Machine Learning based Interference Modelling in Cloud-Native Applications

Alexandru Baluta
balutaal@yorku.ca
York University

Joydeep Mukherjee
jmukherj@calpoly.edu
California Polytechnic State University

Marin Litoiu
mlitoiu@yorku.ca
York University

ABSTRACT

Cloud-native applications are often composed of lightweight containers and conform to the microservice architecture. Cloud providers offer platforms for container hosting and orchestration. These platforms reduce the level of support required from the application owner as operational tasks are delegated to the platform. Furthermore, containers belonging to different applications can be co-located on the same virtual machine to utilize resources more efficiently. Given that there are underlying shared resources and consequently potential performance interference, predicting the level of interference before deciding to share virtual machines can avoid undesirable performance deterioration. We propose a lightweight performance interference modelling technique for cloud-native microservices. 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. We evaluate our technique against realistic microservices on AWS EC2. Our technique outperforms baseline and competing techniques in MAPE by at least 1.45% and at most 92.04%.

CCS CONCEPTS

• **Software and its engineering → System administration**.

KEYWORDS

Microservice, Interference, Cloud Computing, Machine Learning

ACM Reference Format:
Alexandru Baluta, Joydeep Mukherjee, and Marin Litoiu. 2022. Machine Learning based Interference Modelling in Cloud-Native Applications. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3489525.3511677

1 INTRODUCTION

The DevOps paradigm promotes teams to manage the end-to-end lifecycle of an application themselves in order to promote ownership and enable continuous delivery of their application [14]. DevOps favours the microservice architecture in which an application is broken down into several smaller services so that each service has a single responsibility. The microservice architecture is well suited towards efficient scaling and in addition, horizontal scaling. Application developers are increasingly shifting away from the monolithic architecture in favor of the microservice architecture. Furthermore, microservice applications are frequently deployed as cloud-native applications on cloud platforms like Amazon Web Services (AWS) [3] or Google Cloud Platform [6].

Cloud-native microservices are frequently deployed in lightweight containers. Cloud providers offer container platforms tailored to the microservice architecture on which the application owner may even delegate management of the application runtime to the provider. In order to efficiently utilize resources, cloud-native applications are often co-located on the same underlying Virtual Machine and consequently compete for shared VM-level resources. While co-location is cost effective, it can negatively impact an application's Quality of Service (QoS). We refer to degradation of application performance due to resource competition from a co-located application as *performance interference*. Quantifying and mitigating performance interference is therefore important in sustaining good QoS. Furthermore, performance interference modelling techniques can be employed in application deployment strategies to predict whether co-locating two applications will result in performance degradation.

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, the 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. The static models are typically 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 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.

ICPE '22, April 9–13, 2022, Bejing, China

ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511677

Regression models are models that can either be static or dynamic, 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. Dynamic 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, dynamic 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 dynamic 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. We attempt to answer the following research questions with our approach:

RQ-1: 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 RQ1, 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. Our results show that our approach outperforms competing interference modelling techniques by at least 14.13% and at most 1271.37%.

RQ-2: How effective is a dynamic 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 RQ2, 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%.

This paper makes the following three contributions:

(1) We develop a static machine learning based interference modelling technique that outperforms competing state-of-the-art static modelling techniques by at least 14.13% and at most 1271.37%.
(2) We develop a dynamic machine learning based interference modelling technique that generalizes to unknown interfering applications at runtime, has minimal model training overhead, and outperforms competing state-of-the-art dynamic modelling techniques by at least 1.45% and at most 92.04%.
(3) We present a comparative analysis between our static and dynamic ML techniques.

The remainder of this paper is organized as follows. Section 2 discusses methodology. Section 3 describes the common environment and application setup used in addressing our research questions. Section 4 details the experimental setup and results of RQ1. Section 5 presents the experimental setup and results of RQ2. Section 6 frames the threats to validity. Section 7 details related works. Section 8 concludes this paper and discusses future work.

2 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 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.

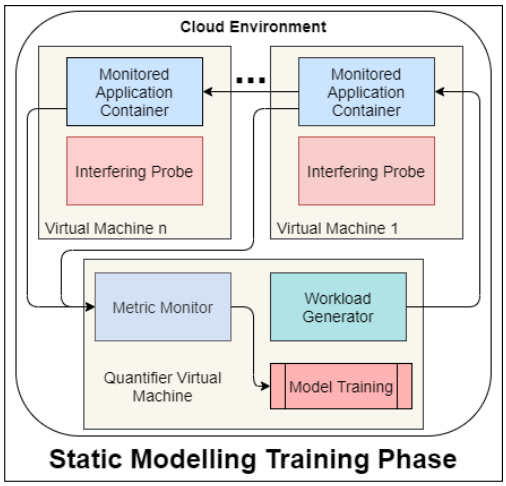

Figure 1: Overview of Interference Modelling Training Phase

2.1 Static Models for Interference

Our static modelling methodology consists of two phases: the *training phase* and the *runtime phase*. Figure 1 presents an overview of our static interference modelling training phase. 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 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.

The static models are deployed in the runtime phase to quantify the impact of interference on our target application's response time. In the runtime phase, as seen in Figure 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 further leverages deployed models to make predictions. As previously mentioned, a benefit of static modelling 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.

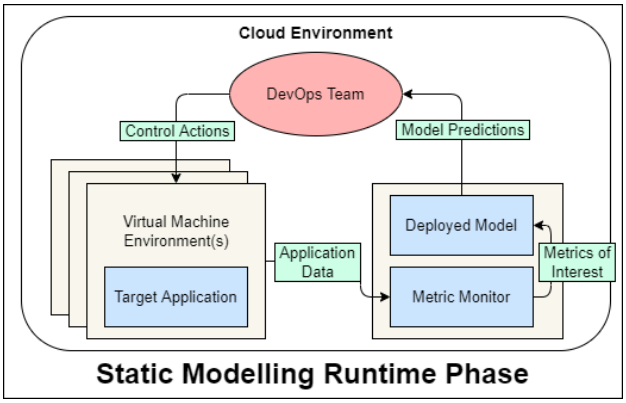

Figure 2: Overview of Interference Modelling Runtime Phase

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

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

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

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.

Suppose an application b is co-located on the same VM and also stresses resource k. Application b imposes its own resource utilization on the VM. The multi-class form of equation 1 that predicts the response time of application a is expressed as:

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

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 2, 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 [11]. 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 [11].

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.

We leverage OPERA [18], a tool that has been used in prior work for constructing LQMs for performance modelling of cloud-native applications [10]. 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 [18] is employed in the training phase to construct the LQM models. The LQM models are used for performance interference modelling of the target application a.

2.1.2 Static Machine Learning Models. Automatic Machine Learning (AutoML) frameworks aim to 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 dataset and output the best performing model. In addition, AutoML optimizes the model search space and may 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 [17] to construct ML models for performance interference modelling.

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 in an attempt to produce the best performing models.

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 as a regression problem.

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 modelling of the target application a.

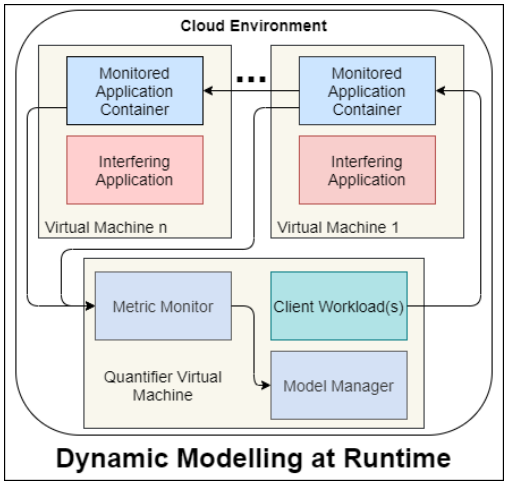

Figure 3: Overview of Dynamic Interference Modelling

2.2 Dynamic Models for Interference

Our dynamic modelling technique consists of one single phase; the runtime phase. All operations, including model training and deployment, are conducted at runtime. Figure 3 depicts an overview of our dynamic interference modelling technique. As in the static modelling approaches, the dynamic approach has a target application, interfering application, client workload(s), and Metric Monitor. In addition to these components, the dynamic modelling technique has a *Model Manager* as depicted in Figure 4.

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. When model training is done at runtime, this constitutes a dynamic modelling 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.

2.2.1 Gaussian Process Models. Gaussian Process (GP) [21] models are probabilistic models that utilize Bayesian theory to derive their predictions. Prior works [16, 22] have employed GP models with sliding windows in modelling response time of containerized applications. In particular, the recent work of Kang and Lama [16] 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 [16], 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 [16] formulate their GP models using

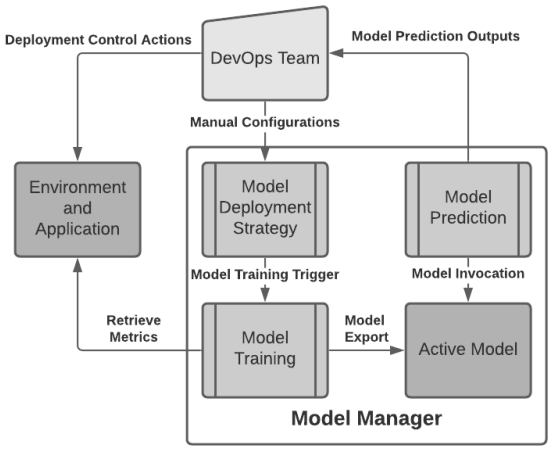

Figure 4: Overview of Model Manager

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.

2.2.2 Dynamic Machine Learning Models. Our Dynamic Machine Learning technique utilizes a Sliding Window Model Deployment Strategy in conjunction with the AutoML framework as detailed in section 2.1.2. Our dynamic 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 2.1.2. 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 [16, 22]. 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 training, 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.

3 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.

3.0.1 Target Application. The target application we use is called Acme Air [1] which is an e-commerce benchmark microservice previously used in other related works [24, 25]. 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.

3.0.2 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 [9]. stress-ng has been used in prior work [12] 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 [2]. 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.

3.0.3 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 [19] 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 [4] as the Workload Generator. The step size increase for the Air Quality Monitor is 20% CPU utilization.

3.0.4 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 VM level metrics, we utilize a metrics exporter known as Node exporter [8]. With respect to container level metrics, we use cAdvisor [5], 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 [7] is utilized to query, merge, and export VM 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 VM.

3.0.5 VM Deployment Strategy. We construct performance interference models for two deployment strategies. The 1VM deployment strategy consists of deploying our target application, Acme Air, along with one of the interfering applications on a single VM. 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. 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.

4 RQ1: STATIC ML FOR KNOWN INTERFERING APPLICATIONS

To quantify performance interference with respect to RQ1, we utilize the setup as presented in section 3 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 3 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 4.1 describes our experimental setup. Section 4.2 presents the results of our experiments.

4.1 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 2.1.2, 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 2.1.1. As mentioned before, LQMs have frequently been used in performance modelling of monolithic and microservice applications [10, 13, 23].

4.2 Results Analysis

4.2.1 Observed Resource Utilization. By varying the target and interfering application workloads, we observed a wide range of resource utilization values. The VM CPU and Memory utilizations each ranged from 2% to 100%. The target Acme Web container's CPU utilization, which is the primary resource it uses, ranged from 3% to 82%. The target Acme DB's Memory, which is the primary resource it uses, ranged from 1% to 59%. Incurring a wide range of resource utilization 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.

4.2.2 Observed Response Times. We report the response time ranges observed in our experimentation as shown in Table 1. 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.

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 1: Observed Response Time Ranges

4.2.3 Model Evaluation. We evaluate the effectiveness of models by the Mean Absolute Percentage Error (MAPE) as used in prior works [16, 20]. 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

$$MAPE = \frac{1}{n} * \sum_{i=1}^{n} \frac{R_{A,i} - R_{P,i}}{R_{A,i}} \quad (3)$$

We calculate the MAPE for each combination of interfering application and deployment strategy in use. Table 2 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, 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 2: 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 1, we observe a lower MAPE as seen in Table 2. 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.

5 RQ2: DYNAMIC 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 where static modelling techniques were employed, in this section we evaluate the performance of dynamic performance interference modelling techniques. Section 5.1 discusses experimental setup. Section 5.2 highlights the results of our dynamic modelling technique. Finally, section 5.3 presents a comparative analysis and

practical considerations between static and dynamic AutoML models and their performance across our different scenarios.

5.1 Experiment Setup

We compare the performance of our dynamic modelling technique as described in Section 2.2.2 against baseline and competing techniques. To that end, we leverage the static LQM technique as depicted in section 2.1.1 as our baseline technique. Furthermore, we leverage a state-of-the-art Gaussian Process technique as presented in section 2.2.1 which itself is a dynamic technique. Dynamic AutoML models are constructed following the methodology in section 2.2.2.

We employ 1VM and 2VM deployment strategies for our target application Acme Air as described in Section 3. 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 dynamic 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.

5.2 Results Analysis

Response time ranges are presented in Table 1 which are also consistent across our experimentation. In Table 3, 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.

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 3: Dynamic ML vs. Dynamic GP and Static LQM

5.3 Comparative Analysis and Practical Considerations

We present a comparative analysis of the best static and dynamic AutoML techniques as captured in Table 4. The table is partitioned by 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 Dynamic AutoML.

It's noteworthy that the static AutoML technique with interference visibility outperforms its counterpart dynamic AutoML technique. However, the applicability of static techniques with interference visibility is limited to scenarios where the application owner knows what interfering applications 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, we observed the MAPE degrades by approximately 2x - 6x. It is therefore unwise to use static models to predict interference impact induced by unknown applications.

The dynamic AutoML technique performs comparatively well versus its static AutoML counterparts. While it does not outperform the static AutoML models that assume interference visibility, the dynamic AutoML models perform within 2.16% and 12.66% of their counterpart. The dynamic 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 the dynamic modelling is highly recommended for modelling the interference cause by applications unknown at runtime.

VMs	Interfering App.	AutoML Technique	MAPE
1VM	stress-ng	Static	3.87%
1VM	stress-ng	Dynamic	6.03%
1VM	Acme Air	Static	26.53%
1VM	Acme Air	Dynamic	38.29%
1VM	IoT	Static	19.26%
1VM	IoT	Dynamic	26.74%
2VM	stress-ng	Static	5.88%
2VM	stress-ng	Dynamic	18.54%
2VM	Acme Air	Static	14.92%
2VM	Acme Air	Dynamic	24.04%
2VM	IoT	Static	29.39%
2VM	IoT	Dynamic	38.85%

Table 4: Comparison of Static and Dynamic ML

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. [26].

We note that an internal threat to validity is the configuration of the fixed number of datapoints considered in the sliding window

of our dynamic 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.

7 RELATED WORK

Several performance engineering techniques model an application's runtime behavior in order to predict application performance [11, 15, 16, 20, 22]. Prominent performance engineering techniques often construct Layered Queuing Models (LQM) which are an extension of Queuing Network Models (QNM) [10, 13, 23] or Regression models [15, 16, 20, 22].

Prior works have utilized LQMs as performance models for both monolithic and microservice software applications [10, 13, 23]. Barna et al. [10] 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 [23] similarly utilize the LQM for application response time predictions. These serve as an input to their proposed genetic algorithm for optimizing cloud-native application scale and placement. Gias et al. [13] propose ATOM, an autoscaling controller for microservices that leverages the LQM to evaluate potential performance gain resulting from deployment decisions.

Iqbal et al. [15] build polynomial regression models to predict response time of applications hosted on virtual machines. They subsequently leverages response time predictions to assess risk of breaking service-level agreements and if so, their technique autoscales the application. Rahman and Lama [20] 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. [22] propose an online Gaussian Process method that uses sliding windows to make response time predictions for containerized monolithic applications. Based on those predictions, their method decides whether to vertically scale the application or not. Kang and Lama [16] 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.

8 CONCLUSIONS AND FUTURE WORK

We propose static and dynamic Machine Learning techniques for performance interference modelling in cloud-native applications. No application instrumentation is required to obtain these metrics and collecting the metrics imposes minimal overhead. We evaluate our technique 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. Likewise, our dynamic ML technique is effective in quantifying performance interference and outperforms competing state-of-the-art techniques by 1.45%-92.04%. Furthermore, our dynamic 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 modelling technique with microservice placement strategies. Quantifying the potential impact of microservice co-location provides a meaningful signal to derive suitable microservice placement.

REFERENCES

[1] [Online]. Acme Air. https://github.com/acmeair
[2] [Online]. Air Quality Monitor. https://github.com/jlofw/air-quality-monitor
[3] [Online]. *Amazon Web Services.* https://aws.amazon.com/
[4] [Online]. *Apache JMeter.* https://jmeter.apache.org/usermanual/component_reference.html
[5] [Online]. *cAdvisor.* https://github.com/google/cadvisor
[6] [Online]. *Google Cloud.* https://cloud.google.com/
[7] [Online]. *Grafana.* https://grafana.com/
[8] [Online]. *Node Exporter.* https://github.com/prometheus/node_exporter
[9] [Online]. *Stress-ng.* https://kernel.ubuntu.com/~cking/stress-ng/
[10] Cornel Barna, Marin Litoiu, Marios Fokaefs, Mark Shtern, and Joe Wigglesworth. 2018. Runtime Performance Management for Cloud Applications with Adaptive Controllers. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 176–183.
[11] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. 2008. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering* 35, 2 (2008), 148–161.
[12] Ruoyu Gao and Zhen Ming Jiang. 2017. An exploratory study on assessing the impact of environment variations on the results of load tests. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 379–390.
[13] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1994–2004.
[14] Michael Httermann. 2012. *DevOps for Developers (1st ed.).* Apress, USA.
[15] Waheed Iqbal, Abdelkarim Erradi, Muhammad Abdullah, and Arif Mahmood. 2019. Predictive auto-scaling of multi-tier applications using performance varying cloud resources. *IEEE Transactions on Cloud Computing* (2019).
[16] Peng Kang and Palden Lama. 2020. Robust Resource Scaling of Containerized Microservices with Probabilistic Machine learning. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 122–131.
[17] Erin LeDell and Sebastien Poirier. 2020. H2O AutoML: Scalable Automatic Machine Learning. *7th ICML Workshop on Automated Machine Learning (AutoML)* (July 2020). https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf
[18] Marin Litoiu. 2013. Optimization, Performance Evaluation and Resource Allocator (OPERA). (2013). http://www.ceraslabs.com/technologies/opera
[19] David Mosberger and Tai Jin. 1998. Httperf—a Tool for Measuring Web Server Performance. 26, 3 (Dec. 1998), 31–37. https://doi.org/10.1145/306225.306235
[20] Joy Rahman and Palden Lama. 2019. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 200–210.
[21] C. E. Rasmussen and C. K. I. Williams. 2005. *Gaussian Processes for Machine Learning*. the MIT Press.
[22] Shashank Shekhar, Hamzah Abdel-Aziz, Anirban Bhattacharjee, Aniruddha Gokhale, and Xenofon Koutsoukos. 2018. Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 82–89.
[23] Yasir Shoaib and Olivia Das. 2019. Cloud VM provisioning using analytical performance models. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 68–72.
[24] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE, 1–10.
[25] Yohei Ueda and Moriyoshi Ohara. 2017. Performance competitiveness of a statically compiled language for server-side Web applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–22.
[26] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen. 2000. *Experimentation in Software Engineering*. Kluwer Academic Publishers.