MACHINE LEARNING INTERFERENCE MODELLING FOR CLOUD-NATIVE APPLICATIONS

ALEXANDRU BĂLUŢĂ

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

YORK UNIVERSITY
TORONTO, ONTARIO

NOVEMBER 2021

Abstract

Modern cloud-native applications use microservice architecture patterns, where fine granular software components are deployed in lightweight containers that run inside cloud virtual machines. To utilize resources more efficiently, containers belonging to different applications are often co-located on the same virtual machine. Co-location can result in software performance degradation due to interference among components competing for resources. In this thesis, we propose techniques to detect and model performance interference. To detect interference at runtime, we train Machine Learning(ML) models prior to deployment using interfering benchmarks and show that the model can be generalized to detect runtime interference from different types of applications. Experimental results in public clouds show that our approach outperforms existing interference detection techniques by 1.35%-66.69%. To quantify the intereference impact, we further propose a ML interference quantification technique. The technique constructs ML models for response time prediction and can dynamically account for changing runtime conditions through the use of a sliding window method. Our technique outperforms baseline and competing techniques by 1.45%-92.04%. These contributions can be beneficial to software architects and software operators when designing, deploying, and operating cloud-native applications.

Co-Authorship

This thesis is based on my published work as detailed below. My contributions to these publications consisted of: proposing research ideas and direction, conducting literature reviews, constructing experimental setups, designing experiments, collecting data, performing data analysis and modelling, and serving as the primary author in writing the publications themselves.

The publications are listed as follows:

1. **Alexandru Baluta**, Joydeep Mukherjee, Zhen Ming Jiang, Marin Litoiu, Machine Learning based Interference Detection in Cloud-Native Applications: *Submitted to IEEE Transactions on Cloud Computing*

2. **Alexandru Baluta**, Joydeep Mukherjee, Marin Litoiu, Machine Learning based Interference Modelling in Cloud-Native Applications: *Submitted to 13th ACM/SPEC International Conference on Performance Engineering*

Acknowledgements

I would like to sincerely thank my supervisor, Professor Marin Litoiu for all his guidance and mentorship. His insight in the fields of Machine Learning, Software Engineering, and Adaptive Systems was instrumental in expanding my knowledge and realizing my thesis. I am greatly appreciative of the support and counsel of Professor Joydeep Mukherjee. His support enabled me to continuously improve my skills and research output. I would also like to thank Professor Zhen Ming (Jack) Jiang for the invaluable guidance and feedback in my research works. I would like to express my gratitude to the team members I got to work alongside at CERAS labs. Thank you to Yar Rouf, Raphael Rouf, Kim Long Ngo, Calin Armenean, Christianne Huber, and Farzin Zaker. Our collaborations and knowledge sharing were of great benefit.

I would like to dedicate this work to my friends and family that supported me in my graduate studies. My success was made possible by your unending support and encouragement.

Table of Contents

Abstract ii

Co-Authorship iii

Acknowledgements iv

Table of Contents v

List of Tables vii

List of Figures viii

1 Introduction 1
- 1.1 Research Questions . . . 4
- 1.2 Contributions . . . 6

2 Related Work 8
- 2.1 Performance Interference Detection . . . 8
- 2.2 Performance Interference Quantification . . . 9
- 2.3 Performance Interference Mitigation . . . 11
  - 2.3.1 Container Placement . . . 11
  - 2.3.2 Virtual Machine Placement . . . 12

3 Performance Interference Detection 13
- 3.1 Methodology . . . 13
  - 3.1.1 AIOps and Interference Detection . . . 14
  - 3.1.2 Interference as a Classification Problem . . . 15
  - 3.1.3 General Methodology . . . 18
  - 3.1.4 Data Collection and Pre-Processing for ML . . . 20
  - 3.1.5 ML Model Training . . . 22
  - 3.1.6 ML Model Runtime Deployment . . . 24
- 3.2 RQ-1: Performance Interference Characterization . . . 24
  - 3.2.1 Experiment Setup . . . 24
  - 3.2.2 Results Analysis . . . 29
- 3.3 RQ-2: ML for Known Interfering Applications . . . 31
  - 3.3.1 Experiment Setup . . . 32

3.3.2 Results Analysis . . . 33
3.4 RQ-3: ML for Unknown Interfering Applications . . . 37
3.4.1 Experiment Setup . . . 38
3.4.2 Results Analysis . . . 39
3.5 Threats to Validity . . . 40
3.6 Summary . . . 40

4 Performance Interference Quantification **42**
4.1 Methodology . . . 42
4.1.1 Static Models for Interference . . . 43
4.1.2 Runtime Models for Interference . . . 50
4.2 General Experiment Setup . . . 53
4.3 RQ-4: Static ML for Known Interfering Applications . . . 56
4.3.1 Experiment Setup . . . 56
4.3.2 Results Analysis . . . 57
4.4 RQ-5: Static ML for Unknown Interfering Applications . . . 60
4.4.1 Experiment Setup . . . 60
4.4.2 Results Analysis . . . 61
4.5 RQ-6: Runtime ML . . . 61
4.5.1 Experiment Setup . . . 62
4.5.2 Results Analysis . . . 63
4.5.3 Comparative Analysis and Practical Considerations . . . 63
4.6 Threats to Validity . . . 65
4.7 Summary . . . 66

5 Conclusions and Future Work **68**

Bibliography **76**

List of Tables

3.1 Details of ML Modeling used by the H2O AutoML Framework 19
3.2 Acme Air + stress-ng Interference in Single VM 30
3.3 Acme Air + Acme Air Interference in Single VM 31
3.4 Acme Air + IoT Interference in Single VM 31
3.5 Acme Air + stress-ng Interference in Dual VM 32
3.6 Top Performing ML Model Per Dataset 33
3.7 Acme Air + Acme Air Interference in Dual VM 34
3.8 Acme Air + IoT in Dual VM 34
3.9 Evaluation of Competing Model 35
3.10 Sensitivity Analysis with stress-ng Interference 36
3.11 Evaluation for Unknown Interference 38

4.1 Observed Resource Utilization Ranges 57
4.2 Observed Response Time Ranges 58
4.3 Static Model with Known Interfering App 59
4.4 Static Model with Unknown Interfering App 62
4.5 Runtime ML vs. Runtime GP and Static LQM 64
4.6 Comparison of Static and Runtime ML 65

List of Figures

3.1 AIOps and Interference Detection 14
3.2 Effect of Interference on Resource Utilization 16
3.3 Overview of Interference Detection Technique 18

4.1 Overview of Interference Quantification Training Phase 43
4.2 Overview of Interference Quantification Runtime Phase 44
4.3 Overview of Runtime Modelling Interference Quantification 49
4.4 Overview of Model Manager 50

Chapter 1

Introduction

DevOps is an emerging paradigm that integrates the development and operations teams to enable fast and efficient continuous delivery of software [1]. 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 cloud instance. 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 [2] and Kubernetes [3] are increasingly gaining popularity and are commonly used in public cloud platforms such as Amazon Web Services (AWS) [4] and Google Cloud Platform [5].

Multiple microservices are generally consolidated on a single cloud instance. This is done to improve resource consumption levels inside a cloud instance and thereby optimize cloud costs. When multiple microservices are deployed on the same instance, they can often compete with each other for shared host-level resources. Such shared resource contention can

alter the application behavior and make it deviate from the development time specifications and performance. This interference has been observed before in applications running on public cloud environments [6–13]. Interference can be also generated by malicious code or applications running concurrently along with legitimate applications [14]. It is therefore important that an application detect, quantify, and mitigate when its execution environment is perturbed by other malicious applications.

Detecting runtime interference in cloud-native applications is a challenging task. Cloud providers typically do not provide any built-in support for interference detection. Researchers [15] have used application instrumentation in the past to record mean request response times for the different services consolidated on the same instance. These runtime service response times were then compared against baseline service response times for significant deviations to detect the presence of interference. However, past research has shown that continuously instrumenting application code and monitoring the runtime response times of services [16] can significantly increase the cost of development and maintenance. In addition, continuous monitoring of service response time can incur a prohibitive overhead when the service is facing a heavy workload [17, 18]. Furthermore, interference detection techniques that model the containerized system environment may not generalize well to scenarios where the interfering application changes at runtime. Finally, since the incoming workload to services typically vary at runtime, it is often difficult to interpret by response time monitoring alone if the increase in response time is due to interference or workload fluctuation.

In light of the above challenges, we propose an interference detection technique that uses Machine Learning (ML) models to classify interference in cloud-native applications. Our ML based interference detection approach is non-intrusive and does not require extensive application modification. We use resource utilization metrics that can be easily collected at runtime from within each application container and the cloud instance along with simple application level metrics such as request throughput as the input to our ML model. Collecting these metrics is lightweight in nature, i.e. it does not incur any substantial performance

overhead on the service's response time. Our ML model can be easily integrated into the DevOps architecture which can then be used to automate interference detection and subsequently maintain a desired level of QoS.

When performance interference is detected, modelling the impact of that interference allows an application owner to quantify how significant the application performance degradation is and derive a suitable mitigation strategy. To quantify performance interference in cloud-native applications, performance engineering techniques are utilized. Performance engineering techniques typically leverage runtime metrics that describe the runtime environment as well as the cloud-native application of interest in order to construct performance models. In cloud-native applications, there are several layers of abstraction from which metrics can be derived, for example, the physical machine (PM), the virtual machine (VM), the containers, or even the application itself. As a consequence, varying degrees of instrumentation are required of the environment and application to obtain the necessary runtime metrics for modelling. These metrics typically include request response time, throughput, as well as container and VM utilizations. State-of-the-art performance engineering techniques often leverage Queuing Networks or Regression models.

Queuing Network models are static models that leverage queuing theory to express application behavior as a function of runtime metrics. Static modelling techniques often have a training and runtime phase. The static models are pre-trained in the training phase and deployed in the runtime phase where they are leveraged to make predictions. A benefit of static modelling is that model training can consider a large amount of performance data, often resulting in a well-performing model. However, the two phase nature of static modelling can be overly cumbersome. If the training phase takes a significant amount of time to complete, static modelling techniques may not keep up with frequent changes in cloud environments.

Regression models are trained models, that can either be trained in a static pre-deployment fashion or at runtime, and are derived from runtime metrics often used in statistical or machine learning based approaches. Regression models are favored for being able to capture application

behavior that is otherwise difficult to express explicitly. Runtime regression models are trained at runtime and can quickly capture changes in the cloud environment. At runtime, model training must be done quickly in order to leverage the new model that accounts for current environment conditions. Accordingly, the amount of data required to train models at runtime tends to be substantially less than that of a static model. As a consequence, runtime models may be less accurate than their static model counterparts.

We propose a Machine Learning (ML) based technique to quantify performance interference in cloud-native applications. Our technique can be leveraged to construct both static and runtime ML models, is non-intrusive, and outperforms competing state-of-the-art techniques. We utilize a realistic e-commerce application benchmark as well as an Internet of Things (IoT) microservice to generate performance interference in our experiments. Furthermore, our approach leverages runtime monitoring metrics that can be captured easily from both the environment and the application.

We attempt to answer the following research questions with our approach:

1.1 Research Questions

RQ-1: How severe is the impact of performance interference in cloud-native applications deployed on public cloud instances? To address this, we run experiments to characterize the impact of interference on a realistic microservice benchmark called Acme Air [19] hosted on the AWS EC2 cloud platform. 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 our machine learning based detection approach perform when the interfering application used for model training and deployment is the same? To answer RQ-2, we train ML models to detect interference caused by a single benchmark application. Our models perform binary classification (*interference or no interference*) and are trained

using readily available system metrics without response time instrumentation. Results show that our approach outperforms competing regression based and threshold based interference detection techniques by at least 2.18% and at most 96.27%. Furthermore our method incurs minimal overhead of only 1% to 2% on the service response time at runtime.

RQ-3: How well does our machine learning based detection approach perform when the interfering applications used for model training and deployment are different? To address RQ-3, we show that we can train our ML model to detect interference caused by an 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. As presented in our results, our technique outperforms competing regression based and threshold based interference detection techniques by at least 1.35% and at most 66.69%.

RQ-4: How effective is a static ML model based approach to quantify the impact of performance interference in cloud-based microservices at runtime when the interfering application used for model training and deployment is the same? We address the use case in which the application owner wants to predict the impact of co-locating applications in a cloud environment. In this scenario, the interfering application and system resource utilization details are known. To answer RQ-4, we train static ML models to quantify interference caused by an interfering application that stresses the same resources as our monitored application. The same interfering application is used at training time and also at runtime. The ML models are trained using easily obtainable system metrics. Our results show that our approach outperforms competing interference quantification techniques by at least 14.13% and at most 1271.37%.

RQ-5: How effective is a static ML model based approach to quantify the impact of performance interference in cloud-based microservices at runtime when the interfering application used for model training and deployment are different? This serves to address the use case in which an application owner does not have visibility into the application(s) co-located along their own application. For RQ-5, we evaluate how well our static ML models

generalize in runtime scenarios where a different interfering application is present as opposed to the one encountered at training time. Our technique outperforms competing techniques by at least 2.49% and at most 668.81% as shown in our results.

RQ-6: How effective is a runtime ML model based approach to quantify the impact of performance interference in cloud-based microservices at runtime? This addresses the use case where an application owner lacks visibility of interfering applications in the environment hosting their own application. To address RQ-6, we leverage a sliding window technique to continuously train ML models at runtime for fixed time intervals. Next, these ML models are used to quantify interference at runtime for a fixed interval and then interchanged with another ML model. As highlighted in our results, our technique outperforms competing state-of-the-art techniques by at least 1.45% and at most 92.04%.

Five contributions are made in this thesis as follows:

1.2 Contributions

1. We develop a machine learning based interference detection technique that generalizes to unknown interfering applications.

2. We construct our machine learning models for performance interference detection using readily available microservice, instance, and application metrics that impose minimal runtime overhead.

3. We develop a static machine learning based interference quantification technique that generalizes to unknown interfering applications at runtime and outperforms competing state-of-the-art static quantification techniques by at least 14.13% and at most 1271.37%.

4. We develop a runtime machine learning based interference quantification technique that generalizes to unknown interfering applications at runtime, has minimal model training overhead, and outperforms competing state-of-the-art runtime quantification

techniques by at least 1.45% and at most 92.04%.

5. We present a comparative analysis between our static and runtime ML quantification techniques.

The remainder of this Thesis is organized as follows. Section 2 discusses related work. Section 3 details the methodology, experimental setup, results, and threats to validity of our proposed ML Interference Detection technique. Section 4 discusses the methodology, experimental setup, results, and threats to validity of our proposed ML Interference Quantification technique. Section 5 concludes this paper and discusses future work.

Chapter 2

Related Work

2.1 Performance Interference Detection

Y. Koh et al. [7] investigate the impact of VM-level performance interference from co-located workloads and characterize interference impact on low level system metrics. They further cluster workloads by interference type and construct performance prediction models for co-located workloads using weighted means and regression analysis approaches. I. Paul et al. [8] similarly characterize performance interference in co-located VMs and further measure the application performance degradation and impact on low-level system metrics. The authors go on to highlight types of workloads that perform well or not when co-located. As opposed to the aforementioned papers, we characterize the impact of interference among containerized microservices.

Novaković et al. [9] 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. S. Wang et al. [10] 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. These works 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.

D. N. Jha et al. [11] measured intramicroservice and intermicroservice interference using four benchmark microservices. The results of their experiments showed significant container interference present in both intra-container and inter-container cases. S. K. Garg, and J. Lakshmi. [12] ran experiments using well known containerized benchmarks to detect microservice interference and accordingly identified the shared resources subject to contention in these scenarios. K. Joshi et al. [13] propose Sherlock, a method for detecting long-lived performance interference in containerized services caused by co-located VMs. The authors employ a statistical regression detection technique to model performance interference using VM-level and application-level metrics at runtime. We differentiate our work from Sherlock in that we focus on use of ML techniques to detect interference as opposed to traditional statistical regression techniques. Our work further differentiates as we explore machine learning as a means to re-use interference detection models in varying environments.

2.2 Performance Interference Quantification

Several performance engineering techniques attempt to model an application's runtime behavior in order to predict application performance [20–24]. Some techniques leverage static modelling, where a model is constructed pre-deployment and subsequently utilized at runtime to predict application performance [22, 25]. Static modelling does not require at runtime instrumentation which can be intrusive and add prohibitive overhead itself. However, static modelling may not adequately model point in time application behavior. Other techniques employ runtime modelling in which models are continuously constructed and replaced at runtime based on certain criteria [23, 24]. These runtime techniques can in fact capture point

in time performance however as mentioned, careful attention is required for the additional instrumentation and overhead that is added at runtime.

Prominent performance engineering techniques often construct Layered Queuing Models (LQM) which are an extension of Queuing Network Models (QNM) [26–28] or Regression models [21–24]. These models serve to predict a metric of interest like response time or resource utilization.

Prior works have utilized LQMs as performance models for both monolithic and microservice software applications [26–28]. Barna et al. [26] leverage the LQM as a performance model of their target cloud-native application. Particularly, they utilize the LQM for response time predictions of their proposed model identification adaptive controller technique to maintain system goals with robustness and cost-effectiveness. Shoaib and Das [27] similarly utilize the LQM for performance modelling of a cloud-native application. The LQM's response time predictions serve as an input to their proposed genetic algorithm that aims to aid application providers in optimizing cloud-native application scale and placement. Gias et al. [28] propose ATOM, an autoscaling controller for microservices that leverages the LQM to evaluate potential performance gain resulting from deployment decisions. In their work, the LQM serves to make throughput and utilization predictions.

Iqbal et al. [21] build polynomial regression models to predict response time of applications hosted on virtual machines. They subsequently leverages response time predictions to decide whether the application is at risk of breaking service-level agreements and if so, their technique auto-scales the application. Rahman and Lama [22] construct several types of Machine Learning models to predict microservice performance. These models utilize metrics from the multiple abstraction layers of the cloud including the VM, container, and application layers.

Shekhar et al. [23] propose an online Gaussian Process method that uses sliding windows to make response time predictions for containerized yet monolithic applications. Based on those predictions, their method decides whether to vertically scale the application or not.

Kang and Lama [24] similarly propose an online probabilistic Machine Learning method that leverages Gaussian Process models and sliding windows to predict microservice performance. The models use metrics from multiple abstraction layers and are trained in real time to adapt to the dynamically changing cloud environment.

2.3 Performance Interference Mitigation

To mitigate performance interference, container and VM placement and re-assignment strategies are often used. That is, they assess where and how to place an application to minimize performance interference. If deemed necessary, a container may be migrated to another VM or a VM itself may be migrated to another physical machine in order to minimize interference. Placement strategies often employ heuristics to ensure computational tractability given problem constraints.

2.3.1 Container Placement

Mao, Ying, et al. [29] proposed a resource aware container placement algorithm that considers container resource needs against hosts' resource availability. Their algorithm, DRAPS, better utilized resources compared to the default placement algorithm in Docker known as Spread, which simply assigns containers to whichever host has the fewest containers irrespective of resource needs.

Lv, Liang, et al. [30] broke down container placement into two phases; the initial container placement and the dynamic container reassignment. They formulated a cost function subject to a set of constraints for which resources and network communication are highlighted. They stated resources need to be efficiently managed to co-locate complementary containers and network communication, or the transit latency of a request between two containers, needs to be minimized for overall optimal application latency. A Communication Aware Worst Fit Decreasing algorithm is proposed to drive initial container placement and a Sweep and

Search algorithm is proposed for container placement re-balancing at runtime.

Sampaio et al. [31] considered container placement as a Bin-packing problem to drive container placement re-balancing at runtime. With respect to container placement, solutions to the Bin-packing problem require utilizing as little hosts as needed while satisfying constraints. Sampaio et al. propose an optimal solution and heuristic solution to this bin-packing representation which incorporated resource constraints, application network affinities, as well as application resource usage history. Their implementation used a MAPE-K loop to provide an autonomic solution.

2.3.2 Virtual Machine Placement

Slama, Wafa Ben, et al. [32] estimated a virtual machine interference score based on application execution times and consequently leveraged the estimate and fuzzy formal concepts analysis to derive a virtual machine placement strategy that minimized their resource interference score as well as network latency.

Alves, Maicon Melo, et al. [33] constructed a virtual machine interference score based on application slowdown times which they attempted to minimize with an Iterated Local Search algorithm that informed placement of virtual machines.

Chapter 3

Performance Interference Detection

In this chapter, we address Research Questions 1-3 with respect to Performance Interference Detection. We characterize the impact of performance interference and propose an ML based performance interference detection technique. Section 3.1 details the motivation and methodology. Section 3.2 describes the experimental setup and results of RQ-1. Section 3.3 presents experimental setup and results of RQ-2. Section 3.4 details experimental setup and results of RQ-3. Section 3.5 presents threats to validity. Section 3.6 is a summary of our Performance Interference Detection work.

3.1 Methodology

We describe how our proposed interference detection technique can fit inside an AIOps platform in Section 3.1.1. In section 3.1.2, we provide an intuitive reasoning behind using ML models to classify interference in cloud-based microservice applications. In section 3.1.3 we define the assumptions our technique makes about the environment and applications contained therein. In section 3.1.4 we detail data generation and pre-processing. We then describe the methodology behind training an ML model for interference detection in Sec 3.1.5. Finally, we detail the application of an ML model for interference detection at runtime in Sec 3.1.6.

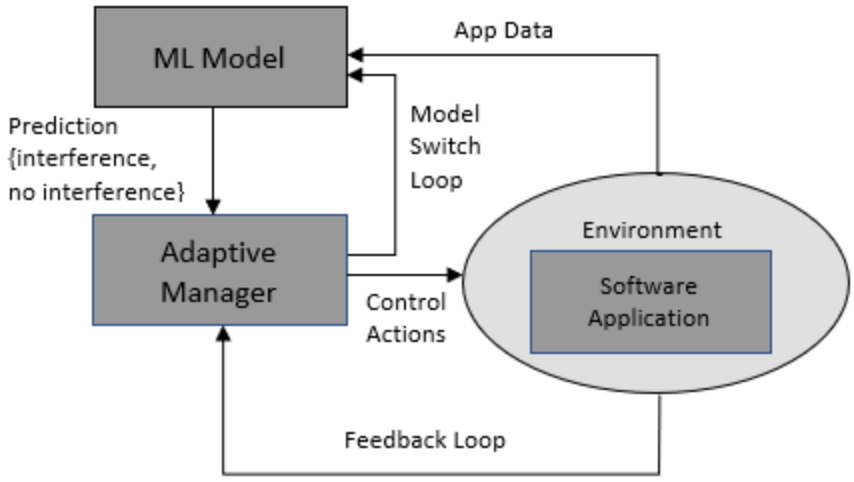

Figure 3.1: AIOps and Interference Detection

3.1.1 AIOps and Interference Detection

DevOps is a cross-community of practice that is targeted towards planning, building, evolving and operating rapidly-changing large scale software systems. The ever increasing scale and complexity of modern software services pose significant challenges for building non-intrusive interference techniques that can be efficiently integrated with current DevOps practices aimed towards effective runtime operations for such services. In this regard, AIOps (Artifical Intelligence for IT Operations) is the latest set of practices that enables DevOps engineers to support and maintain services by using artificial intelligence and machine learning techniques [34, 35].

Figure 3.1 shows how our ML Model based approach can be integrated with several components that manage a software application's operation. As seen in the figure, this involves two loops, a feedback loop and a model switch loop. The environment represents a one VM or multi VM distributed deployment system hosting the software application. The *ML Model* serves to make interference predictions (interference or no interference) based on the inputs obtained from monitoring the application and the cloud environment. The prediction from the ML Model describes the state of the microservice application and is forwarded to the *Adaptive Manager*. The Adaptive Manager is in charge of taking *control*

actions by prescribing corrective measures to the cloud environment. This corrective action is meant to drive the microservice application towards the desired state and knowledge about the cause of current state can help the Adaptive Manager to make better decisions. For example, if the deviation from a performance target is caused by interference, then the Adaptive Manager could migrate the microservice to an environment with greater resource availability. If the deviation is not caused by interference but by the workload, then the Adaptive Manager could scale up the microservice instances. The feedback loop is used by the Adaptive Manager to continuously monitor for further corrective action until the desired state is reached. We note that these corrective actions are orthogonal to this work. The model switch loop is used by the ML Model to input the control actions taken by the Autonomic Manager to adapt the model, and eventually to switch to a different model if needed. For example, the ML Model can switch between one VM or multi VM deployments based on the output of the Adaptive Manager. In this paper, we aim to build a non-intrusive ML model that can be used by AIOps practitioners to develop an interference detection technique by using simple monitoring metrics from the cloud and application environment.

3.1.2 Interference as a Classification Problem

In this section, we explain how interference can be formulated as a ML classification problem by using the fundamentals of Queuing Network Models (QNM). To begin with, we consider a containerized Web application a running inside a cloud instance without any interference, as seen in Figure 3.2. We assume that the application a stresses a single resource k in the instance 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:

$$R_a = \frac{D_{a,k}}{1 - U_k} \tag{3.1}$$

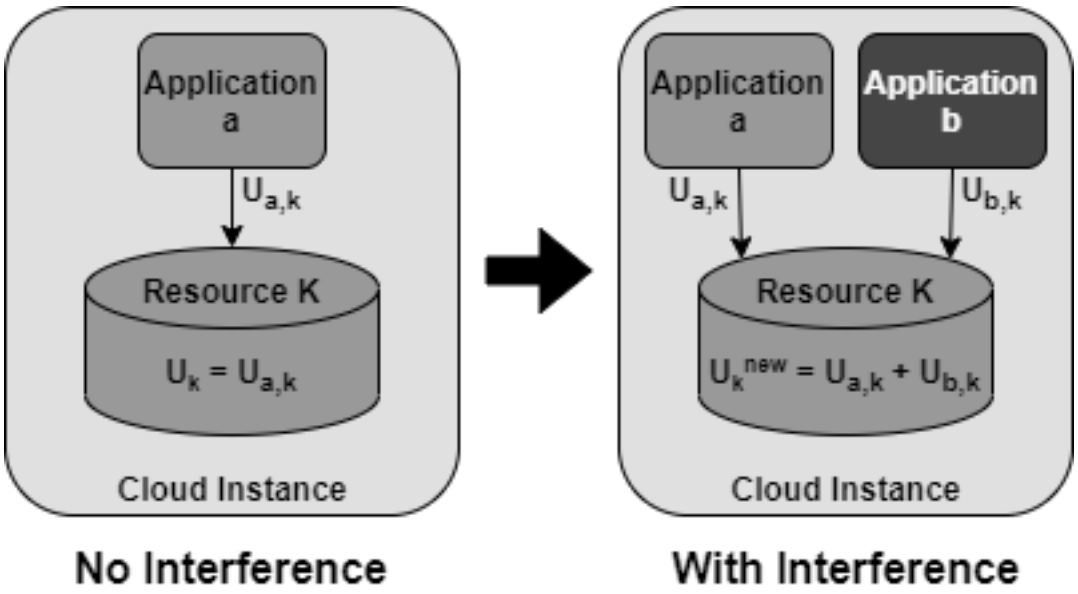

Figure 3.2: Effect of Interference on Resource Utilization

where U_k is the total utilization incurred at resource k in the instance. 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. 3.1 to obtain the response mean time R_a of application a in a no-interference environment as given by:

$$R_a = \frac{D_{a,k}}{1 - U_{a,k}} \tag{3.2}$$

Next, consider another interfering application b running inside the same cloud instance, as seen in Figure 3.2. Application b incurs an utilization of $U_{b,k}$ on resource k inside the instance. Accordingly, from eqn. 3.1, the new response time R_a^{new} of application a when it faces interference from application b is given by:

$$R_a^{new} = \frac{D_{a,k}}{1 - U_k^{new}} \tag{3.3}$$

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:

$$U_k^{new} = U_{a,k} + U_{b,k} \tag{3.4}$$

Based on eqn. 3.2 and eqn. 3.3, 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. We note that these equations can be extended to multiple resources and many different kinds of applications. Based on this observation, we aim to use resource utilization metrics to train our ML model 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}. We note that we use ML models instead of QNMs for interference detection since it is practically 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 [36], which can be difficult. In contrast, our 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. 3.3, 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.

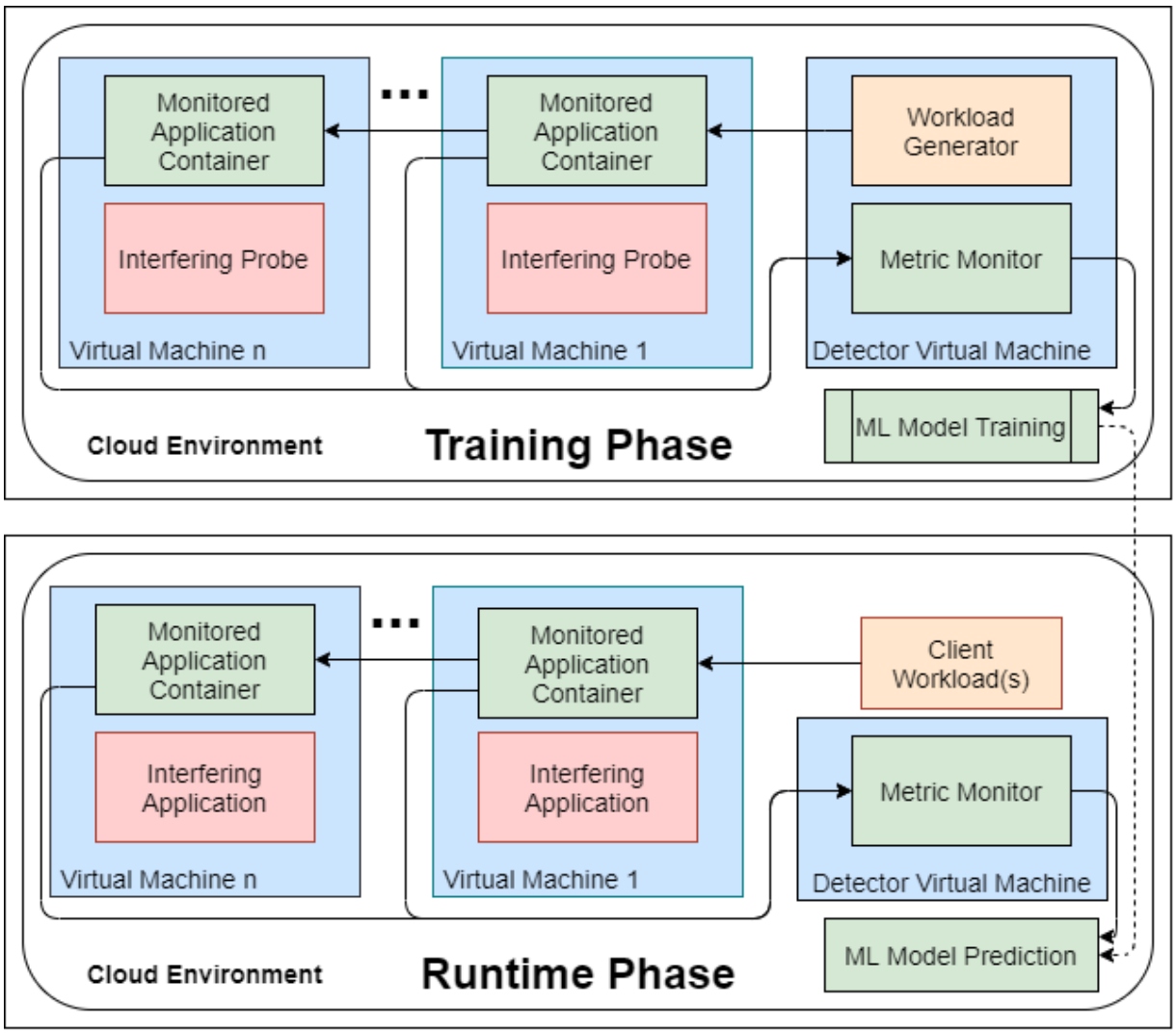

Figure 3.3: Overview of Interference Detection Technique

3.1.3 General Methodology

We refer to the application owner's cloud-based microservice application as the *target application*. The application owner deploys their target application to run on managed cloud services. In doing so, the application owner no longer manages physical servers or even VMs as that responsibility is delegated to the cloud provider. Consequently, the application owner only has guaranteed visibility into the containers of the target application and access to VM level metrics, but not other application containers. Our target application consists of multiple tiers of microservices, with each microservice bundled inside a lightweight container. We note

ML Algorithms	Algorithm Hyperparameters
XGBoost	booster, col_sample_rate, col_sample_rate_per_tree, max_depth, min_rows, ntrees, reg_alpha, reg_lambda, sample_rate
Generalized Linear Model (GLM)	alpha
Random Forest (DRF)	N/A
Gradient Boosting Machine (GBM)	col_sample_rate, col_sample_rate_per_tree, learn_rate, max_depth, min_rows, min_split_improvement, ntrees, sample_rate
Deep Neural Net (DL)	activation, epochs, epsilon, hidden, hidden_dropout_ratio, rho
Extremely Randomized Forest (XRF)	N/A
Stacked Ensemble (All Models)	N/A
Stacked Ensemble (Best of Family)	N/A

Table 3.1: Details of ML Modeling used by the H2O AutoML Framework

that it is typical for microservice applications to be distributed over several cloud instances, with each instance hosting one or more microservice containers. In our study, we consider an *interfering application* as one that runs on the same cloud instances as the target application and competes for shared instance resources.

Our detection technique monitors resource utilization data from inside the cloud instance and the application containers that are easy to collect, does not require any application-level instrumentation and incurs extremely low performance overhead. To this end, we record container utilization metrics at each target application container along with instance utilization metrics collected at a *sampling interval* of t seconds. We also record the overall application throughput, measured in terms of requests per second, at each sampling interval. We note that the sampling interval needs to be tuned for each target application so as to incur minimal levels of performance overhead on the system.

3.1.4 Data Collection and Pre-Processing for ML

Figure 3.3 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 and build our ML model in an offline model training environment. In the training phase, we run our target microservice application on the cloud in a controlled environment where its response time is not impacted by performance interference. This is done by running the target application in isolation on the n cloud instances hosting the microservice without the presence of any interfering application. The objective is to obtain the baseline response times where 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 container hosting the application. This is done through a *Workload Generator* tool running inside a Detector instance which resides alongside the application instances on the same cloud platform, as seen in Figure 3.3. 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 [6]. 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 instance, as shown in Figure 3.3, collects measurable metrics from the target application and its environment such as application response time and throughput, container resource usage, and VM resource usage. We note that instrumenting the application response time is done only in the training phase for constructing the ML model. We also note that although we choose the average request response time of a target application as our performance metric, other metrics such as the 90th percentile response time or the mean throughput values can also be used. Using only the application throughput along with instance 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. We note that in order to ensure that our baseline response time is not impacted by 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 instances* which have same specification as general instances and are offered by public cloud platforms to provide stable performance. Although dedicated instances are more expensive than general instances, 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 cloud instances hosting the target application as shown in Figure 3.3. Doing so imposes additional stress on shared resources which is expected to increase response time as depicted in Equation 3.3. Similar to before, we send the same step-wise increasing workload to our target application from our workload generator tool from inside the Detector instance. Additionally, 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. We note that the application throughput along with the container and VM utilization metrics obtained at each

sampling interval represent a single row of 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. For this purpose, in our data set, we first pair the average response time of the target application R_t along with its corresponding throughput, container and VM utilization metrics in a record obtained at the exact same timestamp. 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 use the 95% CI of the baseline response time of the target application recorded at each step of workload generation to label our training data set. As done in previous anomaly detection work [36], 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. 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.

3.1.5 ML Model Training

AutoML [37–39] 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 [39] in our ML model construction. This framework can be easily integrated with the DevOps feedback loop to automate runtime interference detection. The details of the ML modeling done by the H2O AutoML framework are shown in Table 3.1.

As shown in the table, the framework trains multiple models including but not limited to XGBoost, Generalized Linear Models, Deep Neural Nets, and Random Forests. Additionally, two stacked ensemble models are also constructed and evaluated. An ensemble model is composed of several ML models and combines the predictions of the underlying models to

construct a prediction. The expectation is that the resultant prediction is better than the predictions of individual underlying models. H2O AutoML's first ensemble strategy is to build a stacked ensemble model consisting of all trained models. The second strategy is to construct a stacked ensemble consisting of only the best performing models per algorithm. 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 observe that our labelled data set is imbalanced. Hence, we configure the H2O AutoML training job to balance classes. Doing so enables the classifiers to learn indicators of performance interference as opposed to reporting the majority class. Furthermore, we employ an 80-20 stratified train-test set split to preserve class ratios. We evaluate our models by *Precision*, *Recall*, and F_1 *score*. These metrics are defined as follows:

- *True Positive (TP)* denotes a correct positive prediction.
- *False Positive (FP)* denotes an incorrect positive prediction.
- *True Negative (TN)* denotes a correct negative prediction.
- *False Negative (FN)* denotes an incorrect negative prediction.

$$Precision = \frac{TP}{TP + FP} \tag{3.5}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.6}$$

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.7}$$

When a model makes positive predictions indicating interference as present, the precision scores 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 respective model. F_1 score is the harmonic mean of precision and recall and summarizes their joint behavior in a single metric.

3.1.6 ML Model Runtime Deployment

Once the 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 3.3. In this phase, we continuously monitor throughput, instance 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'.

3.2 RQ-1: Performance Interference Characterization

We design experiments to characterize the impact of performance interference on a target application in Section 4.3.1. Section 4.3.2 details the results of these experiments.

3.2.1 Experiment Setup

Our experiments were run in the AWS EC2 Cloud. We use multiple m5.large EC2 instances residing in the same availability zone on the EC2 cloud platform. These instances 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 instances.

Web Benchmarks

As mentioned before, our target Web application is an open-source e-commerce benchmark microservice application called Acme Air. The Acme Air benchmark 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 [40, 41]. We run these 2 microservices in their own Docker containers. We refer to these containers as the *Acme-Web* and *Acme-Db* containers respectively.

We leverage two setups to host our target and interfering applications: a single VM setup and a dual VM setup. To induce performance interference on our benchmark Acme Air application, we ran experiments with three different interfering applications. These are application benchmarks designed to run along side Acme Air and compete for shared host resources. To this end, we use the stress-ng application [42] that serves as a configurable artificial application benchmark which can be used to generate a wide range of resource utilization levels. Prior work [43] has used the stress-ng benchmark to incur varying level of resource utilization by stressing the system under study. We used a containerized version of stress-ng as the interfering application in our experiments. For our second interfering application, we use a second copy of the Acme Air application that consists of running both the Acme-Web and the Acme-Db containers on the same VM instance under test. Running a Web microservice benchmark like Acme Air provides a more realistic deployment scenario as compared to running the command line stress-ng tool as the interfering application. For our third interfering application, we used the Air Quality Monitor application [44] which is an Internet of Things (IoT) application. Running an IoT microsevice benchmark like the Air Quality Monitor represents a deployment scenario in which diverse and distinct microservices are co-located.

Workload Characteristics

We use *httperf* [45] as the Workload Generator tool to send workload to our target Acme Air application. The httperf client runs inside a separate VM instance and is not containerized. We use the *httperf* tool to submit the default workload obtained from the official Acme Air project [19] 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 variation in application utilization. Our Acme Air workload has a step size increase of approximately 12.5% CPU utilization. We configure the stress-ng application to incur CPU utilization levels at 20%, 40%, 60%, 80%, and 100% respectively. The Air Quality Monitor workload is similarly configured with a step size increase of approximately 20% CPU utilization. The workloads were chosen such that for scenarios without interference present, there are other scenarios with interference present that have comparable Acme Air utilization levels. Otherwise, lacking overlap between interference and non-interference scenarios, a simple utilization threshold algorithm could detect interference with ease. We use our step size increases in workload to that end. Next, we set the number of workload repetitions $N = 40$. As such, each workload variation is run 40 times per environment to capture variance. Finally, for the purpose of this study, we set the workload duration $x = 120$ seconds.

Metrics Monitoring

To monitor our environment, we leveraged prominent industry solutions for running the *Metric Monitor*. Prometheus is an open-source monitoring solution that integrates with multiple metrics exporters [46]. Node exporter is one such metrics exporter that we incorporate in our setup to get virtual machine metrics [47]. cAdvisor, another metrics exporter, natively supports Docker and so we use it to record our Docker container metrics [48]. Grafana [49], a metrics analysis and querying solution that integrates with Prometheus, is used to collect the aforementioned metrics, merge them based on timestamp, and export the resultant data

set. Additionally, application metrics averages were output in httperf logs. The application metrics are also joined with the container and virtual machine metrics by timestamp. These metrics are used in analysis, data labelling, and ML model construction.

With cAdvisor, we record the CPU and memory utilization at each target application container collected at a sampling interval of $t = 5$ seconds. In our study, we set the value of the sampling interval t to 5 seconds since we incurred a maximum response time overhead of only 2% at this setting. We choose to collect only the CPU and memory utilization from inside our target application containers since our target application incurs significant levels of resource utilization for just these two resources. Furthermore, we collect the average end-to-end baseline response time of our target microservice application.

Single Instance Experiments Design

We first conduct experiments on a single VM instance. In this scenario, we consolidate our target microservice application along with the interfering application on the same VM instance. As mentioned before, we use Acme Air as our target application. We run experiments where the interfering application is either stress-ng, another copy of Acme Air, or our IoT application which refers to our Air Quality Monitor application. Using another copy of Acme Air as the interfering application is done to simulate a realistic scenario where interference is caused when a realistic Web microservice application is scaled up and deploys additional containers. In this case, we run the second copy of the Acme-Web and Acme-Db containers inside the same VM instance on which the original target Acme Air application containers are running. Using the IoT application as the interfering application simulates realistic scenarios in which another diverse and distinct application is co-located along our target application. The IoT application's containers run along side the target application's containers on the same VM instance(s). We use our *httperf* 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.

In particular, when stress-ng is the interfering application, our target Acme Air application's web-server container Acme-Web CPU utilization varies in range of 3% to 82% and Memory utilization in range of 1% to 2%. The target application's database container Acme-Db CPU utilization varies in range of 1% to 50% and Memory utilization in range of 1% to 59%. Furthermore, CPU utilization of the VM instance varies in range of 4% to 95%.

In our experiments where another copy of Acme Air serves as the interfering application, our target application's web-server CPU utilization is varied in range of 3% to 73% and Memory utilization in range of 1% to 2%. The target application's database CPU utilization varies in range of 1% to 47% and Memory utilization in range of 1% to 41%. CPU utilization of the VM instance varies in range of 4% to 100%.

Furthermore, we run experiments in which our IoT application serves as the interfering application. Our target application's web-server CPU utilization ranges from 3% to 80% and Memory utilization ranges from 1% to 2%. The target application's database CPU utilization ranges from 2% to 59% and Memory utilization ranges from 2% to 36%.

Dual Instance Experiments Design

We also conduct experiments on a dual instance setup where our target Acme Air application is running on a dual VM instance setup. This is done to motivate large-scale real world use cases where a microservice application like Acme Air is typically hosted in different microservice containers running across different cloud instances. In this scenario, we run the Acme-Web and Acme-Db containers on 2 separate VM instances respectively. Similar to the one instance scenario, we first use stress-ng as the interfering application. To this end, we run 1 stress-ng container on each of the 2 instances hosting the Acme-Web and Acme-Db containers respectively. We send the same workload to these 2 stress-ng containers when they are running alongside the Acme Air containers. We vary the workload to the target and interfering applications to incur a wide range of resource utilization on our host instances. Specifically, we vary CPU utilization on the 2 host instances from 12% to 85% in increments

of 10%.

Similar to our single instance scenario, we further run experiments with copies of Acme Air as interfering applications in the dual VM instance seup. In this case, each of the two virtual machines has its own interfering Acme Air application running on it. We vary workload to our applications to incur a wide resource utilization range in both instances. Specifically, we incur CPU utilization ranges from 12% to 94% and 8% to 84% in the 2 VM instances respectively. By varying the workload to our applications, we incur varying amounts of interference on Acme Air.

In addition, we run experiments with the IoT application serving as the interfering application. Each of the two virtual machines run a copy of the IoT application. Workloads to each of these two copies are varied to incur a wide range of resource utilization. We incur CPU utilization in range of 4% to 99% and 4% to 90% in the 2 VM instances respectively.

3.2.2 Results Analysis

Single Instance Experiments Results

We first evaluate the results where stress-ng served as the interfering application. Table 3.2 shows the target Acme Air application's utilization values along 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 table, depending on the resource utilization and workload levels, the average response time of Acme Air is impacted by interference. At heavy interference scenario, the average response time of Acme Air increases from 9ms to 12.53ms, which is a 39% increase.

Table 3.3 captures the target Acme Air application's utilization values along with associated interference impact on average request response time of Acme Air when it is running alongside

Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	Req/s
14.36%	1.31%	7.56%	41.18%	1.33	1.34	238.9
28.66%	1.35%	13.65%	41.83%	1.91	1.93	469.40
37.68%	1.39%	18.09%	41.99%	2.58	2.87	631.49
53.66%	1.51%	27.88%	42.00%	9.00	12.53	937.03

Table 3.2: Acme Air + stress-ng Interference in Single VM

a second copy of Acme Air. As seen in the table, the average response time of Acme Air is heavily impacted by different levels of interference incurred by the interfering application. Even at comparatively light interference levels, the response time of Acme Air more than doubles, as can be seen in the first row in Table 3.3. In the worst case scenario, the average Acme Air response time increases from its baseline response time of 7.35 ms to a very high value of 518.53 ms which is an increase of 6954.83%.

Similarly, Table 3.4 details the target Acme Air application's utilization values and associated interference impact from the IoT Air Quality Monitor application. Again, the average response time of Acme Air is heavily impacted. In the worst case scenario, the average Acme Air response time increases from its baseline of 16.48 ms to 590.88 ms which is an increase of 3485.44%.

Dual Instance Experiments Results

Table 3.5 shows the impact of interference on the average request response time of Acme Air running along with the stress-ng containers in the dual VM instance setup. As seen in the table, the impact of interference on the average response time of Acme Air in a dual instance setup can be significant. In the worst case scenario, the average Acme Air response time increases from 5.66 ms to 24.1 ms, a 325.8% increase.

Table 3.7 shows the effect of interference by running a second copy of Acme Air on the response time of Acme Air in a dual instance scenario. As seen in the table, the response time of Acme degrades significantly when running alongside a second copy of the Acme application.

Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	Req/s
17.93%	1.25%	9.19%	26.65%	1.22	2.66	238.90
34.15%	1.41%	16.90%	27.35%	1.78	24.74	469.29
41.80%	1.44%	21.99%	27.66%	2.40	104.13	631.48
45.95%	1.49%	23.18%	28.01%	7.35	518.53	673.49

Table 3.3: Acme Air + Acme Air Interference in Single VM

Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	Req/s
15.38%	1.43%	10.04%	20.56%	2.38	8.10	238.90
28.08%	1.32%	17.25%	20.64%	2.98	31.83	469.39
37.19%	1.40%	23.28%	20.86%	4.16	135.66	631.40
54.04%	1.50%	36.50%	20.78%	16.48	590.88	938.97

Table 3.4: Acme Air + IoT Interference in Single VM

In the worst case scenario, the response time of Acme increases from 3.46 ms to 202.47 ms.

Table 3.8 details the impact of interference when running the IoT Air Quality Monitor application as the interfering application along side our target Acme Air application. Again, Acme's response time degrades significantly in this scenario as well. 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 Acme Air in all cases. For both the single and dual instance scenarios, response time degrades by at least a factor of 39% and at most a factor of 6955% at moderate CPU utilization levels between 40% and 60%.

3.3 RQ-2: ML for Known Interfering Applications

We first attempt to evaluate our ML based approach to detect performance interference where the interfering application is known in the training and runtime phases. Section 3.3.1 describes our experimental setup. Section 3.3.2 details results of our experiments.

Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	Req/s
26.03%	1.56%	11.82%	45.20%	1.11	1.81	469.39
34.83%	1.61%	15.56%	45.39%	1.57	2.80	631.48
48.88%	1.66%	21.90%	45.39%	3.02	12.23	937.00
54.63%	1.66%	24.29%	45.78%	5.66	24.10	1099.85

Table 3.5: Acme Air + stress-ng Interference in Dual VM

3.3.1 Experiment Setup

We conduct experiments to compare our interference detection technique with baseline and state-of-the-art interference detection techniques. To this end, we first set up experiments where we compare our technique against a simple utilization-threshold based detection technique as used in previous work [50]. This serves as a baseline technique where we continuously 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 their pre-defined threshold values. To this end, 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 from our 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 container utilization thresholds. If so, interference is inferred. We evaluate the Precision, Recall, and F_1 score when this baseline technique is applied to the single and dual instance experiments.

We also conduct another set of experiments to compare our interference detection technique with a state-of-the-art regression detection technique used in current research [13]. We chose this technique since it is a statistical regression based approach commonly used in interference and anomaly detection. To this end, we adopt the detection method outlined in [13] to construct logistic regression models for interference detection at runtime. We leverage

the scikit-learn library [51] for our regression implementation. Tunable parameters of this implementation include, but are not limited to, the penalty, stopping criteria tolerance, solver, and maximum iterations for solver convergence. The regression models are fit on the same datasets that were constructed and used for our ML approach as described in section 3.1.4. At runtime, we use the logistic regression model in our single and dual instance experiment setups to predict whether interference is present or not in our Acme Air application. Similar to before, we evaluate the Precision, Recall, and F_1 score of the logistic regression model.

3.3.2 Results Analysis

We report the ML models with highest F_1 Score for both the single and dual instance experiments in Table 3.6. Datasets represented in this table are described by the number of VMs in the environment as well as the interfering application. As seen in the table, GBM models performs best for interference detection in all single and dual VM scenarios in which a second copy of Acme Air serves as the interfering application. In the dual VM scenario where stress-ng is the interfering application, a stacked ensemble model performs best. In all scenarios where the IoT Air Quality Monitor was the interfering application, a stacked ensemble model also performed the best. These ensembles were composed of all trained models of type: GBM, DRF, XRT, DL, GLM.

Dataset	**Best ML Model**	**F_1 Score**	**Precision**	**Recall**
stress-ng / 1VM	GBM	87.41%	83.54%	91.66%
stress-ng / 2VM	Ensemble	96.38%	95.56%	97.21%
Acme / 1VM	GBM	98.94%	99.21%	98.67%
Acme / 2VM	GBM	99.09%	99.09%	99.09%
IoT / 1VM	Ensemble	96.27%	94.02%	98.63%
IoT / 2VM	Ensemble	94.55%	91.72%	97.56%

Table 3.6: Top Performing ML Model Per Dataset

Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	Req/s
17.52%	1.31%	9.81%	33.70%	1.11	2.46	238.9
33.16%	1.44%	17.6%	33.94%	1.33	12.91	469.34
41.92%	1.49%	22.22%	34.28%	1.55	28.47	631.50
57.11%	1.56%	30.12%	34.67%	3.46	202.47	907.29

Table 3.7: Acme Air + Acme Air Interference in Dual VM

Web CPU	Web Mem	Db CPU	Db Mem	R_{base} (ms)	R_t (ms)	Req/s
14.18%	1.42%	9.35%	26.19%	1.24	5.46	238.9
25.98%	1.33%	15.22%	26.44%	1.23	13.67	469.4
34.42%	1.38%	20.74%	26.52%	1.45	26.68	631.50
49.55%	1.49%	30.97%	26.54%	2.32	90.22	939.21

Table 3.8: Acme Air + IoT in Dual VM

Single Instance Experiment Results

Table 3.9 captures algorithm performance of our ML approach and competing techniques as described in section 3.3.1. Notably, the best model in the single instance scenario with stress-ng is the ML model which achieves an F_1 score of 87.41%, precision of 83.54%, and recall of 91.66%. Furthermore, as seen in Table 3.9, in the single instance scenario with Acme Air as the interfering application, our best model is the ML model which achieves an F_1 score of 98.94%, precision of 99.21%, and recall of 98.67%. Finally, as seen in Table 3.9, in the single instance scenario with the IoT Air Quality Monitor as the interfering application, our best model is the ML model which achieves an F_1 score of 96.27%, precision of 94.02%, and recall of 98.63%. In Table 3.9, we observe that our ML approach outperforms all other competing methods in the one virtual machine environment. Irrespective of what interfering application is used, our proposed ML technique still achieves the highest F_1 score, precision, and recall.

Dataset	Approach	F_1 Score	Precision	Recall
stress-ng / 1VM	Threshold	29.14%	21.6%	44.72%
stress-ng / 1VM	Regression	12.16%	34.04%	7.41%
stress-ng / 1VM	**ML**	**87.41%**	**83.54%**	**91.66%**
Acme / 1VM	Threshold	10.60%	94.41%	5.61%
Acme / 1VM	Regression	95.83%	96.03%	95.77%
Acme / 1VM	**ML**	**98.94%**	**99.21%**	**98.67%**
IoT / 1VM	Threshold	0.00%	0.00%	0.00%
IoT / 1VM	Regression	87.39%	78.96%	97.84%
IoT / 1VM	**ML**	**96.27%**	**94.02%**	**98.63%**
stress-ng / 2VM	Threshold	30.99%	39.31%	25.57%
stress-ng / 2VM	Regression	61.57%	67.35%	56.71%
stress-ng / 2VM	**ML**	**96.38%**	**95.56%**	**97.21%**
Acme / 2VM	Threshold	39.82%	88.04%	25.73%
Acme / 2VM	Regression	96.91%	97.42%	96.41%
Acme / 2VM	**ML**	**99.09%**	**99.09%**	**99.09%**
IoT / 2VM	Threshold	19.27%	91.38%	10.77%
IoT / 2VM	Regression	87.04%	83.59%	90.79%
IoT / 2VM	**ML**	**94.55%**	**91.72%**	**97.56%**

Table 3.9: Evaluation of Competing Model

Dual Instance Experiment Results

The dual instance experiments observed a high F_1 score, precision, and recall for both interfering applications. For experiments using the stress-ng application, we observed as high as a four times increase in response time when interference was present. The best model trained on the dual instance with stress-ng actuator was the ML model which achieved an F_1 score of 96.38%, precision of 95.56%, and recall of 97.21% as per Table 3.9.

For experiments using the Acme Air interfering application, at worst we observe a fifty times response time increase when interference is present. The best model for this scenario was again the ML model which achieves an F_1 score of 99.09%, precision of 99.09%, and recall of 99.09% as seen in Table 3.9. Finally, when using the IoT Air Quality Monitor as the interfering application, the best model is the ML model with an F_1 score of 94.55%, precision of 91.72%, and recall of 97.56%.

In Table 3.9, we observe that our ML approach outperforms all other competing methods

in the dual instance environment. Again, irrespective of what interfering application is used, ML still achieves the highest F_1 score, precision, and recall. Second to ML, the logistic regression technique performs relatively well, beating the simple threshold technique.

Scenario	Interval (%)	F_1 Score	Precision	Recall
1VM	0-10	0	0	0
1VM	10-20	0.89	1.0	0.8
1VM	20-30	0.71	0.83	0.63
1VM	30-40	0.81	0.74	0.90
1VM	40-50	0.75	0.75	0.75
1VM	50-60	0.99	1.0	0.98
1VM	60-70	0.96	0.95	0.98
1VM	70-80	1.0	1.0	1.0
2VM	0-10	0.85	0.84	0.86
2VM	10-20	0.95	0.98	0.92
2VM	20-30	0.99	0.98	0.99
2VM	30-40	0.97	0.98	0.95
2VM	40-50	0.95	1.0	0.91
2VM	50-60	0.98	0.99	0.97
2VM	60-70	1.0	1.0	1.0
2VM	70-80	1.0	1.0	1.0

Table 3.10: Sensitivity Analysis with stress-ng Interference

Sensitivity Analysis

We conduct sensitivity analysis to see how well our approach detects performance interference when the target application is facing light, medium and heavy levels of incoming workload. To this end, we partition the test data set into resource utilization intervals. Each interval spans a utilization range of 10% to which the data points are assigned, for e.g., all data points are assigned to the interval 0%-10% whose CPU utilization falls between these 2 bounds. Since the Acme-Web and Acme-Db containers are CPU and memory intensive, we focus on only these 2 resources for this analysis. We calculate the F_1 score, precision, and recall values obtained from our ML model for each interval in both single and dual instance scenarios.

Our sensitivity analysis results using the stress-ng interfering application is shown in

Table 3.10. We partition the data set based on the Acme-Web CPU utilization. In the single VM data set, 11% of data points were labelled as interference present. In the dual VM data set, 38% of data points were labelled as interference present. As seen in the table, our ML model based approach achieves the highest F_1 score, precision, and recall at medium Acme Web CPU utilization levels between 50%-80%. When partitioning the same data set by the Acme-Db Memory utilization, we notice a similar observation pattern.

We conduct a similar analysis on our data set partitioned again by the Acme-Web CPU utilization for scenarios where a second copy of Acme Air was run as the interfering application. In both the single and dual VM data sets, 83% of data points were labelled as interference present. We note that there is more interference present across all segments when a second Acme Air application serves as the interfering application as opposed to stress-ng. This is because the two Acme Air replicas are competing for exactly the same shared instance resources. F_1 score, precision, and recall were consistent across all intervals. F_1 score ranged from 0.97 to 1.0, precision ranged from 0.93 to 1.0, and recall ranged from 0.96 to 1.0. We observe similar pattern when partitioning based on the Acme-Db Memory utilization for scenarios with Acme Air as the interfering application.

RQ-2: We validate our ML approach against competing interference detection techniques where the interfering application is the same in the training and runtime phases. The logistic regression technique outperforms the simple threshold technique by sizeable margins in all but one experiment. Our ML models outperform both the logistic regression and simple threshold techniques in each of our evaluated experiments in F_1 score by at least 2.18% and at most 96.27%.

3.4 RQ-3: ML for Unknown Interfering Applications

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

Scenario	Interfering Probe / Interfering App	Approach	F_1 Score	Precision	Recall
1VM	stress-ng / Acme	Threshold	10.6%	94.41%	5.61%
1VM	stress-ng / Acme	Regression	37.47%	77.18%	24.74%
1VM	stress-ng / Acme	**ML**	**44.78%**	**96.67%**	**29.14%**
1VM	stress-ng / IoT	Threshold	0.0%	0.0%	0.0%
1VM	stress-ng / IoT	Regression	0.0%	0.0%	0.0%
1VM	stress-ng / IoT	**ML**	**1.35%**	**96.29%**	**0.68%**
1VM	Acme / IoT	Threshold	0.0%	0.0%	0.0%
1VM	Acme / IoT	Regression	86.4%	77.72%	97.29%
1VM	Acme / IoT	**ML**	**89.87%**	**90.26%**	**89.49%**
2VM	stress-ng / Acme	Threshold	39.82%	88.04%	25.73%
2VM	stress-ng / Acme	Regression	20.73%	99.89%	11.56%
2VM	stress-ng / Acme	**ML**	**72.64%**	**98.9%**	**57.4%**
2VM	stress-ng / IoT	Threshold	17.80%	85.43%	9.93%
2VM	stress-ng / IoT	Regression	34.46%	92.32%	21.18%
2VM	stress-ng / IoT	**ML**	**54.61%**	**92.52%**	**38.74%**
2VM	Acme / IoT	Threshold	17.80%	85.43%	9.93%
2VM	Acme / IoT	Regression	83.27%	77.53%	89.92%
2VM	Acme / IoT	**ML**	**84.49%**	**82.91%**	**86.12%**

Table 3.11: Evaluation for Unknown Interference

stress the underlying instances. 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 interfering application in the runtime phase as opposed to its training phase. Section 3.4.1 describes our experimental setup. Section 3.4.2 presents our experiment results.

3.4.1 Experiment Setup

We run experiments to evaluate the effectiveness of ML models in detecting performance interference when the interfering application is unknown. We evaluate models by their F_1 score, precision, and recall. Competing techniques considered are the same Regression and Threshold based techniques as introduced in 3.3.1. To this end, we use the same methodology as described in section 3.1.5 to train ML models. Our ML models are trained in either a single

VM instance environment or dual VM instance environment as discussed in section 3.2.1 and section 3.2.1 respectively. For these environments, we choose one of our three applications to serve as the interfering probe benchmark application in the ML model training. Next, we deploy these trained ML models in the runtime phase as described in section 3.1.6. 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.

The goal is to evaluate the model performance against an unknown interfering application that has not been seen by the ML model before. To measure the effectiveness of our generalized ML model, we evaluate our method against competing models of the logistic regression and simple threshold techniques described in section 3.3.1. The logistic regression model follows a similar methodology as our ML model in that the interfering application seen at runtime is different than interfering probe application used at training time. For the simple threshold technique, given its thresholds are static, the model does not change across datasets for making predictions. To compare performance across these competing models, we evaluate and compare their respective F_1 scores, precision, and recall values.

3.4.2 Results Analysis

Table 3.11 shows our results from the experiments conducted in section 3.4.1. Table 3.11 denotes the interfering probe used at training time as well as the interfering application present at runtime. In this way the ML models were evaluated in scenarios where the interfering application is unknown and previously unseen. Notably, ML outperformed competing methods in both single VM and dual VM environments. In each scenario, ML obtained the highest F_1 score when compared to competing techniques. It's also notable that the models trained using the second copy of Acme Air as the interfering probe at training time and tested at runtime with IoT interference resulted in a better F_1 score than the models where the interfering probe at training time was stress-ng. Models where the interfering application is known perform substantially better over their counterparts where it is unknown. However,

using a model trained for use with unknown interfering applications does not suffer from long pre-deployment or training phases.

RQ-3: We validate our ML approach against competing interference detection techniques where the interfering application is different in the training and runtime phases. Our ML models outperform competing techniques in F_1 score by at least 1.35% and at most 66.69%.

3.5 Threats to Validity

We identify some threats to validity of our work in this section and classify them as per Wohlin et al [52]. We note as an external threat that our approach of training a ML model using a benchmark interfering application and using this model to detect interference from unknown interfering applications might not work if the resource utilization signature of the benchmark interfering application is significantly different from that of the interfering application at runtime. In other words, if the benchmark interfering application used in the training phase and the interfering application at runtime stresses different instance-level resources in different ways, our technique may be unable to classify interference. Furthermore, as another external threat, in multi-instance deployment strategies frequently used for microservice deployment, if the instance 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.

3.6 Summary

In Chapter 3, we characterize the impact of performance interference in cloud-native applications with a realistic microservice benchmark to address RQ-1. At moderate CPU utilization levels, we observe target application performance degradation of as much as 6955%. To detect such interference, we propose a Machine Learning based technique that does not require application instrumentation and imposes minimal overhead of 1-2% CPU utilization. We

evaluate our ML detection technique in cloud environments where an interfering application is known to address RQ-2. In this scenario, our technique outperforms baseline and competing detection techniques by at least 2.18% and at most 96.27%. Furthermore, we evaluate our technique in cloud environments where an interfering application is unknown to address RQ-3. This was done to measure how well our ML models generalize in dynamic cloud environments. Under these conditions, our ML technique outperforms baseline and competing detection techniques by at least 1.35% and at most 66.69%.

Chapter 4

Performance Interference Quantification

In this chapter, we address Research Questions 4-6 with respect to Performance Interference Quantification. We propose an ML based performance interference quantification technique usable for both static and runtime modelling. Section 4.1 discusses methodology. Section 4.2 describes the common environment and application setup used in addressing our research questions. Section 4.3 details the experimental setup and results of RQ-4. Section 4.4 describes the experimental setup and results of RQ-5. Section 4.5 presents the experimental setup and results of RQ-6. Section 4.6 frames the threats to validity. Section 4.7 is a summary of our Performance Interference Quantification work.

4.1 Methodology

We use the methodology described in this section to address our research questions. The microservice application deployed by the application owner is denoted as the *target application* and is the application for which we want to maintain a good QoS. The target application is hosted in one or more containers which we refer to as *Monitored Application Containers* as seen in Figure 4.1. These containers are distributed across n VMs. An *interfering application* denotes an application distinct from the target application but whose container instances have been or might be co-located on the same VM as the target application. Accordingly,

the interfering application competes with the target application for shared resources.

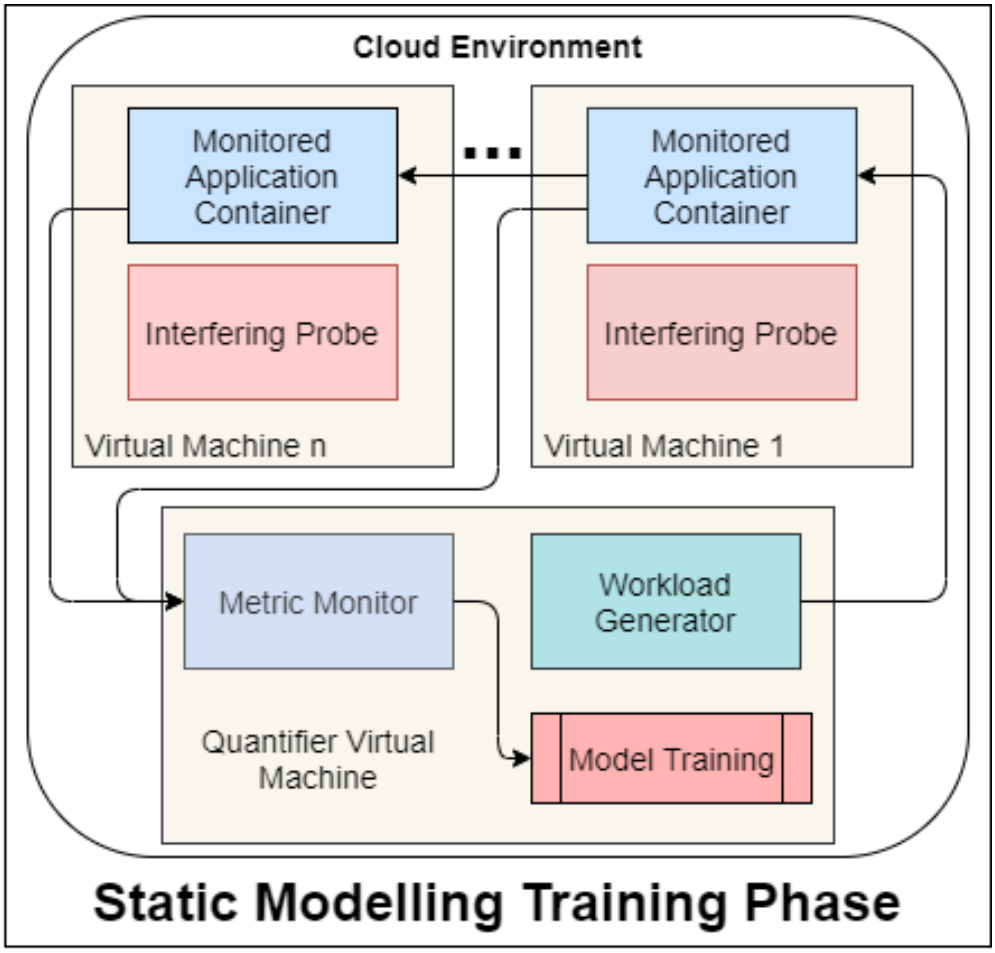

Figure 4.1: Overview of Interference Quantification Training Phase

4.1.1 Static Models for Interference

For static quantification, our methodology consists of two phases: the *training phase* and the *runtime phase*. Figure 4.1 presents an overview of our interference quantification training phase in which we construct static models. The training phase is an offline phase where controlled experiments are run to simulate an environment with and without inference present. The *Workload Generator* is a configurable tool that sends traffic to the target application at varying intensities. In that fashion, a wide range of utilization levels can be generated for the target application. In addition, the interfering application, called *probe*, can be co-located on

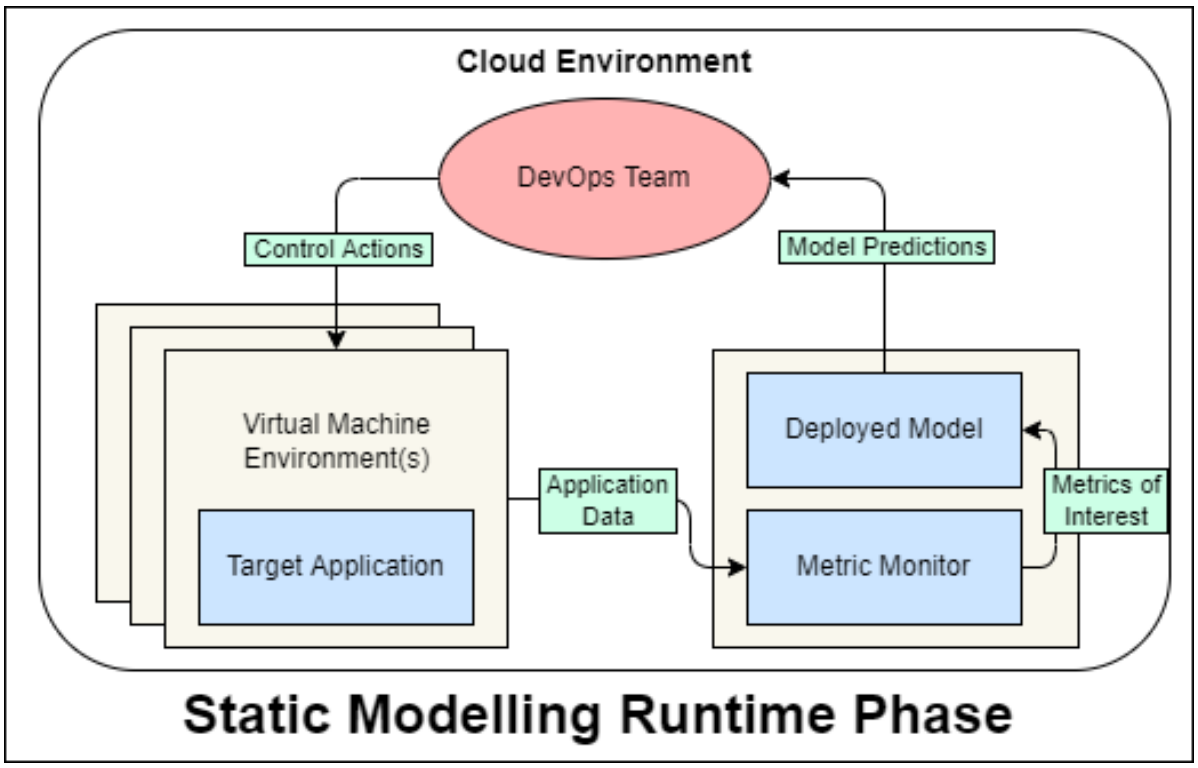

Figure 4.2: Overview of Interference Quantification Runtime Phase

the same VM as the target application. Similarly to the target application, the utilization of the interfering probe can be varied on demand to cover a large space of resource utilization.

Furthermore, in the training phase, the environment and the target application are monitored by the *Metric Monitor*. The Metric Monitor collects data from the VM(s), container(s), and target application at a fixed time interval t. The metrics obtained by the Metric Monitor are used to construct static models. In our experimentation, the static models predict the response time of the target application. Other metrics of interest can be predicted with our proposed technique however.

These static models are subsequently deployed in the runtime phase to quantify the impact of interference on our target application's response time. Models can be deployed for use in scenarios where the interfering application is known and is the same one that was encountered in the training phase. In addition, models can also be deployed in scenarios where the interfering application is unknown and where the interfering application may be

different from the one encountered in the training phase.

The runtime phase is similar in nature to the training phase, with a few key distinctions. In the runtime phase, as seen in figure 4.2, instead of a Workload Generator and Interfering Probes, we have the Client Workload and Interfering Applications which in practice are not controllable. The runtime phase furthermore leverages deployed models to make predictions. As previously mentioned, a benefit of static quantification is that we do not have frequent model re-training although as a consequence, static models may be less accurate in their predictions if they do not generalize well.

Layered Queuing Models

We draw from Queuing Theory to motivate the use of Queuing Networks for quantifying performance interference as a static quantification technique. The LQM is an extension of the QNM and so we start with a discussion of the Queuing Network model accordingly.

In a QNM, the response time of an application a running on a virtual machine is expressed as a function of service demand as well as the total utilization of a resource k as follows:

$$R_a = \frac{D_{a,k}}{1 - U_k} \tag{4.1}$$

where $D_{a,k}$ is the service demand of application a at resource k and U_k is the total utilization incurred at resource k.

The total utilization U_k is further expressed as a function of utilization incurred by all applications in the system that utilize resource k. Consider a system in which an application a and an application b both utilize resource k. The total utilization U_k is expressed as follows:

$$U_k = U_{a,k} + U_{b,k} \tag{4.2}$$

Now suppose an application b is co-located on the same virtual machine and also stresses resource k. Application b imposes its own resource utilization on the virtual machine. The

multi-class form of equation 4.1 that predicts the response time of application a is expressed as:

$$R_a = \frac{D_{a,k}}{1 - U_{a,k} - U_{b,k}} \tag{4.3}$$

where $U_{a,k}$ represents the utilization incurred by application a on resource k and likewise $U_{b,k}$ represents the utilization incurred by application b on resource k.

From equation 4.3, it follows that an application b co-located on the same VMs as our target application, application a, and which utilizes the same resource k as does application a, can impact the response time of application a.

LQMs are static, non-linear queuing models that represent both the software and hardware components of a system as a set of layers [20]. In addition, they explicitly express the topology of an application to represent the different tiers of the application as queues. In an LQM the response time, or service time, at each tier of an application is expressed as a function of the response times of the previous tiers and any additional queuing time incurred [20].

As per Franks et. al. [20], applications, or services, are broken down into two main components; clients and servers. Clients make requests to servers and servers process those requests accordingly. Its important to note that a service can be both a client and a server. Clients and servers each represent the different layers of a service. As mentioned, hardware components are also represented as layers. A client makes requests to hardware layers to obtain resources as need.

Consider that an application a is composed of two tiers, say tier i and tier j where tier i calls tier j. Let R_a denote the response time of application a, $R_{a,i}$ the response time of tier i in application a, and $R_{a,j}$ the response time of tier j in application a. We can express $R_{a,i}$ as:

$$R_{a,i} = R_{a,j} + Q_{i,j} \tag{4.4}$$

where $Q_{i,j}$ represents the queuing time incurred by tier i while waiting for a response

from tier j of application a. It follows that the total response time of application a as viewed by a client c is then:

$$R_a = R_{a,i} + R_{a,j} + Q_{c,i} \tag{4.5}$$

where $Q_{c,i}$ represents the queuing time incurred by the client waiting on a response from tier i of application a.

Given that an application's response time is influenced by the response times of each of its tiers as per equation 4.5 and furthermore influenced by the amount of utilization imposed on the system's resources as per equation 4.1, it follows that each tier is susceptible to performance degradation which can have cascading effects on subsequent tiers.

Given the aforementioned characteristics of LQMs, they are well suited for modelling microservice applications. Accordingly, we leverage LQMs as a competing static model that we evaluate our static ML technique against. Our method uses the aforementioned Metric Monitor to obtain utilization values of the target application.

In addition to utilization metrics, LQMs require some prior knowledge of the demand imposed on the system to make a response time prediction as per equation 4.1. Drawing from Queuing Theory again, we can estimate the demand of a system as:

$$D_{a,k} = \frac{U_{a,k}}{X_a} \tag{4.6}$$

where $D_{a,k}$ is the service demand of application a on resource k, $U_{a,k}$ is the utilization of application a on resource k, and X_a is the throughput of application a. We utilize Formula 4.6 in conjunction with the utilization and throughput metrics of our target application to estimate the service demand imposed by our application. With service demand estimates and utilization metrics, we have the necessary inputs to construct LQMs that model our target application.

We leverage OPERA [53], a tool that has been used in prior work for constructing LQMs

for performance modelling of cloud-native applications [26]. OPERA constructs LQMs given workload specifications, application topology, resource utilization metrics, and system demand estimates. The workload specification defines the client throughput to the target application a. The application topology is representative of application a and the environment. Resource utilization metrics are obtained from the Metric Monitor and consist of CPU utilization metrics from the VMs and containers in use. Finally, the system demand estimates are derived from resource utilization metrics and the workload specification. OPERA [53] is employed in the training phase to construct the LQM models. These LQM models are subsequently deployed in the runtime phase where the interfering application may either be known or unknown. The LQM models are used for performance interference quantification of the target application a.

Static Machine Learning Models

Automatic Machine Learning (AutoML) frameworks aim to automate and abstract away the complexity required to train well performing ML models. These frameworks train and evaluate a variety of ML models on a user's provided dataset and outputs the best performing model. In addition, AutoML optimizes the model search space and may even make trade offs between exploration and exploitation. Furthermore, popular AutoML frameworks can be run with several constraints configured. For instance, user's can constrain the AutoML training time to a maximum time budget. In our work, we leverage the H2O AutoML framework [39] in both a static and runtime fashion to construct ML models for performance interference quantification.

The H2O AutoML framework considers several ML algorithms like Deep Neural Networks, XGBoost, and Stacked Ensembles. Each algorithm may have one or more hyperparameters that the AutoML framework tunes. H2O AutoML performs a random grid search over the set of hyperparameters when training models of each ML algorithm. It does so in an attempt to find hyperparameter values that result in the best performing model for the particular ML

algorithm.

Our ML technique relies on CPU and Memory utilization metrics from the VMs and containers in the environment as well as the throughput of the target application a. This combination of multi-layer metrics make up the feature set for our ML models. These metrics are obtained from the Metric Monitor. Our models predict the response time of our target application which makes the prediction task a regression problem.

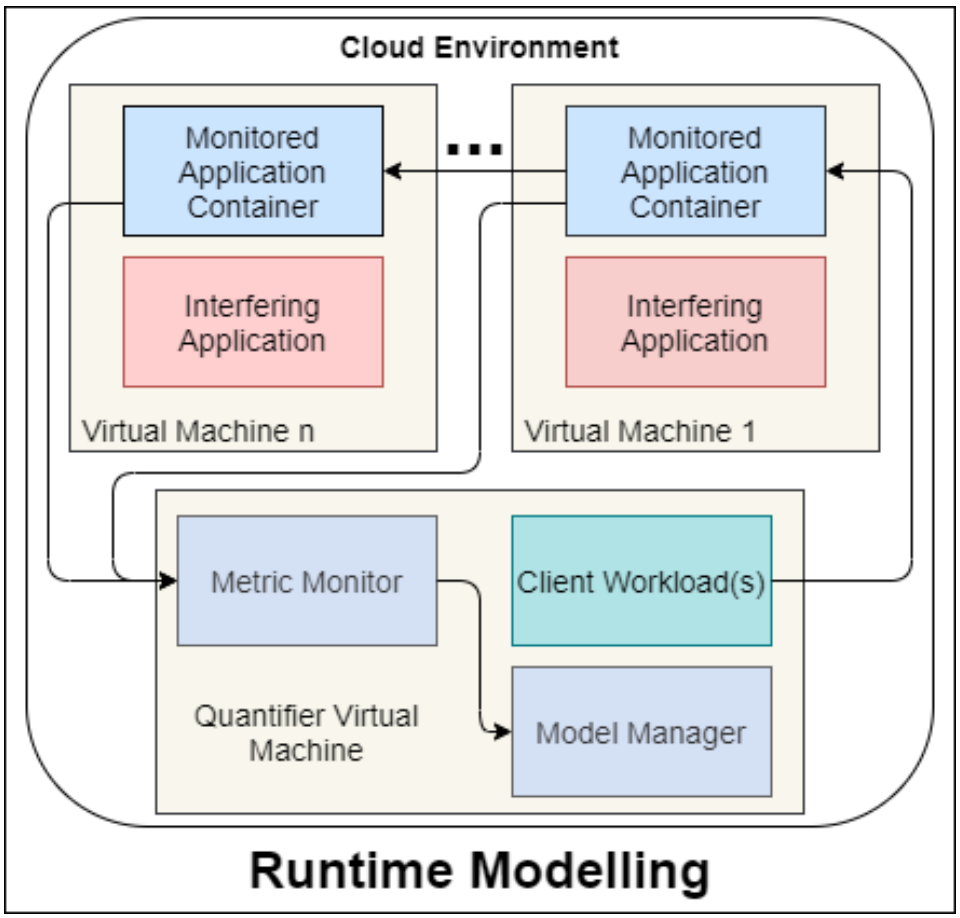

Figure 4.3: Overview of Runtime Modelling Interference Quantification

Given the data obtained from the Metric Monitor, we utilize the H2O AutoML framework to train ML models in the training phase of our technique. Trained ML models are subsequently deployed in the runtime phase to be used for performance interference quantification of the target application a where the interfering application is either known or unknown.

4.1.2 Runtime Models for Interference

Our runtime quantification technique consists of one single phase; the runtime phase. All operations, including model training and deployment, are conducted at runtime. Figure 4.3 depicts an overview of our runtime interference quantification technique. As in the static interference quantification approaches, the runtime approach also has a target application, interfering application, and client workload(s). Our runtime quantification technique also leverages a Metric Monitor which serves the same purpose as it does in the static interference quantification technique. That is, the Metric Monitor polls the environment and target application for metrics of interest at a fixed time interval t. In addition to these components, the runtime quantification technique has a *Model Manager* as depicted in 4.4.

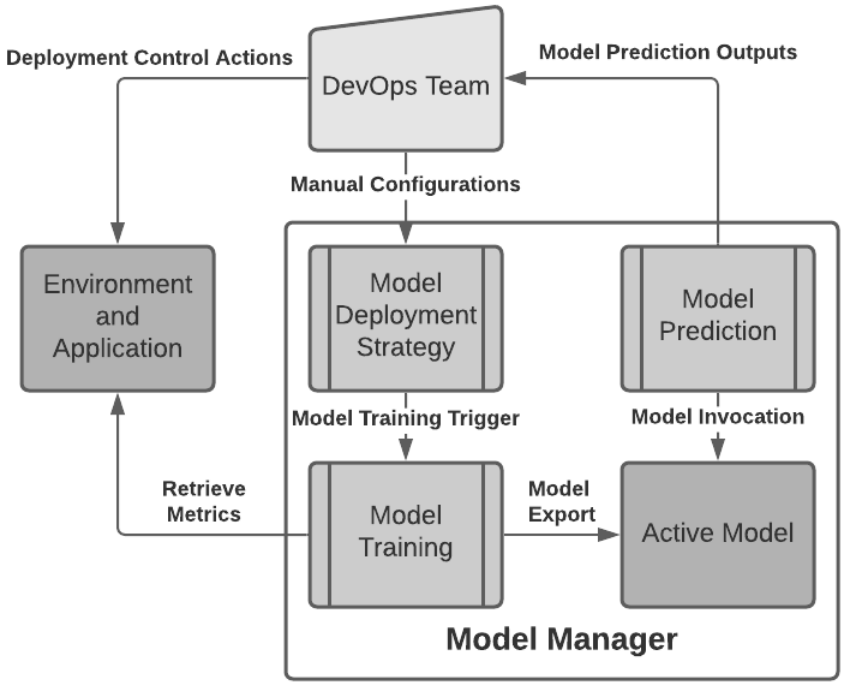

Figure 4.4: Overview of Model Manager

The Model Manager is responsible for the end to end life cycles of models trained and deployed in our runtime phase. The Model Manager employees a *Model Deployment Strategy*,

as defined by the DevOps team, that defines when a model should be trained or by extension, re-trained. When model training and re-training is done at runtime, this constitutes a runtime quantification technique. If the Model Manager indicates a new model is required, the Model Manager triggers the Model Training process. In the Model Training process, the Model Manager queries the Metric Monitor for the relevant metrics and trains a new model at runtime. When a new model is trained, the Model Manager swaps out the old model for newly trained one, which we refer to as the *Active Model*. The Model Prediction process invokes the Active Model to obtain runtime performance predictions. We predict the response time of our target application in our experimentation although our technique can be used to predict other metrics of interest as well.

Gaussian Process Models

Gaussian Process (GP) [54] models are probabilistic models that utilize Bayesian theory to derive their predictions. Prior works [23, 24] have employed GP models with sliding windows in modelling response time of containerized applications. In particular, the recent work of Kang and Lama [24] uses GP models for the modelling of microservice performance interference. As such, we consider their implementation of the GP model as a competing technique in our work.

Kang and Lama [24], as with other prior works, leverage VM, container, and application metrics for their feature sets. These metrics are used to train GP models at runtime as well as utilizing those same models for response time predictions. In their model definition, Kang and Lama [24] formulate their GP models using the sum kernel of the Radial Basis Function (RBF) kernel plus the Rational Quadratic kernel. Furthermore, they employ a sliding window technique in which a limited number of datapoints, in this case 200, are used to train GP models at runtime such that the training time as a result is less than 30 seconds. Their motivation for doing so is to keep overhead like the data collection and model training times less than their sampling interval.

Runtime Machine Learning Models

Our Runtime Machine Learning technique utilizes a Sliding Window Model Deployment Strategy in conjunction with the AutoML framework as detailed in section 4.1.1. Our runtime AutoML technique's feature set is made up of the same metrics used in our static AutoML technique's feature set as mentioned in section 4.1.1. That is, it consists of CPU and memory utilization from the VM and target application containers as well as the target application's throughput. We predict the response time of our target application which again constitutes a regression task.

Prior works have employed a sliding window technique to trigger model training and re-training at runtime [23, 24]. Sliding window techniques configure a fixed time interval t. Model training is conducted with respect to this time interval t. That is, model training only considers a dataset of points acquired within the fixed time interval t. The intent is to set t such that there are enough points to construct a well performing model while also keeping t small enough as to minimize required model training time. With a small enough t, it follows that sliding window techniques can be employed at runtime.

By utilizing a sliding window for model re-training and deployment, changes in the cloud environment can be captured in a dynamic fashion. For instance, assume a new unknown interfering application is co-located alongside the target application. The use of a sliding window technique allows for new models to capture the interference impact this new interfering application has on the target application at runtime.

AutoML frameworks like H2O have parameters in which a user can set a fixed training time budget. Model training time cannot exceed this budget and so the framework optimize how best to spend the training time. In our technique, we set the H2O AutoML training time budget equal to t. That is, the Model Training process does not exceed the sliding window interval. For each interval, a new model is trained and deployed at runtime.

4.2 General Experiment Setup

We run experiments in the AWS cloud on EC2 VMs running in the same availability zone. We utilized m5.large VMs running Ubuntu 16.04 and Docker 19.03.13. Each VM was allocated 2 VCPUs, 8GiB of Ram, and 64GiB of Elastic Block Storage. We conduct two sets of experiments. The first set of experiments deploys the target application on a single VM, which we refer to as the 1VM deployment strategy. The second set deploys the target application across 2 VMs, which we refer to as the 2VM deployment strategy.

Target Application

The target application we use is called Acme Air [19] which is an e-commerce benchmark microservice previously used in other related works [40, 41]. Acme Air consists of a NodeJs Web Server and a MongoDB database, each of which are each containerized and denoted as *Acme-Web* and *Acme-Db* respectively.

Interfering Applications

In our experiments, we leverage three different interfering applications. Only one interfering application is ever deployed concurrently with our target application.

The first interfering application we leverage is a containerized version of the stress-ng benchmark [42]. stress-ng has been used in prior work [43] to induce stress on system resources at varying configurable levels. stress-ng is well suited for our experiments given we can configure the level of stress induced on the system. We can evaluate how well our models work at varying levels of performance interference.

Our second interfering application is a second copy of Acme Air that runs concurrently with the target application. The interfering application copy of Acme Air has its own distinct and configurable workload making for a more realistic deployment scenario. We use a second copy of Acme Air to model scenarios with high levels of performance interference. Given that the interfering application is a copy of the target application, their resource utilization

patterns are the same. Therefore, there will be abundant resource competition.

Finally, our third interfering application is an Internet of Things (IoT) microservice application called the Air Quality Monitor [44]. We leverage the Air Quality Monitor as an interfering application to represent a realistic scenario in which two distinct microservices are co-located. We refer to the Air Quality Monitor application as IoT.

Client Workloads

We devise workloads to each target and interfering application such that they incur incremental step size increases in utilization. Doing so allows us to capture performance interference behavior across a wide range of resource utilization levels. Each workload is run for $N = 40$ repetitions in each environment to ensure variance is captured. Furthermore, each workload is run for a duration $x = 120$ seconds.

For our target application, Acme Air, we leverage httperf [45] to serve as the Workload Generator. The workload itself represents the default workload mix provided with the Acme Air application. The step size increase of this workload is approximately 12.5% CPU utilization. For our interfering application, stress-ng, we configure the application itself through a command line script to stress CPU resources with a step size increase of 20% CPU utilization. The command line script represents the Workload Generator for stress-ng. Next, when we use a second copy of Acme Air as the interfering application, we leverage the same default workload mix previously mentioned to incur a step size increase of 12.5% CPU utilization. The target application copy of Acme Air and the interfering application copy of Acme Air each have their own distinct workloads that may be configured at the same or different levels at any one point in time. Finally, for Air Quality Monitor, we leverage JMeter [55] as the Workload Generator. The step size increase for the Air Quality Monitor is 20% CPU utilization.

Metrics Monitor

Prometheus, a frequently used open-source monitoring tool serves as the Metrics Monitor in our experiments. Prometheus integrates with metrics exporters to obtain metrics as configured by the user. To obtain virtual machine level metrics, we utilize a metrics exporter known as Node exporter [47]. With respect to container level metrics, we use cAdvisor [48], a metrics exported that integrates with Docker. Application level metrics for our target application, Acme Air, are obtained from the logs output by its Workload Generator, httperf. Grafana [49] is utilized to query, merge, and export virtual machine and container metrics.

We configure the metrics exporters to record only CPU and Memory utilization metrics as our target application, Acme Air, heavily utilizes just these two resources. Metrics are configured in Prometheus to be collected at sampling interval $t = 5$ seconds as to incur a minimal overhead of 2% CPU utilization on each virtual machine.

1VM Deployment Strategy

We construct performance interference models for two deployment strategies. The first consists of deploying our target application, Acme Air, along with one of the interfering applications on a single VM. As previously mentioned, we leverage stress-ng, a second copy of Acme Air, and the IoT Air Quality Monitor as interfering applications in our experimentation. The target application is deployed to the single VM and a single interfering application is co-located alongside the target application on the same VM. Each application's Workload Generator is used to incur utilization on system resources and consequently induce performance interference. Meanwhile, our Metric Monitor, Prometheus, captures relevant metrics from the single VM and containers running on the VM.

2VM Deployment Strategy

The 2VM deployment strategy distributes our target application, Acme Air, across two VMs. Acme Air's Web Server is deployed on one of the VMs and Acme Air's Database is

deployed on the other VM. Microservices are frequently deployed across several VMs so we account for this deployment strategy in our experimentation. With respect to the interfering application, each of the two VMs has its own copy of the interfering application deployed to it. Therefore, both VMs can be subject to performance interference. Each interfering application copy has its own Workload Generator. Again, we vary the workloads sent by the Workload Generators to target and interfering applications to incur varying levels of resource utilization. Concurrently, the Prometheus Metric Monitor records metrics from both VMs and the containers that run on the two VMs.

4.3 RQ-4: Static ML for Known Interfering Applications

To quantify performance interference with respect to RQ-1, we utilize the setup as presented in section 4.2 in which we can induce performance interference on a target application. For the experimentation in this section, we assume the interfering application is known to the target application owner. We induce performance interference as described in section 4.2 and collect the resultant target application's performance metrics. With these metrics, we evaluate the performance of our static technique versus baseline and competing techniques. Section 4.3.1 describes our experimental setup. Section 4.3.2 presents the results of our experiments.

4.3.1 Experiment Setup

For each deployment strategy, we sent varying workloads to both target and interfering applications while monitoring the environment. The metrics collected are used in training static performance interference models. We evaluate our proposed static AutoML technique, as detailed in section 4.1.1, against baseline and competing state-of-the-art techniques. We leverage Linear Regression as a simple, statistical baseline technique. We further evaluate our technique against the competing Layered Queuing Model (LQM) as described in section

4.1.1. As mentioned before, LQMs have frequently been used in performance modelling of monolithic and microservice applications [26–28]. The interference models are subsequently used at runtime to predict our target application Acme Air's response time when subject to interference from a known interfering application.

4.3.2 Results Analysis

Observed Resource Utilization

By varying the target and interfering application workloads, we observed a wide range of resource utilization values as depicted in Table 4.1. This enables us to model performance interference impact at varying levels of resource consumption. Notably, we are able to evaluate periods of little to no interference, periods of high interference, and levels of interference between those two extremes.

VMs	Interfering App.	Component	CPU Range	Memory Range
1VM	stress-ng	VM	4%-95%	9%-57%
1VM	stress-ng	Acme Web	3%-82%	1%-2%
1VM	stress-ng	Acme Db	1%-50%	1%-59%
1VM	Acme Air	VM	4%-100%	10%-90%
1VM	Acme Air	Acme Web	3%-73%	1%-2%
1VM	Acme Air	Acme Db	1%-47%	1%-41%
1VM	IoT	VM	5%-100%	15%-39%
1VM	IoT	Acme Web	3%-80%	1%-2%
1VM	IoT	Acme Db	2%-59%	2%-36%
2VM	stress-ng	VM	2%-94%	7%-53%
2VM	stress-ng	Acme Web	2%-73%	1%-2%
2VM	stress-ng	Acme Db	1%-40%	2%-66%
2VM	Acme Air	VM	2%-100%	1%-100%
2VM	Acme Air	Acme Web	3%-79%	1%-2%
2VM	Acme Air	Acme Db	2%-49%	1%-43%
2VM	IoT	VM	4%-100%	4%-41%
2VM	IoT	Acme Web	3%-77%	1%-2%
2VM	IoT	Acme Db	2%-58%	2%-45%

Table 4.1: Observed Resource Utilization Ranges

VMs	Interfering App.	$R_{baseline}$ (ms)	$R_{interf.}$ (ms)	Percent Increase
1VM	stress-ng	1.33-9.00	1.34-12.53	39.22%
1VM	Acme Air	1.22-7.35	2.66-518.53	6954.83%
1VM	IoT	2.38-16.48	8.10-590.88	3485.44%
2VM	stress-ng	1.11-5.66	1.81-24.10	325.80%
2VM	Acme Air	1.11-3.46	2.46-202.47	5751.73%
2VM	IoT	1.24-2.32	5.46-90.22	3788.79%

Table 4.2: Observed Response Time Ranges

Observed Response Times

We further report the response time ranges observed in our experimentation as shown in Table 4.2. We again partition our table by the VM deployment strategy and interfering application in use. The $R_{baseline}$ column depicts the range of baseline response times for our target Acme Air application when there is no interference present. The $R_{interf.}$ column depicts the range of response times for our target Acme Air application subject to performance interference. The Percent Increase column represent the worst case percent increase in response time subject to interference relative to the baseline response time. When a second copy of Acme Air is used as the interfering application we see the largest percent increase in response time. This is followed by the IoT interfering application. stress-ng interference causes the least impact on the target Acme Air application's response time.

Model Evaluation

We evaluate the effectiveness of models by the Mean Absolute Percentage Error (MAPE) as used in prior works [22, 24]. MAPE is defined as:

- Let n denote total number of records observed
- Let $R_{A,i}$ denote actual response time observed for record i
- Let $R_{P,i}$ denote predicted response time made for record i

$$MAPE = \frac{1}{n} * \sum_{i=1}^{n} \frac{R_{A,i} - R_{P,i}}{R_{A,i}} \tag{4.7}$$

Our metrics are delineated by scenario. That is, we calculate the MAPE for each combination of interfering application and deployment strategy in use. Table 4.3 details the MAPE of our static AutoML approach as well as baseline and competing techniques. Across all scenarios, our static AutoML technique outperforms baseline and competing techniques by as little as 14.13% and as much as 1271.37%. In all scenarios, AutoML selected the Gradient Boosting Machine (GBM) as the best performing ML model considered. Furthermore, as per Table 4.3, QNM ranks in second place by outperforming the corresponding baseline Linear Regression models in all scenarios.

VMs	**Interfering App.**	**Model**	**MAPE**
1VM	stress-ng	Regression	109.06%
1VM	stress-ng	LQM	98.07%
1VM	**stress-ng**	**AutoML**	**3.87%**
1VM	Acme Air	Regression	1297.9%
1VM	Acme Air	LQM	50.28%
1VM	**Acme Air**	**AutoML**	**26.53%**
1VM	IoT	Regression	701.37%
1VM	IoT	LQM	69.76%
1VM	**IoT**	**AutoML**	**19.26%**
2VM	stress-ng	Regression	122.83%
2VM	stress-ng	LQM	38.59%
2VM	**stress-ng**	**AutoML**	**5.88%**
2VM	Acme Air	Regression	558.52%
2VM	Acme Air	LQM	48.39%
2VM	**Acme Air**	**AutoML**	**14.92%**
2VM	IoT	Regression	243.45%
2VM	IoT	LQM	43.52%
2VM	**IoT**	**AutoML**	**29.39%**

Table 4.3: Static Model with Known Interfering App

In scenarios with less interference impact observed on response times relative to baseline response times as measured by the Percent Increase column in Table 4.2, we observe a lower

MAPE as seen in Table 4.3. Conversely, the greater the impact on response time relative to baseline response times, the larger we observe MAPE to be. We observe the least amount of performance interference impact on response time when stress-ng is the interfering application and likewise the smallest MAPE scores. When a second copy of Acme Air is the interfering application, we observe the most amount of interference impact and consequently the largest MAPE scores. Finally, the IoT interfering application generates more impact on response time than stress-ng but less than a second copy of Acme Air and the same pattern is observed in MAPE scores.

4.4 RQ-5: Static ML for Unknown Interfering Applications

In a cloud environment, an application owner may not have visibility into the other applications co-located on the same physical server or even VM. Accordingly, we evaluate the effectiveness of performance interference quantification using static models where the interfering application is unknown. In other words, we evaluate how well our technique generalizes when a new interfering application is encountered at runtime which was not previously seen at training time. Section 4.4.1 covers our experimental setup and section 4.4.2 details our results.

4.4.1 Experiment Setup

We leverage the experimental setup detailed in section 4.2. With that setup, we evaluate our static ML technique against baseline and competing techniques. Our baseline technique again is Linear Regression. LQM serves as the competing technique. Again, we compare techniques by MAPE to measure their effectiveness. We leverage the same ML model training methodology as described in section 4.1.1. We assess the techniques across the 1VM and 2VM deployment strategies for our target application. Furthermore, we ensure the interfering application encountered at runtime is different from the interfering application encountered

at training time. In that fashion we can evaluate how well techniques perform when faced with an unknown interfering application.

4.4.2 Results Analysis

The workloads used in experimentation across our research questions are fixed and so we observe the same levels of resource utilization as detailed in Table 4.1. Likewise, we observe the same levels of response time as detailed in Table 4.2. We report the MAPE of each technique and scenario in Table 4.4. AutoML consistently outperformed the Regression and LQM models across all scenarios by as little as 2.49% and as much as 668.81%. For each 1VM scenario, a stacked ensemble model was the best performing ML model in our AutoML technique. For our 2VM scenarios, when stress-ng was the training time interfering application and Acme Air was the runtime interfering application, a GBM model performed best among the considered ML models. The same was true in the 2VM scenario in which Acme Air is encountered at training time and the IoT application is encountered at runtime. In the 2VM scenario when stress-ng is encountered at training time and the IoT application at runtime, a stacked ensemble was selected by our AutoML technique.

4.5 RQ-6: Runtime ML

As previously mentioned, cloud-native application owners may not have visibility of other co-located interfering applications on their physical server or even VM. In contrast to the work presented in section 4.4 where static quantification techniques were employed, in this section we evaluate the performance of runtime performance interference quantification techniques. Section 4.5.1 discusses experimental setup. Section 4.5.2 highlights the results of our runtime quantification technique. Finally, section 4.5.3 presents a comparative analysis and practical considerations between static and runtime AutoML models and their performance across our different scenarios.

VMs	Training Interf. App	Runtime Interf. App	Model	MAPE
1VM	stress-ng	Acme Air	Regression	455.63%
1VM	stress-ng	Acme Air	LQM	72.19%
1VM	stress-ng	Acme Air	AutoML	60.0%
2VM	stress-ng	Acme Air	Regression	131.48%
2VM	stress-ng	Acme Air	LQM	128.34%
2VM	stress-ng	Acme Air	AutoML	85.41%
1VM	stress-ng	IoT	Regression	106.72%
1VM	stress-ng	IoT	LQM	84.47%
1VM	stress-ng	IoT	AutoML	74.0%
2VM	stress-ng	IoT	Regression	103.86%
2VM	stress-ng	IoT	LQM	68.0%
2VM	stress-ng	IoT	AutoML	65.39%
1VM	Acme Air	IoT	Regression	820.16%
1VM	Acme Air	IoT	LQM	83.86%
1VM	Acme Air	IoT	AutoML	71.96%
2VM	Acme Air	IoT	Regression	737.55%
2VM	Acme Air	IoT	LQM	71.23%
2VM	Acme Air	IoT	AutoML	68.74%

Table 4.4: Static Model with Unknown Interfering App

4.5.1 Experiment Setup

We compare the performance of our runtime quantification technique as described in Section 4.1.2 against baseline and competing techniques. To that end, we leverage the static LQM technique as depicted in section 4.1.1 as our baseline technique. Furthermore, we leverage a state-of-the-art Gaussian Process technique as presented in section 4.1.2 which itself is a runtime technique. Runtime AutoML models are constructed following the methodology in section 4.1.2.

We leverage the same setup as presented in section 4.4.1. That is, we employ 1VM and 2VM deployment strategies for our target application Acme Air. In addition, we leverage the same Prometheus Metric Monitor. We utilize stress-ng, a second copy of Acme Air, and the IoT application as interfering applications in our experiments.

For the runtime techniques, model training occurs at runtime. Consequently, the Active Model is likely to be employed when there is runtime interference from the same interfering application encountered in model training. Should the interfering application change at runtime, the new interfering application is unknown to the Active Model. However, given the dynamic nature of the sliding window, subsequent models trained at runtime will detect the new interfering application's impact in their respective model training.

4.5.2 Results Analysis

Resource utilization levels are detailed in Table 4.1 and again consistent across all our experimentation. Response time ranges are presented in Table 4.2 which are also consistent across our experimentation. In Table 4.5, we detail the MAPE of our technique and competing techniques across each scenario. Notably, AutoML outperforms both the LQM and GP models across all scenarios. In four of the six scenarios, the GP model comes in second place by outperforming the LQM. In two of the six scenarios, the LQM outperforms the GP model to come in second place.

4.5.3 Comparative Analysis and Practical Considerations

Static and runtime performance models each have benefits and drawbacks. Having employed techniques from both categories, we compare the performance of the best static and runtime AutoML techniques as captured in Table 4.6. The table is partitioned the runtime interfering application and the VM deployment strategy in use for the target application. Furthermore, we denote the evaluated technique; either Static AutoML or Runtime AutoML. Then we denote whether or not the technique in use had visibility of the interfering application in use in the Interference Visibility column. This column is only applicable to static techniques as runtime techniques construct new models at runtime periodically. Therefore there are periods in which the Active Model in a runtime technique may or may not have been trained using data with interference impact from the current interfering application(s).

VMs	Interfering App.	Model	MAPE
1VM	stress-ng	LQM	98.07%
1VM	stress-ng	GP	7.48%
1VM	**stress-ng**	**AutoML**	**6.03%**
1VM	Acme Air	LQM	50.29%
1VM	Acme Air	GP	50.67%
1VM	**Acme Air**	**AutoML**	**38.29%**
1VM	IoT	LQM	69.75%
1VM	IoT	GP	36.13%
1VM	**IoT**	**AutoML**	**26.74%**
2VM	stress-ng	LQM	38.59%
2VM	stress-ng	GP	20.9%
2VM	**stress-ng**	**AutoML**	**18.54%**
2VM	Acme Air	LQM	48.39%
2VM	Acme Air	GP	32.81%
2VM	**Acme Air**	**AutoML**	**24.04%**
2VM	IoT	LQM	43.52%
2VM	IoT	GP	54.46%
2VM	**IoT**	**AutoML**	**38.85%**

Table 4.5: Runtime ML vs. Runtime GP and Static LQM

It's particularly noteworthy that the static AutoML technique with interference visibility outperforms its counterpart static AutoML technique lacking interference visibility as well as the runtime AutoML technique. However, the applicability of static techniques with interference visibility is limited to scenarios where the application owner knows what interfering applications may be or are co-located with the target application. Nevertheless, this is a useful scenario when the application owner wants to scale manually or automatically known applications and would like to see the effects on another application. These static AutoML models were trained using data collected over a period of 40 hours and as previously noted, the cloud-native application owner may not have visibility into the co-located interfering applications. If we consider the generalized form of our static AutoML technique, where the interfering application is unknown at runtime, the MAPE degrades by approximately 2x - 6x. It is therefore unwise to use static models to predict interference impact induced by unknown applications.

The runtime AutoML technique performs comparatively well with respect to its static AutoML counterparts. While it does not outperform the static AutoML models that assume interference visibility, the runtime AutoML models perform within 2.16% and 12.66% of their counterpart. The runtime AutoML technique does not require visibility of co-located interfering applications so it can account for changing environment conditions. Also, it does not require a long training time unlike the static technique counterparts. Therefore runtime quantification is highly recommended for modelling the interference cause by applications unknown at runtime.

VMs	**Interfering App.**	**AutoML Technique**	**Interference Visibility**	**MAPE**
1VM	stress-ng	Static	Known	3.87%
1VM	stress-ng	Runtime	N/A	6.03%
1VM	Acme Air	Static	Known	26.53%
1VM	Acme Air	Static	Unknown	60.0%
1VM	Acme Air	Runtime	N/A	38.29%
1VM	IoT	Static	Known	19.26%
1VM	IoT	Static	Unknown	83.86%
1VM	IoT	Runtime	N/A	26.74%
2VM	stress-ng	Static	Known	5.88%
2VM	stress-ng	Runtime	N/A	18.54%
2VM	Acme Air	Static	Known	14.92%
2VM	Acme Air	Static	Unknown	85.41%
2VM	Acme Air	Runtime	N/A	24.04%
2VM	IoT	Static	Known	29.39%
2VM	IoT	Static	Unknown	65.39%
2VM	IoT	Runtime	N/A	38.85%

Table 4.6: Comparison of Static and Runtime ML

4.6 Threats to Validity

We describe some threats to validity with respect to our work. We leverage the threats to validity classification of Wohlin et. al. [52].

We note that an internal threat to validity is the configuration of the fixed number

of datapoints considered in the sliding window of our runtime AutoML technique. If this configuration is set too small, the MAPE of our trained models may deteriorate. If this configuration is set too large, model training may become too slow to keep up with changing environment dynamics.

We further note as an external threat to validity that our experiments were conducted in a single availability zone in the AWS cloud. Our experimentation considers multi-VM deployment strategies within the same availability zone. We do not consider deployment strategies cross multiple availability zones.

4.7 Summary

In Chapter 4, we propose a Machine Learning based Performance Interference Quantification technique. We utilize the technique in both static and runtime modelling. With respect to RQ-4, we evaluate our static ML technique to quantify performance interference when the interfering application is known. That is, when the interfering application encountered at deployment time is the same as the one encountered at model training time. Results show our static ML technique outperformed competing interference quantification techniques by at least 14.13% and at most 1271.37%. Additionally, we evaluate the static ML technique when the interfering application is unknown to answer RQ-5. This constitutes evaluating our models at runtime when the interfering application is different than the one encountered at model training time. Our static ML technique outperforms competing interference quantification techniques by at least 2.49% and at most 668.81% in this scenario. Furthermore, we evaluate our runtime ML technique against competing interference quantification techniques as per RQ-6. The runtime ML technique trains ML models at runtime using a sliding window method to account for changing cloud environment dynamics. Our runtime ML technique outperforms competing techniques by at least 1.45% and at most 92.04%. Finally, we present a comparative analysis between our static and runtime ML models. We favor the runtime

ML models for their ability to account for changing runtime dynamics while also maintaining good accuracy.

Chapter 5

Conclusions and Future Work

In this thesis, we first characterize the impact of performance interference on a cloud-native microservice. We measure the performance degradation of a target application subject to varying levels of interference from an interfering application. At moderate levels of interference, we observed a response time degradation up to 6955%. As a first step towards mitigating this performance interference, we propose a performance interference detection technique that leverages Machine Learning. We design a suite of experiments in which our target application is subject to varying levels of performance interference. Models of our ML interference detection technique are trained and evaluated on the microservice resource utilization metrics collected from these experiments. 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. When the runtime interfering application is known, our ML detection technique outperforms baseline and competing techniques by 2.18%-96.27%. Our ML models are also effectively used in varying cloud environments where the interfering application is unknown. Accordingly, they outperform baseline and competing techniques by 1.35%-66.69%.

In addition, we propose static and runtime Machine Learning techniques for performance interference quantification in cloud-native applications. The static ML technique consists of

two phases; the training phase and runtime phase. The runtime ML technique consists of a single phase, the runtime phase. The runtime ML technique leverages a sliding window method to continuously train and re-train ML models at runtime. Both static and runtime ML models are trained using easily obtainable environment and application metrics. No application instrumentation is required to obtain these metrics and collecting the metrics imposes minimal overhead. We evaluate our techniques against competing state-of-the-art techniques using realistic microservice application benchmarks and across varying deployment strategies. Our static ML models outperform competing methods by 14.13%-1271.37% when the interfering application is known and by 2.49%-668.81% when the interfering application is unknown. Our static ML quantification technique are highly accurate at deployment time provided the interfering application is the same as the one encountered in the model training phase. Our runtime ML technique is effective in quantifying performance interference and outperforms competing state-of-the-art techniques by 1.45%-92.04%. Furthermore, our runtime ML technique is practical in that it does not require long training times and can efficiently account for changes in the cloud environment.

With respect to future direction, we intend to integrate our performance interference quantification technique with microservice placement strategies. Quantifying the potential impact of microservice co-location provides a meaningful signal to derive suitable microservice placement. By utilizing our proposed runtime ML performance interference quantification technique, we can measure how significant target application performance degradation is or would be in a system. That is, we can define both reactive and proactive microservice placement strategies. In the reactive case, a placement strategy would migrate an already running microservice to another cloud instance provided a significant level of performance interference is detected at runtime. In the proactive case, a placement strategy would predict how much performance interference could be present if two microservices were co-located. If the level of interference is significant, the placement strategy could avoid co-location of the two microservices and seek an alternate placement.

Bibliography

[1] M. Httermann, *DevOps for Developers (1st ed.).* Apress, USA., 2012.

[2] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[3] ([Online]) Production-grade container orchestration. [Online]. Available: https://kubernetes.io/

[4] ([Online]) Amazon web services. [Online]. Available: https://aws.amazon.com/

[5] ([Online]) Google cloud. [Online]. Available: https://cloud.google.com/

[6] J. Mukherjee and D. Krishnamurthy, "Subscriber-driven cloud interference mitigation for network services," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS).* IEEE, 2018, pp. 1–6.

[7] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *2007 IEEE International Symposium on Performance Analysis of Systems & Software.* IEEE, 2007, pp. 200–209.

[8] I. Paul, S. Yalamanchili, and L. K. John, "Performance impact of virtual machine placement in a datacenter," in *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC).* IEEE, 2012, pp. 424–431.

[9] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments,"

in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. USA: USENIX Association, 2013, p. 219–230.

[10] S. Wang, W. Zhang, T. Wang, C. Ye, and T. Huang, "Vmon: Monitoring and quantifying virtual machine interference via hardware performance counter," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2015, pp. 399–408.

[11] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A holistic evaluation of docker containers for interfering microservices," in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 33–40.

[12] S. K. Garg and J. Lakshmi, "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, 2017, pp. 1–6.

[13] K. Joshi, A. Raj, and D. Janakiram, "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, 2017, pp. 522–530.

[14] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel, "Cryptomining detection in container clouds using system calls and explainable machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 674–691, 2020.

[15] Y. Amannejad, D. Krishnamurthy, and B. Far, "Detecting performance interference in cloud-based web services," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 423–431.

[16] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017, p. 469–478. [Online]. Available: https://doi.org/10.1145/3038912.3052649

[17] V. Horký, J. Kotrč, P. Libič, and P. Tůma, "Analysis of overhead in dynamic java performance monitoring," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 275–286. [Online]. Available: https://doi.org/10.1145/2851553.2851569

[18] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan, "Qmon: Qos-and utility-aware monitoring in enterprise systems," in *2006 IEEE International Conference on Autonomic Computing*. IEEE, 2006, pp. 124–133.

[19] "Acme air," [Online]. [Online]. Available: https://github.com/acmeair

[20] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2008.

[21] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood, "Predictive auto-scaling of multi-tier applications using performance varying cloud resources," *IEEE Transactions on Cloud Computing*, 2019.

[22] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 200–210.

[23] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applica-

tions," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 82–89.

[24] P. Kang and P. Lama, "Robust resource scaling of containerized microservices with probabilistic machine learning," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2020, pp. 122–131.

[25] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, "Benchmarking machine learning methods for performance modeling of scientific applications," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2018, pp. 33–44.

[26] C. Barna, M. Litoiu, M. Fokaefs, M. Shtern, and J. Wigglesworth, "Runtime performance management for cloud applications with adaptive controllers," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 176–183.

[27] Y. Shoaib and O. Das, "Cloud vm provisioning using analytical performance models," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 68–72.

[28] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1994–2004.

[29] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, "Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2017, pp. 1–8.

[30] L. Lv, Y. Zhang, Y. Li, K. Xu, D. Wang, W. Wang, M. Li, X. Cao, and Q. Liang, "Communication-aware container placement and reassignment in large-scale internet

data centers," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 540–555, 2019.

[31] A. R. Sampaio, J. Rubin, I. Beschastnikh, and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," *Journal of Internet Services and Applications*, vol. 10, no. 1, pp. 1–30, 2019.

[32] W. B. Slama, Z. Brahmi *et al.*, "Interference-aware virtual machine placement in cloud computing system approach based on fuzzy formal concepts analysis," in *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 2018, pp. 48–53.

[33] M. M. Alves, L. Teylo, Y. Frota, and L. M. Drummond, "An interference-aware virtual machine placement strategy for high performance computing applications in clouds," in *2018 Symposium on High Performance Computing Systems (WSCAD)*. IEEE, 2018, pp. 94–100.

[34] Y. Dang, Q. Lin, and P. Huang, "Aiops: real-world challenges and research innovations," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 4–5.

[35] A. Masood and A. Hashmi, "Aiops: Predictive analytics & machine learning in operations," in *Cognitive Computing Recipes*. Springer, 2019, pp. 359–382.

[36] J. Mukherjee, A. Baluta, M. Litoiu, and D. Krishnamurthy, "Rad: Detecting performance anomalies in cloud-based web services," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 493–501.

[37] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," ser. KDD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 847–855. [Online]. Available: https://doi.org/10.1145/2487575.2487629

[38] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 2755–2763.

[39] E. LeDell and S. Poirier, "H2O AutoML: Scalable automatic machine learning," *7th ICML Workshop on Automated Machine Learning (AutoML)*, July 2020. [Online]. Available: https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf

[40] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[41] Y. Ueda and M. Ohara, "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, 2017, pp. 13–22.

[42] ([Online]) Stress-ng. [Online]. Available: https://kernel.ubuntu.com/ cking/stress-ng/

[43] R. Gao and Z. M. Jiang, "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, 2017, pp. 379–390.

[44] "Air quality monitor," [Online]. [Online]. Available: https://github.com/jlofw/air-quality-monitor

[45] D. Mosberger and T. Jin, "Httperf—a tool for measuring web server performance," vol. 26, no. 3, p. 31–37, Dec. 1998. [Online]. Available: https://doi.org/10.1145/306225.306235

[46] ([Online]) Prometheus. [Online]. Available: https://prometheus.io/

[47] ([Online]) Node exporter. [Online]. Available: https://github.com/prometheus/node_exporter

[48] ([Online]) cadvisor. [Online]. Available: https://github.com/google/cadvisor

[49] ([Online]) Grafana. [Online]. Available: https://grafana.com/

[50] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*.

[51] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

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

[53] M. Litoiu, "Optimization, performance evaluation and resource allocator (opera)." 2013. [Online]. Available: http://www.ceraslabs.com/technologies/opera

[54] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. the MIT Press, 2005.

[55] ([Online]) Apache jmeter. [Online]. Available: https://jmeter.apache.org/usermanual/component$_r eference.html$