2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) | 979-8-3503-1192-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/SEAMS59076.2023.00018

Towards a Robust On-line Performance Model Identification for Change Impact Prediction

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

Joydeep Mukherjee
California Polytechnic State University
CA, United States
jmukherj@calpoly.edu

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

***Abstract*—In self-adaptive systems, model-based control assumes decisions are taken based on a model that is identified at run-time. The model is built by measuring the control inputs, disturbances, and outputs of the controlled system and fitting the data into a function. Models can be accurate locally, that is, for data already seen by the system and by the model identification method. However, many times an Autonomic Manager (AM) needs to move the cloud-native applications into new operational points, e.g. by adding new applications to the shared environment, scaling applications or consolidating resources. There are no data points yet for these new operational regions to have any certainty that the prediction models are accurate. In this paper, we propose a method to identify a model that predicts metrics at any unexplored operational point of a cloud-native application. The method is based on a lightweight Look-Ahead Scanner (LAS) mechanism that explores different operational points by injecting controlled short-lived load. We evaluate our method on realistic applications deployed on public clouds. We show that the proposed method can build models that outperform the state of the art ML models by 42%.**

***Index Terms*—Microservices, Performance Modeling, Cloud Computing, Machine Learning**

I. Introduction

Modern applications are often composed of lightweight containers that are hosted on Virtual Machine (VM) instances on public cloud platforms. These applications are increasingly following a service architecture and are being deployed as services running inside containers on cloud platforms such as Amazon Web Services or Google Cloud Platforms [1], [2]. Containers belonging to different applications are often co-located on the same VM to utilize resources more efficiently. Sharing resources by co-locating application(s) and their containers can reduce the cost and energy footprint [3], [4]. However, these co-located containers can often compete for VM-level shared resources, which can in turn negatively impact the Quality of Service (QoS) for the applications running inside these containers. Therefore, it is important to model the performance impact of co-locating containers at run-time so that adaptive actions can be taken to ensure good QoS for the cloud-native applications.

Static models have been used in the past to quantify the impact of co-locating different cloud-native containers on the same VM. Static models are trained in a training phase and then deployed at run-time to leverage their prediction ability. Static models benefit from their training on a large amount of historical data which results in better accuracy. However, modern applications deployed in a DevOps environment are dynamic in nature, i.e., the state of such applications in terms of number of containers, incoming workload, and underlying features change frequently at run-time. Static models do not lend themselves well to frequent changes in application state since these models have to be re-trained every time the application state changes. Furthermore, it is challenging to obtain historical training data for static models that capture the future dynamics of a cloud-native application for all of its possible states. In light of this observation, dynamic models that can be quickly constructed at run-time by taking into account the current state of the application are often preferable over static models [5].

Model based control in self-adaptive systems utilizes dynamic models created at run-time to make adaptive decisions [6]. These models are built around an *Operational Point*, defined as the performance utilization, workload patterns and other configurations that describe the software system at its normal behavior. However, no studies have shown how these models extrapolate to regions far from the current operational point. A running 'in-production' cloud application can experience a change in its operational point when there is a change in the cloud environment. Changes in the cloud environment are frequent, e.g. co-location, patching, scaling, etc. When a new co-located application is deployed and share the same environment, containers sharing the same VMs, network or services can interfere with each other and affect the performance of in-production applications running on those containers [7], [8]. We need to be able to predict the performance impact on production applications when new applications are co-located. Similar situations arise when consolidating resources, that is, dynamically re-distributing the containers of the same applications, or when the autonomic manager anticipates an increase in the load and evaluates the impact on the application Service Level Agreement (SLA). We define this problem as Change Impact Prediction (CIP). CIP is akin to an autonomous driving radar scanning of future positions, such as intersections, obstacles, etc that informs the self-driving software on the impact of its decisions. Being able to scan and look ahead from the current operational point and predict dynamically the impact of a drastic change will allow us to make appropriate adaptation decisions to prevent end-

979-8-3503-1192-1/23/$31.00 ©2023 IEEE
DOI 10.1109/SEAMS59076.2023.00018

user performance from being significantly affected and keep our cloud application(s) within the SLA.

In this work, we conduct experimental studies to verify our hypotheses that models trained only on historical data are inefficient in predicting the performance impact of a significant change such as when another application is deployed on the same VM(s) along production applications. Then, we propose a dynamic modeling technique for the Change Impact Prediction in cloud native applications. This technique uses a *Look-Ahead Scanner (LAS)* approach that injects controllable disturbances at run-time to scan ahead different operational points, and therefore enables a model that is accurate around these operational points. These controllable disturbances should be injected with care so that production applications do not breach the SLA. Therefore, our research questions for this work are as follows:

- **RQ-1:** What is the prediction accuracy of change impact models built with only historical data and how do they compare with those built using a *Look-Ahead Scanner (LAS)* mechanism?
- **RQ-2:** How effective is the LAS-based model in predicting the impact of changes on production cloud applications?

For **RQ-1**, we build models around operational points and run change prediction experiments. We also develop a method to inject controllable disturbances to an application environment to produce data-points for unexplored target operational data points without breaching the SLA. These data points are used as supplemental training data for the model to cover the target operational points of the cloud environment. For **RQ-2**, we evaluate how performance models trained with the data points produced by these injected controllable disturbances can predict the effect of co-locating a new application along a production one.

The original contributions of this paper are as follows:

- We show that models built on historical data do not predict accurately metrics outside of operational regions; we show that on Machine Learning, Regression and Queuing Network models.
- We propose a *Look-Ahead Scanner (LAS)* that injects a short-lived load and collects additional data at the target operational points;
- We show that models built with the data collected through our LAS are accurate and outperform those built with historical data by reducing the mean absolute percentage error (MAPE) by up to 42%.
- We validated the findings through experiments on public clouds and across many operational point changes.

The remainder of the paper is organized as follows. We discuss the background of current research in performance modeling in self-adaptive cloud systems in Section II. We discuss the motivations and foundations of our work in Section III. We then present our methodology architecture and approach in Section IV. We then discuss our experimental setup and implementations in Section V. Finally, section VI shows the results of our approach compared to other models.

II. Related Work

There is past work that studied the performance analysis and modeling of microservice based architecture for cloud native applications [9]–[12]. Jindal et al. [9] build a methodology to identify the maximal rate of requests of a microservice without violating the Service Level Objective through a performance model build using microservice sandbox simulations. Khazae et al. designed a microservice platform to study performance indicators and then designed an analytical performance model which can be used for capacity planning for their microservice platform [11]. However, these past research works focus on modeling and planning for existing microservice applications as the current operational point. There are techniques that profile and measure performance interference in cloud environments [13], [14] and for cloud-native microservice applications [15], [16]. However, while we look at the performance impact due to an interference, we focus on modeling to predict the performance impact of a change in the cloud environment rather than detecting the interference in the cloud environment. In addition, these research works mainly focus on building a performance model through a simulation-based methodology or evaluation on an offline VM environment. In Self-Adaptive systems, run-time performance models are used by the MIAC to make adaptive decisions when deploying new services. Wang et al. [17] propose a self-adaptive resource management framework that uses a combination of a QoS model trained through historical data on the cloud and PSO algorithm to perform on-line resource allocation. There has been research on self-adaptive managers and techniques that use performance models focusing on cloud-native applications [18]. Machine learning can also be used to train models to be used for self-adaptive systems [19]. However, the run-time models used by self-adaptive managers utilize historical data and can be unaware of new deployments. There are research works similar to this concept of deploying exploration mechanisms [20]–[22] that are in different domains outside of cloud-native microservice based architectures. The main inspiration for our Look-Ahead Scanner mechanism originates from the Networking domain rather than cloud and self-adaptive systems. Chen et al. [23] state the difficulty of quantifying the effects of changing network characteristics on end-to-end performance, specifically logical link latency. Logical Link Latency is the network latency from one application to another that spans multiple physical links. Chen et al. present a measurement-based approach to quantifying the impact of change in logical link latency on end-to-end performance and utilize delay injection and spectral analysis on an existing link gradient to explore end-to-end performance in new and untested configurations. Their method is simple, unobtrusive to production environments, and can be isolated from the other transactions in the network. We take inspiration from their work and apply it to the domain of cloud-native microservice based applications through a microservice based Look-Ahead

Scanner mechanism deployed on a cloud environment to predict the change impact of new deployments to share the environment.

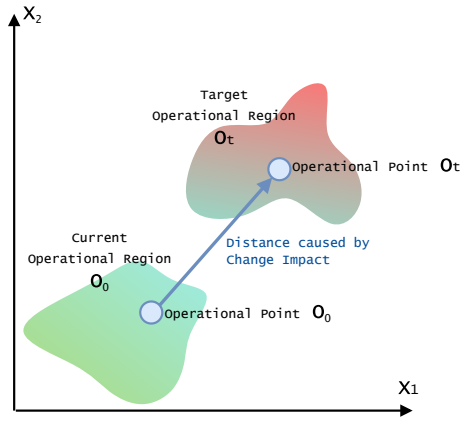

Fig. 1: Operational Regions and Points O_0 and Target Operational Regions and Points O_t

III. Motivation and Problem Formulation

In this section, we describe the concepts and definitions behind the problem and our proposed solution to enable Change Impact Prediction on existing cloud services. To illustrate the points, we use the co-location of applications as a use case; however, the proposed method can be used in other adaptation scenarios. Assume an application that runs on a VM instance(s). We can define the *Operational Point* as a time snapshot of the application and its environment: $O = (Env, C, P, W)$, where O includes the cloud environment with its set of VMs and configurations, $Env = (VM_1...VM_K)$, the set of containers and their configuration, $C = (C_1...C_n)$, the application's performance resource utilization $P = (R, X, U_1, ...U_K)$, where R and X are the application response time and throughput respectively and U_K is the utilization of VM_K. The workload, W, is represented by the number of requests per second or the number of simultaneous users.

The operational point is affected by the cloud environment variability, the changes in performance utilization due to the workload fluctuations, and other software and hardware configurations. It is therefore consequential that the application runs in many operational points that form an *operational region*. Figure 1 visualizes the operational points in their respective operational regions and the distance between the *Current Operational Point* and the *Target Operational Point* in a simplified two-dimensional space, with x_1 and x_2 two variables from P.

1) Current Operational Point O_0: The Current Operational Point, O_0, is the operational point of the application at its normal behavior before any change in the application or cloud environment occurs. This is where the application functions at its average day-to-day operation, where we can observe the application utilization, workload, and end-user metrics at its normal behavior.

Example. Consider an E-commerce application, a, with an Env = VMs, VM_1 and VM_2, running on containers, C= C_1 and C_2, which includes front-end server(s) and a back-end database. The performance utilization of the VM(s) are $U_1 = 12\%, U_2 = 5\%$, and the response time and throughput at that point are $R = 2ms$ and $X = 108req/s$ at the day-to-day browse and purchase workload patterns $W = 15$ requests per second. Daily changes in W can move the current operational point O_0 around, but will remain within the operational region O_0 as seen in Figure 1.

2) Target Operational Point O_t: The Target Operational Point, O_t, is the unexplored operational point that the cloud environment will move towards from O_0 when a change occurs. For a self-adaptive system, it is important to understand the performance impact on the production cloud environment and applications before enacting the change. The cloud environment moving towards O_t may have a negative impact on performance metrics that will ultimately lead to poor QoS. An example of a target operational point O_t can be a deployment of a new co-located IoT analytics application, b, to share the same environment as the e-commerce application.

This IoT analytics application has not been deployed before on Env, therefore, its effects are unknown. The co-located deployment would increase the number containers C_n from $n = 2$ to $n = 5$. It changes the utilization of VM1 and VM2 to $U_1 = 23.54\%$ and $U_2 = 23.87\%$, pushing the operations of the e-commerce application a from O_0 to an unexplored region with target operational point(s) O_t.

Figure 1 shows the current operational point O_0 and the target operational point O_t, in a simplified two-dimensional space (x_1, x_2). The two operational points reside in operational regions, created by cloud variability and workload fluctuations.

3) Change Distance: It is expected that the performance of the application at the new target operation point depends on how much load we add to the shared environment. We can define the change distance DI between the points as the difference between the current and target VM utilization. Therefore,

$$[U_1^t...U_K^t]^T = [U_1^0...U_K^0]^T + DI \quad (1)$$

where U_K^0 is the VM utilization at O_0 and U_K^t is the VM utilization at O_t. DI is the change distance load vector we add to U_K^0 to move the environment to a new operational point O_t.

Example: Following our example, the distance introduced by the new deployed IoT application, is $DI = [11.54, 18.87]^T$.

4) Problem Formulation: Since the application is running around O_0, we can collect data and train a model around that point. A model $M(O_0, O_0)$, is trained to predict performance in and around the region O_0. We can use this model to predict performance in O_t, denoted by $M(O_0, O_t)$. Our conjecture is that the prediction will not be accurate since it uses historical data that do not capture the O_t region. Therefore, we need to be able to sample around the target point to augment the data and retrain a new model. A model retrained by using data points in and around the target point and used

to predict performance in the training point is denoted by $M(O_t, O_t)$. Our conjecture is that this model is more accurate than $M(O_0, O_t)$. Our problem can be formulated as follows:

Given an operational point O_0 and the distance D to a target operational point, O_t, find a method to build a new model around O_t, $M(O_t, O_t)$, such that the existing SLAs are not breached.

To solve the problem, we propose to build a Look-Ahead Scanner (LAS) mechanism that causes load perturbations along the distance from the current operational point towards O_t. The Look-Ahead Scanner mechanism can be used to incrementally increase the utilization at O_0 in small steps such that the end-user metric does not exceed a predetermined threshold that violates the application SLA. In this way, we are able to collect several data points of future positions that can capture the impact on existing applications at O_t.

IV. Look-Ahead Scanner Methodology

In this section, we discuss the overview of our methodology to construct the Look-Ahead Scanner, deploy it on a production cloud environment along existing applications, and build a more robust run-time performance model for change prediction.

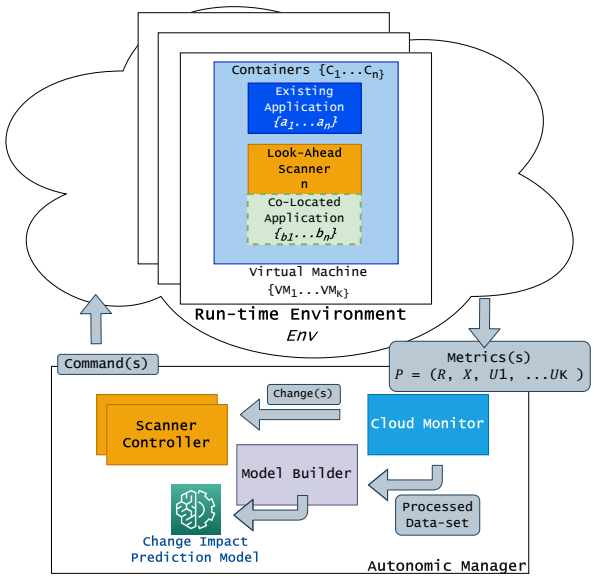

Fig. 2: Overview of Look-Ahead Scanner in Production

A. Methodology and Architecture Overview

Figure 2 shows the architecture of the methodology of the LAS mechanism. We have three key components in our methodology, (1) a Look-Ahead Scanner, (2) a Scanner Controller, and (3) a Model Builder. The Look-Ahead Scanner is deployed on the VM(s) in Env of O_0. The Scanner Controller is part of the Autonomic Manager and sends configuration command(s) that instruct the Look-Ahead Scanner(s) on what to do on the VM(s). The performance metrics produced by the Look-Ahead Scanner are collected by the Cloud Monitor and the processed data is streamed to Scanner Controller and Model Builder. The Model Builder uses the data to build the run-time performance model(s).

B. Look-Ahead Scanner

The Look-Ahead Scanner explores the performance space of the existing cloud environment and its services at unexplored target operational points O_t. It needs to be lightweight and be able to inject CPU, memory, or I/O perturbations.

Since we target cloud-native application, deployed as a set of containers C in the cloud environment Env, we can use containerization to inject our mechanism into the cloud environment. Therefore, our run-time *Look-Ahead Scanner (LAS)* is a lightweight containerized service that can be deployed on the cloud environment where cloud-native in-production application(s) resides. The LAS can induce disturbances on the cloud environment by stressing resources. As a container, it can easily be deployed, managed by a Scanner Controller, independent of the modeled applications life cycle.

C. Scanner Controller

LAS is managed by a Scanner Controller located externally and separately from the existing cloud environment and is part of an Autonomic Manager that manages the system. The Scanner Controller is responsible for managing the Look-Ahead Scanner container(s) on a single or multi-VM architecture. By incorporating a Scanner Controller as the central controller, we are able to orchestrate multiple Look-Ahead Scanner(s) by sending unique instructions to each of them. This allows our methodology to mimic applications deployed on multiple VMs. The LAS has three states that are managed and controlled by the Scanner Controller: Deployed, Active, Destroyed. When the LAS is *deployed* by the Scanner Controller, it becomes *active* and awaits instructions from the Scanner Controller. The Scanner Controller instructs the LAS(s) on which resources to stress, the length of time for each stress, and in what increments to increase the stress value, supporting multi-resource control and tunable resource perturbation. Constraints can also be implemented by the Scanner Controller when handling the LAS. The Scanner Controller can periodically check the outputs produced by LAS, and be able to stop or modify the LAS if the constraints are violated. When the constraints are violated or the LAS has completed its perturbations, then the Scanner Controller can *destroy* the LAS from the run-time environment.

D. Cloud Monitor

The Cloud Monitor collects the utilization metrics, U_K, of the virtual machine(s) VM_K and the container(s) C_n, such as the CPU Utilization, memory Utilization, I/O, the applications' response time, R, and throughput, X, required by the Service Level Agreement. It can use existing cloud-native instrumentation to minimize overhead. It streams measured data to the Model Builder.

E. Performance Model Builder

The Model Builder is a framework that can build run-time performance models for use in self-adaptive scenarios. The Model Builder produces a Change Impact Prediction Model that can be used to predict performance impacts at the target operational point O_t. In this paper, we evaluate performance-specific models, such as Queuing Network Models (QNM), Machine Learning (ML) models, and Linear Regression Models (LM), however other models can work as well.

1) Queuing Networked Models: represent the software and hardware components of a system as a network of queues. Co-locating two applications can be modeled as follows. Assume an existing in-production application *a* that runs on its own cloud server and does not share that server with other application(s). The end-to-end delay or response time of one request is given by the processing time of the application software and hardware resources, plus waiting time at the same processing resource. Each hardware resource *k* can be used to calculate the Demand D of application *a*, $D_{k,a}$. Without loss of generality, the waiting time can be measured by a state variable of the resource *k*, called utilization U_k. In absence of other applications on the cloud server, the utilization of *k* is produced by application *a*, that is $U_k = U_{k,a}$. Using the Open Mean Value Analysis Model from Queuing Networks [24], the end-to-end delay of an application R_a can be expressed as:

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

where *k=1 ... K* denotes the software and hardware resources used by the application *a*.

A co-located application *b* will impact the in-production application *a* through utilization U_k, which can be expressed as:

$$U_k = U_{k,a} + U_{k,b} \tag{3}$$

where $U_{k,b}$ is the additional utilization brought by the application *b*. With application *b* now sharing the cloud environment, the response time of application *a* will become:

$$R_a = \sum_{k=1}^{K} \frac{D_{a,k}}{1 - (U_{k,a} + U_{k,b})} \tag{4}$$

From Equation (4), we can infer that we can use utilization as a proxy for moving the application to a target operational point. The operational point O_0 is characterized by $U_{k,a}$ and O_t by $U_{k,a} + U_{k,b}$, while the distance DI between the operational points is $U_{k,b}$. Additional co-located application(s) can also be implemented in Equation(s) (3)-(4) to model the impact on application a. A QNM model can be tuned at the operational point O_0 (cf. Eq. (2)) by sampling the throughput X and the utilization U and estimating D using the Kalman filter [25], [26]. Once calculated, D can be used to evaluate the response time at point O_t using Equation (4). The assumption is that the demands $D_{k,a}$ do not change with operational points. The QNM model tuned at O_0 to evaluate the response time at O_t is defined as $QNM(O_0, O_t)$. A similar approach can be used to tune a QNM model in O_t. LAS can generate controlled perturbations, Kalman filter will estimate D by sampling through X and U of the application in O_t. This model tuned at O_t to evaluate the response time at O_t is defined as $QNM(O_t, O_t)$.

2) Machine Learning Models: model the system as a black box. We can explore many models, [27], to explore a wide variety of configurations, parameters, and options. Using the dataset provided by Cloud Monitor, periodically or on-demand, we can train multiple ML models. The features used for training are U_k, *k=1 ... K* and the predicted metric is R_a.

$$R_a = f(U_1..U_K) \tag{5}$$

We are able to build two types of ML models. The initial model, $ML(O_0, O_t)$, is defined as a ML model trained using the dataset where application *a* is at O_0 to predict the R_a at O_t. Using the LAS mechanism, we can build $ML(O_t, O_t)$ which is defined as the ML model trained using the dataset generated by the LAS to move the application towards O_t and predict R_a at O_t.

3) Linear Regression Models: make the assumption that the systems are linear around the operational point. We build gradient models, that is functions that approximate the estimated metric R_a with a linear combination of utilization gradients. That is equivalent to approximating Equation (2) with a first-order Taylor series [28]:

$$R_a = R_a^{O_0} + \sum_{k=1}^{K} m_k * \Delta R_a / \Delta U_k \tag{6}$$

where $R_a^{O_0}$ is the response time in the operational point O_0, m_k are the coefficients to be identified and ΔR_a are the changes in response time caused by LAS ΔU_k changes in utilization. To identify m_k, we inject changes around the target operational point. The initial model, $LM(O_0, O_t)$, is defined by building the model with a dataset at O_0 to predict the response time at O_t. The LAS model, $LM(O_t, O_t)$, is defined by building the model with a dataset generated through the LAS at O_t to predict the response time at O_t.

F. Look-Ahead Scanner Run-Time Algorithm

We now describe how we can use LAS to explore what if we move the operational point to O_t. As such, LAS acts as a proxy for a co-located application that is to be deployed on the cloud environment or for any other run-time performance adaptation we would like to make. The main motivation of our methodology is to be able to provide a more accurate run-time performance model on the existing cloud environment and application while staying within the SLA. Essentially, we need to find the balance between accuracy, low overhead, and keeping the end-user delay within the SLA requirements.

To start with, we assume that we already have a model that is valid at the current operational point, O_0, utilized by the MIAC. Our model does not have data points for O_t, and therefore it is uncertain that the current model can predict

performance around that point. In this paper, LAS follows a Rolling Hill algorithm [29] to sample new data points toward O_t. However, the algorithm can be replaced with another algorithm.

Algorithm 1 LAS Data Collection

```
1: Input O_0, DI = [DI_1..DI_K], ΔU, R_threshold
2: Output DS = {}
3: Start LAS(s) in Env
4: for i=1 to K do
5:    Instruct LAS to add ΔU load to VM_K within distance DI_K
6:    Read P_m = (R_m, X_m, U_m,1...U_m,n)
7:    if R_m > R_threshold then
8:       Stop LAS(s) and EXIT
9:    else
10:      Add P_m to DS
11:   end if
12: end for
```

The Scanner Controller deploys the LAS on the existing production environment where our running application *a* resides and where other applications *b* will be co-located. We define a threshold value, $R_{threshold}$, below the SLA requirements, that LAS cannot exceed in its exploration. We can express $R_{threshold}$ as:

$$R_{threshold} = R_{SLA} \times X\% \tag{7}$$

where *X* is a configurable value, for example, 80%. This allows us to deploy the LAS dynamically at run-time and explore the operational points of the exact production environment and its configurations without breaching SLAs.

Algorithm 1 is executed by the Scanner Controller. It has the following inputs: the current operational point O_0, the distance to O_t, $DI = [DI_1..DI_K]$, a load increment ΔU and an upper limit for the response time, $R_{threshold}$. We select our $R_{threshold}$ based on the $X\%$ value that application *a* will not exceed while the LAS is running. The LAS will move the application *a* towards O_t to collect the data points to be added to the output dataset, $DS = ()$.

The Scanner Controller begins the process by deploying LAS and then instructing LAS to induce each of the perturbations on the VMs. LAS will generate additional ΔU load on the VM(s) at each step of the algorithm (line 5), where ΔU is the incremental increase in utilization. In each step, the Scanner Controller collects measured performance metrics (line 6), P, including the measured response time, R_m and measured utilization values, $U_{m,1}...U_{m,n}$, and adds them to the output dataset, DS. R_m is compared with the threshold response time, $R_{threshold}$, (line 7) and if the measured response time, R_m, is under the threshold value, $R_{threshold}$, then the Scanner Controller continues the algorithm. However, if R_m is greater than $R_{threshold}$, then the Scanner Controller stops the LAS and removes it from the VM(s).

This process can be repeated multiple times, and DS can then be continually updated and passed to Model Builder to build a more robust and accurate model. In this work, we do not focus on the optimal exploration of the points; we focus only on feasibility. Other algorithms can be used to explore the space within DI in a more efficient or optimized approach.

V. Experimental Validation

In this section, we describe the testbed setups and experiments that answer our research questions. We run our experiments in the AWS Cloud on EC2 VMs. We utilized m4.large VMs running Ubuntu 18.04 and Docker 20.10.12. Each of the VMs was allocated 2 VCPUs, 8 GB of memory, and 20GB of Elastic Block storage. The first set of experiments runs an in-production application *a* and the Look-Ahead Scanner on one and two VM deployment configuration. The second set of experiments runs the application *a*, the co-located application *b* and the Look-Ahead Scanner(s) on one and two VM configurations.

A. Test-bed Setup

1) In-Production Application a*:* As the in-production application *a*, we use a Web benchmark application called Acme-Air [30] developed by IBM. Acme-Air is an implementation of a multi-tier airline e-Commerce application composed of two service components. The application-tier component is a front-end Node.JS server connected to a data-tier component, a MongoDB Database. These components are deployed as Docker containers that can run on different cloud platforms.

2) Co-locating Application b*:* The application *b* we intend to co-locate with Acme-Air, is an IoT Air Quality Monitoring application that utilizes an MQTT workload. The IoT application is composed of three container components: Mosquitto, NodeRed and InfluxDB. Mosquitto [31] is an open source message broker for the MQTT protocol for sensors to publish and subscribe messages. NodeRed [32] is a flow-based development tool built in Node.JS to connect APIs, hardware services, and sensors. InfluxDB [33] is an open source time series database platform. Air quality sensor data is published through Mosquitto. NodeRed collects and processes data from Mosquitto whenever it is published, and the sensor data is then stored in InfluxDB to be viewed. The in-production and co-located application(s) have been selected in our experimentations as they are well-known industry type representative application(s) [34], [35].

3) Look-Ahead Scanner: The Look-Ahead Scanner was built and deployed as a lightweight Node.JS [36] application, and its dependencies are containerized to be deployed on the Docker platform. The Node.JS application interacts with Stress Tools [37] that are packaged inside the container. For the Cloud Monitor, we deploy Prometheus to monitor the VM(s). Prometheus [38] is an open-source monitoring system and time-series database that can collect performance metrics at run-time. We utilized three models in our experimentation. We use scikit-learn [39], a machine learning library for the Python, to build Regression Models. We use Opera [40] to build our QNM models. We use the H20 AutoML frameworks [27] to build our AutoML models. The H20 AutoML frameworks train

and evaluate a variety of ML models based on the dataset from our LAS, and outputs the best performing model. The framework considers several ML algorithms such as Deep Neural Networks, Gradient Boosting, XGBoost and Stacked Ensembles.

4) Operational Points and Regions: The workloads W will emulate different loads and load mixes for the in-production application *a* and the co-located application *b*, Acme-Air and IoT Air Quality Monitoring, respectively. We use Httperf as the workload generator for Acme-Air. The Acme-Air workload represents a default workload mix that is provided by the Acme-Air application. To validate the answers to our research questions, we consider three *current operational regions* for the application *a*: light, medium and heavy. The light workload is within 5 to 15% CPU Utilization, the medium workload is within 15 to 20% CPU Utilization, and the heavy workload is within 40 to 60% CPU Utilization. The regions are covered in step sizes that emulate different current operational points, O_0: for the light and medium workloads the step size is approximately 5 to 8% CPU Utilization, and the step size for the heavy workload is 10 to 12% CPU Utilization. For the Air Quality Monitoring Application, we use Jmeter [41] as the workload generator. The workload for the Air Quality Monitoring application is MQTT-based and, to cover many target operational points, it is increased approximately at a step size of 10% CPU Utilization. We configured each Httperf and Jmeter workload to run for a duration of *x* = 100 seconds and for N = 100 iterations so that we account for the cloud variability. The value of *x* is configurable depending on the type of application and cloud environment for experimentation.

B. Experiment Setup

To account for deployment variability, we consider that each application is deployed in two configurations. The 1VM setup hosts the Node.JS and MongoDB containers of Acme-Air on a single VM instance. The 2VM setup hosts the Node.JS and MongoDB containers on two separate VM instances. In the remainder of the section, the VM that hosts Node.JS will be *VM 1*, and the VM that hosts MongoDB will be *VM 2* for the 2VM setup. The Look-Ahead Scanner(s) is deployed on these VMs to inject controlled load. For our AutoML machine learning model building, we do an 80/20% split of our dataset for the testing and training data.The training data for our models consists of the CPU utilization of each of the containers, and the host CPU Utilization of each CPU core.

1) Experiment One: Feasibility of the Performance Impact Model: Our first experiment answers **RQ-1** by comparing the prediction accuracy of change models built in the operational regions of O_0 with those build around the target region O_t. The distance between O_0 and O_t is variable and emulated by an utilization increase on the VMs. In this experiment, we only show the Machine Learning Models results; the other models perform worse, as will be evident in the second experiment. An $ML(O_0, O_t)$ is a model trained around the operational point O_0 used to predict performance metrics in O_t; An $ML(O_t, O_t)$ is a model trained using sampled data from both O_0 and O_t regions and used to predict performance metrics in O_t.

To train $ML(O_0, O_t)$, we run Acme-Air application running on the 1VM and 2VM setup in three operational regions (light, medium and heavy workloads) and collect datasets in these regions. The datasets are then used to train the ML models in these regions for the 1VM setup and 2VM setup. We then use $ML(O_0, O_t)$ to predict the response time in the target O_t, outside the current operational regions. The models use as input the container and host utilization distances to the target operational points. The results will be presented in Section VI-A.

To train $ML(O_t, O_t)$ models, we use data generated by Algorithm 1. The algorithm will assume several target points, generate new dataset and combine it with dataset in the O_0 region. In our experiment, we set *X*% of $R_{threshold}$ value to 30%, thus $R_{threshold}$ will not exceed 30% when the LAS is injecting the disturbances. We set the resource that our LAS will stress as the CPU, and ΔU in increments of 3% in DI. We selected U_{CPU} in O_t as 70% CPU utilization as the maximum the LAS will affect the Host Utilization of VM in Env, thus $DI = (0, 3, 6...70)$. The LAS datasets are used as input to build the $ML(O_t, O_t)$ model when changing the operational point from light, medium, and heavy operational regions. The results are presented in Section VI-A.

2) Experiment Two: Performance Impact Model for Co-location of a New Application: Our second experiment answers **RQ-2** by evaluating the prediction accuracy of the change models when a new co-located application is deployed on the same Env as *a*. The distance between the operational region O_0 to the operational region O_t is caused in this case by the deployment of the co-located IoT Air Quality application. Since O_t is induced by the co-located application *b*, the operational region of O_t will be more complex than experiment one where the O_0 was artificially emulated by an utilization increase. We run this experiment in 3 steps:

1) We first run the IoT application in isolation and collect its performance profile, that is, the container utilization for different workloads. In this way, we determine the distance DI in the utilization space.
2) We use IoT application DI and run the LAS algorithm to build prediction models $LM(O_t, O_t)$, $ML(O_t, O_t)$ and $QNM(O_t, O_t)$, similar to experiment 1.
3) We then deploy the IoT Application alongside Acme-Air and run both the Httperf and Jmeter workloads simultaneously and measure the response time of Acme-Air and IoT applications.
4) We compare the metrics predicted by the model prior to deployment with the measured metrics after the deployment

In our 1VM setup, we deploy the containers of the IoT Application alongside Acme-Air. Similar to experiment one, the LAS stress the CPU at increments of 3% in DI. In our 2VM setup, we deploy the NodeRed container with the Acme-Air Node.JS Web Server container on *VM 1* and the InfluxDB + Mosquitto containers with the Acme-Air MongoDB container

on *VM 2*. Based on the performance profile (cf. Step 1) of the IoT application, the InfluxDB container has the highest utilization, followed closely by the NodeRed container, while the Mosquito container has the lowest utilization. We run LAS to have different perturbations representing NodeRed on *VM 1*, and InfluxDB + Mosquitto and *VM2*. The LAS on VM 1 will start at 5% CPU Stress and increase the perturbation by 4%, and the LAS on *VM 2* will start at 6% and increase the perturbation by 5%, thus $DI_{VM1} = (0, 5, 9...70)$ and $DI_{VM2} = (0, 6, 11...70)$. These incremental increases has been determined by the performance profile of the IoT application and these values can be configurable. The LAS datasets are used as input to build $LM(O_t, O_t)$, $ML(O_t, O_t)$ and $QNM(O_t, O_t)$ models when changing the operational point from light, medium, and heavy operational regions of the IoT Application. The selected benchmark application(s) and the cloud deployment are industrial strength, and we covered a wide range of workload intensities to emulate extreme situations. The results are presented in Section VI-B. The datasets for both experiments are available on Github[1].

VI. Results and Discussion

In this section, we evaluate the accuracy of our $M(O_0, O_t)$ and $M(O_t, O_t)$ performance impact model(s) by comparing it with the actual deployment and discuss the results. We evaluated the effectiveness and accuracy of our models by using Mean Absolute Percentage Error (MAPE). As discussed in our experiment setups in Section V, we compare the $R_{measured}$ of application *a* and application *b* sharing the same cloud environment with the $R_{predict}$ outputted by our $M(O_0, O_t)$ and $M(O_t, O_t)$ models. MAPE is defined as:

- Let n denote the total number of records observed
- Let $R_{measured,i}$ denote the actual response time, observed for record *i*
- Let $R_{predict,i}$ denote the predicted response time, $R_{predict}$, made for record *i*

$$MAPE = \frac{1}{n} * \sum_{i=1}^{n} \frac{|R_{measured,i} - R_{predicted,i}|}{|R_{measured,i}|} * 100 \quad (8)$$

Each record *i* is the combination of application *a* and application *b* or LAS with their respective workload(s), workload intensities and step size within the workload(s). We evaluated the three models using historical data at normal operational point, O_0, used by the $M(O_0, O_t)$ models, and the LAS data at target operational point, O_t, used by the $M(O_t, O_t)$ models to compare the MAPE of each model.

A. RQ-1 Results: Feasibility of the Performance Impact Model

Figure 3 and Figure 4 present the MAPE violin plots for models built with O_0 and O_t data. On the violin, the horizontal bar represents the median MAPE across all operational points for different load types. The lower the median, the shorter and wider the violin, the better. The $ML(O_t, O_t)$ models

[1]https://github.com/yar-yorku/Change-Impact-Prediction-Datasets

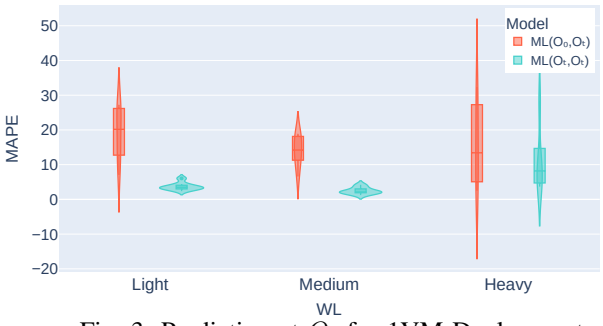

Fig. 3: Prediction at O_t for 1VM Deployment

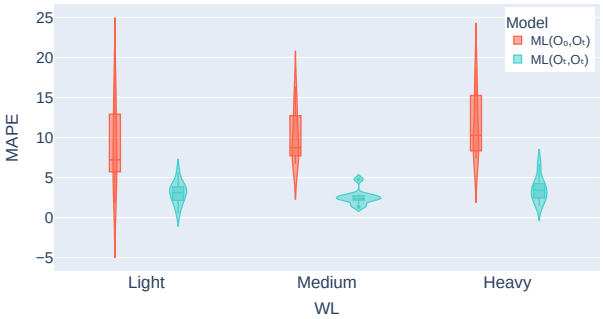

Fig. 4: Prediction at O_t for 2VM Deployment

outperforms the $ML(O_0, O_t)$ models for all the workload types and VM deployments. The MAPE of $ML(O_t, O_t)$ has significantly decreased from the $ML(O_0, O_t)$ and there is less variability within the MAPE values. This shows that the $ML(O_t, O_t)$ model can predict response times with higher accuracy and more consistently than the $ML(O_0, O_t)$ models. At heavy loads, the models have higher variation since the VMs are closer to saturation. Conforming to Equation (4), when close to saturation utilization, 1, the denominators tend to 0, explaining the variability.

Table I presents a closer look at the MAPE values as the

			$ML(O_0, O_t)$	$ML(O_t, O_t)$
WL	**Utilization at O_0**	**Utilization at O_t**	**MAPE**	**MAPE**
Light	**0 - 5 %**	10 - 20 %	12.04	**6.05**
		20 - 30 %	22.75	**2.39**
		30 - 40 %	25.77	**3.09**
		40 - 50 %	**27.22**	**3.40**
	5 - 10 %	10 - 20 %	7.04	**4.40**
		20 - 30 %	13.44	**2.94**
		30 - 40 %	17.60	**3.72**
		40 - 50 %	26.59	**3.56**
Medium	15 - 20 %	20 - 30 %	12.08	**4.18**
		30 - 40 %	14.21	**2.09**
		40 - 50 %	18.70	**1.26**
	20 - 25 %	20 - 30 %	6.59	**2.13**
		30 - 40 %	11.02	**1.85**
		40 - 50 %	16.41	**2.37**
		50 - 60 %	18.87	**3.24**
Heavy	25 - 30 %	30 - 40 %	2.49	**3.69**
		40 - 50 %	5.93	**5.07**
		50 - 60 %	13.41	**28.34**
	30 - 40 %	40 - 50 %	25.58	**10.12**
		50 - 60 %	32.36	**8.19**

TABLE I: Prediction at O_t for 1VM Deployment

distance DI moves away from O_0. $ML(O_0, O_t)$ model can accurately predict the response time when the distance of O_t is closer to O_0. However, as we move the O_t further away from O_0, the performance of the $ML(O_0, O_t)$ model decreases, while the $ML(O_t, O_t)$ model has a consistently low error until highly saturated points in the heavy workload. Experiments on 2VM deployments shows similar results.

RQ-1: Based on experimental results, models built with only historical data do not extrapolate well beyond the operational points; the accuracy error, MAPE, increases with the distance from the operational point. By incorporating the data produced through the Look-Ahead Scanner (LAS), it is feasible to build a model that can be used to predict end-user metrics at different unexplored operational points, O_t. LAS-based models, compared to models built on historical data, reduce the MAPE by as much as 24% (e.g. from 27.22% to 3.40% in Table I). Overall, Machine Learning outperforms Queuing and Regression models.

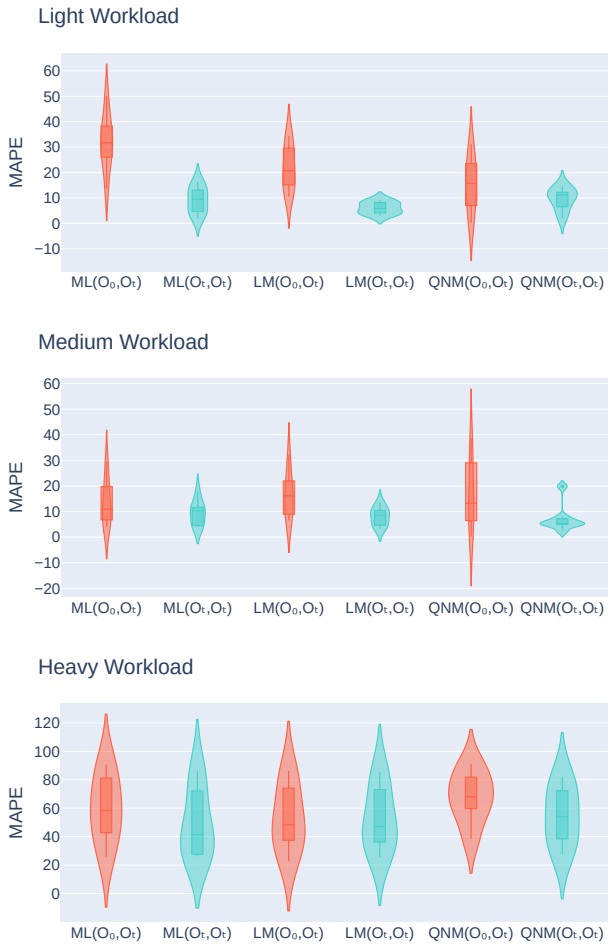

Fig. 5: Prediction of Error Improvement of Co-locating Applications on 1VM Deployment

B. RQ-2 Results: Effectiveness of LAS to Predict Performance Impact of Co-location

We now compare the prediction accuracy of the models built in O_0 with the LAS models built in O_t when we deploy

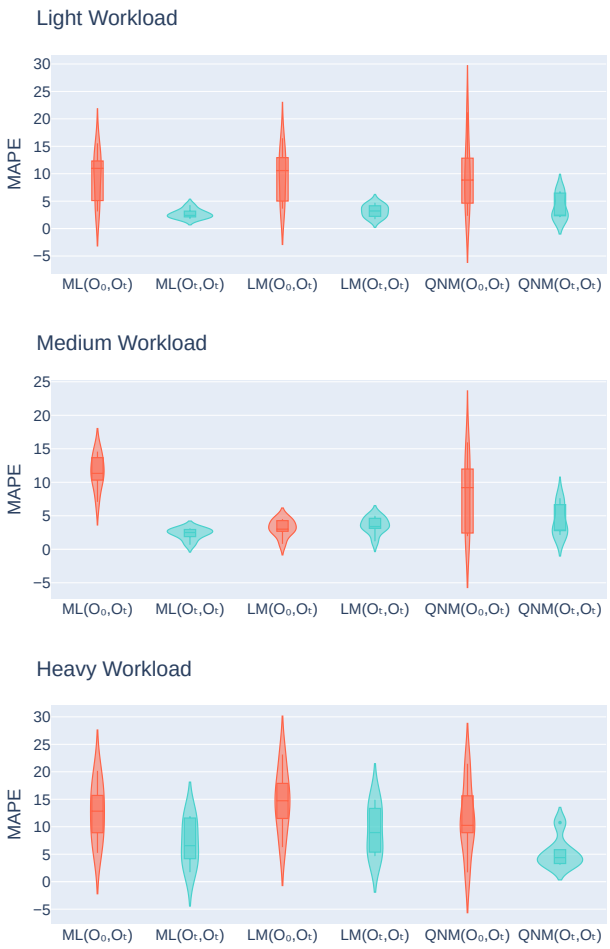

Fig. 6: Prediction of Error Improvement of Co-locating Applications on 2VM Deployment

a real IoT application that shares the same VMs as AMCE-Air. Figure 5 and Figure 6 compare the distribution of the accuracy error, MAPE, for all the models built in O_0 and O_t. For the 1VM deployment in Figure 5, all $M(O_t, O_t)$ models outperform $M(O_0, O_t)$ models and have significantly less variability at different O_t. This shows that $M(O_t, O_t)$ models can consistently predict more accurately the impact of changes when co-locating applications. For the 2VM deployment co-location in Figure 6, the $ML(O_t, O_t)$ and $QNM(O_t, O_t)$ models outperform their respective $M(O_0, O_t)$ models as well. The $LM(O_t, O_t)$ outperforms the $LM(O_0, O_t)$ in the light workload, but there is an insignificant difference in the medium workload. The QNM models have the most variability of MAPE distribution compared to the other models. The $ML(O_t, O_t)$ outperforms the $ML(O_0, O_t)$ models at the medium and light workloads, and shows the least variability of MAPE distribution. While there is more variability in the heavy workload for 2VM deployment, all $M(O_t, O_t)$ models have a smaller first and third quartiles, and median line.

Table II and Table III show MAPE at different co-location operational regions. The light workload can be seen in the

		QNM		LM		ML	
		(O_0, O_t)	(O_t, O_t) **(LAS)**	(O_0, O_t)	(O_t, O_t) **(LAS)**	(O_0, O_t)	(O_t, O_t) **(LAS)**
Utilization at O_0	**Utilization at O_t**	**MAPE**	MAPE	**MAPE**	**MAPE**	**MAPE**	**MAPE**
0 - 5 %	10 - 20 %	3.32	**20.80**	2.77	**2.31**	5.97	**13.43**
	40 - 50 %	16.32	**10.68**	17.09	**8.64**	36.34	**4.29**
	50 - 60 %	21.00	**1.82**	26.53	**9.12**	**50.12**	**7.30**
	60 - 70 %	31.00	**11.84**	34.46	**6.93**	28.12	**16.32**
5 - 10 %	20 - 30 %	8.00	**13.24**	2.07	**8.86**	2.87	**25.91**
	40 - 50 %	3.29	**5.18**	14.19	**4.03**	24.07	**11.67**
	50 - 60 %	15.16	**7.88**	24.37	**4.78**	**40.23**	**2.09**
	60 - 70 %	26.05	**11.70**	32.62	**4.22**	34.95	**11.78**
15 - 20 %	20 - 30 %	12.54	**18.21**	2.79	**4.46**	4.01	**5.53**
	40 - 50 %	3.29	**5.32**	10.80	**9.54**	6.72	**12.09**
	50 - 60 %	9.60	**5.03**	13.33	**13.50**	8.18	**11.16**
	60 - 70 %	12.35	**2.29**	19.31	**10.53**	13.70	**4.55**
20 - 25 %	20 - 30 %	6.18	**18.88**	1.56	**2.09**	1.88	**4.52**
	40 - 50 %	26.18	**6.50**	18.91	**4.29**	17.25	**4.46**
	50 - 60 %	31.80	**7.91**	24.60	**5.04**	22.20	**9.49**
	60 - 70 %	38.61	**19.83**	32.23	**10.26**	29.35	**17.90**
20 - 30 %	40 - 50 %	38.60	**27.40**	22.70	**25.36**	25.84	**26.07**
	50 - 60 %	62.89	**45.54**	37.46	**36.17**	42.82	**27.51**
	60 - 70 %	73.22	**62.61**	52.52	**49.82**	61.45	**42.88**
30 - 40 %	40 - 50 %	59.69	**38.54**	44.14	**44.22**	55.63	**40.14**
	50 - 60 %	81.83	**72.30**	74.10	**73.10**	81.17	**72.11**
	60 - 70 %	91.06	**82.05**	86.18	**85.33**	90.62	**86.10**

TABLE II: Prediction of Error Improvement of Co-locating Applications on 1VM Deployment

		QNM		LM		ML	
		(O_0, O_t)	(O_t, O_t) **(LAS)**	(O_0, O_t)	(O_t, O_t) **(LAS)**	(O_0, O_t)	(O_t, O_t) **(LAS)**
Utilization at O_0	**Utilization at O_t**	**MAPE**	MAPE	**MAPE**	**MAPE**	**MAPE**	**MAPE**
0 - 5 %	10 - 20 %	11.04	**2.13**	7.09	**5.69**	7.04	**2.87**
	40 - 50 %	6.91	**6.82**	10.61	**1.96**	11.33	**1.76**
	50 - 60 %	14.67	**2.33**	13.68	**3.41**	13.23	**3.15**
	60 - 70 %	21.19	**2.35**	16.46	**3.00**	15.54	**3.21**
5 - 10 %	20 - 30 %	2.57	**2.55**	2.25	**3.78**	2.45	**4.82**
	40 - 50 %	2.55	**2.45**	3.65	**4.11**	3.14	**4.28**
	50 - 60 %	6.73	**2.09**	6.39	**4.70**	6.59	**1.92**
	60 - 70 %	10.78	**6.35**	12.18	**1.73**	11.42	**2.35**
15 - 20 %	20 - 30 %	7.60	**7.70**	1.83	**1.81**	3.71	**3.79**
	40 - 50 %	15.95	**6.77**	4.32	**4.74**	11.53	**2.63**
	50 - 60 %	11.95	**2.81**	3.05	**3.05**	11.12	**3.12**
	60 - 70 %	11.95	**7.62**	4.51	**4.98**	14.27	**3.08**
20 - 25 %	20 - 30 %	2.85	**2.80**	2.46	**2.47**	4.35	**2.93**
	40 - 50 %	6.45	**6.55**	2.95	**3.20**	10.93	**2.38**
	50 - 60 %	1.97	**2.14**	2.43	**3.29**	13.08	**1.53**
	60 - 70 %	2.00	**2.90**	2.93	**3.50**	14.57	**2.82**
20 - 30 %	40 - 50 %	1.71	**3.10**	6.30	**4.66**	5.25	**1.74**
	50 - 60 %	10.26	**5.83**	11.51	**6.29**	8.91	**4.19**
	60 - 70 %	10.23	**3.30**	13.29	**5.38**	10.49	**4.88**
30 - 40 %	40 - 50 %	8.92	**4.88**	16.17	**13.34**	15.18	**11.58**
	50 - 60 %	21.45	**10.76**	17.89	**11.58**	15.70	**8.23**
	60 - 70 %	15.63	**3.89**	23.13	**14.90**	20.16	**11.95**

TABLE III: Prediction of Error Improvement of Co-locating Applications on 2VM Deployment

rows where the *Utilization of Acme-Air at* O_0 column is at 0 - 5% and 5 - 10%, the medium workload is at 15 - 20% and 20 - 25%, and the heavy workload is at 20 - 30% and 30 - 40%. When the cloud environment's load moves further towards the unexplored operational points at 40 - 50% and beyond, the models trained at O_t outperform the models trained at O_0 for most of the O_t levels. For the 1VM deployment, the $ML(O_t, O_t)$ outperforms the $ML(O_0, O_t)$ model from 11.46% up to 42.82%. At heavy load, close to saturation points, similar to Experiments 1, all models have high variability in predicting the effects of co-location. Overall, the LAS models are more robust, especially when covering the unexplored operational points at higher CPU ranges.

RQ-2: Experimental results show that by incorporating the data produced through the Look-Ahead Scanner, we can build a model that is consistently more accurate at predicting the effect of co-locating applications. This is especially noticeable in higher utilization values of the shared infrastructure when the target operational point is further from the current point. The LAS-based models compared to models built on historical data reduce MAPE by as much as 42.82% (e.g. from 50.12% to 7.30% in Table II). Overall, Machine Learning outperforms Queuing and Regression models. Experiments also show that pushing the applications close to saturation points (heavy load) yields uncertain behavior.

C. Threats to Validity

We note that an internal threat to validity is the balance of data points between the historical data and the Look-Ahead Scanner. Increasing or decreasing the amount of data points of the Look-Ahead Scanner in the training/testing data of AutoML may affect the accuracy of the model, such that the model may begin over or under estimating. An external threat to validity is that our experiments were conducted on a single availability zone in the AWS cloud. We do not consider deployment across multiple availability zones or other cloud platforms.

D. Computational Complexity and Overhead

Model	Number of Samples	Training Time
LM	K*2*v	0.00046 ms
QNM	K*5*v	1.266 s
ML	K*DI*v / ΔU	190 s

TABLE IV: LAS Complexity

Complexity. Table IV shows the sampling and model building complexity. For LM, we need to collect two points for each derivative, for all K VMs, therefore the number of samples is K*2 samples. For LQM, when tuned with Kalman filter, practically, it converges in less than 5 samples [25] for each VM, therefore the number of samples is 5*K. For ML, the number of samples calculated based on Algorithm 1 is $K * DI/\Delta U$; where ΔU is the chosen increment in Alg. 1. ΔU is 3% in our experiments. To account for the variability of the cloud, the sampling can be repeated v times. In our experiments, v was 4. The 3rd column in Table IV shows the average training time for our models.

Overhead. Besides the controlled load introduced as per Alg. 1, there is no additional overhead. The Autonomic Manager (cf. Fig 2) runs externally and we use the instrumentation available on containers.

VII. Conclusion

In this work, we propose a method and a controllable Look-Ahead Scanner (LAS) that can induce stress on an in-production cloud-native application while keeping end-user metrics within the SLA. Through this, we are able to collect the performance data of the cloud-native application at unexplored operational points. Using the LAS dataset, the performance models outperform the models built with historical data. New models can help better runtime adaptation. Further work includes optimizing the data collection process and considering more features (e.g. i/o, memory utilization).

REFERENCES

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

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

[3] [Online]. Available: https://azure.microsoft.com/en-ca/overview/what-is-a-container/

[4] [Online]. Available: https://www.ibm.com/cloud/learn/containerization

[5] A. Baluta, J. Mukherjee, and M. Litoiu, "Machine learning based interference modelling in cloud-native applications," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 125–132. [Online]. Available: https://doi.org/10.1145/3489525.3511677

[6] T. B. Sheridan, "Adaptive automation, level of automation, allocation authority, supervisory control, and adaptive control: Distinctions and modes of adaptation," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 41, no. 4, pp. 662–667, July 2011.

[7] P. Patros, S. A. MacKay, K. B. Kent, and M. Dawson, "Investigating resource interference and scaling on multitenant paas clouds," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, 2016, pp. 166–177.

[8] 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)*, 2017, pp. 1–6.

[9] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–32. [Online]. Available: https://doi.org/10.1145/3297663.3310309

[10] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 2114–2129, 2019.

[11] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2016, pp. 261–268.

[12] M. Gokan Khan, J. Taheri, A. Al-dulaimy, and A. Kassler, "Perfsim: A performance simulator for cloud native microservice chains," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2021.

[13] A. O. Ayodele, J. Rao, and T. E. Boult, "Performance measurement and interference profiling in multi-tenant clouds," in *2015 IEEE 8th International Conference on Cloud Computing*, 2015, pp. 941–949.

[14] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov, "Stay-away, protecting sensitive applications from performance interference," in *Proceedings of the 15th International Middleware Conference*, ser. Middleware '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 301–312. [Online]. Available: https://doi.org/10.1145/2663165.2663327

[15] 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)*, June 2019, pp. 200–210.

[16] 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)*, Dec 2017, pp. 522–530.

[17] H. Wang, Y. Ma, X. Zheng, X. Chen, and L. Guo, "Self-adaptive resource management framework for software services in cloud," in *2019 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, Dec 2019, pp. 1528–1529.

[18] V. Podolskiy, M. Mayo, A. Koay, M. Gerndt, and P. Patros, "Maintaining slos of cloud-native applications via self-adaptive resource sharing," in *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, June 2019, pp. 72–81.

[19] T. Chen and R. Bahsoon, "Self-adaptive and online qos modeling for cloud-based software services," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 453–475, May 2017.

[20] A. O. Strube, D. Rexachs, and E. Luque, "Software probes: A method for quickly characterizing applications' performance on heterogeneous environments," in *2009 International Conference on Parallel Processing Workshops*, Sep. 2009, pp. 262–269.

[21] H. Moradi, W. Wang, and D. Zhu, "Adaptive performance modeling and prediction of applications in multi-tenant clouds," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Aug 2019, pp. 638–645.

[22] J. Viktorin, P. Korcek, T. Fukac, and J. Korenek, "Network monitoring probe based on xilinx zynq," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 237–238. [Online]. Available: https://doi.org/10.1145/2658260.2661769

[23] S. Chen, K. R. Joshi, M. A. Hiltunen, W. H. Sanders, and R. D. Schlichting, "Link gradients: Predicting the impact of network latency on multitier applications," in *IEEE INFOCOM 2009*, 2009, pp. 2258–2266.

[24] 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.

[25] T. Zheng, C. M. Woodside, and M. Litoiu, "Performance model estimation and tracking using optimal filters," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 391–406, 2008.

[26] E. Brookner, *Tracking and Kalman filtering made easy*. Wiley New York, 1998.

[27] 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

[28] L. Perko, *Differential equations and dynamical systems*. Springer Science & Business Media, 2013, vol. 7.

[29] S. Edelkamp and S. Schrödl, *Heuristic Search - Theory and Applications*. Academic Press, 2012. [Online]. Available: http://www.elsevierdirect.com/product.jsp?isbn=9780123725127

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

[31] "Mosquitto," [Online]. [Online]. Available: https://mosquitto.org/

[32] "Nodered," [Online]. [Online]. Available: https://nodered.org/

[33] "Influxdb," [Online]. [Online]. Available: https://www.influxdata.com/

[34] P. D'silva, "Deploying acme air microservices application on red hat openshift container platform," Sep 2020. [Online]. Available: https://developer.ibm.com/tutorials/deploy-acme-air-on-openshift-on-power/

[35] J. Bagur and T. Chung, "Data logging with mqtt, node-red, influxdb and grafana: Arduino documentation," Mar 2023. [Online]. Available: https://docs.arduino.cc/tutorials/portenta-x8/datalogging-iot

[36] "Node.js," [Online]. [Online]. Available: https://nodejs.org/en/

[37] "Stress-ng," [Online]. [Online]. Available: https://wiki.ubuntu.com/Kernel/Reference/stress-ng

[38] "Prometheus," [Online]. [Online]. Available: https://prometheus.io/

[39] 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.

[40] "Opera," [Online]. [Online]. Available: http://www.ceraslabs.com/technologies/opera

[41] "Jmeter," [Online]. [Online]. Available: https://jmeter.apache.org/