makes positive predictions indicating interference as present, the precision score measures how often interference is truly present in the environment. Recall on the other hand measures how well a model identifies true instances of interference relative to all positive predictions made by the model. F_1 score is the harmonic mean of precision and recall and summarizes their joint behavior in a single metric.

3.3.4 ML Model Runtime Deployment. Once ML model construction is complete, we move on to the second phase, i.e., the *runtime phase*, where we deploy the ML model to detect container interference at runtime as seen in Figure 3. In this phase, we continuously monitor throughput, VM and container resource utilization at runtime as the microservice serves client traffic. Subsequently, this data is fed as input to the ML model, which in turn classifies the current runtime state of the target application as either 'interference' or 'no-interference'.

4 EXPERIMENTAL VALIDATION

The experiments are designed to answer research questions RQ-1 to RQ-3 for a range of deployments and interference types. We consider: a) multiple target and interference applications; b) different deployments for the target and interference applications; c) different types of interference. In this section, we show the design of the experiments and the implementation setup. Section 5 details the results of these experiments.

4.1 Target and Interference applications

In the experiments, we use 'in-production' applications such as ACME Air and Boutique as our Target application. Acme Air emulates transactions for an airline website and consists of 2 microservices, a NodeJS Web server, and a MongoDB database. Acme Air is a well known and frequently used benchmark [33, 34]. We run these 2 microservices in their own Docker containers. We refer to these containers as the *Acme-Web* and *Acme-Db* containers respectively. Online Boutique [6] is a popular open-source benchmark application developed by Google. The e-commerce application is composed of multiple microservices deployed on Kubernetes, an open-source container orchestration system.

As interference applications, we use stress-ng and Air Quality Monitor. Stress-ng [9] is a linux stress tool that allows the user to stress the CPU, memory, disk and other resources. We use stress-ng as a configurable artificial application benchmark which can be used to generate a wide range of resource utilization levels. Air Quality Monitor [2] is an Internet of Things (IoT) application that processes air quality sensor data.

4.2 Interference Types

We consider the following interference types:

(1) *Correlated and similar workloads (CSI).* In this use case, we consider that the interference is generated by applications in which the load at different VMs is *correlated* with the load of an ingress service/gateway). The interfering applications belong to the same class as the in-production application, therefore the workloads of the in-production and interfering application are *similar*. We consider e-commerce applications, like ACME, for both training and inference.
(2) *Correlated but dissimilar workloads (CDI).* Here, we consider that the interfering applications belongs to a different class than the in-production target application. We use ACME as the in-production application and the Air Quality Monitor application, [2] which is an Internet of Things (IoT) application, as the interference.
(3) *Non-correlated and dissimilar workloads (NDI).* We consider that interference can happen at any VM, independently (*uncorrelated*). Also, the interference is not similar to the load of the in-production application. We use stress-ng [9] for training and inference. Prior work [16] has used the stress-ng benchmark to incur varying level of resource utilization by stressing the system under study.

4.3 Deployment types

We want to show that we can distinguish the interference from normal load in multiple application configurations. For that we consider several deployment types for the in-production application.

4.3.1 Single Virtual Machine (1VM). In this deployment, we consolidate our in-production microservice application and the interfering load on the same VM. We run experiments where the interfering application is either 1.) stress-ng (NDI scenario), 2.) another copy of Acme Air (CSI scenario), or 3.) Air Quality Monitor application (CDI scenario). We use our workload generator tool to send a workload to Acme Air while also running our interfering application to incur a wide range of resource utilization and interference levels on our target application.

4.3.2 Dual Virtual Machine (2VM). Here, Acme Air application is distributed across two VMs. This is done to motivate distributed use cases where a microservice application like Acme Air is hosted in containers running across different VMs. In this scenario, we run the Acme-Web and Acme-Db containers on 2 separate VMs respectively. We run NDI, CSI, and CDI scenarios, similar to the single VM experiments. In each of these scenarios we run 1 interfering application container on each of the 2 VMs hosting the Acme-Web and Acme-Db containers respectively. Workloads to these two copies are varied to incur a wide range of resource utilization.

4.3.3 At-scale Deployments. To demonstrate the scalability of our methodology, we selected Online Boutique, a large 11-tier microservice e-commerce application. In our experimentation, we deploy Online Boutique on an Amazon EKS (Elastic Kubernetes Service) Cluster with 4 cluster nodes using an EC2 node group composed of EC2 m5.large VMs. Our Amazon EKS uses Amazon Linux 2 and Docker as the container runtime. We have a total of 14 Kubernetes pods of Online Boutique distributed between the 4 Cluster Nodes. For our interference application, we selected Acme-Air and stress-ng from our previous experiments. For the Acme-Air deployment, we distributed Acme-Air and MongoDB pods on two of our EKS nodes where the Boutique front-end pods reside. This deployment pattern is also repeated with two stress-ng pods. We have in total 9 pods responsible for monitoring our cluster at the container, node and application level.

4.4 Experimental Setup and Methodology

Our experiments were run in the AWS EC2 Cloud. We use multiple m5.large EC2 VMs residing in the same availability zone on the EC2 cloud platform. These VMs run Ubuntu 16.04 and each have 2 VCPUs, 8GiB of RAM, and 64GiB of Elastic Block Storage. We used Docker version 19.03.13 as the containerization platform on our EC2 VMs.

4.4.1 Workload Characteristics. We use *httperf* [26] as the Workload Generator tool to send workload to our target Acme Air application. The httperf client runs inside a separate VM and is not containerized. We use the *httperf* tool to submit the default workload obtained from the official Acme Air project [1] as the workload transaction mix submitted to Acme Air.

We use the number of concurrent connections and inter-request arrival time settings in *httperf* to drive a wide range of application resource utilization. Acme Air workload has a step size increase of approximately 12.5% CPU utilization. stress-ng has a step size increase of 20% CPU utilization. The Air Quality Monitor has a step size increase of 20% CPU utilization.

The workloads were chosen such that comparable Acme Air utilization levels are generated with and without interference present. Otherwise, lacking overlap between interference and non-interference scenarios, a simple utilization threshold algorithm could detect interference with ease. Next, we set the number of workload repetitions $N = 40$ to capture variance. Finally, for the purpose of this study, we set the workload duration $x = 120$ seconds.

For our at-scale deployment, we use Locust, a python-based workload generator [5], to generate the workload on Online Boutique. The workload generator increases the utilization on each of the nodes by 15% increments, ranging from 15% to 80% utilization on the Kubernetes Cluster. We use the same workloads of Acme Air and stress-ng in the previous experiments as the interference workload while the Boutique Workload is running for our experiments.

4.4.2 Metrics Monitoring. To monitor the applications and their environment, we leveraged prominent industry solutions for the *Metric Monitor*. Prometheus is an open-source monitoring solution that integrates with multiple metrics exporters [8]. Additionally, application metrics averages were output in httperf logs. The application metrics are joined with the container and VM metrics from Prometheus by timestamp. These metrics are used in analysis, data labelling, and ML model construction.

4.4.3 RQ-2 Setup. We conduct experiments to compare our interference detection technique with baseline and state-of-the-art techniques. To this end, we first set up experiments where we compare our technique against a simple utilization-threshold detection technique used in previous work [31]. This serves as a baseline technique where we monitor the resource utilization metrics from the Acme-Web and Acme-Db containers at runtime, and indicate the presence of interference if these metrics exceed pre-defined threshold values. We set the pre-defined thresholds of CPU and memory utilization in Acme-Web to 38% and 1.5%, and in Acme-Db to 18% and 38% respectively. We chose these pre-defined threshold values since we observed in earlier experimentation that at these utilization levels, the baseline response time of Acme Air without interference is 3 ms, which is the average baseline response time recorded over all utilization levels incurred by Acme Air. We compare the container utilization metrics of Acme to see if they exceed the pre-defined utilization thresholds. If so, interference is inferred. In addition to the threshold model, we evaluate Queuing Network Models to provide another baseline.

We also conduct another set of experiments to compare our interference detection technique with a state-of-the-art detection technique used in current research [21]. We chose this technique as a regression based approach commonly used in interference and anomaly detection. We adopt the detection method outlined in [21] to construct logistic regression models for interference detection at runtime. Logistical regression predicts the probability of occurrence of a binary classification by using a logistic function. The regression models are fit on the same datasets that were constructed and used for our ML approach as described in section 3.3.2. At runtime, we use the regression model in our single and dual VM experiment setups to predict whether interference is present or not in our Acme Air application. We compare the performance of our QNM and ML approach with the logistical regression model. We evaluate the Precision, Recall, and F_1 score when these baseline technique are applied in our experiments.

4.4.4 RQ-3 Setup. We run experiments to evaluate the effectiveness of ML models in detecting performance interference when the interfering application class at runtime is different from training time. State-of-the-art techniques considered are the same Regression and Threshold based techniques as introduced in the RQ-2 Setup. QNM models are omitted as they explicitly define interfering applications whereas in these experiments we assume no knowledge of the interfering application class at runtime. To train our ML models, we use the same methodology as described in section 3.3.3. For these environments, we choose one of our three applications to serve as the interfering probe benchmark application in the ML model training. These three applications correspond to the NDI, CSI, and CDI scenarios. Next, we deploy these trained ML models in the runtime phase as described in section 3.3.4. However, at runtime, we evaluate the resultant ML models where a different interfering application is present in the environment than as seen at training time. Accordingly, a different interference scenario is encountered at runtime than training time. The goal is to evaluate the model performance against an interfering application class not yet seen by the ML model. To measure the effectiveness of our generalized ML model, we evaluate our method against state-of-the-art models of the logistic regression and simple threshold techniques described in the RQ-2 Setup. The logistic regression model follows a similar methodology as our ML model in that the interference scenario encountered at runtime is different than that used at training time.

5 RESULTS AND DISCUSSION

In this section, we evaluate the interference scenarios for our research questions and present our findings.

5.1 RQ-1 Results: How Severe is the Impact of Interference?

5.1.1 Single VM Experiments Results Analysis. Tables 1, 2, and 3 shows the target Acme Air application's utilization values along

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	14.36%	1.31%	7.56%	41.18%	1.33	1.34	0.75%
469.4	28.66%	1.35%	13.65%	41.83%	1.91	1.93	1.05%
631.49	37.68%	1.39%	18.09%	41.99%	2.58	2.87	11.24%
937.03	53.66%	1.51%	27.88%	42.00%	9	12.53	39.22%

Table 1: NDI Interference in Single VM

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	17.93%	1.25%	9.19%	26.65%	1.22	2.66	118.03%
469.29	34.15%	1.41%	16.90%	27.35%	1.78	24.74	1289.89%
631.48	41.80%	1.44%	21.99%	27.66%	2.4	104.13	4238.75%
673.49	45.95%	1.49%	23.18%	28.01%	7.35	518.53	6954.83%

Table 2: CSI Interference in Single VM

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	15.38%	1.43%	10.04%	20.56%	2.38	8.1	240.34%
469.39	28.08%	1.32%	17.25%	20.64%	2.98	31.83	968.12%
631.4	37.19%	1.40%	23.28%	20.86%	4.16	135.66	3161.06%
938.97	54.04%	1.50%	36.50%	20.78%	16.48	590.88	3485.44%

Table 3: CDI Interference in Single VM

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
469.39	26.03%	1.56%	11.82%	45.20%	1.11	1.81	63.06%
631.48	34.83%	1.61%	15.56%	45.39%	1.57	2.80	78.34%
937	48.88%	1.66%	21.90%	45.39%	3.02	12.23	304.97%
1099.85	54.63%	1.66%	24.29%	45.78%	5.66	24.10	325.80%

Table 4: NDI Interference in Dual VM

the impact of interference on the average request response time of Acme Air. Accordingly, Web CPU and Web Mem refer to the target Acme Air's Web server. Similarly, DB CPU and DB Mem refer to the target Acme Air's Database server. R_{base} and R_t refer to the baseline response time of our target Acme Air application without interference and the runtime response time of Acme Air with interference respectively. As seen in the tables, depending on the resource utilization and workload levels, the average response time of Acme Air is heavily impacted by different levels of performance interference. Even at comparatively light interference levels, the response time of Acme Air more than doubles, as seen in the first row in Table 2.

5.1.2 Dual VM Experiments Results Analysis. Tables 4, 5 and 6 shows the impact of interference on the average request response time of Acme Air in dual VM scenarios. For the dual VM NDI Scenario, the impact of interference on the average response time of Acme Air can be significant. In the worst case, the average Acme Air response time increases from 5.66 ms to 24.1 ms, a 325.8% increase. For the dual VM CSI Scenario, the response time of Acme degrades significantly when running alongside a second copy of the Acme application. In the worst case scenario, the response time of Acme increases from 3.46 ms to 202.47 ms. For the dual VM CDI scenario, Acme's response time degrades significantly in this scenario as well.

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	17.52%	1.31%	9.81%	33.70%	1.11	2.46	121.62%
469.34	33.16%	1.44%	17.60%	33.94%	1.33	12.91	870.68%
631.5	41.92%	1.49%	22.22%	34.28%	1.55	28.47	1736.77%
907.29	57.11%	1.56%	30.12%	34.67%	3.46	202.47	5751.73%

Table 5: CSI Interference in Dual VM

Req/s	Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	R% Change
238.9	14.18%	1.42%	9.35%	26.19%	1.24	5.46	340.32%
469.4	25.98%	1.33%	15.22%	26.44%	1.23	13.67	1011.38%
631.5	34.42%	1.38%	20.74%	26.52%	1.45	26.68	1740.00%
939.21	49.55%	1.49%	30.97%	26.54%	2.32	90.22	3788.79%

Table 6: CDI Interference in Dual VM

In the worst case, the response time of Acme increases from 2.32 ms to 90.22 ms.

RQ-1: Through extensive experimentation, we characterize interference through its impact on container utilization metrics. Results show significant impact of interference on the response time of in-production applications for all interference types considered. For both single and dual VM deployments, response time degrades by at least a factor of 39% and at most a factor of 6955% when the normal CPU utilization levels are between 40% and 60%. We also conclude that the higher the utilization, the higher the impact of interference.

5.2 RQ-2 Results: When Interfering Workloads for Model Training and Deployment are Similar

In this section, we evaluate the performance of ML, QNM and state-of-the-art models to detect interference when the interfering application load is known in the training phase. The applications used for the interfering workloads are the same for both model training and deployment for the RQ-2 results. This is indeed an ideal case, however, it is important in defining a baseline. We compare the effectiveness of three different approaches for all our application scenarios; Regression, QNM and ML.

Figure 4 presents a F_1 score box-plot for ML approach compared to the other state-of-the-art approaches described in Section 4.4.3. On the box-plot, the horizontal bar represents the median F_1 scores across all scenario types and VM deployments. The higher the median, the shorter the box, the better the performance. The Threshold model has the lowest performance and the Regression model has high variability of F_1 scores, showing that the Regression has difficulty consistently detecting interference across different scenarios and deployment types. We performed the ANOVA test on our data to to confirm the significant differences between the approaches (f-value = 16.343801, p-value = 0.000013). Overall, we observe from the box-plot that our ML approach outperforms all state-of-the-art methods with the highest F_1 scores and lowest variability.

Table 7 captures a closer look on the details of the performance of ML approach and state-of-the-art techniques described in Section 4.4.3. The Threshold and QNM approaches performs the worst in all

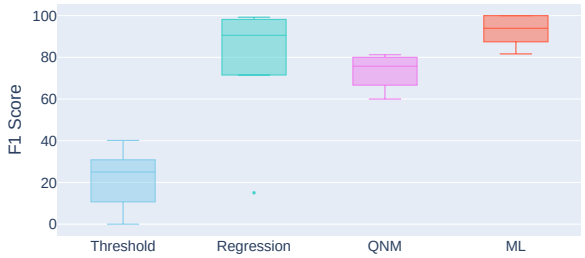

Figure 4: Performance of ML versus State-of-the-Art Models

Scenario	Approach	F_1 Score	Precision	Recall
NDI in 1VM	Threshold	30.75%	22.69%	47.69%
NDI in 1VM	Regression	15.04%	40.0%	9.26%
NDI in 1VM	QNM	60.0%	46.15%	85.71%
NDI in 1VM	**ML**	**81.64%**	**100.0%**	**68.98%**
CSI in 1VM	Threshold	16.67%	87.5%	9.21%
CSI in 1VM	Regression	97.99%	100.0%	96.05%
CSI in 1VM	QNM	80.0%	80.0%	80.0%
CSI in 1VM	**ML**	**98.67%**	**100.0%**	**97.37%**
CDI in 1VM	Threshold	0.00%	0.00%	0.00%
CDI in 1VM	**Regression**	**89.59%**	**92.79%**	**86.6%**
CDI in 1VM	QNM	81.25%	100.0%	68.42%
CDI in 1VM	ML	87.46%	100.0%	77.71%
NDI in 2VM	Threshold	41.94%	46.43%	38.24%
NDI in 2VM	Regression	73.85%	77.42%	70.59%
NDI in 2VM	QNM	66.67%	66.67%	66.67%
NDI in 2VM	**ML**	**97.14%**	**94.44%**	**100.0%**
CSI in 2VM	Threshold	40.13%	89.85%	25.84%
CSI in 2VM	Regression	99.24%	99.27%	99.21%
CSI in 2VM	QNM	77.78%	77.78%	77.78%
CSI in 2VM	**ML**	**99.91%**	**99.82%**	**100.0%**
CDI in 2VM	Threshold	19.27%	86.45%	10.84%
CDI in 2VM	**Regression**	**91.5%**	**95.44%**	**87.87%**
CDI in 2VM	QNM	73.68%	87.5%	63.64%
CDI in 2VM	ML	87.97%	100.0%	78.52%

Table 7: Performance of ML vs. State-of-the-Art Models

the scenarios. The Regression approach overall has a significantly better performance compared to the Threshold and QNM in the CSI and CDI scenarios. However, the Regression model showed significant performance degradation in the NDI scenario with a F_1 score of 15.04 %, being on par with the Threshold model. Compared to the Regression model F_1 score in all the other scenarios in both VM deployments, this can be considered an outlier. The poor performance of the QNM model when detecting performance interference might be impacted by the lack of precision in estimating alpha and assuming ideal arrival rates and service time distributions. The ML approach outperforms all the models in each of the scenarios. The Regression model does outperform out ML approach in the CDI scenario by 2.13 % and 3.53 %. However, the ML approach is more consistent in performance with a high F1-score, Precision and Recall across all the scenarios, unlike the Regression approach. This demonstrates that the ML approach can consistently classify

Scenario	Approach	F_1 Score	Precision	Recall
NDI in 4VM	QNM	40.45%	34.05%	49.81%
NDI in 4VM	Regression	85.71%	81.03%	90.98%
NDI in 4VM	**ML**	**99.12%**	**99.25%**	**99.00%**
CSI in 4VM	QNM	42.5%	36.31%	51.24%
CSI in 4VM	Regression	90.14%	87.48%	92.98%
CSI in 4VM	**ML**	**99.17%**	**99.0%**	**99.26%**
CDI in 4VM	QNM	0%	0%	0%
CDI in 4VM	Regression	97.57%	95.82%	99.38%
CDI in 4VM	**ML**	**99.79%**	**99.79%**	**99.79%**

Table 8: Boutique subject to Interference

interference with high accuracy on a variety of different types of applications and deployments.

Table 8 show the F_1 score, Precision and Recall for the Boutique deployment experiments. Since the Threshold model performance was the lowest of the models, we omitted it from our experiments. The F_1 score for the ML approach has less variability than the regression method which shows that the ML approach is more consistently able to detect performance interference for large at-scale deployments. Our QNM model performed the worst when detecting interference for at-scale deployments. Specifically, the CDI scenario demonstrates the limitation of the QNM approach and it's inflexibility in handing variability and incomplete metrics which caused the QNM approach unable to classify interference. Overall our ML approach outperforms the other approaches from 2.22% up to 13.41%.

Scenario	Best ML Model	F_1 Score	Precision	Recall
NDI in 1VM	GBM	81.64%	100.0%	68.98%
NDI in 2VM	Ensemble	97.14%	94.44%	100.0%
CSI in 1VM	Ensemble	98.67%	100.0%	97.37%
CSI in 2VM	Ensemble	99.91%	99.82%	100.0%
CDI in 1VM	GBM	87.46%	100.0%	77.71%
CDI in 2VM	GBM	87.97%	100.0%	78.52%

Table 9: Top Performing ML Model Per Scenario

Interference Base Utilization	Interference Step Size	F_1 Score	Precision	Recall
20 %	10 %	97.95%	100%	96%
10 %	6 %	91.30%	100%	84%
6 %	2 %	89.36%	91.30%	87.50%

Table 10: ML Accuracy with Different Interference Step Size

As the ML approach has the best overall performance, we then investigated the step size of the workload and it's effect on the performance of the ML approach. Since the same workload is used across all the experiments, we investigate how the F_1 score of the ML approach are affected with different workload step sizes. Table 10 shows differing workloads of the interfering application with the Base Utilization as the starting point of the interfering application and its respective step size. A higher base utilization and step size increases causes the ML model to classify the interference with more ease. As the step size of the workload is reduced to smaller

Env	Training / Runtime Scenarios	Approach	F_1 Score	Precision	Recall
1VM	NDI / CSI	Threshold	10.3%	94.15%	5.45%
1VM	NDI / CSI	Regression	21.91%	1.0%	12.31%
1VM	**NDI / CSI**	**ML**	**75.77%**	**99.26%**	**61.27%**
1VM	NDI / CDI	Threshold	0.0%	0.0%	0.0%
1VM	NDI / CDI	Regression	53.8%	1.0%	36.8%
1VM	**NDI / CDI**	**ML**	**82.77%**	**78.18%**	**87.93%**
1VM	CSI / CDI	Threshold	0.0%	0.0%	0.0%
1VM	CSI / CDI	Regression	87.46%	83.68%	91.61%
1VM	**CSI / CDI**	**ML**	**90.21%**	**82.17%**	**100.0%**
2VM	NDI / CSI	Threshold	39.88%	88.04%	25.78%
2VM	**NDI / CSI**	**Regression**	**91.81%**	**93.91%**	**89.8%**
2VM	NDI / CSI	ML	89.53%	98.5%	82.1%
2VM	NDI / CDI	Threshold	17.8%	85.43%	9.93%
2VM	NDI / CDI	Regression	78.63%	93.3%	67.95%
2VM	**NDI / CDI**	**ML**	**82.62%**	**93.86%**	**73.79%**
2VM	CSI / CDI	Threshold	17.8%	85.43%	9.93%
2VM	CSI / CDI	Regression	90.98%	84.41%	98.67%
2VM	**CSI / CDI**	**ML**	**94.37%**	**91.68%**	**97.22%**

Table 11: Evaluation of Training/Runtime Interference

values, the F1-score decreases. However even at very minimum step size increases of 1.5 - 3 %, the ML approach has an 89.36 % F1-Score with relatively high precision and recall when classifying interference.

Finally, we report the ML models with highest F_1 Score for both the single and dual VM experiments in Table 9. Scenarios represented in this table are described by the number of VMs in the environment as well as the characteristics of interfering application. As seen in the table, GBM models performs best for interference detection in all single and dual VM CSI scenarios. In the dual VM NDI scenario, a stacked ensemble model performs best. In all CDI scenarios, a stacked ensemble model also performed the best.

RQ-2: We evaluated interference detection techniques on different deployment and interference types where the interfering load is the same in the training and runtime phases. ML models outperform logistic regression, QNM and simple threshold techniques in each of our evaluation experiments in F_1 score by at least 0.67% and at most 23.29%. The high ML performance is maintained at-scale deployments and we can conclude ML models can better handle the variability of the cloud.

5.3 RQ-3 Results: When Interfering Workloads for Model Training and Deployment are Different

Cloud environments of scale are subject to frequent change. Containerized microservices may be co-located and scaling actions may introduce additional containers that in turn stress the underlying VMs. It is impractical to train an interference model for every possible deployment combination. Accordingly, we attempt to construct a ML model that performs well when our target application is subject to a different interference scenario in the runtime phase as opposed to its training phase.

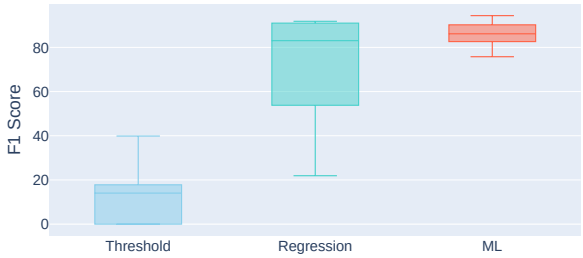

Figure 5: Evaluation for Unknown Interference

Training / Runtime Scenarios	Approach	F_1 Score	Precision	Recall
NDI / CSI	Regression	86.19%	88.54%	83.96%
NDI / CSI	**ML**	**98.44%**	**98.69%**	**98.18%**
NDI / CDI	Regression	86.05%	99.19%	75.99%
NDI / CDI	**ML**	**99.10%**	**99.50%**	**98.72%**
CSI / CDI	Regression	80.18%	99.57%	67.12%
CSI / CDI	**ML**	**97.64%**	**99.48%**	**95.87%**
NDI + CDI / CSI	Regression	83.20%	94.25%	74.46%
NDI + CDI / CSI	**ML**	**97.11%**	**99.19%**	**95.12%**
NDI + CSI / CDI	Regression	80.72%	99.58%	67.86%
NDI + CSI / CDI	**ML**	**99.65%**	**99.47%**	**99.83%**

Table 12: Boutique Training/Runtime Interference

Table 11 and Figure 5 shows our results from the experiments conducted in Section 4.4.4. Table 11 denotes the interference scenario encountered at training time as well as the interfering scenario present at runtime. In this way the ML models were evaluated in scenarios where the interference scenario is unknown and previously unseen. Notably, ML outperformed state-of-the-art methods in both single VM and dual VM environments. In each scenario, ML obtained the highest F_1 score when compared to state-of-the-art techniques. As seen in Figure 5, the ML approach consistently able to classify interference and significantly outperformed the ML and Regression. We performed the ANOVA test on the data to to confirm the significant differences between the approaches (f-value = 24.650731, p-value = 0.000018).

It's also notable that the models trained in CSI scenarios and tested at runtime with CDI interference resulted in a better F_1 score than the models with NDI interference at training time. Models where the interference scenario is the same at training and runtime perform substantially better over their counterparts where interference scenarios are different. However, using a model trained for use with different interference scenarios does not suffer from long pre-deployment or training phases.

For the large at-scale deployment, we compared the highest performing approaches in the previous experiments. When the interfering application is unknown with large at-scale deployments, Table 12 shows the Regression approaches begin to struggle in detecting the performance interference. Our ML approach outperforms the F_1 score of the regression up to 20.57%.

RQ-3: We evaluated ML versus the state-of-the-art interference detection techniques where the interference scenario is different in the training and runtime phases. The ML models outperform state-of-the-art techniques in F_1 score by at least 2.75 % and at most 53.86 % for all but one interference types and different deployments. We can conclude that training with interfering applications yields better performance than if we train with random interference (NDI with stress-ng). The performance of the ML approach (F_1 score), although lower than in the case of RQ-2, is high enough and scales well so it can be applied in production.

6 THREATS TO VALIDITY

We identify threats to validity of our work as per Wohlin et al [37]. We note as an external threat that our ML model training approach might not detect interference from unknown interfering applications well if the resource utilization signature of the benchmark interfering application is significantly different from the interfering application used at runtime. If the interfering application used in the training phase and the interfering application at runtime stresses different VM-level resources, our technique may be unable to classify interference. Furthermore, as another external threat, in multi-VM deployment strategies frequently used for microservice deployment, if the VM characteristics at runtime change relative to what was used in the training phase, our ML models may not be successful and will need to be re-trained.

7 CONCLUSIONS AND FUTURE WORK

In this work, we propose a runtime performance interference detection technique that leverages Machine Learning. Our ML models are trained on microservice resource utilization metrics subject to varying environments and different interfering applications. Our approach does not require expensive service instrumentation and does not pose prohibitive monitoring overhead. Our proposed technique is effective in detecting performance interference for a realistic microservice benchmark running on the EC2 cloud platform and also outperforms baseline and state-of-the-art interference detecting techniques. Our ML models are also effectively used in varying cloud environments where the interference characteristics change at runtime. In the future, we plan to investigate model maintenance over time with respect to an application's behavioural drift through continuous learning or time series techniques. Furthermore, we will integrate our ML Models into an AIOps platform that takes deployment actions to mitigate performance interference.

REFERENCES

[1] [Online]. Acme Air. https://github.com/acmeair
[2] [Online]. Air Quality Monitor. https://github.com/jlofw/air-quality-monitor
[3] [Online]. *Amazon Web Services.* https://aws.amazon.com/
[4] [Online]. *Google Cloud.* https://cloud.google.com/
[5] [Online]. *Locust.* https://locust.io/
[6] [Online]. *Online Boutique.* https://github.com/GoogleCloudPlatform/microservices-demo
[7] [Online]. *Production-Grade Container Orchestration.* https://kubernetes.io/
[8] [Online]. *Prometheus.* https://prometheus.io/
[9] [Online]. *Stress-ng.* https://kernel.ubuntu.com/~cking/stress-ng/
[10] Sandip Agarwala, Yuan Chen, Dejan Milojicic, and Karsten Schwan. 2006. QMON: QoS-and utility-aware monitoring in enterprise systems. In *2006 IEEE International Conference on Autonomic Computing*. IEEE, 124–133.
[11] Yasaman Amannejad, Diwakar Krishnamurthy, and Behrouz Far. 2015. Detecting performance interference in cloud-based web services. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 423–431.
[12] Alexandru Baluta, Joydeep Mukherjee, and Marin Litoiu. 2022. Machine Learning based Interference Modelling in Cloud-Native Applications. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 125–132.
[13] R.J. Boucherie and N.M. van Dijk. 2011. *Queueing Networks: A Fundamental Approach*. Springer US. https://books.google.ca/books?id=C98YswEACAAJ
[14] Eli Brookner. 1998. *Tracking and Kalman filtering made easy*. Wiley New York.
[15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 2755–2763.
[16] Ruoyu Gao and Zhen Ming Jiang. 2017. An exploratory study on assessing the impact of environment variations on the results of load tests. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 379–390.
[17] Surya Kant Garg and J Lakshmi. 2017. Workload performance and interference on containers. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 1–6.
[18] Vojtěch Horký, Jaroslav Kotrč, Peter Libič, and Petr Tůma. 2016. Analysis of Overhead in Dynamic Java Performance Monitoring. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2851553.2851569
[19] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-Hosted Web Applications. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 469–478. https://doi.org/10.1145/3038912.3052649
[20] Devki Nandan Jha, Saurabh Garg, Prem Prakash Jayaraman, Rajkumar Buyya, Zheng Li, and Rajiv Ranjan. 2018. A holistic evaluation of docker containers for interfering microservices. In *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 33–40.
[21] Kartik Joshi, Arun Raj, and Dharanipragada Janakiram. 2017. Sherlock: Lightweight detection of performance interference in containerized cloud services. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 522–530.
[22] Peng Kang and Palden Lama. 2020. Robust Resource Scaling of Containerized Microservices with Probabilistic Machine learning. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 122–131.
[23] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. 2007. An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 200–209.
[24] Erin LeDell and Sebastien Poirier. 2020. H2O AutoML: Scalable Automatic Machine Learning. *7th ICML Workshop on Automated Machine Learning (AutoML)* (July 2020). https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf
[25] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014).
[26] David Mosberger and Tai Jin. 1998. Httperf—a Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev.* 26, 3 (Dec. 1998), 31–37. https://doi.org/10.1145/306225.306235
[27] Joydeep Mukherjee, Alexandru Baluta, Marin Litoiu, and Diwakar Krishnamurthy. 2020. RAD: Detecting Performance Anomalies in Cloud-based Web Services. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE,

493–501.

[28] Joydeep Mukherjee and Diwakar Krishnamurthy. 2018. Subscriber-driven cloud interference mitigation for network services. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–6.

[29] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA) *(USENIX ATC'13)*. USENIX Association, USA, 219–230.

[30] Indrani Paul, Sudhakar Yalamanchili, and Lizy K John. 2012. Performance impact of virtual machine placement in a datacenter. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*. IEEE, 424–431.

[31] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. 2012. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*. 285–294. https://doi.org/10.1109/ICDCS.2012.65

[32] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 847–855. https://doi.org/10.1145/2487575.2487629

[33] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE, 1–10.

[34] Yohei Ueda and Moriyoshi Ohara. 2017. Performance competitiveness of a statically compiled language for server-side Web applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–22.

[35] Fotios Voutsas, John Violos, and Aris Leivadeas. 2023. Filtering alerts on cloud monitoring systems. In *2023 IEEE International Conference on Joint Cloud Computing (JCC)*. IEEE, 34–37.

[36] Sa Wang, Wenbo Zhang, Tao Wang, Chunyang Ye, and Tao Huang. 2015. Vmon: Monitoring and quantifying virtual machine interference via hardware performance counter. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 2. IEEE, 399–408.

[37] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen. 2000. *Experimentation in Software Engineering*. Kluwer Academic Publishers.

[38] Tao Zheng, C. Murray Woodside, and Marin Litoiu. 2008. Performance Model Estimation and Tracking Using Optimal Filters. *IEEE Transactions on Software Engineering* 34, 3 (2008), 391–406. https://doi.org/10.1109/TSE.2008.30