Self-Adaptive Strategies for Cloud Applications

Yar Rouf

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

YORK UNIVERSITY

TORONTO, ONTARIO

April 2025

Abstract

Modern Software has become increasingly complex with the use of microservice architectures and cloud computing becoming essential practices in the continuous deployment cycle. Self-Adaptive Systems can help with automating service deployment, maintenance, and optimization of such applications. However, these cloud applications are becoming large-scale and complex in nature, often deployed on multi-node clusters on multiple cloud platforms. To improve the self-adaptive process for large-scale cloud applications, we introduce four main contributions in this thesis: (1) We present a Self-Adaptive MAPE-K (Monitor, Analysis, Planning, Execution) framework that is built with existing Components-off-the-Shelf (COTS) that interacts with each other to perform self-adaptive actions on multi-cloud environments. (2) We propose a novel method to identify a performance model that predicts metrics at any unexplored operational point of a cloud environment. (3) We introduce an interference detection method for industrial strength at-scale deployments and evaluate the method using several model types. (4) We propose a proactive dynamic model identification technique to predict the impact of cloud consolidation and co-location for large at-scale deployments. We evaluated each contribution with an extensive set of experiments ranging from feasibility to prediction accuracy.

Acknowledgment

I express deep gratitude to my supervisor, Professor Marin Litoiu, for his unwavering support, guidance, and kindness during this research journey. His extensive knowledge, insights, and new perspectives were invaluable to my progress as a Ph.D. researcher. My opportunity with Professor Marin Litoiu was filled with many new experiences and lessons that have become core memories. This thesis would not have been possible without Professor Marin Litoiu. I am beyond words grateful.

I am grateful for the assistance of Joydeep Mukherjee and for the valuable research and technical experience that I gained working under him. I give my thanks to the supervisory committee members Professor Manos Papagelis and Professor Zhen Ming for providing valuable feedback and viewpoints on my research. I express my gratitude to all of the members with whom I worked in the Adaptive Systems Research Lab at York University. Alex Baluta, Kim Long Ngo, Hamza Hussain, Farhoud Jafari, Raphael Rouf, Sara Fehresti, Jaime Dantas and Sahasra Kokkula. Our research collaboration and knowledge sharing was of great benefit.

Finally, I want to dedicate this work to my father Niaz, my mother Shahina, my brother Raphael, and many of my friends who supported me throughout my journey as a researcher. All of your presences have brought immeasurable positive energy into my life. Thank you for all the unconditional love and constant support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Organizations have been leveraging various cloud platforms and APIs to build and deploy their applications to better support the DevOps cycle. These applications are commonly designed to have a microservice architecture, where the application is broken down to several independent individual services characterized by its own specific functionality. Containerization has become highly prevalent in application development and deployment and enables the developer to easily encapsulate their application services and its libraries and dependencies in a lightweight executable container. To manage and automate containerized applications in a dynamic environment, container orchestration tools and platforms have been used to create, modify, and remove cloud resources in cloud environments. Container orchestration can involve both large monolith applications and hundreds of services from multiple modern microservice-based applications designed specifically for the cloud, known as cloud-native applications, to run alongside each other in an organizations cloud ecosystem. Container orchestration tools make it easier to support distributed applications on such clusters. Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and Google Ku-

bernetes Engine (GKE) are some of the most popular deployment services to manage containerized applications and clusters on the cloud. These container orchestration tools have found themselves to play an essential role in the DevOps lifecycle.

DevOps is a popular paradigm that integrates development and operations teams to enable fast and efficient continuous delivery of software. Applications and services deployed on cloud platforms can benefit from implementing the DevOps practice. This involves using different tools for enabling end-to-end automation to ensure continuous deployment and maintain good Quality-of-Service. Self-Adaptive systems can support the DevOps process by automating service deployment and maintenance without manual intervention by employing a MAPE-K (Monitoring, Analysis, Planning, Execution-Knowledge) framework. While industrial MAPE-K tools are robust and built for production environments, they have limited analysis based on basic threshold-based planning and lack the flexibility to create more sophisticated automation plans for large multi-cloud applications [1]. Academic models are more flexible and can be used to perform sophisticated self-adaption during analysis, such as machine learning models, control loops, etc. However, academic models can lack the robustness and scalability that are needed to be used in production environments [2].

Performance models can be classified into different types based on how they are trained. 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 cloud applications and their environments in the DevOps lifecycle are dynamic in nature. Static models do not perform well to frequent changes and are unable to capture the future dynamics of a cloud-native application [3]. Dynamic models can be constructed at run-time by taking into account the current state of the application, and are often

preferable over static models when used with self-adaptive systems. In self-adaptive systems, model-based control assumes decisions are taken based on a dynamic 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. Models are built around the known operational points (e.g. the performance utilization, workload patterns, and configurations) of an application and its environment at its normal behavior. However, there is a lack of studies when these models extrapolate to regions far from the current operational point. We need to be able to predict the performance impact on the cloud environment when we plan to co-locate new applications with the production application, consolidate resources, redistribute containers or when the autonomic manager anticipates an increase in load which can impact the Service Level Objectives (SLO).

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 [4, 5]. 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. When multiple microservices are deployed on the same VM, they can often compete with each other for shared host-level resources. Such shared resource contention can alter the application behavior and make it deviate from the development

time specifications and performance. We refer to application performance degradation due to shared resource contention as *performance interference*. Detecting the cause of performance degradation at runtime is crucial to decide the correct remediation action such as, but not limited to, scaling or migrating. Automatically detecting runtime interference in cloud-native applications is a challenging task. Application workload variability and unpredictability can produce the same effects as interference, and hence it is difficult to distinguish between them. Furthermore, interference detection techniques that model the baseline behavior may not generalize well to scenarios where the interfering application changes at runtime.

Cloud operations are an important part of the cloud ecosystem for cloud-native applications and extend the DevOps principles of continuous deployment and automation. Applying cloud operations for cloud-native applications is an important practice for the DevOps team to continuously meet evolving requirements by deploying, managing, and optimizing of cloud services that run on a public, private, or multi-cloud environment. This practice is called CloudOps [6] [7] and supports automation, scalability, and continuous deployment of services. Automation in CloudOps provides the organization with the ability to construct an operational framework for a new addition, update, or removal of cloud resources [6]. Automation in cloud operations can push the environment into operational regions the system has not seen, thus the performance model may not accurately extrapolate into unseen regions. The unexplored operational regions can be the result of an expansion in the environment with the deployment of a co-located application or a reduction in environment resources with cloud consolidation. With more modern applications deployed on large-scale Kubernetes clusters, scaling up and down of applications is quite common.

In this thesis, we investigate self-adaptive techniques and strategies for cloud ap-

plications and environments. We explore the feasibility of developing an autonomic MAPE-K framework for multi-cloud platforms, building a dynamic modeling technique to look ahead from the current operational point of a cloud-native application, a modeling technique to detect interference for at-scale cloud applications and performance modeling technique for cloud operations of multi-node at-scale cloud deployments. We address these following research objectives:

1. Building Self-Adaptive Systems for Multi-Cloud Environments
2. Performance Modeling and Identification for Cloud Applications
3. Interference identification for Cloud Applications
4. Proactive Identification of Performance Models for Cloud Operations, Cloud Consolidation and Co-location

To address the first objective, we present a MAPE-K framework, built with existing Components-off-the-Shelf (COTS) that interacts with each other to perform self-adaptive actions on multi-cloud environments. By integrating existing COTS, we are able to deploy a MAPE-K framework efficiently to support DevOps for applications running on a multi-cloud environment. To address the second objective, 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. To address the third objective, we propose a non-intrusive detection technique that differentiates between degradation caused by load and by interference. To address the fourth objective, we propose a proactive performance sampling and modeling technique for at-scale cloud native applications on multi-node clusters to be used by self-adaptive systems for scaling-up, co-location and consolidation operations.

The original contributions of this thesis are as follows:

1. We introduce an industrial MAPE-K framework that can support the automation of services through the integration of Components-off-the-Shelf (COTS) to support self-* properties on Multi and Hybrid-cloud environments. This contribution was published in ICPE '21: ACM/SPEC International Conference on Performance Engineering titled "A Framework for Developing DevOps Operation Automation in Clouds using Components-off-the-Shelf" and is presented in Chapter 2.

2. We demonstrate that models built on historical data do not predict accurately metrics outside of operational regions, and introduce a method that collects additional data at the target operational points to build a robust performance model to predict performance impacts. The robust model at target operational points outperforms those built with historical data by reducing the mean absolute percentage error (MAPE) by up to 42%. This contribution was published in 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) titled "Towards a Robust On-line Performance Model Identification for Change Impact Prediction" and is presented in Chapter 3.

3. We introduce an interference detection method that generalizes across interference scenarios and outperforms the state-of-the-art for industrial strength at-scale deployments and evaluate the method using several model types. This contribution was published in ICPE '24: 15th ACM/SPEC International Conference on Performance Engineering titled "Disambiguating Performance Anomalies from Workload Changes in Cloud-Native Applications" and is presented in Chapter 4.

4. We introduce a proactive performance sampling method to build a performance model to explore the operational regions of consolidation and co-location for an at-scale deployment. This contribution was submitted to IEEE on Software Engineering titled "Proactive Identification of Performance Models for Cloud Consolidation and Co-location" and is presented in Chapter 5.

We designed a MAPE-K architecture for modern cloud applications to support self-adaptive systems. We then focus on expanding the analysis phase of the MAPE-K framework and learn how to build a performance model that can predict change impact and classify interference. Finally, we combine all our contributions and apply our models on supporting cloud operations such as co-location and consolidation on large at-scale cloud deployments.

The organization of this thesis is as follows: In Chapter 2, we investigate the first objective for a MAPE-K framework for multi-cloud environment. In Chapter 3, we investigate the second objective for building dynamic performance models using a Look-Ahead Scanner mechanism. In Chapter 4, we investigate the third objective for a interference identification technique for at-scale cloud deployments. In Chapter 5, we investigate the fourth objective for a proactive performance modeling technique for cloud operations of multi-node deployments. In each chapter, we discuss the research questions, background and related works, methodology and architecture, implementation, experimental set-ups and experimental results.

Chapter 2

Self-Adaptive Framework for Multi-Cloud Environments

DevOps [8] is a popular practice that enables the development and operation teams to continuously coordinate and collaborate with each other. By adopting DevOps practices and tools, it is possible to build, deploy and manage services and applications seamlessly while providing good Quality-of-Service (QoS) to the end users. With more and more commercial enterprise software increasingly being deployed on the cloud, it is important to adapt DevOps principles for assuring good QoS for cloud-based applications.

Multi-cloud platforms aggregates multiple cloud providers (for e.g., Amazon EC2, Microsoft Azure, Google Cloud Platforms, Private Clouds) to achieve faster, easier and better application building and deployment [9]. However, ensuring good QoS in multi-cloud platforms requires a substantial level of automation and adaptation in such heterogeneous hybrid platforms. This can be realized by employing a MAPE-K (Mon-

itoring, Analysis, Planning, Execution- Knowledge) framework [10] [11] [12], which deploys a feedback loop that cycles through Monitoring, Analysis, Planning, and Execution phases of a system, and shares data through a common Knowledge base. To support autonomic computing, there are existing enterprise products that follow the MAPE-K framework [13]. These autonomic frameworks are robust and reliable as they are designed to handle production applications. However, industrial MAPE-K frameworks support limited analysis based on basic threshold-based monitoring and planning [1]. This lack of flexibility limits the DevOps teams to create more sophisticated automation plans for complex multi-cloud applications. On the other hand, autonomous frameworks proposed in academia are more flexible and can be applied to a wider variety of analysis including machine learning models, analytical models, control loops, etc. However, they are less robust and non-scalable as many of them are not designed for production environments [2].

2.1 Research Questions

In this chapter, we explore the feasibility of developing an autonomic MAPE-K framework for multi-cloud platforms by integrating existing services and Components-Off-the-Shelf (COTS) software. When we incorporate MAPE-K into applications to support DevOps operations, we need to satisfy several requirements. Multi-cloud application deployments must be dynamic in nature and require automatic scaling capabilities in which service components can be added or removed in response to incoming traffic without manual intervention. They also require adaptive management that enables deployment and movement of service components across different cloud providers. Providing good QoS in a multi-cloud platform needs large-scale automated resource

monitoring across several cloud platforms, which is challenging to do since resource monitoring metrics are often tightly coupled with individual cloud providers. Additionally, the autonomic framework needs to be robust and should be able to handle failures as applications in production need to have high availability and stability. Essentially, we need to satisfy the requirements of availability, scalability, autonomic management, multi-cloud integration and ease of deployment. Keeping these requirements in mind, we answer the following research questions in this work:

- **RQ-1:** Is it feasible to build a MAPE-K framework for multi-cloud applications by integrating existing COTS and services?
- **RQ-2:** What are the challenges of building such a framework?
- **RQ-3:** What is the performance and efficiency of this framework?

RQ-1 explores COTS and available industrial services to determine if they meet our requirements. The interactions between each of technologies has to follow the MAPE-K model and needs to be connected together seamlessly. **RQ-2** identifies the various tasks needed for integration and the challenges we experienced when merging each of the services. **RQ-3** evaluates the framework on different self-adaptive use cases on a IBM benchmark application. We applied our framework to scale services depending on the workload, repair the application by redeploying a failed service and scale services in response to varying workloads on a multi-cloud setup.

We support the DevOps lifecycle by incorporating the MAPE-K framework with Continuous Deployment, which can be seen in Figure 1. Continuous deployment refers to automatically releasing the developer's changes of the application into production [14]. During application deployment, the DevOps team needs to monitor and manage their infrastructure to assure the services are properly provisioned and are

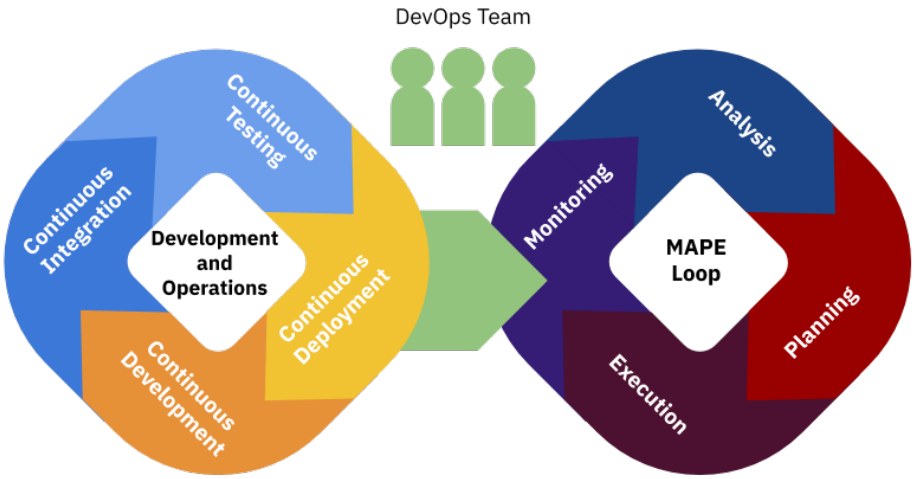

Figure 1: DevOps to MAPE Loop

able to perform well [15]. Our framework incorporates the COTS and available services to perform each of the phases in the MAPE-K loop. The DevOps teams can specify and develop the requirements that need to be satisfied for their managed application. The deployment and movement of the services on the cloud(s) can then be autonomously managed by our framework. This chapter is the work published in ICPE '21: ACM/SPEC International Conference on Performance Engineering titled "A Framework for Developing DevOps Operation Automation in Clouds using Components-off-the-Shelf".

2.2 Related Work and Background

A self-adaptive system can handle changes and uncertainties in the environment in order to achieve certain goals with minimum or no human intervention. Self-adaption provides the software system the ability to adapt to internal dynamics and the dynamics in the environment [16]. The self-adaptive system consists of two parts: a **managed system** and a **managing system**. We can consider our cloud environment and the de-

ployed services the managed system, and the software that monitors and executes the adaptation of the managed system as the managing system [16]. Weyns [16] defines two basic principles of a self-adaptive system, the **external principle** and **internal principle**. The external principle is where a system can handle changes and uncertainties autonomously of the environment, system and its goals. The internal principle is where the self-adaptive system comprises of two parts, the first is where the system interacts with the environment, and the second part is the feedback that monitors the environment and is responsible for the adaptation.

MAPE-K is an influential feedback control loop for self-adaptive systems [12] [17] [10] [11] that cycles through Monitoring, Analyze, Planning, and Execution phases of a system, and shares data through a common knowledge base (K). The Monitor component (M) collects data and metrics from the underlying managed system and environment through the sensors. The Analyze component (A) performs the analysis on the data, and checks if there is a violation that requires adaptation. The Plan component (P) is then triggered, and creates the adaptation actions for the managed system to achieve the adaptation goals. The Execution component (E) executes the plan through an actuator/effector of the managed system. The Knowledge base contains all the data and metrics of the managed system, the environment, adaptation goals and other relevant states that are used by the MAPE components. Each phase of the MAPE-K can be achieved through multiple services that coordinate with each other (such as cloud monitors or container orchestration platforms) [17].

Cloud-native architecture and technologies represent an approach to designing, constructing, and managing workloads specifically tailored for the cloud [18]. The Cloud Native Computing Foundation [19] official definition of cloud-native technologies is that they empower organizations to create and operate scalable applications

in contemporary, dynamic environments including public, private, or hybrid clouds. Key components of this approach include containers, service meshes, microservices, immutable infrastructure, and declarative APIs. By adopting these techniques, organizations can build loosely coupled systems that are both resilient and manageable. Additionally, robust automation enables engineers to implement impactful changes frequently and predictably, minimizing operational overhead.

2.2.1 Self-Adaptive Systems in Cloud

Autonomous Systems on Clouds has been a popular research area by practitioners and researchers, and integrated for DevOps environments is a developing area. Cukier [20] present their experience in using cloud services to scale a web application by following DevOps and development patterns. They provide different solutions when scaling a complex web application that has a mix of cloud services in PaaS, SaaS and IaaS layer. Guerriero et. al. [21] designed a DevOps tool to support QoS assessment, optimization and QoS-aware runtime capacity allocation of cloud applications. To support DevOps operations with an Autonomous Management System (AMS), Barna et. al [22] proposed a method to develop an (AMS) for multi-tier, multi-layer data-intensive containerized applications. They employ a self-tuning performance model that outputs the performance metrics based on the application's topology, and plan for potential adaptive actions on the system if there are any problematic situations based on the performance model. In our research, we support the DevOps Operations by building an autonomic framework, which explores COTS and existing tools to provide the DevOps teams with a high-level language equivalent to design adaptive actions for their system.

2.2.2 Self-Adaptive Systems in Multi and Hybrid Cloud

In Multi and Hybrid Cloud Autonomous Systems, different methods have been proposed to help scale deployments across multiple clouds. Namjoshi et. al [23] present a framework for elastic scaling of cloud resources that is portable across a wide range of private and public cloud providers and can be easily integrated with other frameworks. Their framework uses a Monitoring Engine, Decision Engine, Provisioning Manager, and a Database to auto-scale. Their decision engine is both rule-based and schedule-based, allowing them to scale at a particular time or when the monitored resource satisfy rules or policies. Miglierina et. al [24] propose a self-adaptive technique for both in-cloud scaling policies and traffic routing among the different cloud providers in a multi-cloud setup with a control-theoritical approach. Loreti and Ciampolini [25] propose an autonomic method to scale clusters of virtual machines over a hybrid cloud for MapReduce jobs. Kang et al. [26] propose a auto-scaling method to provide efficient resource utilization for Hybrid clouds. Their auto-scaler method is designed to handle variable workloads of modern applications, which must also meet SLAs such as deadlines, cost-oriented and performance-oriented policies. Ahn et. al design an autoscaler that dynamically allocates resources depending on the two patterns of jobs, Bag-of-Tasks and workflow, in Hybrid Clouds [27]. Ahn and Kim also extend this work by focusing on dynamically allocating VMs in Hybrid Clouds in order to maximize resource utilization within a deadline and dealing with task dependency in workflow application [28]. Yunchun Li and Yumeng Xia [29] design a platform which can auto-scale web applications in hybrid cloud based on docker. They built a hybrid scheduling controller, and use a combination of prediction and reaction algorithms to scale docker containers of a web application on a hybrid cloud. In this work, our architecture follows the MAPE-K framework, utilizing COTS and open-source services performing

the Monitor, Analysis, Planning and Execution stages to autonomously manage Multi and Hybrid cloud applications.

2.2.3 Finite-State Machines

Finite-state machine (FSM) is a design pattern that handles the change from one state to another in response to some inputs, based on a finite number of changes. Drumea and Popescu [30] discuss implementing finite state machines in the software application domain, with a focus on web technologies. Rules and the ruleflow can be modeled as finite-state machine with rules being used to transition from one state to the next [31]. Moran et. al [32] use a rule engine to apply adaptive changes in Federated clouds, with the adoption of Rule Interchange Format (RIF) to allow the portability of reusing the rules on a different rule engine. In our previous research, we used business rules to define security rules at an operational level and created a flow of rules to assist developers and security analysts in finding security vulnerabilities [33]. We extended this work by incorporating COTS and existing infrastructure management tools to ensure automated DevOps management for enterprise applications.

2.3 Requirements for DevOps Autonomic Framework

In this section, we have established a set of requirements for a MAPE-K framework that automates cloud deployment and supports the DevOps process. These requirements were selected based on current challenges in Cloud Automation for DevOps operations and multi-cloud deployment for industry applications.

REQ-1: Availability Availability is a measure of how the system is able to run without failure and how fast it recovers from failure. Availability of a service is of

primary importance in order to provide good QoS to its end users. Services deployed on a multi-cloud environment can often suffer from faults and must have mechanisms in place to recover quickly. This is challenging since a multi-cloud environment spans multiple cloud providers, with each provider offering different mechanisms of control and management over its own virtual infrastructure, which significantly increases the room for faults throughout multiple services and cloud platforms. Thus, we need highly reliable components that can be used by multi-cloud services in a production environment. This is most likely to be achieved by using established software components that have been already tested in production.

REQ-2: Multi-cloud Integration In a multi-cloud environment, service components are distributed across multiple cloud providers. In such environments, features from different public cloud provider platforms are required to be integrated with the service's internal private cloud. Since each cloud provider offers different mechanisms to control its infrastructure, integrating features and services as well as managing and automating service deployments becomes complicated in a multi-cloud environment. This involves runtime monitoring of data from multiple heterogeneous sources, deploying services on different cloud provider platforms and networking them with the proper security protocols in place.

REQ-3: Autonomous Configuration for Cloud Applications Services deployed on multi-cloud environments require an autonomic framework that is able to (a) deploy an application on multi-clouds and (b) self-manage the deployed application at runtime. To this end, the framework needs to handle a wide variety of input metrics from different service components deployed on multiple cloud platforms for sophisticated decision-making. Based on the decision, the framework should be able to perform a wide variety of different adaptive actions that can make changes to the service at run-

time for enabling better service management and optimizing its QoS. Additionally, we also need to keep track of changes to our multi-cloud resources so that they can be used for implementing autonomic decision-making for managing our services in the future.

REQ-4: Scalability The autonomous framework needs to handle monitoring millions of metrics and deploying thousands of servers and applications across multiple data centers or cloud providers. When new services are deployed or new cloud platforms are integrated, the MAPE-K components needs to scale accordingly and seamlessly. We need to be able to scale up services at runtime in response to increased workload. Similarly, we should also be able to scale down services when needed so as not to incur additional cloud resource costs. This scaling has to be done automatically and efficiently so as to not react to transient changes in the state of the service.

REQ-5: Ease of Deployment We need to provide the DevOps teams an autonomous framework that can be integrated with the service without incurring significant levels of development, maintenance or learning effort. For this purpose, each of the COTS tools selected for the MAPE-K framework must be intuitive to learn, and easy to configure and implement. Selecting COTS tools with extensive documentation and online support will help significantly in this regard.

2.4 DevOps Autonomic Framework Architecture

The architecture of our proposed framework as shown in Figure 2 has three main components: (1) Cloud Monitor, (2) State Rule Engine, and (3) Workflow Engine. These components interact with each other in a continuous MAPE-K loop to conduct DevOps automation and integration for cloud-based services. For the monitoring stage of MAPE-K, we deploy a Cloud Monitor that monitors and records performance metrics

of the managed services running on different cloud platforms in a multi-cloud environment. The State Rule Engine is responsible for the analysis and planning stages of the MAPE-K loop where it queries events, i.e. a set of metrics from the Cloud Monitor, to automate operational decisions for changing the state of the deployed service, if needed. Finally, the execution stage of the MAPE-K loop is automated by implementing these operational changes by the Workflow Engine. We describe the Cloud Monitor, State Rule Engine and Workflow Engine components of our framework in details in Sections 2.4.1, 2.4.2 and 2.4.3 respectively. We also describe how our architecture adequately satisfies the 5 requirements (REQ-1 to REQ-5) as discussed before in Section 2.3.

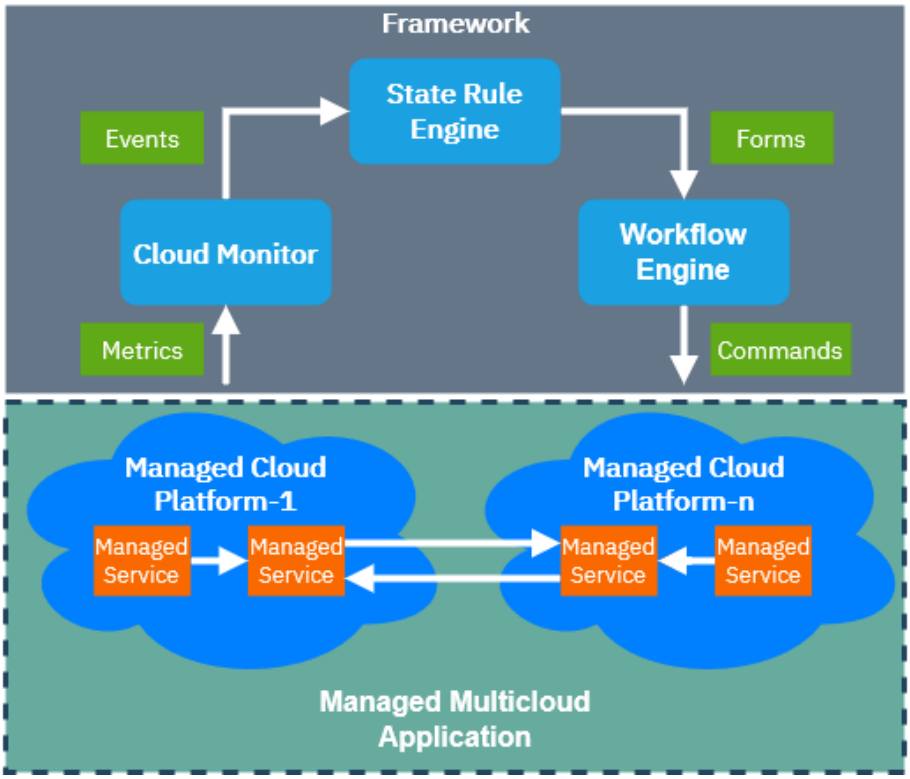

Figure 2: Conceptual Framework

2.4.1 Cloud Monitor

The Cloud Monitor is used to monitor and collect performance metrics for our managed service on the cloud. The Cloud Monitor deploys a distributed cloud monitoring tool on each cloud provider platform that runs the individual service components of the managed service. The Cloud Monitoring tool runs continuously along with the service to collect performance metrics at runtime. It can be configured to collect a wide variety of such metrics at both the application, platform and infrastructure level. In our case, we designed the Cloud Monitor to collect resource utilization metrics for each service component continuously at runtime at an interval of t seconds. We note that the value of t is configured by the DevOps team. This value needs to be tuned for each service component and cloud platform, depending on what kind of metrics the DevOps team wants to monitor as well as the performance overhead imposed by the Cloud Monitor on the service. In our case, we set the value of t based on the observation that collecting the resource utilization metrics from each service tier once every t seconds does not impose a maximum performance overhead of more than $2\% - 3\%$. For multi-cloud setups, the cloud monitoring tool is distributed on all the cloud platforms, and the performance metric data from each cloud platform is streamed to the State Rule Engine and aggregated.

The Cloud Monitor uses *Service Discovery* to find new instances running the service for collecting the desired performance metrics from these instances. Service Discovery is a process for automatic detection of newly deployed services. In order to collect metrics from an instance, the metrics have to be exposed to the Cloud Monitor. This is done by using an *exporter*. An exporter exposes the performance metrics of the instance on which a service tier is running. The exporter is deployed on the same cloud instance as the service component. When an exporter is initialized, it has an

exposed port which can be accessed by the Cloud Monitor to capture the metrics from the instance. When a new instance is deployed along with its exporter, the Cloud Monitor uses Service Discovery to find the instance first. We note that the Cloud Monitor component in our framework can also be used to work with the microservice architecture. To this end, the exporter is deployed as a container on each cloud instance running the service. We note that for containers, the container technology itself exposes the performance metrics of its containers, rather than each individual container. Thus, Service Discovery is not required for new containers. For multi-cloud setups, each of the instances and container platform on the different cloud platforms have their own exporter

The ability to deploy the Cloud Monitor on multiple cloud platforms allows the DevOps team to enable multi-cloud integration (REQ-2). The Cloud Monitor can automatically handle a large selection of input metrics from heterogeneous cloud platforms, which fulfills the requirements of autonomic management (REQ-3) and scalability (REQ-4). Finally, the DevOps team can easily deploy the Cloud Monitor and integrate it with the rest of the components in our framework (REQ-5). The Cloud Monitor provides an API for the other components of our framework to access. The State Rule Engine periodically queries the Cloud Monitor API. The query can include functions, for e.g. sum and average of all metrics collected over the past T seconds. The aggregated data is then sent to the State Rule Engine as an event for the Analysis and Planning stage, as described in detail in Section 2.4.2 next.

2.4.2 State Rule Engine

The State Rule Engine component is responsible for the Analysis and Planning part of the MAPE-K loop to automate the deployment of the services. The State Rule Engine

is used to execute and manage operational rules. These rules are composed of a set of conditional and consequential "When - Then" rules, for e.g., *When* a condition occurs, *Then* perform a consequence or action. Since rules are simple to learn and can be easily understood by the DevOps teams (REQ-5), we implement the State Rule Engine to facilitate autonomic decision-making. The rules are written by both operations and development teams based on the events that are input from the Cloud Monitor to the State Rule Engine. The State Rule Engine enables the DevOps teams to collaborate for maintaining a desired level of QoS for the deployed service at runtime in a large multi-cloud environment. The operations team can utilize the rules to manage and deploy the required infrastructure and platform for the services, while the development team can autonomously configure their application as services on the infrastructure at run-time through the rules. As mentioned earlier, the State Rule Engine queries events from the Cloud Monitor periodically every T seconds. These events are then validated against the conditionals of the rules, where first the "When" statement is checked to be satisfied against the current values of the events. If the conditional is met, the consequential statement is triggered, which performs the action from the "Then" statement mentioned in the rules.

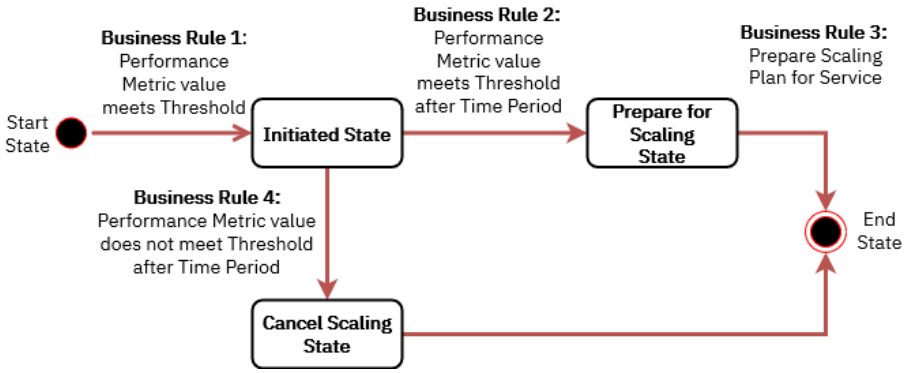

Figure 3: Analysis Stage of State Rule Engine

The consequential "then" statement can perform the Analysis stage or Planning

stage. We can take advantage of the stateful and temporal properties of the State Rule Engine. The State Rule Engine can store objects in memory. These objects include states, metrics and deployment descriptions. Since these objects are stored in memory, we can perform analysis by having rules that compare new objects with the previous objects that are stored in memory. The temporal property means that the objects can have a timestamp, which allows for time-based conditionals in the rules.

For the Analysis stage, we use states in the State Rule Engine that are used to define the current event of the service. The states take advantage of the temporal property which allows us to compare the timestamps of the state and another object. When a business rule is triggered, it can begin the Analysis stage by inserting a new state. An example of this can be seen in Figure 3, where in the beginning Business Rule 1 compares the performance metric event against a threshold value. If the metric meets or exceeds its threshold value, the state of the service is transitioned to an initiated state and an adaptive action needs to be taken, for e.g., scaling up the cloud service. The Business Rule 2 is triggered after another sampling interval and uses the temporal property to compare the current performance metric event with the initiated state to check if the metric still meets or exceeds the threshold value. If so, the business rule causes the state to transition to the next state. This triggers the Business Rule 3 in the sequence and starts the Planning stage which prepares the service for scaling. However, if the performance metric no longer meets the threshold value, Business Rule 3 is triggered that cancels the state from the State Rule Engine, as shown in the figure. The use of states in the State Rule Engine shows forward chaining in action, where the rules are connected with each other in sequence.

For the Planning stage, service deployment and modification plans can be defined in the consequential ”Then” statement. Plans are the deployment details of the service.

Examples of these deployment details include the number of container replicas, hostname, cloud platform location, etc. The cloud operator can specify these input values of the instance that will need to be deployed or modified. The new values will be sent as a POST request to the Workflow Engine's API, where the changes will be applied. The Workflow Engine will then return a response, which gives the details of the service. These details include the name and ID of the deployment, the output value of the services (e.g. IP address) and the input values that were specified by the rule. We then store these details into the State Rule Engine, which can be used for future conditionals and consequential. For example, if we need to deploy a second service that requires the previous service, we can create a conditional statement that will trigger if first service has already been deployed by checking if the deployment details are in the State Rule Engine.

Using the State Rule Engine allows the DevOps team the robustness (REQ-1) to maintain QoS in large services spanning multiple cloud platforms by taking adaptive actions through a ruleset. Using the State Rule Engine, we can orchestrate the scaling of multiple services (REQ-4) deployed on different cloud platforms with rules that are easy to understand. This means integration in the DevOps process will be more efficient with minimal impact on the actual application which improves the ease of deployment (REQ-5).

2.4.3 Workflow Engine

The Workflow Engine is responsible for deploying and managing the services on the cloud through end-to-end automation. It is responsible for the execution stage of MAPE-K by deploying the changes based on the analysis and planning decisions from the State Rule Engine. The Workflow Engine uses Templates, which is a blueprint of

the service and infrastructure details described in high-level configuration syntax. We use Templates to describe the instances and configurations required for running the service. The services are then deployed by the commands issued by the Workflow Engine based on the Template descriptions. Additionally, the Workflow Engine is also responsible for automating the initialization and runtime of the service. Once the service is deployed, the Workflow Engine can modify and apply changes to the running instances, for e.g. applying new updates to the service, making changes in the requirement or taking auto-scaling decisions. Hence, we can automate DevOps operation by integrating the convenience of rules along with the Infrastructure-as-Code (IaC) capability of the Workflow Engine.

The Templates are written in IaC scripts, and the IaC engine is used to deploy and manage these Templates. The deployment variables, such as the access keys, number of VMs, resource allocation, etc, are specified by the DevOps teams and are obtained as input from the State Rule Engine component. The Template can select one or multiple cloud providers where the services will be deployed, thus automating service deployments in a multi-cloud environment (REQ-2). As Templates are high level descriptions of our services, it is easy to keep track of a large number of growing services, supporting the scalability of those services (REQ-4). Using IaC scripts in the Workflow Engine also provides more flexibility for the DevOps teams to describe, modify and manage the service configurations at runtime with ease (REQ-5). We note that this is better than designing, building, testing and maintaining an adaptive framework from scratch, since each of the components has already been extensively tested and designed for production environments. Additionally, the industrial tools used for building the framework components have the added advantage of good documentation and tool support.

RQ-1: It is feasible to develop MAPE-K frameworks for multi-cloud applications by reusing COTS. There is an abundance of commercial and open source services, especially for implementing the Monitoring and Execution components. The offering for Analysis and Planning is weaker, which suggests that more research is needed in developing reusable services for these two activities. A mitigation strategy is to use state-full rule engines, which can encompass both Monitoring and Analysis. Rule engines, given their low complexity but high expressiveness, can also act as a language common denominator for a Development and Operations team.

2.5 Concrete Architecture

Component	**COTS**
Cloud Monitor	Prometheus, Graphite, InfluxDB, Nagios, Zabbix
Rule Engine	Drools, IBM Operational Decision Manager, Red Hat Decision Manager, Grule
Automation Manager	IBM Cloud Automation Manager, Ansible, AWS CloudFormation
Container Platform	Docker, Kubernetes
VM Exporter	NodeExporter
Container Exporter	cAdvsior

Table 2.1: COTS for Each Component

In this section, we present the COTS that we selected to build our MAPE-K framework and describe the integration process for each of the COTS. Table 2.1 show the COTs that we used for each component, as well as other potential alternatives COTS.

2.5.1 Prometheus

For our Cloud Monitor implementation, we used Prometheus [1], an open-source monitoring system and time-series database that can collect runtime performance metrics

[1] https://prometheus.io/

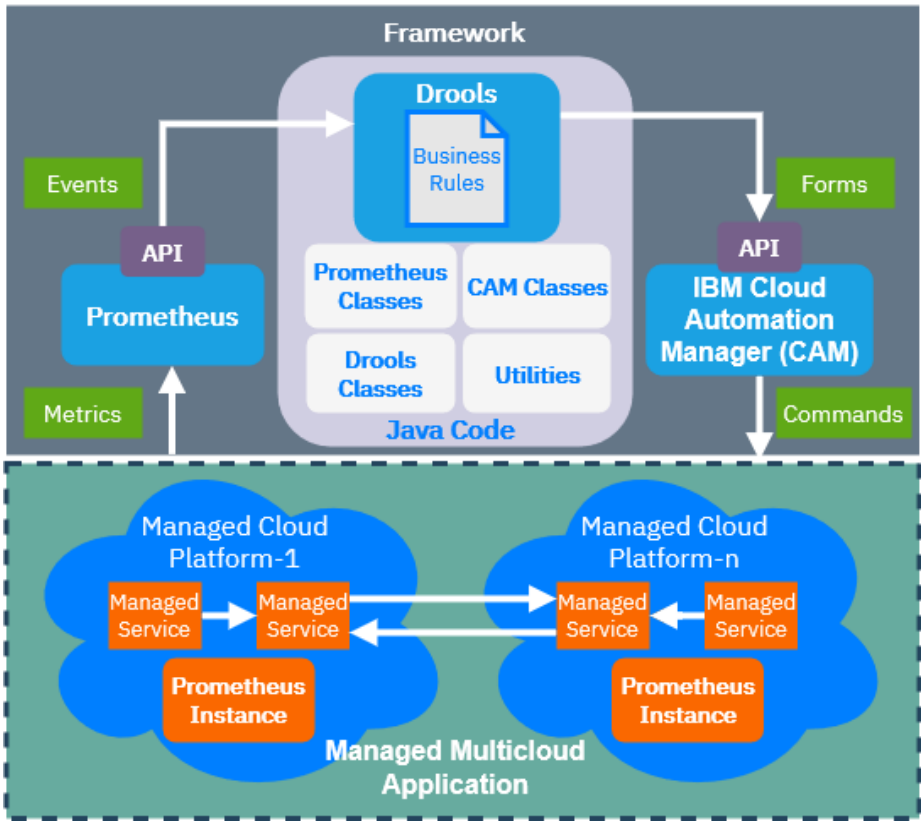

Figure 4: Concrete Architecture

at runtime. Prometheus stores runtime performance metrics collected from an exporter into its local memory and allows us to query the stored metrics. We can use query functions provided by Prometheus to get the aggregate time series data in real-time.

Prometheus integrates data exporters, including third party data exporters, depending on the cloud platform where the service will reside on. For capturing the performance metrics of containers, we use a exporter from Google called cAdvisor [2] which provides the resource usage of running containers. For Virtual Machine, we use Prometheus' NodeExporter to capture the performance metrics of the hardware and the Operating System. In our implementations, we deploy a cAdvisor container on each of the Docker Nodes on our Docker Swarm Cluster. On the Docker Nodes,

[2]https://github.com/google/cadvisor

the cAdvsior exposes its metrics on a specified port (the default port is 8000). In the Prometheus configuration file, we define the cAdvisor port for Prometheus. When new containers are deployed or old containers are deleted, the performance metrics of these will automatically be reflected on Prometheus. For our Hybrid Cloud implementation, we use both cAdvisor on an Openstack-based community cloud, which will be described in more detail in Section 2.6.1, and NodeExporter on the EC2 Cloud. When we deploy a MongoDB instance on EC2 through the IaS script, we also specify in the Terraform script to install and enable NodeExporter on the MongoDB EC2 instance. In the Prometheus configuration, we enable Service Discovery to filter VMs with the String "MongoDB" in its name. NodeExporter metrics are exposed on port 9100 by default, which we specify in the Prometheus Configuration for Service Discovery. For both Node Exporter and cAdvisor, we have to specify the time interval in which we want to collect the metrics in the Prometheus configuration. For our experiments, we collected metrics of cAdvisor and NodeExporter every 5 seconds. The default retention time of the data is configured as 15 days.

2.5.2 Drools

For State Rule Engine implementation, we use Drools [3], an open-source Business Rules Management System (BRMS). Drools is implemented and located on-premise in the IBM Cloud. Drools is Java-based, and it is implemented with our own Java code. This is through Java Libraries provided by Drools that contain the Drools core and compiler. We use the classes provided by the Drools Libraries in our Java Code which will be explained in more detail in Section 2.5.4. The Drools Engine handles a Drools file that contains a set of Business Rules. To get the metrics from the performance monitor,

[3] https://www.drools.org/

we have an initial rule that calls the Prometheus API with a query range that gets the utilization metrics from the last 30 seconds. The rule also uses one of our custom Java methods that calculates the average CPU utilization from the query range of values. To insert the metric into the memory, we create a Drools Type Declaration. While Drools works out of the box with plain Java Objects as facts, the user may want to define a fact directly in the Drools File instead of creating a new object in Java. Drools provides the ability to declare new types in the Drools File to be used with the business rules. In our use cases, we defined a new type called Metrics with the attribute cpuAverage to store the CPU Average from the query range and insert it into the Working Memory.

The auto-scaling decisions in the Business Rules are executed using States and rule chaining. The first rule evaluates the performance metrics against a threshold value in the rule conditional. If the rule is triggered, a state is created and inserted into the working memory. In the state, we set the name and state value. In our use cases, the state value is binary, either the state is new and has not completed the task, or the state has completed the task. Using states, we can create a sequence of rules that react to each others changes, known as Forward Chaining. Instead of reacting instantly to the change in the performance metrics, we insert the state for the next rule to evaluate. If the performance metric is still within the threshold value of the condition and the state has been in the working memory for 30 seconds, the next rule in the chain is triggered. This rule sets the state as FINISHED for the third rule in the sequence. When the state is set as FINISHED, the rule is triggered and begins to call the Automation Manager API and scale the services accordingly. The Drool Rules are able to call the CAM API through our custom Java class to send the deployment and modification details for the services.

2.5.3 CAM

For our Workflow Engine, we use IBM's Cloud Automation Manager (CAM) [4] to handle the deployment of services. It uses the Terraform Engine [5] to deploy services that are described in IaC scripts. Using IBM CAM strengthens the robustness in our autonomic framework as it is an industry component rigorously tested and deployed on actual production environments. CAM is deployed on an internal IBM VMware Cloud. We first need to initialize a cloud connection to our Cloud Platforms. We set the authentication values of our Openstack-based and EC2 cloud connections. For SAVI, we require the Project Name, Domain Name, Region Name and Login credentials. For connecting to EC2, we need the Access Key ID and Secret access key. We have to create a template for our services that we plan on deploying on the cloud platforms. In practice, the Template and its associated files is located in a GitHub repository. When a new commit is placed on the repository, the changes are reflected dynamically in the CAM Catalog.

```
resource "docker_service" "web-application" {
  name = "${var.webname}"
  mode {
    replicated {
        replicas = "${var.web_replica}"
    }
  }
    task_spec {
        container_spec {
            image = "webapp/image-name"
            env {
                MONGO_PORT_27017_TCP_PORT="${var.mongodbport}"
            }
        }
    }
}
```

Figure 5: CAM Template of Web Application

[4] https://www.ibm.com/us-en/marketplace/cognitive-automation
[5] https://www.terraform.io/

Our services and their configurations are described in the Terraform template. An example of a portion of our service Template for a web application can be seen in figure 5. This web application resource is a deployment for Docker Swarm. As seen in the figure, the var.variablename in the template are the input deployment variables. The number of replicas is one of the variables that we modify with our Drools engine for scaling. Other configuration values that can be seen are the name of the web application image and the environment variables in the *env* block. When the CAM API receives the POST request to plan, modify and apply the Template, it begins the process of deploying the instance on the cloud. Depending on the service, it may take a few or several minutes for the deployment to be complete. Once complete, CAM displays the output variables automatically, such as the IP address.

2.5.4 Java Components

To be able to get Drools functioning with Prometheus and CAM, we had to code and implement our own components to support the framework. Since Drools is Java-based, all our components were coded in Java. As seen in Figure 4, there are four main components: (1) Drools Classes (2) Prometheus Classes (3) CAM classes and the (4) Utilities. The Java libraries provided by Drools are imported in a Core java component that we also coded. This core component runs the Drool Engine and orchestrates between all the other components. These Java components are used to connect Prometheus, the Drools Engine and CAM in a working MAPE-K process.

The core components build the Drools Engine and input the Drool rules file into the engine. Building the Drools engine with our Java code first requires creating a Drools memory file system which is provided by the Drools library. We then import our Drools rule file location into the Drools memory file system. A Drools configuration class is

also declared to specify the STREAM option which adds a duration attributes to the rules and memory management techniques to optimize real-time streaming. Finally, we initiate a Drools session from the Drools file system detailed above. The session is used to fire the rules periodically every 30 seconds. The core component also periodically checks if there is any change in the rule files. When there is a change in the Drools file, it rebuilds the session with the modified Drools file.

While we provided alternatives for each of the COTS we selected in the concrete architecture, all the components require a degree of effort and learning curve for implementation. Each component can be replaced with an alternative as long as it meets the list of requirements for multi-cloud deployments. The most generalizable component is the Cloud Monitor with Prometheus, as any monitoring tool that provides metrics to meet our self-adaptive goals can potentially be used. The Rule Engine can be generalized but requires the most amount of effort as it will need to be designed to handle both incoming metrics for the selected cloud monitor and output an adaptation plan that can be understood by the Automation Manager.

RQ-2: The main challenge in developing a MAPE-K from COTS is the development effort for integrating components. While the integration of the Monitoring and Execution components with the underling cloud and the application is of medium complexity, integrating the Rule Engine with Monitoring Execution requires some effort. For our particular case, we needed to write and test 1760 lines of code. A second challenge is the steep learning curve of particular components as well as of the underlying cloud infrastructure.

2.6 Experiments and Evaluation

In this section, we describe the testbed for our evaluations and how we applied our MAPE-K framework to three self-adaptive use cases. We measure the performance of the MAPE-K framework by its ability to cover the major self-adaptive properties of a self-adaptive system, including Self-Configuration, Self-Optimization and Self-Healing. The goal of the evaluation is to test the feasibility of the MAPE-K framework in covering self-optimization, self-configuration, and self-healing. The addition of using the conceptual autonomic framework with existing COTS will allow for DevOps teams to cover multiple Self-* use cases for multi-cloud environments.

2.6.1 Testbed Applications

Acme-Air

In our work, we use a Web benchmark application called Acme-Air [6] developed by IBM. Acme-Air is an implementation of multi-tier airline e-commerce microservice application composed of two service components. The application-tier component is a front-end NodeJS server connected to a data-tier component, a MongoDB Database. These components are deployed as Docker containers that can be run on different cloud platforms. We selected the micro-service application mode and containerized the Acme-Air components by enabling a Docker Swarm Cluster. This allows us to to easily scale the Acme-Air web and database servers using our framework. We used the httperf and JMeter workload generator tools to generate traffic to our Acme-Air application which allows us to control and stress the system metrics (CPU, memory, disk, network) for our use cases.

[6] https://github.com/acmeair/acmeair

HAProxy Loadbalancer

We implemented a containerized HAproxy loadbalancer [7] for our Acme-Air web containers. We deployed the HAProxy container on the Docker Swarm Cluster where the Acme-Air services reside. We used the containerized HAProxy loadbalancer to distribute the workload to the Acme-Air containers within the cluster. We enabled service discovery in the HAproxy configuration file by looking for newly deployed AcmeAir containers by their container name and connecting them to the loadbalancer. In our use cases, we instruct HAProxy to look for container services that has "Acme-Web" as their service name. When a new Acme-Web container is deployed, HAproxy looks for that container that matches the name and port number that is specified in the server-template setting. The workload generator sends the requests to HAproxy loadbalancer as the entry point to Acme-Web containers and we used the default round-robin load balancing algorithm.

Docker Swarm

The Acme Air Service components service components were containerized to be deployed on a Docker Swarm Cluster. We set up a Docker swarm cluster with three nodes for our deployment. All the nodes were on medium-sized VMs (2 VCPUs, 4GB Ram, and 40GB disk). Two of the nodes were worker nodes while one of the nodes was both the master and worker. This allowed us to run Acme-Web in a distributed environment with a large amount of CPU and disk resources to stress using the workload generator. Since the web service and the mongoDB containers connect with each other from different nodes, we create a Docker overlay network. An overlay network sits on top of the host-specific networks allowing all containers in the network to communicate.

[7] https://hub.docker.com/r/haproxytech/haproxy-debian

The HAProxy loadbalancer is deployed on the master node which has a public IP address and acts as the entry-point. All the other worker nodes are internal and only have private IP addresses.

Cloud Platform

We deployed our services on SAVI Cloud [8] and EC2 Cloud[9], while the BRE and Automation Manager were deployed on premise in an internal VMWare Cloud [10]. SAVI is a community cloud provided as a partnership between universities, research and industry in Canada. It is built on Openstack [11], which is one of the many clouds supported by the services we used to build our framework. For our Hybrid Cloud use case, we deploy the Acme-Air database services on EC2. Since Workflow Engine will need to use the API of both SAVI and EC2, the authentication values are required by the Workflow Engine. For Openstack, these values can be found in the Openstack RC File V3. The Authentication URL, Region Name, Project Name, Domain Name, Username and Password are required for the Workflow Engine to connect to the Openstack API. For EC2, the Access Key ID, Secret Access Key and Availability Zones are required for the Workflow Engine. Since the State Rule Engine and Workflow Engine are located on the IBM internal cloud, these components also need to also connect to the performance monitor and the Docker Swarm API to scale the containers. Public floating IP is associated for the Docker Swarm Master node and the Cloud Monitor instance. We use security groups for both these instances, and create rules such that only IP addresses of State Rule Engine and Workflow Engine can access the exposed ports of the Cloud Monitor and Docker Swarm APIs.

[8] https://www.savinetwork.ca/
[9] https://aws.amazon.com/ec2/
[10] https://cloud.vmware.com/
[11] https://www.openstack.org/

2.6.2 Use Cases

Self-Configuration

It is important for a web application to prevent the end user from experiencing high latency and availability issues. To this end, web applications should be able to self-configure automatically for providing good quality of service while optimizing costs incurred due to resource utilization. This allows for the support of Continuous Deployment by keeping the application within good performance requirements. We demonstrate the self-configuration capability of our framework in the first use case using the Acme-Air Web application. Our Acme-Web and MongoDB containers are all deployed on the three-node Docker Swarm Cluster. We test our framework's self-configuration ability to see if it can keep the average CPU utilization of the Acme-Web and MongoDB containers within an acceptable threshold range. For this purpose, we set the upper and lower threshold limits for the average CPU utilization in Acme-Web to 50% and 25% respectively. For our MongoDB container, we set the upper and lower threshold limits for the average CPU utilization to 10% and 5% respectively. We first deploy our performance monitor and initiate service discovery, as mentioned in Section 2.5. Next, we write business rules in Drools that can handle the self-configuration scenario.

The first rule we create is to insert the Prometheus metric into the Drools engine. This rule triggers periodically after an interval set up by the DevOps team. In our case, this was done by setting the timer attribute for the rule to 30 seconds. In the "Then" statement, we get the query range from the last 30 seconds of the average CPU utilization from our Prometheus component, which we insert into a Metric object and store it in the Drools memory, as mentioned in Section 2.5.2. The first business rule for self-configuration checks if the Acme-Web containers have an average CPU utilization

over 40% and whether there is any existing state in the memory for scaling. When the threshold and requirements of the ”When” statement in figure 6 are met, we initialize the state. As seen in the figure, we give the state an identifying name ”ScaleUp” to allow the other rules responsible for scaling up to look for this state. We set the state value to NOTRUN, which is the initial value of the state. We then insert this state into the working memory. The next rule checks if the ”ScaleUp” state exists in the working memory, and if it is in the initial state. This rule executes if the average CPU utilization is still over 50% after 30 seconds of the initial state. The state is set to FINISHED for the final rule in the sequence to trigger. The reason why we have these two rules is to check if the average CPU usage is consistently above the upper threshold limit of 50% since we want to prevent reacting to sudden and transient utilization spikes.

```
rule "Prepare for Scale Up"
   when
      metric: Metric (metric.getCpuAverage() > 50)
      state : State (name == "ScaleUp" && state == State.NOTRUN, this
          before[30s] metric)
   then
      state.setState(State.FINISHED);
      update(state);
end
```

Figure 6: Business Rule for Finishing State

The rule that is responsible for the execution stage and sending the scaling plan to CAM is shown in Figure 7. This rule checks if the State is complete and inserts a camJson() object as seen in the figure. This camJson object includes deployment value of the number of current Acme-web replicas. We initialize the CamTemplateAPI from the Cam Caller Component in the ”Then” statement, specifying the endpoint of our CAM deployment. We get the current value of the replicas for our Acme-Web Service using the getNumericalValue() function, which is a simple function that parses

```
rule "Scale Up" salience 10
   when
      camJson : CamJson ()
      state : State (name == "ScaleUp" && state == State.FINISHED)
   then
      CamTemplateAPI camAPI = new CamTemplateAPI("<HOSTNAME>");
      int value = camJson.getNumericalValue("acme.json", "web_replica");
      camJson.changeValue("acme.json", "web_replica", value + 1);

      CamJson ModifyJson = new CamJson();
      camAPI.ModifyInstance("<instance-id>","acme.json")
      camAPI.ApplyInstance("<instance-id>")
      retract(state)
end
```

Figure 7: Business Rule for calling CAM to Scale

the Json of the Acme-Web deployment. We increase the value of the replica in the Json file. We send the Json file through a POST request for modifying an instance to the CAM API and then send a POST request for applying the modification on the SAVI cloud. We then retract the state, allowing the process to accept new ”ScaleUp” states. This process is same for scaling down as well. If the average CPU usage does not meet the threshold value while a ”ScaleUp” or ”ScaleDown” state is already in the memory after 30 seconds, another rule is triggered that removes the state, and restarts the rule sequence.

There were challenges when setting up our Self-configuration use case with our framework. When we first began to deploy the Acme-web infrastructure through Terraform, both our MongoDB and Acme-Web containers resided in a single Template. However, due to Terraform's behavior, this caused some issues. As mentioned before, the Automation Manager may need to recreate the instance when a Modify plan is applied. When having Acme-Web and MongoDB services in a single template file, we observed that when scaling the Acme-Web containers, it would remove the MongoDB container each time. We decided to create two different templates, one for MongoDB

and another for Acme-Web. This would allow us to scale without causing an increased unavailability time by removing the MongoDB container. We also needed to figure out a way to keep track of the number of replicas when we scale. Originally, we started our BRE with no services deployed on SAVI and wrote rules that deployed the initial infrastructure before scaling. However, most of the time there is already an existing service on CAM that we may want to modify. For this reason, we choose to deploy and modify services by using Json files. With the acme.json file, we can keep track of the web replicas of an existing service. However, one related challenge that is unresolved is tracking changes to services that occur through the CAM UI with our BRE. If the DevOps team makes changes to the services through the CAM UI, we need to be able to sync those changes while the BRE is running.

Self-Healing

When a service goes down, we need the ability to automatically respond and re-deploy the service. We demonstrate our framework's capability to self-heal a service in the second use case. This can support the DevOps team and Continuous Deployment by maintaining good availability while potentially logging and alerting the DevOps team when the service went down. To this end, we use our framework for self-healing Acme-Air by automating the redeployment of a MongoDB container when it goes down instead of redeploying MongoDB manually. We check the status of MongoDB for Acme-Web to see if we get a response. This is done through our Prometheus instance, which is periodically monitoring the MongoDB instance. If we receive an empty or irregular JSON array from Prometheus, we conclude that the MongoDB service is down. Instead of inserting the CPU usage average metric into the working memory, we insert the JSON output when we query Prometheus every 30 seconds.

Similar to the rule that initializes the state in the Self-Configuration use case, we

```
rule "Prepare for Self-Healing"
   when
      jObj : JsonObject
          (jObj.get("data").getAsJsonObject().get("result")
          .getAsJsonArray().size() == 0)
      state : State (name == "HealDB" && state == State.NOTRUN, this
          before[30s] jObj)
   then
      state.setState(State.FINISHED);
      retract(jObj);
      update(state);
end
```

Figure 8: Business Rule for Self-Heal State Initialization

run a rule that checks if the query of our MongoDB service is empty. If empty, the rule inserts a new state named ”HealDB”. The next rule in this sequence is shown in Figure 8. The ”When” statement checks if the query is still empty after 30 seconds of the initial state. The rule is then triggered, and the state is complete. The time duration between the two rules that modify the state is to confirm that the MongoDB service stays down consistently. The final rule in the sequence checks if the state is complete, and uses the ModifyInstance and ApplyInstance functions seen in Figure 7 to MongoDB.

Hybrid Cloud Scaling

One of the key features of CAM is that it allows the DevOps team to automatically deploy services in a Hybrid cloud setup. An organization can have private data that they prefer not to store on a public cloud and store it on their internal private cloud instead. There also may be unique services that are only offered by a specific cloud platform and the organization may want to use different services from different cloud providers simultaneously, such as using storage instances from one cloud platform while having front-end services on another cloud platform. Resource and cost constraints can also be

a reason to have a Hybrid Cloud setup. Our framework enables the capability of CAM to allow for automatically managing services in a Hybrid Cloud setup. In this use case, we scale a MongoDB instance in EC2, while simultaneously scaling an Acme-Web container in our SAVI cloud. Similar to the Self-Configuration and Self-Healing use case, we have a rule that checks the CPU average usage of the MongoDB instance deployed on EC2. We implement business rules for scaling up and scaling down these instances in a Hybrid Cloud setup. Since the Acme-Web container needs to connect to a running MongoDB, we first have to deploy a MongoDB instance. When the state is finished after the 30 second timer, the business rule sends a POST request to CAM to deploy a new MongoDB instance on EC2 with a json file. The json file contains the deployment variables for our MongoDB instance. Once we deploy the MongoDB instance on EC2, we can then deploy an Acme-Web container in SAVI that connects to MongoDB through the next business rule in sequence. We write our rules such that they prevent the Acme-Web container to be deployed while the Mongo-DB deployment is still not complete.

When scaling instances on a Hybrid Cloud platform, we had to consider some unique challenges. Since we have our services on two different cloud providers, we have to deploy two cloud monitors, one on SAVI and the other on EC2. Every time we have to deploy a service on a new Cloud Platform, we have to deploy Prometheus with its respective Service Discovery configuration as it can differ for each cloud platform. Next, we needed to automate the connection of the MongoDB instance on EC2 to the Acme-Web containers in SAVI with our framework. By deploying MongoDB as an instance, CAM is able to get deployment output variables from the EC2 instance, and in our case, the public IP address of the MongoDB instance from EC2. We can then specify in our rule that the Acme-Web container will connect to the MongoDB

instance with that IP address. This simplifies the automation process to establish the connections between the two services by using CAM to get the output variables of newly deployed services. Since we have our Database on EC2 and our Web Server on our SAVI cloud, we need to automate ways to secure the connection between Acme-Web and MongoDB. We first have to manually create a security group that only exposes the MongoDB port to the IP address of SAVI Instances used for outbound requests. With CAM, we can specify the unique ID of the security group in the template which it will automatically associate the MongoDB instance with our security group during deployment.

2.7 Results and Discussion

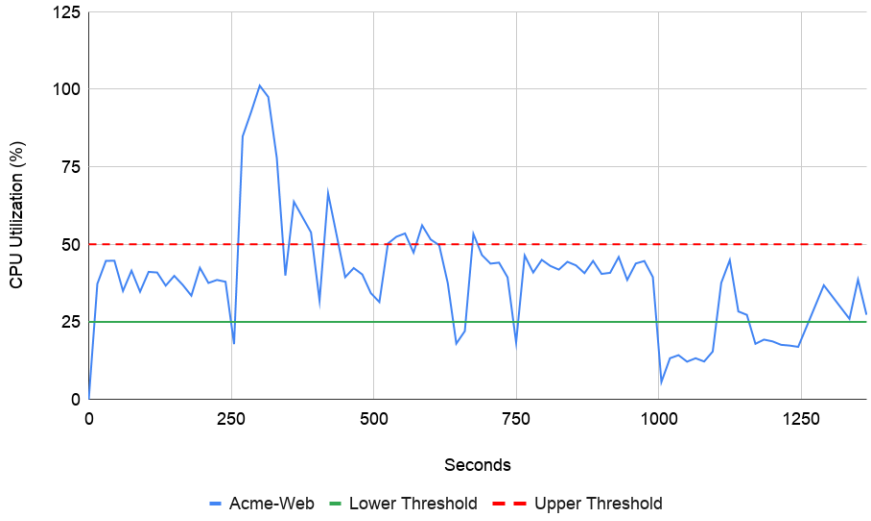

Figure 9: Acme-Web CPU Utilization during Self-Configuration

The results of the Self-Configuration use case are shown in Figure 9 and Figure 10. We start with one Acme-Web container and gradually increase the incoming workload which causes the CPU utilization of the Acme-Web container to exceed its upper

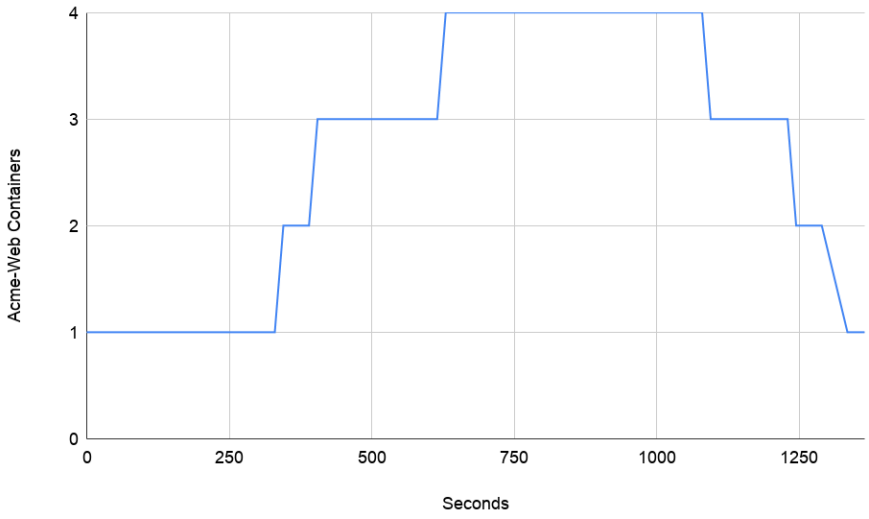

Figure 10: Number of Acme-Web Containers during Self-Configuration

threshold limit, as seen around the 270 second timestamp in Figure 9. Our framework correctly identifies this transgression, and triggers the business rule that waits for an additional sampling time of 30 seconds before scaling. This rule automatically adds a second Acme-Web container to the service at the 345 second timestamp as seen in Figure 10, taking 35 seconds to deploy the container from when the rule to scale up is first triggered. The average CPU utilization is still above the threshold at this point, and the framework adds an additional container at the 405 second timestamp. Our framework successfully self-configures by distributing the incoming workload between the 3 containers such that the average CPU utilization of the Acme-Web containers is maintained within the acceptable threshold limits for 75 seconds until the workload is increased again. To this end, our framework adds a total of 4 Acme-Web containers to the service and the average CPU utilization stays inside its acceptable threshold values from the 630 second timestamp to the 1005 second timestamp as we no longer increase the workload. At this point, the workload generator starts decreasing the workload to the

service, which triggers the scale-down business rule in our framework. The utilization decreases below the lower threshold at the 1020 second timestamp, after which begins the additional 30 second sampling period before scaling down. This rule takes 35 seconds to remove the Acme-Web container. However, even after removing a container, the CPU utilization stayed below the lower threshold limit. Our framework successfully identifies this transgression and further removes two containers, the first container at the 1245 second timestamp and second container at 1335 second, thus maintaining the average CPU utilization within the threshold value.

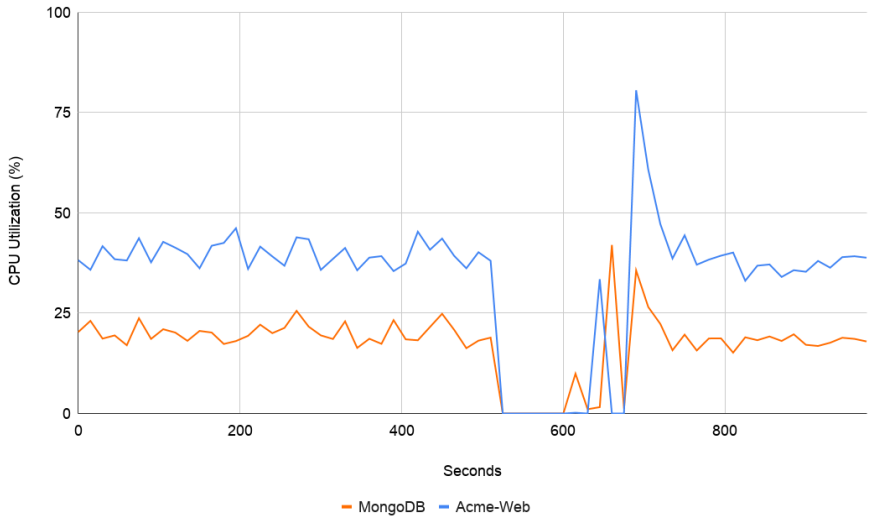

Figure 11: Acme Air CPU Utilization during Self-Healing

Next, we discuss results for the Self-Healing use case as shown in Figure 11 and Figure 12. We configure the Acme-Air service with one Acme-Web and one MongoDB container and send incoming workload to the service such that the CPU utilization values for both the containers are at acceptable levels. At the 525 second timestamp, the MongoDB container goes down, which reduces the CPU utilization values for both the MongoDB and Acme-Web containers to zero, as seen in Figure 11. This indicates

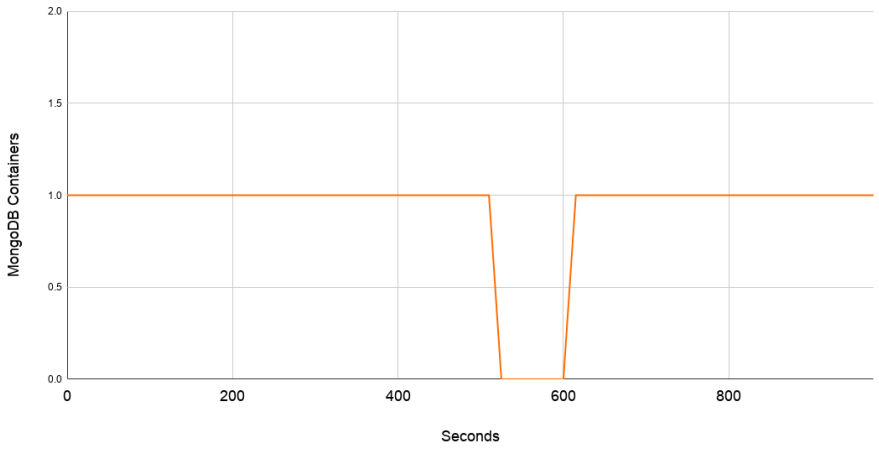

Figure 12: Number of MongoDB Containers during Self-Healing

that at this point, the MongoDB container is not working and the Acme-Web container, which is dependent on the MongoDB, also stops working. Our framework correctly identifies this fault after waiting an additional 30 seconds and triggers the self-healing business rule described previously in Figure 8 to correct this fault. Consequently, the framework automatically re-deploys the MongoDB container, as seen in Figure 12. which takes 42 seconds. Once the container is deployed and running, it takes an additional 90 seconds for the MongoDB container to stabilize, after which the container utilization levels are within acceptable limits.

Finally, we show results for the Hybrid Cloud Scaling use case in Figures 13, 14 and 15. We deploy this scenario using one Acme-Web container on the private SAVI cloud platform and one MongoDB instance on the public EC2 cloud platform. We set the upper and lower average CPU utilization thresholds for MongoDB to be within 5% to 10% CPU Utilization since MongoDB on EC2 incurs lower levels of CPU Utilization than Acme-Web on SAVI. We start by sending incoming workload to the Acme-Air application such that the CPU utilization of both Acme-Web and MongoDB are maintained within the acceptable threshold limits. We then increase the workload so

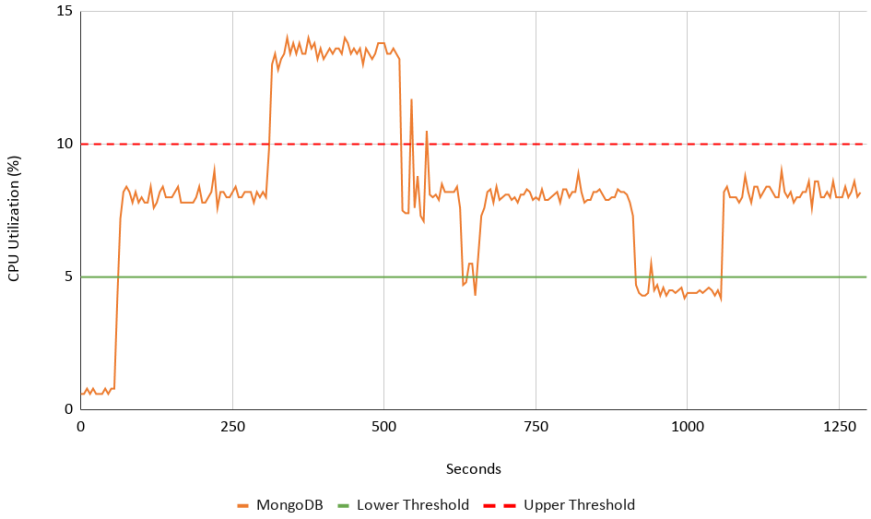

Figure 13: MongoDB Average CPU Utilization on EC2

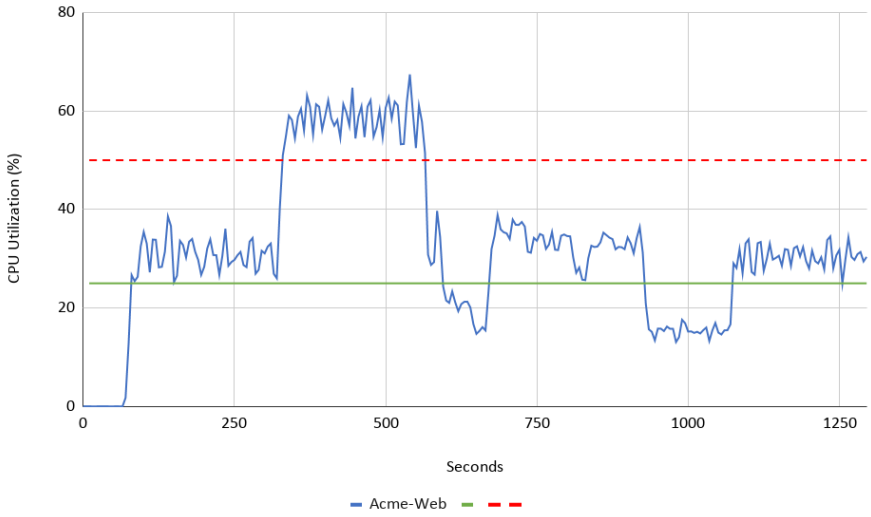

Figure 14: Acme-Web Average CPU Utilization on SAVI

that their CPU utilization values exceed their upper threshold limits at the 325 second timestamp. Our framework correctly senses the need for scaling the hybrid cloud setup and triggers the corresponding business rules. The rules then scales the hybrid cloud automatically by adding a MongoDB instance in EC2 and then an Acme-Web container

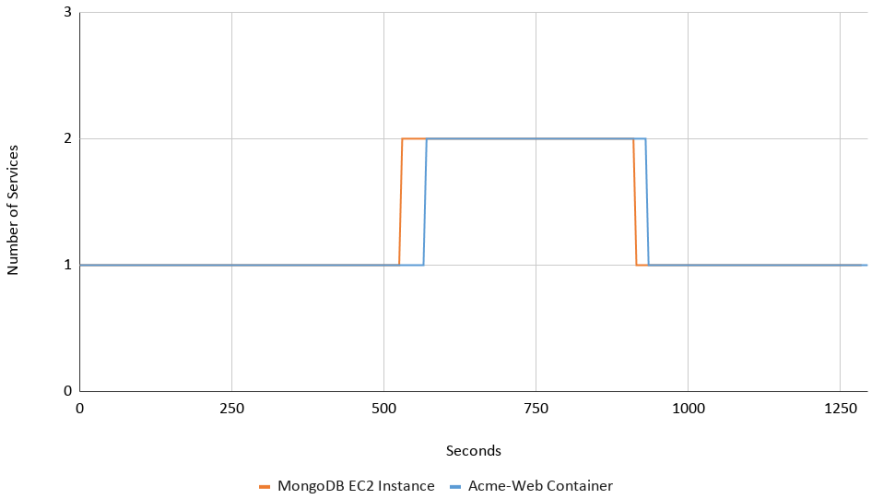

Figure 15: Number of Acme-Web and MongoDB Containers

in SAVI, as seen in Figure 10. Since we are deploying MongoDB as an EC2 instance instead of a container, it takes longer to deploy because it needs to initiate and install MongoDB. The framework then evenly distributes the incoming workload between the 2 Acme-Web containers such that the CPU utilization values of Acme-Web and MongoDB are maintained within their respective threshold limits, as seen from the figures. Finally, we reduce the workload to the application at 925 seconds, which triggers a business rule for scaling down the new deployments. The rule deletes the newly deployed Acme-Web container from SAVI and the MongoDB instance from EC2 at 1070 second timestamp and is able to maintain the average CPU utilization within acceptable limits.

RQ-3: The MAPE-K framework we proposed, has the required performance and efficiency. We showed that it supports deployment and self-configuration in a multi-cloud environment, one of the main tasks of the DevOps pipeline. Furthermore, it can implement self-configuration and self-optimization across multiple clouds. Similarly, it can implement self-healing scenarios. The syntax of rules, used to implement the self-* is accessible to both Development and Operations team, making it appropriate for DevOps.

2.8 Summary

Enabling automation for service deployment and configuration in the cloud for the DevOps practice allows us to maintain good Quality-of-Service and continuous delivery. In this chapter, we presented an industrial framework that can support the automation of services on the cloud. We detailed the three main components of our framework, the Cloud Monitor, State Rule Engine and the Workflow Engine. These components interact with each other in our framework implementation based on the MAPE-K feedback loop.

Chapter 3

Performance Modeling of Change Impact Prediction for Cloud Applications

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 [34, 35]. 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 [4,5]. 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 [3].

Model based control in self-adaptive systems utilizes dynamic models created at run-time to make adaptive decisions [36]. 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 [37, 38]. 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-user performance from being significantly affected and keep our cloud application(s) within the SLA.

3.1 Research Questions

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 chapter 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.

This chapter is the research published in 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) titled "Towards a Robust On-line Performance Model Identification for Change Impact Prediction". The chapter is organized as follows. We discuss the background of current research in performance modeling in self-adaptive cloud systems in Section 3.2. We discuss the motivations and foundations of our work in Section 3.3. We then present our methodology architecture and approach in Section 3.4. We then discuss our experimental setup and implementations in Section 3.5. Finally, section 3.6 shows the results of our approach compared to other models.

3.2 Background and Related Work

3.2.1 Performance Analysis and Modeling of Cloud-Native Application

There is past work that studied the performance analysis and modeling of microservice based architecture for cloud native applications [39–42]. Jindal et al. [39] 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 [41]. 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 [43, 44] and for cloud-native microservice applications [45, 46]. However, while we look at the performance impact due to an in-

terference, 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. [47] 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 [48]. Machine learning can also be used to train models to be used for self-adaptive systems [49]. However, the run-time models used by self-adaptive managers utilize historical data and can be unaware of new deployments.

3.2.2 Performance Exploration and Probing Technique

There are research works similar to this concept of deploying exploration mechanisms [50–52] 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. [53] 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.

Other research works similar to this concept include deploying exploration mechanisms. Strube et al [50] measures the performance of an standard application through a software probe approach that characterizes the application. Moradi et al. [51] build a container-based network performance monitoring system, however, their solution utilizes a probe within the container to inject and receive network packets to characterize the network and traffic dynamics. Victorin et al.l also utilize probes to analyze network traffic at the application layer, which is low-power and able to offload time consuming operations [52].

3.3 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

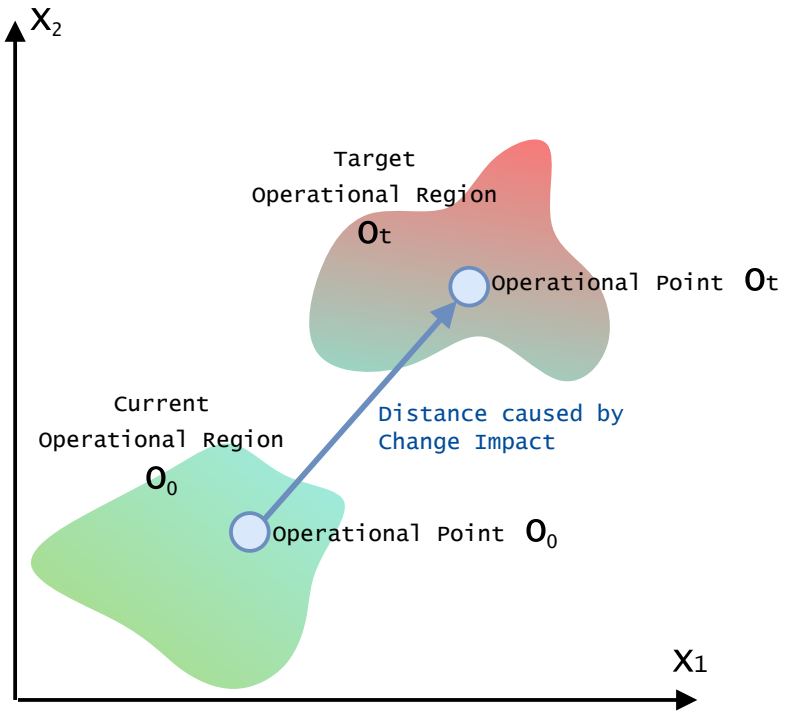

Figure 16: Operational Regions and Points O_0 and Target Operational Regions and Points O_t

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 16 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.

3.3.1 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 16.

3.3.2 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 16 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.

3.3.3 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,

$$[U_1^t ... U_K^t]^T = [U_1^0 ... U_K^0]^T + DI \tag{3.1}$$

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 $DI = [11.54, 18.87]^T$.

3.3.4 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.

3.4 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.

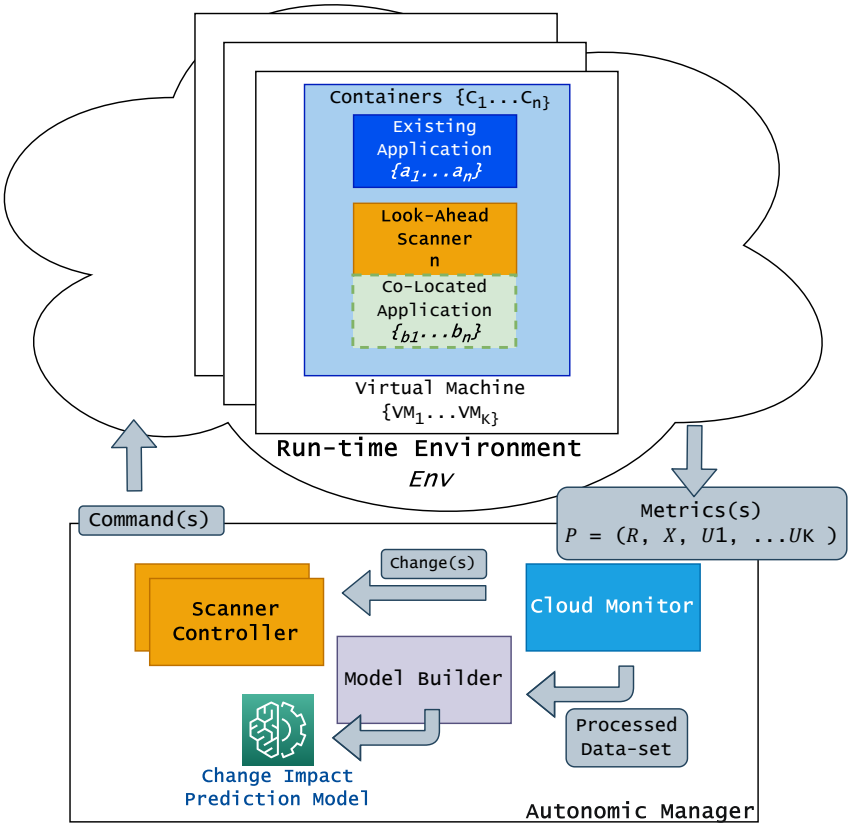

Figure 17: Overview of Look-Ahead Scanner in Production

3.4.1 Methodology and Architecture Overview

Figure 17 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).

3.4.2 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.

3.4.3 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.

The LAS can work in self-adaptive systems discussed in Chapter 2. For example, we can have rules set for co-Location and consolidation based on temporal or performance requirements. The LAS can be deployed by the execution stage of the MAPE-K to generate data during the monitoring stage. Once the LAS has completed its run, the prediction model is used by the analysis and planning stage to co-locate a new application or consolidate the environment if the predicted response time is under the SLO threshold.

3.4.4 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.

3.4.5 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.

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 [54], the end-to-end delay of an application R_a can be expressed as:

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

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:

$$U_k = U_{k,a} + U_{k,b} \tag{3.3}$$

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:

$$R_a = \sum_{k=1}^{K} \frac{D_{a,k}}{1-(U_{k,a}+U_{k,b})} \tag{3.4}$$

From Equation (3.4), 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. (3.2)) by sampling the throughput X and the utilization U and estimating D using the Kalman filter [55, 56]. Once calculated, D can be used to evaluate the response time at point O_t using Equation (3.4). 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 $QNM(O_t, O_t)$.

Machine Learning Models

model the system as a black box. We can explore many models, [57], 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.

$$R_a = f(U_1..U_K) \tag{3.5}$$

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.

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 (3.2) with a first-order Taylor series [58]:

$$R_a = R_a^{O_0} + \sum_{k=1}^{K} m_k * \Delta R_a / \Delta U_k \tag{3.6}$$

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.

3.4.6 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 [59] 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:

$$R_{threshold} = R_{SLA} \times X\% \tag{3.7}$$

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.

3.5 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.

3.5.1 Test-bed Setup

In-Production Application *a*

As the in-production application *a*, we use a Web benchmark application called Acme-Air [60] 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.

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 [61] is an open source message broker for the MQTT protocol for sensors to publish and

subscribe messages. NodeRed [62] is a flow-based development tool built in Node.JS to connect APIs, hardware services, and sensors. InfluxDB [63] 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) [64, 65].

Look-Ahead Scanner

The Look-Ahead Scanner was built and deployed as a lightweight Node.JS [66] application, and its dependencies are containerized to be deployed on the Docker platform. The Node.JS application interacts with Stress Tools [67] that are packaged inside the container. For the Cloud Monitor, we deploy Prometheus to monitor the VM(s). Prometheus [68] 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 [69], a machine learning library for the Python, to build Regression Models. We use Opera [70] to build our QNM models. We use the H20 AutoML frameworks [57] 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.

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 Mon-

itoring, 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 [71] 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.

3.5.2 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.

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 3.6.1.

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 3.6.1.

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 3.6.2. The datasets for both experiments are available on Github[1].

3.6 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. The goal of the evaluation is to observe the prediction accuracy of models built around

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

the current operational regions and models built around the target operational region. The addition of run-time datasets at the target operations regions will improve the performance of the models. We compare the LAS approaches with state-of-the-art models which include several models provided by the AutoML framework suite. Current model identification adaptive systems use such models during run-time, however, they do not use extrapolated data from target operational zones. We demonstrate that by applying the LAS, we can improve the performance of current run-time models.

We evaluated the effectiveness and accuracy of our models by using Mean Absolute Percentage Error (MAPE). As discussed in our experiment setups in Section 3.5, 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

$$MAPE = \frac{1}{n} * \sum_{i=1}^{n} \frac{|R_{measured,i} - R_{predicted,i}|}{|R_{measured,i}|} * 100 \tag{3.8}$$

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.

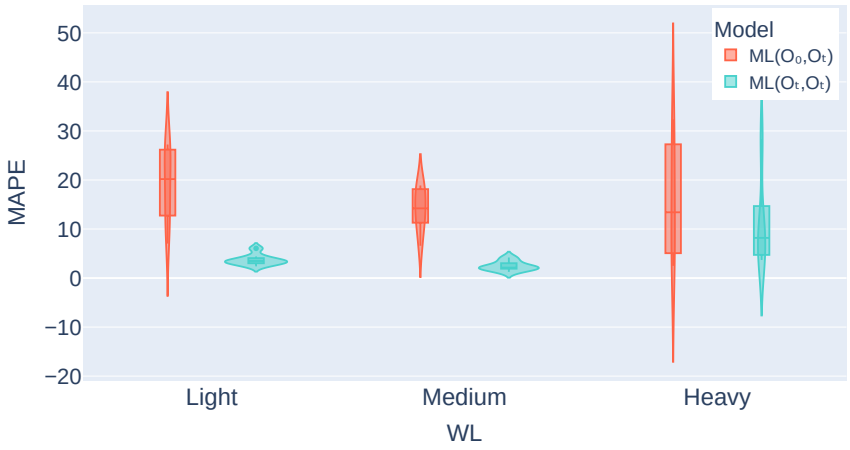

Figure 18: Prediction at O_t for 1VM Deployment

3.6.1 RQ-1 Results: Feasibility of the Performance Impact Model

Figure 18 and Figure 19 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 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 3.1 presents a closer look at the MAPE values as the 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

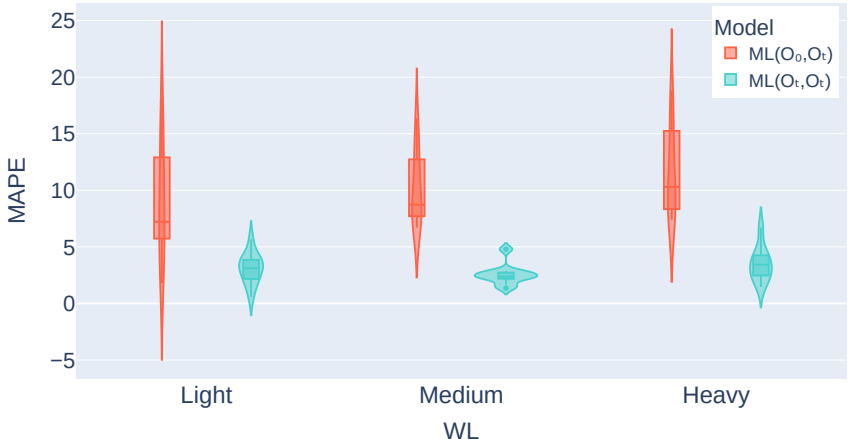

Figure 19: Prediction at O_t for 2VM Deployment

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 3.1). Overall, Machine Learning outperforms Queuing and Regression models.

			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 3.1: Prediction at O_t for 1VM Deployment

3.6.2 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 a real IoT application that shares the same VMs as AMCE-Air. Figure 20 and Figure 21 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 20, 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 21, the $ML(O_t, O_t)$ and $QNM(O_t, O_t)$ models outperform their re-

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

spective $M(O_0, O_t)$ 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 3.2 and Table 3.3 show MAPE at different co-location operational regions. The light workload can be seen in the rows where the *Utilization of Acme-Air at* O_0

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

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 3.2). Overall, Machine Learning outperforms Queuing and Regression models. Experiments also show that pushing the applications close to saturation points (heavy load) yields uncertain behavior.

3.6.3 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.

3.6.4 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 3.4: LAS Complexity

Complexity. Table 3.4 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 [55] 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 3.4 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 17) runs externally and we use the instrumentation available on containers.

3.7 Summary

In this chapter, 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.

Figure 20: Prediction of Error Improvement of Co-locating Applications on 1VM Deployment

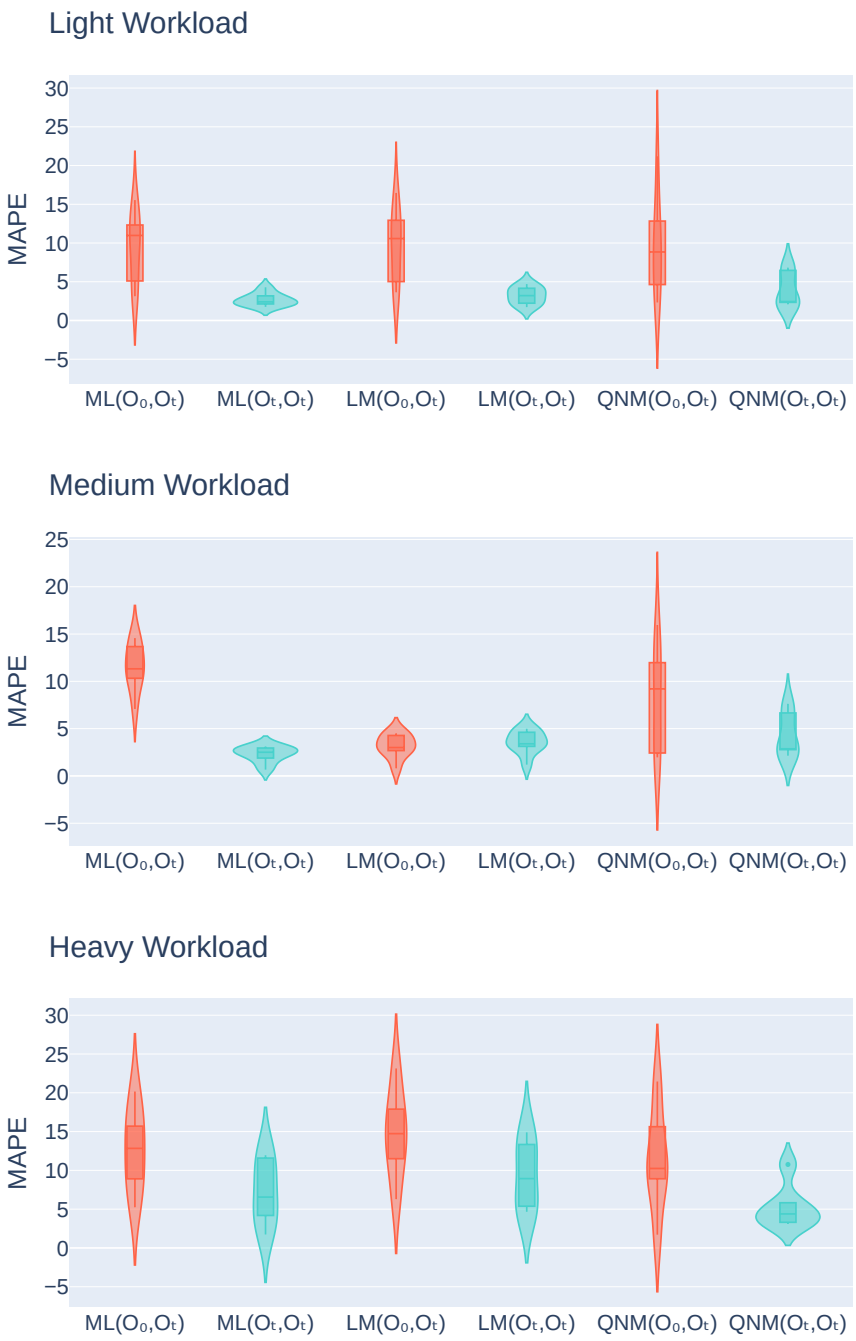

Figure 21: Prediction of Error Improvement of Co-locating Applications on 2VM Deployment

Chapter 4

Interference Identification for Cloud Applications

To improve resource consumption levels and optimize cloud costs, multiple microservices from different applications can be deployed and consolidated on a single VM. However, this can cause microservices to compete with each other for shared host-level resources. Shared resource contention can impact the application behavior and make it deviate from the development time specifications, service level objectives and performance. We refer to application performance degradation due to shared resource contention as *performance interference*. Performance interference has been observed before in applications running on public cloud environments [72–79].

Detecting and disambiguating performance interference anomalies from other causes of performance degradation is very important for application runtime self-management and for avoiding outages. For example, performance degradation due to legitimate workload surge can be automatically mitigated by horizontal scaling, whereas interfer-

ence might be mitigated through application redeployment and relocation. Another use case where detection interference anomalies is important is in reducing the number of alerts the IT operators have to deal with. It is estimated that the an IT operator faces *80-100* alerts per system [80]. We need to detect if there is interference, and disambiguate if the performance degradation is from interference or from normal behavior. Therefore, it is important that an application detects when its execution environment is perturbed significantly by other applications.

Automatically detecting runtime interference in cloud-native applications is a challenging task. Application workload variability and unpredictability can produce the same effects as interference, and hence it is difficult to distinguish between them. Researchers have looked, with limited success, into instrumenting applications [81], recording their mean request response times, and comparing them against their baseline response times to detect the presence of interference. Continuously instrumenting application code and monitoring the response times of running services [82] can significantly increase the cost of development and maintenance. In addition, continuous monitoring of service metrics can incur a prohibitive overhead when the service is facing a heavy workload [83, 84]. Furthermore, interference detection techniques that model the baseline behavior may not generalize well to scenarios where the interfering application changes at runtime.

4.1 Research Questions

To address some of the above challenges, we propose a light-weight method that uses a small number of deployment and runtime performance metrics to automatically disambiguate interference and workload effects on performance. The method involves build-

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

The paper addresses the following research questions:

- **RQ-1:** How well does a disambiguation method perform when the interfering workloads used for model training and deployment are similar for an At-Scale Deployment?
- **RQ-2:** How well does our disambiguation method perform when the interfering workloads used for model training and deployment are not similar for an At-Scale Deployment?

To answer RQ-1, we develop and train models to detect interference caused by ap-

plications with similar performance characteristics for large at-scale deployments on a multi-node cluster. Results show that ML models outperform state-of-the-art regression based and threshold based interference detection techniques by at least 0.67% and at most 23.29%. Furthermore, our method incurs minimal overhead of only 1% to 2% on the service response time at runtime.

To address RQ-2, we show that we can develop and train models to detect interference caused by an arbitrary application that stresses the same resources as used by the target at-scale application on a multi-node cluster. This question covers a common case when we do not know a priori what applications might share the infrastructure with our managed application. In addition, this model generalizes well for detecting interference from different containerized interfering applications. Lastly, we evaluate scalability by using our model to detect interference against a large multi-tired microservice application. As presented in our results, our technique outperforms state-of-the-art regression based and threshold based interference detection techniques by at least 2.75% and at most 53.86%.

This chapter makes the following contributions:

1. We introduce a formal definition of cloud-native application interference disambiguation, rooted in queuing theory.

2. We introduce an interference detection method that generalizes across interference scenarios and outperforms the state-of-the-art.

3. We evaluate the method using several model types, including queuing and machine learning models.

4. We validate the method on four industrial strength applications and show that it scales to large deployments and works in public clouds.

This chapter is the research work published in ICPE '24: 15th ACM/SPEC International Conference on Performance Engineering titled "Disambiguating Performance Anomalies from Workload Changes in Cloud-Native Applications".

4.2 Related Work

4.2.1 Interference Definitions

Paul et al. [74] characterize performance interference in co-located VMs and further measure the application performance degradation and impact on low-level system metrics. Koh et al. [73] investigate the impact of VM-level performance interference from co-located workloads and characterize interference impact on low level system metrics. They cluster workloads by interference type and construct performance prediction models for co-located workloads using weighted means and regression analysis. The authors highlight types of workloads that perform well or not when co-located. In contrast to these work, we characterize the impact of interference among containerized microservices.

4.2.2 Interference Measurement and Classification

Jha et al. [77] measured intra-microservice and inter-microservice interference using four benchmark microservices. The results of their experiments showed significant container interference present in both intra-container and inter-container cases. Garg, and Lakshmi. [78] ran experiments using well-known containerized benchmarks to detect microservice interference and accordingly identified the shared resources subject to contention in these scenarios. Novaković et al. [75] propose DeepDive to mitigate VM-level performance interference. Their approach detects interference by comparing

low-level system metrics against a workload's baseline values and migrates VMs to other physical machines as needed. Wang et al. [76] present Vmon, a system that detects and quantifies VM-level performance interference. Vmon profiles an application running on a VM to observe how Hardware Performance Counters correlate with application performance. DeepDive and Vmons detect VM-level interference through use of low-level system metrics whereas our work detects microservice interference using metrics that are readily available and do not pose prohibitive overhead in collecting. Additionally, our work is differentiated in that we evaluate the effectiveness of model re-use across different scenarios.

4.2.3 Interference Regression

Joshi et al. [79] propose Sherlock, a method for detecting long-lived performance interference in containerized services caused by co-located VMs. The authors employ a regression detection technique to model performance interference using VM and application-level metrics at runtime. Kang and Lama [85] predict microservice interference using Gaussian Process models trained at runtime using a sliding window technique. Baluta et al. [86] predict microservice interference using AutoML models trained at runtime using a sliding window technique. Our work differentiates from these as we explore the reuse of pre-trained interference detection models in varying environments.

4.3 Interference Definitions

Assume that at deployment time, we profile the performance of an application and the interference effects in the space (U, λ, R), where U is the utilization, R is the

response time and λ is the workload (arrival rate of requests) of the application and infrastructure. That profile is captured in a model and the profile is defined over a large workload space and interference levels.

In Figure 22, we illustrate interference in a simplified two-dimensional space, U and R. The blue area is the baseline profile for one workload ($[\lambda{=}\lambda_1]$). The response time of application, with no interference, falls within 95 percentile Confidence interval, denoted $[minR_{base}\ maxR_{base}]$. With the help of tunable slack variable, α, we define a *operational zone* around the operational area (gray area). This parameter is application-dependent and defined by the application owner and can be derived from Service Level Agreements. Interference zones are those areas in the space (U, λ, R) where response time is outside the interval $[minR_{base}\text{-}\alpha, maxR_{base}{+}\alpha]$ but cannot be explained through the change of the workload. For example, red areas in the figure are interference anomalies, whereas green areas depict normal behaviour, non-interference, caused by workload. Although the anomaly depicted in red can be explained by other causes (bugs, misconfigurations, etc.), in this work we focus on interference anomalies. ΔU is the utilization step size between non-interference and what is classified as interference. Although both are outside the operational zone, it is important to know what causes the shift in the operation region. Different root causes of anomalies can be mitigated through different runtime actions.

4.3.1 Queuing Network Models and Interference Modeling

In this section, we explain how interference can be formulated as Queuing Network Models (QNM). To begin, we consider a containerized application a running inside a VM without interference, as seen in Figure 23. We assume that the application a stresses a single resource k in the VM so as to incur an utilization of $U_{a,k}$ on the re-

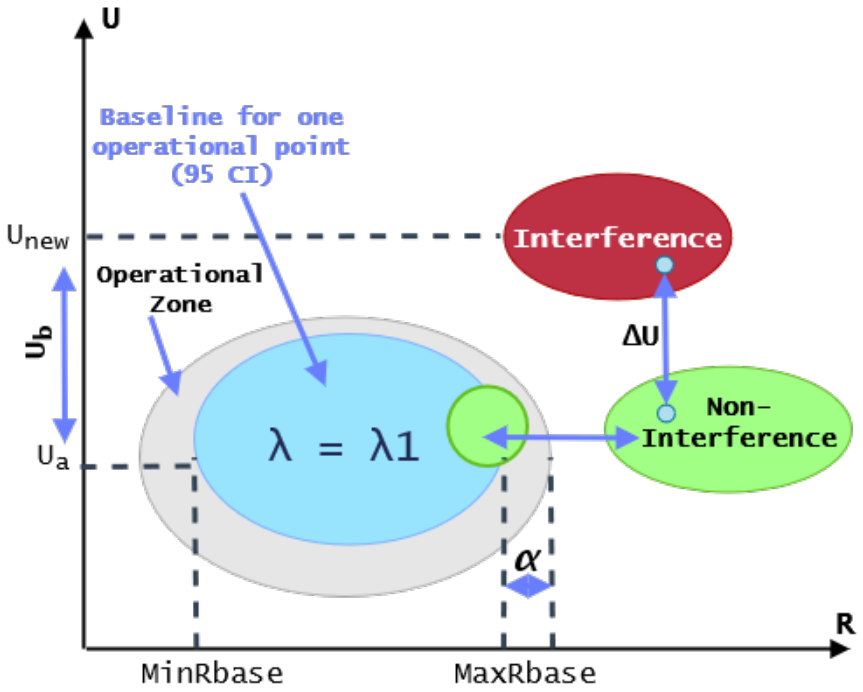

Figure 22: Interference Range

source. We assume a request arrival rate of λ_a to application a. If, at runtime, we had a way to measure service demand $D_{a,k}$ of application a at resource k, we could then predict the no-interference mean request response time R_a of application a as:

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

where U_k is the total utilization incurred at resource k in the VM. Since in a no-interference scenario, the only application stressing resource k is our monitored application a, $U_k = U_{a,k}$. Finally, we substitute U_k with $U_{a,k}$ in eqn. 4.1 to obtain the response mean time R_a of application a in a no-interference environment as given by:

$$R_a = \frac{D_{a,k}}{1 - U_{a,k}} \tag{4.2}$$

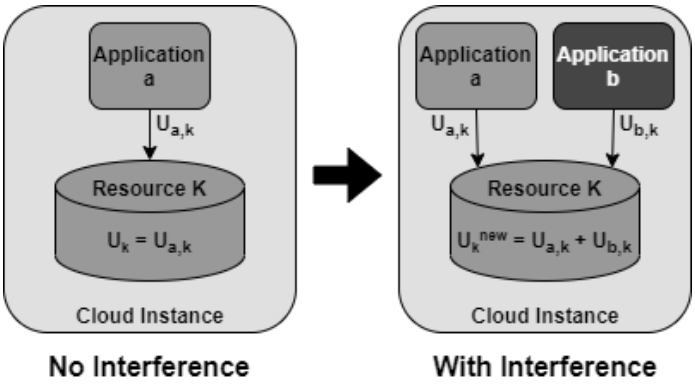

Figure 23: Effect of Interference on Resource Utilization

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

$$R_a = \frac{D_{a,k}}{1 - D_{a,k} * \lambda} \tag{4.3}$$

and this allows us to extrapolate the response time outside of an operational point and for any workload λ (cf. green areas in Figure 22).

To illustrate interference, consider another application b running inside the same VM, as seen in Figure 23. Application b incurs an utilization of $U_{b,k}$ on resource k inside the VM. Accordingly, from eqn. 4.1, the new response time R_a^{new} of application a when it faces interference from application b is:

$$R_a^{new} = \frac{D_{a,k}}{1 - U_k^{new}} \tag{4.4}$$

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

$$U_k^{new} = U_{a,k} + U_{b,k} \tag{4.5}$$

Equations 4.2 and 4.5 explain interference (cf. Figure 22, red zones).

QNM Methodology

We tune QNM following the approaches in [55,56]. Since we work under the assumption that we do not have access to the metrics of other applications and we consider the system is a product form network [87], we can use Eq. 4.3 to estimate the response time R and compare it with the measured response time. If the measured response time is outside the bounds [R-α, R+α] then we label it as affected by interference, otherwise we consider the deviation as caused by application load.

4.3.2 Machine Learning Classification and Interference Modeling

Based on eqn. 4.2 and eqn. 4.4, we observe that a point [λ_a, R_a, U_k] is transposed to [λ_a, R_a^{new}, U_k^{new}] in the same tri-dimensional space under the impact of interference. Based on this observation, we aim to use resource and application metrics to train ML models that can classify two distinct classes of response time values, one class denoting the no-interference scenario represented by R_a, and the other denoting the interference scenario represented by R_a^{new} (corresponding to red and green zones in Figure 22). We note that it appear more practical to use ML models instead of QNMs for interference detection since it is infeasible to accurately measure resource service demands for all containers belonging to a monitored microservice application serving different kinds of workloads. Furthermore, multiple levels of virtualization involved in a cloud-based microservice container can involve estimating service demands for unknown virtualized resources [88], which can be difficult. In contrast, ML models use resource utilization and throughput metrics that are easy to collect at the container and VM levels.

We use our understanding of QNMs as motivation for applying machine learning to detect interference. From eqn. 4.4, since the new response time R_a^{new} of application

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

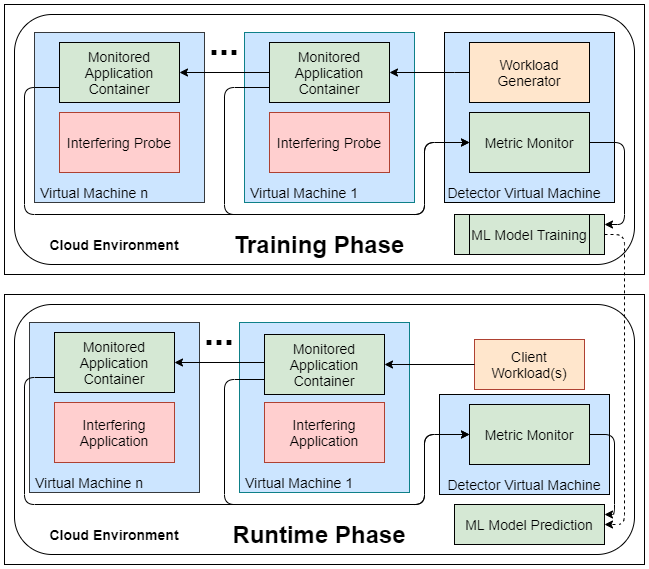

Figure 24: Overview of Interference Detection Technique

4.3.3 Machine Learning Methodology

In this section, we present our methodology focused on ML models since they are central to our work. Section 4.3.3 defines the assumptions of our technique. Section 4.3.3 details data generation and pre-processing. Section 4.3.3 describes the methodology to train an ML model for interference detection. Section 4.3.3 details the use of an ML model for interference detection at runtime.

General Methodology

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

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

Data Collection and Pre-Processing for ML

Figure 24 presents the high level overview of our interference detection technique. As seen in the figure, our technique runs in two phases. The first phase is the *training phase* where we run controlled experiments to train our ML model in an offline model training environment. In the training phase, we run our target microservice application in a controlled cloud environment where its response time is not impacted by perfor-

mance interference. This is done by running the target application in isolation on the n VMs hosting the microservice without any interfering application present. The objective is to obtain baseline metrics when no interference is present. To this end, we increase the arrival rate of the application workload to the target application in steps to incur a wide range of resource utilization observed at each application container. This is done through a *Workload Generator* tool running inside a Detector VM which resides alongside the application VMs, as seen in Figure 24. We ensure that our workload generation setup is free from internal software bottlenecks and network latency issues by following the approach detailed in past work [72]. Since application owners can only access metrics from the VM and their own application containers, we omit including metrics of any other kind in our data set for ML model training. To this end, a *Metric Monitor* tool from inside our Detector VM, as shown in Figure 24, collects measurable metrics from the target application and its environment such as application response time and throughput, and container and VM resource usage. Although we choose the average request response time of a target application as our performance metric, other metrics such as the mean throughput values can also be used. Using only the application throughput and response time along with VM and application resource utilization metrics as input allows for a smaller feature set that is easy to monitor, collect and does not incur prohibitive monitoring overhead at runtime.

We repeat each step of workload generation N times to obtain N estimates of the average application request response time R_{base} at each step. We refer to N as the *number of workload repetitions*. We use this data to construct the 95% Confidence Interval (CI) of the baseline application response time at each step. The upper and lower limits of this CI are denoted as $maxR_{base}$ and $minR_{base}$, respectively. To mitigate against performance variability inherent in public cloud platforms, we repeat each step,

i.e. set the value of N in our training phase such that the width of our CI is within 5% of its sample mean at each step. Alternatively, if the cloud platform has significant performance variability, the application owner may consider deploying the application containers inside *dedicated VMs* which have same specification as general VMs and are offered by public cloud platforms to provide stable performance. Although dedicated VMs are more expensive than general VMs, the application owners only need to use them for a short period of time to collect training data for ML models.

Once we obtain the 95% CI of the application's baseline response time, we next introduce interference into the system. For this purpose, we run a controllable *interfering probe* along with the target application on the n VMs hosting the target application as shown in Figure 24. Doing so imposes additional stress on shared resources which is expected to increase response time as depicted in Equation 4.4. Similar to before, we send the same step-wise increasing workload to our target application from our workload generator tool from inside the Detector VM. We also simultaneously vary load on the interfering probe to diversify its degree of shared resource contention and introduce different levels of performance interference on our target application at each step. We record the average request response time R_t of the target application along with VM and container utilization metrics collected by the Metric Monitor tool when the interfering probe is also running. This is done at our sampling interval of t seconds continuously for a duration of x seconds to obtain the training data set to be used in the ML model. The application throughput, response times, and container and VM utilization metrics for each sampling interval represent a single record in the training data set of our ML model.

We use the baseline response time data obtained in the training phase to label the ML data set in our offline model training. As mentioned before, our ML model outputs

binary classification with 2 labels, where label 1 indicates interference and label 0 indicates no-interference. To this end, we compare the value of R_t at each record in this set with its corresponding value of $maxR_{base}$ obtained at the same step of workload. As depicted in Figure 22, if R_t exceeds $maxR_{base}$, we infer that the application response time suffers from performance interference and accordingly assign the label 1 to the record. Otherwise, the record is assigned with label 0. Once the training data set is labeled, we remove the associated response time pairing from each record to construct the final input data set to the ML model. In our experiments, 30 to 35 % of the data points were labeled as interference. We observe that our labeled data set is imbalanced. Hence, we configure the H2O AutoML training job to balance classes. Doing so enables classifiers to learn indicators of performance interference as opposed to reporting the majority class.

ML Model Training

AutoML [57, 89, 90] has gained popularity as an effective solution for training well-performing ML models. The AutoML framework evaluates several ML models trained on the same data set against one another and outputs the best performing model. We leverage the H2O AutoML framework [57] in our ML model construction. This framework can be easily integrated with the DevOps feedback loop to automate runtime interference detection.

The framework trains multiple models including but not limited to XGBoost and Stacked Ensemble Models. An ensemble model is composition of several ML models and their predictions. The framework automatically tunes ML algorithm hyper-parameters using random grid search over a predefined range of possible hyper-parameter values. H2O AutoML iteratively evaluates models under these varying hyper-parameter

configurations and outputs the best performing model. In addition, the H2O AutoML framework employs 5-Fold cross validation by default to promote models generalizing well to unseen data. We employ an 80-20 stratified train-test set split while preserving class ratios. We evaluate our models by *Precision*, *Recall*, and F_1 *score*.

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

ML Model Runtime Deployment

Once ML model construction is complete, we move on to the second phase, i.e., the *runtime phase*, where we deploy the ML model to detect container interference at runtime as seen in Figure 24. In this phase, we continuously monitor throughput, VM and container resource utilization at runtime as the microservice serves client traffic. Subsequently, this data is fed as input to the ML model, which in turn classifies the current runtime state of the target application as either 'interference' or 'no-interference'.

4.4 Experimental Validation

The experiments are designed to answer research questions RQ-1 and RQ-2 for a range of deployments and interference types. We consider: a) multiple target and interference applications; b) different deployments for the target and interference applications; c) different types of interference. In this section, we show the design of the experiments and the implementation setup. Section 4.5 details the results of these experiments.

4.4.1 Target and Interference applications

In the experiments, we use 'in-production' applications such as Boutique as our Target application. Online Boutique [91] is a popular open-source benchmark application developed by Google. The e-commerce application is composed of multiple microservices deployed on Kubernetes, an open-source container orchestration system.

As interference applications, we use stress-ng, Air Quality Monitor and Acme Air. Stress-ng [67] is a linux stress tool that allows the user to stress the CPU, memory, disk and other resources. We use stress-ng as a configurable artificial application benchmark which can be used to generate a wide range of resource utilization levels. Air Quality Monitor [92] is an Internet of Things (IoT) application that processes air quality sensor data. Acme Air emulates transactions for an airline website and consists of 2 microservices, a NodeJS Web server, and a MongoDB database. Acme Air is a well known and frequently used benchmark [93, 94]. We run these 2 microservices in their own Docker containers. We refer to these containers as the *Acme-Web* and *Acme-Db* containers respectively.

4.4.2 Interference Types

We consider the following interference types:

1. *Correlated and similar workloads (CSI).* In this use case, we consider that the interference is generated by applications in which the load at different VMs is *correlated* with the load of an ingress service/gateway). The interfering applications belong to the same class as the in-production application, therefore the workloads of the in-production and interfering application are *similar*. We consider e-commerce applications, like Boutique and ACME, for both training and inference.

2. *Correlated but dissimilar workloads (CDI)*. Here, we consider that the interfering applications belongs to a different class than the in-production target application. We use Boutique as the in-production application and the Air Quality Monitor application, [92] which is an Internet of Things (IoT) application, as the interference.

3. *Non-correlated and dissimilar workloads (NDI)*. We consider that interference can happen at any VM, independently (*uncorrelated*). Also, the interference is not similar to the load of the in-production application. We use stress-ng [67] for training and inference. Prior work [95] has used the stress-ng benchmark to incur varying level of resource utilization by stressing the system under study.

We select the interference types to evaluate how well our method can classify the interference of different classes of applications. We select an interference type of an e-commerce web application and an IoT sensor monitoring application. These are two realistic applications that have different deployment, performance and workload patterns. Correlated workloads refers to the interfering application services distributed on multiple VM(s), thus interference affecting one VM will affect another VM. In addition, we use a load inducing method to incur stress on VM(s) separately to build a model for interference.

At-scale Deployments

To demonstrate the scalability of our methodology, we selected Online Boutique, a large 11-tier microservice e-commerce application. In our experimentation, we deploy Online Boutique on an Amazon EKS (Elastic Kubernetes Service) Cluster with 4 cluster nodes using an EC2 node group composed of EC2 m5.large VMs. Our Amazon EKS uses Amazon Linux 2 and Docker as the container runtime. We have a total of 14

Kubernetes pods of Online Boutique distributed between the 4 Cluster Nodes. For our interference application, we selected Acme-Air and stress-ng from our previous experiments. For the Acme-Air deployment, we distributed Acme-Air and MongoDB pods on two of our EKS nodes where the Boutique front-end pods reside. This deployment pattern is also repeated with two stress-ng pods. We have in total 9 pods responsible for monitoring our cluster at the container, node and application level.

4.4.3 Experimental Setup and Methodology

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

Workload Characteristics

For our at-scale deployment, we use Locust, a python-based workload generator [96], to generate the workload on Online Boutique. The workload generator increases the utilization on each of the nodes by 15% increments, ranging from 15% to 80% utilization on the Kubernetes Cluster.

We use *httperf* [97] as the Workload Generator tool to send workload to our target Acme Air application. The httperf client runs inside a separate VM and is not containerized. We use the *httperf* tool to submit the default workload obtained from the official Acme Air project [60] as the workload transaction mix submitted to Acme Air.

We use the number of concurrent connections and inter-request arrival time settings in *httperf* to drive a wide range of application resource utilization. Acme Air workload

has a step size increase of approximately 12.5% CPU utilization. stress-ng has a step size increase of 20% CPU utilization. The Air Quality Monitor has a step size increase of 20% CPU utilization.

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

Metrics Monitoring

To monitor the applications and their environment, we leveraged prominent industry solutions for the *Metric Monitor*. Prometheus is an open-source monitoring solution that integrates with multiple metrics exporters [68]. Additionally, application metrics averages were output in httperf logs. The application metrics are joined with the container and VM metrics from Prometheus by timestamp. These metrics are used in analysis, data labelling, and ML model construction.

RQ-1 Setup

We conduct a set of experiments to compare the ML interference detection technique with a state-of-the-art detection technique used in current research [79]. We chose this technique as a regression based approach commonly used in interference and anomaly detection. We adopt the detection method outlined in [79] to construct logistic regression models for interference detection at runtime. Logistical regression predicts the probability of occurrence of a binary classification by using a logistic function. The

regression models are fit on the same datasets that were constructed and used for the ML approach. At runtime, we use the regression model in our at-scale EKS experiment setups to predict whether interference is present or not in the Boutique application. We compare the performance of our QNM and ML approach with the logistical regression model. We evaluate the Precision, Recall, and F_1 score when these baseline technique are applied in our experiments. In addition to the regression model, we evaluate Queuing Network Models to provide another baseline.

RQ-2 Setup

We run experiments to evaluate the effectiveness of ML models in detecting performance interference when the interfering application class at runtime is different from training time. State-of-the-art techniques considered are the same Regression techniques as introduced in the RQ-1 Setup. QNM models are omitted as they explicitly define interfering applications whereas in these experiments we assume no knowledge of the interfering application class at runtime.

To train our ML models, we use the same methodology as described in section 4.3.3. For these environments, we choose one of our three applications to serve as the interfering probe benchmark application in the ML model training. These three applications correspond to the NDI, CSI, and CDI scenarios.

Next, we deploy these trained ML models in the runtime phase as described in section 4.3.3. However, at runtime, we evaluate the resultant ML models where a different interfering application is present in the environment than as seen at training time. Accordingly, a different interference scenario is encountered at runtime than training time. The goal is to evaluate the model performance against an interfering application class not yet seen by the ML model. To measure the effectiveness of our

generalized ML model, we evaluate our method against state-of-the-art models of the logistic regression and simple threshold techniques described in the RQ-2 Setup. The logistic regression model follows a similar methodology as our ML model in that the interference scenario encountered at runtime is different than that used at training time.

4.5 Results and Discussion

In this section, we evaluate the interference scenarios for our research questions and present our findings. The goal of the evaluation is to observe the positive prediction of identifying interference at runtime with models built outside the operational region of interference. The addition of interference datasets will train models with high identification accuracy. In addition, the higher the step size of interference from the base utilization, the greater the ML model can accurately identify interference.

4.5.1 RQ-1 Results: When Interfering Workloads for Model Training and Deployment are Similar for an At-Scale Deployment

We evaluate the performance of ML, QNM and state-of-the-art models to detect interference when the interfering application load is known in the training phase. The applications used for the interfering workloads are the same for both model training and deployment for the RQ-1 results. This is indeed an ideal case, however, it is important in defining a baseline. We compare the effectiveness of three different approaches for all our application scenarios; Regression, QNM and ML.

Table 4.1 show the F_1 score, Precision and Recall for the Boutique deployment experiments. The F_1 score for the ML approach has less variability than the regression method which shows that the ML approach is more consistently able to detect perfor-

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

Table 4.1: Boutique subject to Interference

mance interference for large at-scale deployments. Our QNM model performed the worst when detecting interference for at-scale deployments. Specifically, the CDI scenario demonstrates the limitation of the QNM approach and it's inflexibility in handing variability and incomplete metrics which caused the QNM approach unable to classify interference. Overall our ML approach outperforms the other approaches from 2.22% up to 13.41%.

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

Table 4.2: ML Accuracy with Different Interference Step Size

As the ML approach has the best overall performance, we then investigated the step size of the workload and it's effect on the performance of the ML approach. Since the same workload is used across all the experiments, we investigate how the F_1 score of the ML approach are affected with different workload step sizes. Table 4.2 shows differing workloads of the interfering application with the Base Utilization as the starting point of the interfering application and its respective step size. A higher base utilization and step size increases causes the ML model to classify the interference with more ease. As the step size of the workload is reduced to smaller values, the F1-score decreases.

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

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

Table 4.3: Top Performing ML Model Per Scenario

Finally, we report the ML models with highest F_1 Score for both the single and dual VM experiments in Table 4.3. Scenarios represented in this table are described by the number of VMs in the environment as well as the characteristics of interfering application. As seen in the table, GBM models performs best for interference detection in all single and dual VM CSI scenarios. In the dual VM NDI scenario, a stacked ensemble model performs best. In all CDI scenarios, a stacked ensemble model also performed the best.

RQ-1: We evaluated interference detection techniques on an at-scale deployment and different interference types where the interfering load is the same in the training and runtime phases. ML models outperform logistic regression, QNM and simple threshold techniques in each of our evaluation experiments in F_1 score by at least 2.22% and at most 13.41%. The high ML performance is maintained at-scale deployments and we can conclude ML models can better handle the variability of the cloud.

4.5.2 RQ-2 Results: When Interfering Workloads for Model Training and Deployment are Different

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

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

Table 4.4 shows our results from the experiments conducted in Section 4.4.3. Table 4.4 denotes the interference scenario encountered at training time as well as the interfering scenario present at runtime. In this way the ML models were evaluated in scenarios where the interference scenario is unknown and previously unseen. For the large at-scale deployment, we compared the highest performing approaches in the previous experiments. When the interfering application is unknown with large at-scale deployments, Table 4.4 shows the Regression approaches begin to struggle in detecting the performance interference. Our ML approach outperforms the F_1 score of the regression up to 20.57%.

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

Table 4.4: Boutique Training/Runtime Interference

RQ-2: We evaluated ML versus the state-of-the-art interference detection techniques where the interference scenario is different in the training and runtime phases. The ML models outperform state-of-the-art techniques in F_1 score by at most 20.57% % for all but one interference types and different deployments. We can conclude that training with interfering applications yields better performance than if we train with random interference (NDI with stress-ng). The performance of the ML approach (F_1 score), although lower than in the case of RQ-2, is high enough and scales well so it can be applied in production.

4.6 Threats to Validity

We identify threats to validity of our work as per Wohlin et al [98]. We note as an external threat that our ML model training approach might not detect interference from

unknown interfering applications well if the resource utilization signature of the benchmark interfering application is significantly different from the interfering application used at runtime. If the interfering application used in the training phase and the interfering application at runtime stresses different VM-level resources, our technique may be unable to classify interference. Furthermore, as another external threat, in multi-VM deployment strategies frequently used for microservice deployment, if the VM characteristics at runtime change relative to what was used in the training phase, our ML models may not be successful and will need to be re-trained.

4.7 Summary

In this chapter, we propose a runtime performance interference detection technique that leverages Machine Learning. Our ML models are trained on microservice resource utilization metrics subject to varying environments and different interfering applications. Our approach does not require expensive service instrumentation and does not pose prohibitive monitoring overhead. Our proposed technique is effective in detecting performance interference for a realistic microservice benchmark running on the EC2 cloud platform and also outperforms baseline and state-of-the-art interference detecting techniques. Our ML models are also effectively used in varying cloud environments where the interference characteristics change at runtime.

Chapter 5

Proactive Identification of Performance Models for Cloud Consolidation and Co-location

Organizations have been leveraging various cloud platforms and APIs to build and deploy their applications to better support the DevOps cycle. These applications are commonly designed to have a microservice architecture, where the application is broken down to several independent individual services characterized by its own specific functionality. Containerization has become highly prevalent in application development and deployment and enables the developer to easily encapsulate their application services and its libraries and dependencies in a lightweight executable container. To manage and automate containerized applications in a dynamic environment, container orchestration tools and platforms have been used to create, modify, and remove cloud resources in cloud environments. Container orchestration can involve both large mono-

lith applications and hundreds of services from multiple modern microservice-based applications designed specifically for the cloud, known as cloud-native applications, to run alongside each other in an organizations cloud ecosystem. Container orchestration tools make it easier to support distributed applications on such clusters. Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and Google Kubernetes Engine (GKE) are some of the most popular deployment services to manage containerized applications and clusters on the cloud. Applying cloud operations for cloud-native applications is an important practice for the DevOps team to continuously meet evolving requirements by deploying, managing, and optimizing of cloud services that run on a public, private, or multi-cloud environment. This practice is called CloudOps [6] [7] and supports automation, scalability, and continuous deployment of services.

Automation in CloudOps provides the organization with the ability to construct an operational framework for a new addition, update, or removal of cloud resources [6]. Large cloud clusters that have been provisioned to habitat a set of microservices may not efficiently utilize all its resources overtime, causing unnecessary costs. Service consolidation is when the machine or cluster node needs to be removed and the services of a cloud-native application are consolidated to the remaining machine(s) or node(s). Consolidation occurs when resources are not utilized efficiently by the machines in a cloud environment and there is a need to reduce cost or energy consumption. The virtual machines or cluster nodes will be removed and the microservices will be consolidated to the remaining nodes. In contrast to the consolidation, co-location refers to addition of new services or applications that share the same resources with existing ones. Co-location saves cost on resources, management, and overhead [99]. However, applying consolidation and co-location operations to a cloud cluster can have an unknown performance impact. A co-located application can negatively impact the end-

user performance of the existing application(s) on the cluster. For example, a node in a multi-node cluster can have a resource-intensive microservice with a key functionality. If we deploy a new co-located application on that specific node, we may negatively impact the performance of our entire cloud application environment. With service consolidation, we can negatively impact performance by consolidating services on an insufficient number of nodes. A self-adaptive system needs to predict the impact of consolidation and co-location operations before executing them. A predictive performance model that relies on past data to guide co-location decisions has big limitations, as shown in our previous work [100]. That work also shown that, among logistic regression, queuing and machine learning, machine learning (ML) models are better to model non-linearities outside the operational zone.

In this work, we propose a proactive performance sampling and modeling technique for at-scale cloud native applications on multinode clusters to be used by self-adaptive systems for scaling-up, co-location and consolidation operations. We use a Look-Ahead Scanner (LAS) that injects controllable perturbations at multiple cluster nodes, at run-time, to scan ahead at target operational points representative of a consolidated cluster or a co-located deployment. In addition, to speed up the data collection and model building, we augment the look ahead sampling data with synthetic data generation.

The research questions are as follows:

- **RQ-1:** What is the improvement in accuracy of a proactive model for co-location trained with LAS-approach compared with standard and current approaches?
- **RQ-2:** What is the improvement in accuracy of a proactive model for consolidation trained with LAS-approach compared with standard and current approaches?

- **RQ-3:** How effective is the augmentation of proactive datasets with synthetically generated data?

We construct a methodology that incorporates the LAS approach into three different algorithms that allows for exploring the operational region when cloud resources are both added, updated or removed through cloud operations. For RQ-1, we evaluate prediction accuracy of proactive models trained with the LAS-approach compared with standard models for co-location of at-scale deployments on a multi-node cluster. For RQ-2, we evaluate prediction accuracy of proactive models trained with the LAS-approach compared with standard models for consolidating multi-node clusters. For RQ-3, we use machine learning for data generation on datasets sampled by the LAS approach to generate synthetic data and augment the synthetic data in existing proactive datasets. We then evaluate the prediction accuracy of the proactive models extended through augmented synthetic data.

The original contributions of this chapter are as follows:

- We propose a Look-Ahead Scanner Controller to dynamically inject controlled load perturbations and build a proactive performance model to explore the operational regions of consolidation and co-location for an at-scale deployment.

- We propose three different algorithms for the LAS approach for CloudOps. The first algorithm is the exploration method which probes the environment for scaling up and consolidation. The second algorithm uses the LAS exploration methodology to proactively sample and model consolidation. The third algorithm is a Provisioning Controller that uses the proactive models to apply adaptive changes during run-time.

- We present experimental results that show models built with the LAS approach

outperform the state-of-the-art models built with historical data for large-scale microservice applications on multi-node clusters.

This chapter is the research work submitted to IEEE on Software Engineering titled ”Proactive Identification of Performance Models for Cloud Consolidation and Co-location” and extends from the research work presented in Chapter 3.

The remainder of the chapter is organized as follows. We discuss the background and current research on microservice adaptive strategies and prediction, co-location, consolidation and synthetic data generation in Section 5.1. We discuss the expanded definitions and formulations of our work in Section 5.2. We then present the expanded methodology architecture of our work in Section 5.3. We discuss the experimental setup and implementations in Section 5.4. Finally, we discuss the results of our approach in Section 5.5.

5.1 Background and Related Works

5.1.1 Cloud-Native and Microservice Adaptive Strategies

To reduce the effort of developing, maintaining and managing large and complex systems, using Self-adaptive Systems (SAS) is a popular solution used in industry and has an extensive research domain. SAS provide a wide variety of self-management, such as self-healing, self-configuration and self-optimization. With microservice architectures deployed as cloud-native applications on cloud platforms, there has been research in applying self-adaptive techniques on cloud-native applications. Filho et. al conducted a survey to systematically map self-adaptive techniques used in microservice based systems. They focus on 21 primary studies and found most self-adaptive studies for microservices focus on reactive adaptive strategies in the system infrastruc-

ture level [101]. To address the unique requirements of using self-adaptive approaches for microservice architectures, Zhang et. al propose a multi-level self-adaption model that support the service and instance layers of cloud-native applications and use basic self-adaptive operations with a descriptive language to apply changes [102]. There have been multiple self-adaptive frameworks and architectures for cloud-native applications and managing microservices [103] [104]. The MAPE-K loop has been implemented for the self-adaptive and autonomic systems of large scale microservice applications to improve scalability, availability and reliability [105] [106] [107]. For self-adaptive systems, research has been found in a wide range of the self-* properties that utilizes microservice architectures, including microservice self-healing [108, 109] and self-protection [110] [111] [107].

5.1.2 Techniques on Co-location, Consolidation and Provisioning

The cloud-native application can experience frequent changes in its cloud environment. In previous research, Co-location-as-a-service (CLaaS) is defined as the sharing of computing resources of public cloud providers by multiple customers [99] [112]. Early research explores the co-location of multiple simultaneous applications on the same instance(s) to enables better resource optimization and cost saving [99] [112] [113]. More recent research examines more specific challenges when co-locating cloud applications [114]. Chen et. al explores the interference impact of different co-located workloads on the same server at the hardware and application level, and finds the recommended and non-recommended workload patterns for deployment [115]. Co-location has also been managed by self-adaptive systems, Ashraf et. al use feedback control algorithms to deploy and scale of multiple simultaneous applications per virtual machine [113].

Co-location techniques and frameworks have been proposed in recent research with microservice architectures and cloud-native applications. Li et. al quantifies the interference of co-located containerized service by using scheduling latency metrics. They then train a machine learning model to predict the interference of online workloads, and design a scheduling algorithm to minimize the interference [116]. Gao et. al proposes a method to detect interference in colocation deployments of sidecar clusters, and uses a Histogram-based Outlier Score algorithm to aggregate real-time metrics to detect interference [117]. Cao et. al use application-level telemetry information of deployed containerized service(s) to create graph-based representations which are used to create co-location policies of the application workload [118].

In addition to Service Co-Location, Service Consolidation is a redeployment strategy where instance(s) are removed, and the services are consolidated into the remaining instance(s). VM Consolidation is the replacement of multiple VM(s) into a physical machine when the data centers are at low levels of resource utilization. VM Consolidation can enhance the resource utilization and reduce energy consumption [119] [120]. Consolidation can also be seen as an optimization problem where the objective can be improving total energy consumption, operational costs and efficient resource utilization while maintaining Quality of Service (QoS) and Service Level Objectives (SLO) [121] [122]. With microservice and cloud-native applications, the research has expanded towards container consolidation. One approach is consolidating containers that are deployed directly on bare-metal cloud computing environments, often as Container as a Service, to improve the efficiency of resource utilization and energy consumption and reduce overhead [123] [124] [125]. For cloud-native support, Kubernetes has become a popular container orchestration platform. Wu et. al propose a system that adjusts that dynamically adjusts the number of nodes in a Kubernetes

cluster. Their method maintains the QoS through the use of monitor, QoS, scaling and executing modules to get the ideal number of nodes for the cluster [126].

5.1.3 Techniques on Prediction

Predicting cloud-native application performance is important for making adaptive decisions. There has been past work with predicting future performance of cloud-native applications and microservice architectures. Varela et. al build performance prediction models using probabilistic prediction to address cloud variability and use the model for a resource scheduling algorithm to keep the application within SLA (Service Level Agreement) requirements [127]. Preventing the SLA from violations in cloud has been an important research problem and there has been multiple research using prediction techniques to keep the performance within SLA requirements [128] [129] [130]. Within Cloud systems, there have been research on performance modeling and prediction techniques for microservice architectures and cloud-native applications. Bao et. al develop a performance modeling and prediction method for microservice-based applications and uses a heuristic approach to solve minimum end-to-end delay in a public cloud [131]. Stefano et. al use Prometheus monitoring instances and AIOps for decision-making of the cloud system, and using their approach to predict traffic peaks of cloud-native applications [132]. Pramesti et. al predict the response time of microservice-based applications based on a Support Vector Machine (SVR) algorithm, and develop an autoscaler that keeps the applications within the Service Level Objective [133]. For exploration techniques for prediction, Leitner et. al use micro-benchmarks (e.g. StressNG) to capture the performance behavior of the application and estimate the application performance with a linear regression model trained by using 38 metrics from 23 micro-benchmarks [134]. While most ML models are trained on

interpolation, there has been research on extrapolation techniques [135] [136] [137]. DataFarm [137] generates a large amount of training data for ML-based query optimization by analyzing the execution patterns of existing small query workloads and the distribution of input data, and estimates the labels for each data by actively learning their performance. However, there is limited research in training on extrapolation for ML models, especially for cloud systems.

5.1.4 Data Generation for Performance Models

There has been several diverse research works that explored synthetic data generation for performance modeling. Will et al. [138] explores performance modeling for large-scale data analytics workloads for improving cluster resource allocations. Their research specifies the major challenge that performance models require a large amounts of training data, and not all organizations will publicly disclose the data. Their work proposes an approach for sharing runtime metrics based on differential privacy and data synthesis (Data-Synthesizer [139]) to generate fully anonymized training data that can maintain the performance prediction accuracy of their models. Sharifisadr et. al [140] present Generative Adversarial Networks (GANs) have a lack of comparative analysis for different real-world datasets in cloud workload domain. Their research work focuses on time-series datasets of cloud workloads, and compares multiple data generation architectures, TimeGAN, RGAN, TTS- GAN, and V-GAN, to evaluate their performance. Su et. al [141] explores the use of data generation in time series forecasting models to improve it's improving the forecasting capabilities. They utilize a generative adversarial network on time series data to build their model. Yang et. Al [142] integrates Wasserstein Generative Adversarial Network (WGAN) to synthesize data to accurately forecast future changes in workload for computing platforms. Guo et al. [143]

propose a cloud-based trajectory data management technique using spatio-temporal series data model and use synthetic datasets for their evaluation of their prototype. We present the current research work that showcases the benefits of using synthetic data generation for performance models and challenges of data generation techniques for time series datasets for performance models.

5.2 Motivation and Problem Formulation

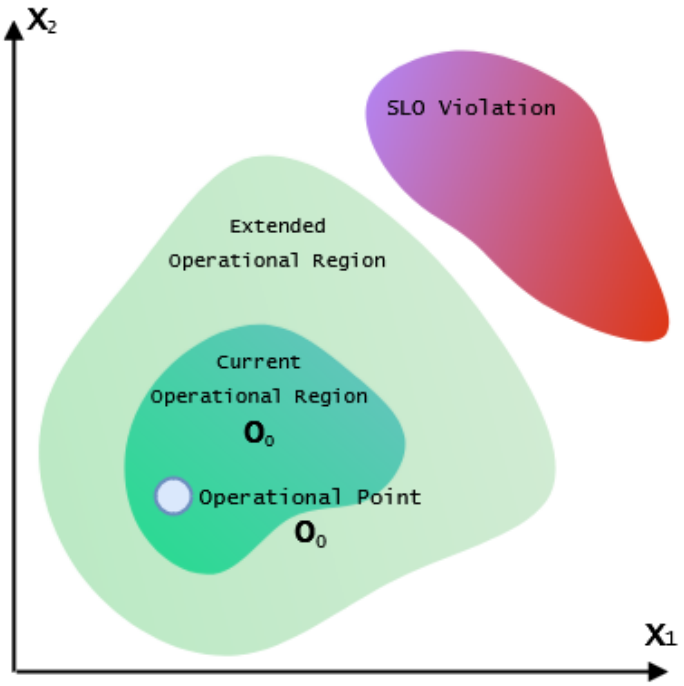

Figure 25: Current Operational Regions within the Extended Operational Region

In this section, we describe the concepts and definitions behind the problem to build a more robust performance model of a cloud environment and application. Assume that an application runs on a VM instance(s) in a cloud environment. We define the *Operational Point* as a time snapshot of the application and it's 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 arrival rate(s) of the application, λ. Due to the variability of the cloud, the production application and the change in workload, multiple *Operational Points* form an *Operational Region*.

5.2.1 Prediction Models

Performance models, M, are built by measuring the (Env, C, P, W) inputs and outputs of the Operational Region. At any point in the operational region, as the system runs normally, we collect data and train prediction models to evaluate any future performance impact. A model is trained with the observed performance metrics, P, of the production application.

5.2.2 Current Operational Point and Operational Region O_0

The *Current Operational Point* O_0 is the operational point of the application at its steady normal behavior. We observe application utilization, workload, and end-user metrics as the application functions during day-to-day operations. Changes in workload and resource utilization for application performance during normal operations result in multiple O_0 forming *Current Operational Region*. This can be the result of cloud and application fluctuations, but more significantly changes in workload's arrival rate λ. We assume that the production application has a set of arrival rates for its workload W:

$$\lambda_{app} = [0, \lambda_{max}] \tag{5.1}$$

where the arrival rate λ_{app} of the production application is between zero and the maximum arrival rate λ_{max}. The production application at O_0 will have an arrival rate from 0 to λ_{max} during current observed operational behaviour. A model, $M(O_0)$, that learns the current operational region will be evidence-based and tested during actual observed λ_{app}.

5.2.3 Extended Operational Region O_e

The *Extended Operational Region* is a region where the production application might move due to an increase in workloads λ, which have not been observed and tested in the environment *Env* so far. The region is a continuous extension of the operational region. Proactively sampling data in the extended region, O_e, can build a valid model in that region and provide the assurance that an increase in the workload will not break SLOs.

5.2.4 Target Operational Point and Region O_t

The *Target Operational Region* O_t is the unexplored operational region of cloud environment caused by a structural change in the cloud environment. The cloud environment moves from the current operational point O_0 to the unexplored target operational point, O_t when a change occurs. These changes include deploying new services, co-locating applications, scaling VMs and containers, consolidating resources, etc. The $M(O_t)$ has not been learned or tested in the target operational region. Although the target operational region can share some space with the extended operational region,

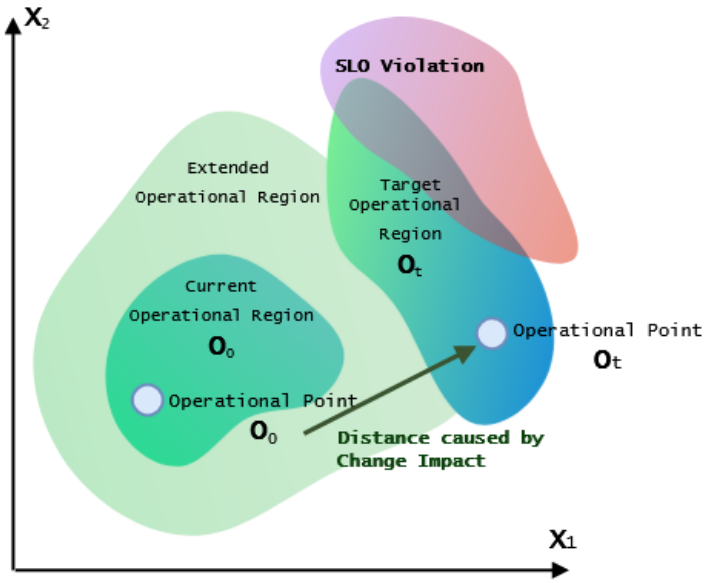

Figure 26: Target Operational Point and Region

in general, it is farther away from the operational region. Moving the cloud environment in a target operational region might have negative implications on Service Level Objectives (SLO) if not done with care.

5.2.5 Change Impact Distance *DI*

The change distance *DI* between the current operational point, O_0 and an operational points outside O_0, such as O_t and O_e is the difference between the current and target utilization.

5.2.6 Problem Formulation

A Current Performance Model, $M(O_0)$, is a model trained in (O_0). When we use the current model to predict in (O_0), we represent the model as $M_(O_0,O_0)$. $M_(O_0,O_t)$, denotes a a model trained at O_0 but used to predict performance at O_t.

To build a more robust model, we need to pro-actively explore the Extended Oper-

ation Region and the Target Operational Region to build a more complete and robust model of the cloud application. We represent this target model as a function $M_(O_t, O_t)$, where the model is trained at O_t to predict performance at O_t. We call this model as the Target Performance Model $M(O_t)$. To explore the target or the extended operational regions we propose inducing load through a set of artificial services that affect the shared resources' utilization. The load will change the utilization of the shared resources by:

$$\Delta U_i = \Delta\lambda_i * D_i \tag{5.2}$$

where i denotes a resource shared by the application, $\Delta\lambda_i$ is the induced change in the arrival rate of the artificial service and the D_i is the demand of the artificial service. To emulate moving the cloud environment to an extended or target operational zone, we propose to use the Look-Ahead Scanner (LAS) approach introduced by us earlier [100]. LAS performs load perturbations on the cloud environment using model (2) and explores the extended and target operational regions by collecting data around unobserved $\Delta\lambda$ through small ΔU_i on the DI and through large ΔU_i on the DI for O_t. Data collected by LAS approach is used to train and build a valid and tested model for the extended and target operational regions, called a Change Impact Prediction (CIP) model. The CIP model can be used to predict the impact on the environment and SLO before consolidation or co-location operations are executed. The second component of our approach is a Provision Controller that uses the information from the CIP model to dynamically execute decisions on the run-time environment. The Provision Controller can scale, co-locate and consolidate services on the environment, and by using the predicted results from the model, keeps the environment within the SLO.

5.3 Methodology

In this section, we discuss the methodology on using the Look-Ahead scanner and Provision Controller to train a valid model for extended and target operational regions, and use the change prediction results to pro-actively provision on a cloud environment.

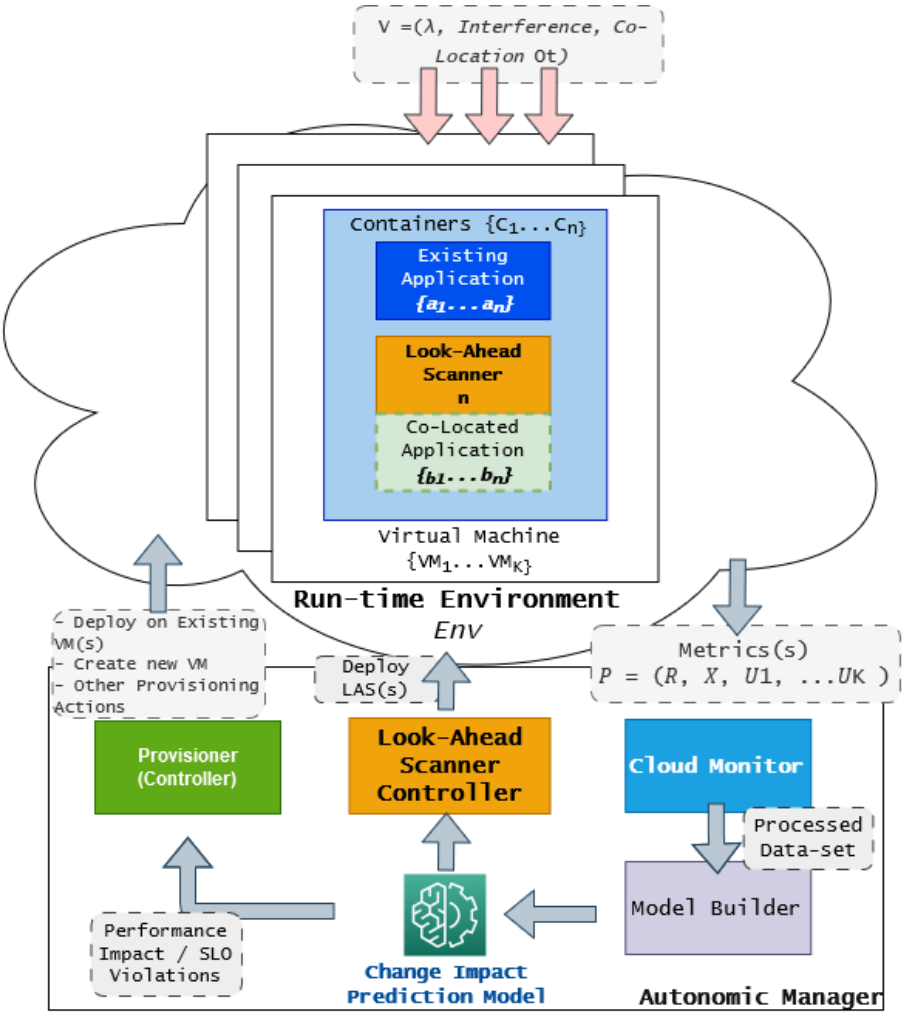

Figure 27: Overview of Look-Ahead Scanner in Production

5.3.1 Methodology and Architecture Overview

The architecture and the key components are presented in figure 2. The Look-Ahead Scanner Controller deploys and manages the LAS(s) in the Run-Time Environment to explore the operational regions by inducing ΔU_i in Env. The Performance Metric(s) of the extended and target operational regions are then collected by the Cloud Monitor to be used by the Model Builder to train the new Change Impact Prediction Model, $M(O_t)$. The Look-Ahead Scanner Controller then uses the predictions of the Model to prevent any SLO violations while continuing to explore the operational regions. When a deployment or scaling decision is required, the Provision Controller uses the prediction results of the model to check the performance impact and SLO violations and executes the appropriate action on the Run-Time Environment.

5.3.2 Look-Ahead Scanner

The Look-Ahead scanner (LAS) is a light-weight containerized service that is deployed on the VM(s) of the cloud environment to explore the operational regions through CPU, memory and I/O perturbations. As Cloud-native applications are deployed as a set of containers C in the cloud environment Env, we can use containerization to inject the LAS into the Env. As a containerized service, the LAS can easily be deployed and managed by the LAS Controller independent of the existing application(s).

The LAS Controller instructs and manages the LAS(s) deployed on the Env for single VM or multi-node cluster deployments, $VM_1 ... VM_k$. The LAS controller sends instructions to each of the LAS to induce ΔU_i to Env. By incorporating the LAS Controller, we orchestrate multiple LAS(s) together on a multi-node cluster to mimic the λ changes and the performance of co-location and consolidation. The LAS has three states that are managed by the LAS Controller: Deployed, Active and Destroyed. The

LAS is first $deployed$ on $VM_1 ... VM_k$ and awaits instructions from the LAS Controller. The LAS Controller then instructs the LAS(s) on which resources to stress for what length of time and in what increments to increase the stress to explore the operational region. The LAS is $active$ when it begins to induce stress on the resources of the Env. Depending on the current workload and utilization of the production application, the LAS will periodically change to the $active$ state to produce performance metrics around the operational regions to be used later for model training. When the LAS has completed its perturbations and needs to be removed from Env, the LAS(s) can be $destroyed$ by the LAS controller.

We use the LAS to explore two types of operational regions; the extended operational region and the target operational region. The extended operational region is affected by unobserved but realistic λ changes, the DI between the current and extended operational region is small. Therefore, the LAS will need to induce small ΔU_i changes to explore the extended operational region. We move the environment to the target operational region when co-locating a new application and consolidating the cloud environment. This has a more significant effect on the cloud environment, and the DI between the current and target operational region is large. Therefore, the LAS will need to induce larger ΔU_i to explore the target operational region.

5.3.3 LAS Methodology

Exploring Operational Regions

The LAS explores the operational regions and generates new data-points to build a prediction model. To explore both the extended and target operational regions, we need to be able to move along the DI. We can define the DI as the difference between current utilization and target utilization:

$$[U_1^t ... U_K^t]^T = [U_1^0 ... U_K^0]^T + DI \tag{5.3}$$

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 an unexplored operational region. In addition, we need to be aware of the SLO such that we do not cause any violations while we induce utilization load with the LAS. The SLO is the maximum response time that the production application will not exceed while the LAS is running:

$$SLO = R_{Max} \tag{5.4}$$

Algorithm 2 Exploration Model (CU, DI, SLO)

1: Define $CU = \{1 \ldots CU\}$
2: Define $DI = [DI_1..DI_{CU}]$
3: **for** each node cu in CU **do**
4: Start LAS(s) on node cu
5: Add increments ΔU load to cu along $F \mid DI$, where F is determined experimentally
6: Read $P_m = (R_m, X_m, U_{m,1} ... U_{m,n})$
7: Add Data P_m to DS
8: **if** $R_m >$ SLO **then**
9: Stop LAS(s) and EXIT
10: **end if**
11: **end for**
12: Fit prediction model $SLO_{pred} = f(DS, DI)$
13: **if** SLO $> SLO_{pred}$ **then**
14: **return** TRUE
15: **else**
16: **return** FALSE
17: **end if**

Algorithm 2 executes the exploration of the LAS towards the target operational regions through the LAS Controller. We first define CU, which is the set of current nodes in the cloud environment. These nodes can be the VMs that make up the cluster. We also input DI, the vector of change distances that will move each of the nodes in CU to

a target operational point. For each of the nodes, *cu* in CU, the LAS Controller activate the LAS (Line 4). The LAS will add ΔU load to node *cu* incrementally following a Rolling Hill algorithm. Using LAS to induce the load, we are unable to explore the whole of *DI*, as it can violate *SLO* or take a long time to increase the utilization of the environment. As such, the LAS does explores along F, which is a factor of *DI* (Line 5). F is determined experimentally based on length of time the LAS will run and the effect on the response time of the environment. In each increments ΔU, the LAS controller collects the measured performance metrics, P_m, which contains the measured response time, (R_m, the measured throughput X_m, and the measured utilization metrics $U_{m,1}...U_{m,n}$) (Line 6). The measured performance are added to the dataset, *DS*, (Line 7) which will be used to train the prediction model once the LAS exploration is complete. If the measured response time (R_m is less than the SLO, we can continue exploring F and adding increments ΔU with the LAS. When the LAS has completed inducing load along the F, we can stop and remove the *LAS*. However, if the measured response time (R_m is greater than the SLO, we stop the LAS (Line 8 - 9) before completing the load along F and remove it from all nodes. We then train the prediction model with the newly collected data-points in the dataset, *DS*. We fit the prediction model, a function of dataset *DS* and distance *DI*, to predict the response time, SLO_{pred}, at the target operational region, O_t, which is the remaining *DI* the *LAS* did not explore. If SLO_{pred} is less than *SLO*, the algorithm returns **True** as there will be no violations at the new operational regions. If SLO_{pred} is greater than *SLO*, the algorithm returns **False**, meaning that there is a major performance change impact on the environment and we are violating the SLO.

For the extended operational region, the LAS will need to capture the unobserved λ changes to train the model to be valid for that region. The *DI* will be derived from

an estimated workload change that the production application will experience based on future projections. Since the distance between the current operational region and the extended operational region is small, the length of the *DI* can be explored with smaller ΔU values. For the target operational region, the LAS will need to capture deployments of co-located applications and major environment changes to train a model to be valid for the target operational region. The ΔU will have to cover a larger area of *DI*. The incremental increase of ΔU will mimic the performance profile of the co-located application. Since the *DI* between the current and target operational region is large, the LAS cannot violate the SLO during its exploration.

Exploring Consolidation

In contrast to co-location expansion on the cloud environment, we can also use the LAS exploration methodology to build a model to explore service consolidation. Service Consolidation occurs when we need to scale down the environment by removing some of the VMs or cluster nodes, and move the services to the remaining nodes. Cost requirements and low levels of resource utilization on VMs are reasons for removal and consolidation.

To begin describing the method for consolidation, we start with CU, the set of all the current nodes cu. We define D as the set of nodes to be removed, d. We define T as the set of target nodes t, the nodes that all the services will be consolidated into from D. Figure 28 shows an example of the consolidation in process. There are currently 3 VMs in the environment which are the nodes of CU. In each of the VMs, there are a set of containers from c_1 to c_5. We select the target nodes, T, which are VM_1 and VM_2, and the nodes to be removed, D, which is VM_3. When consolidation is performed, we are left with the target nodes T, where the service, c_5 from D has been consolidated

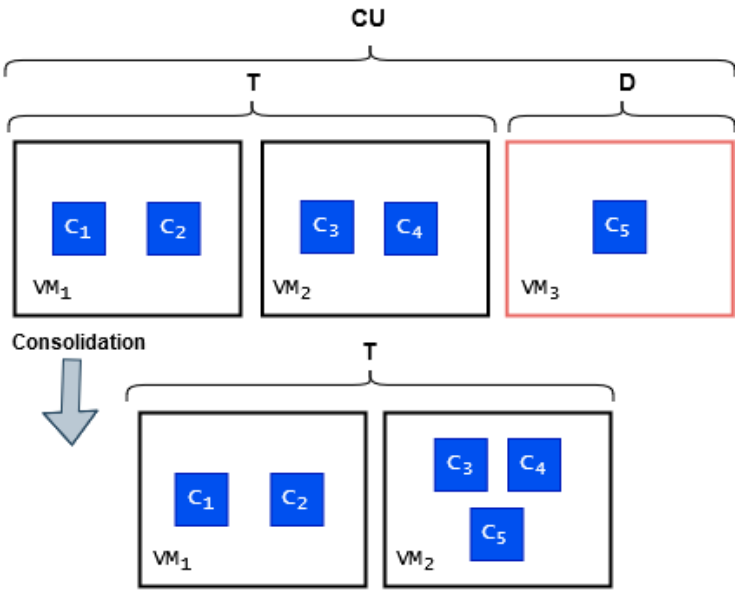

Figure 28: Consolidation of Current Nodes to Target Nodes Example

into one of the nodes in T.

However, we do not know the change impact of consolidation and if there will be any negative effects on the cloud environment. Consolidating a large amount of containers onto the remaining target nodes or selecting too many nodes to be removed can cause performance saturation and cause a violation of the *SLO*. Therefore, it is important to have a method to use the LAS approach to explore service consolidation.

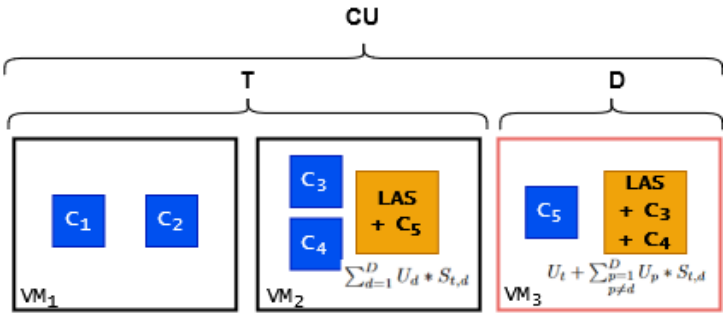

Figure 29: Look-Ahead Scanner for Consolidation

Figure 29 visualizes how the LAS will act as a proxy of the consolidation of services in the T and D nodes when inducing load. We deploy the LAS on the T, where the services from d nodes will move to, and the D. The LAS will need to induce the load across distance DI on the nodes that will let us simulate consolidation when collecting data-points for our dataset. Therefore, we need to determine the distance vectors for the nodes in D and T to set the DI.

Algorithm 3 Consolidation Model (D, T, $S_{t,d}$, DI, SLO)

1: Define $D = \{1 \dots D\}$
2: Define $T = \{1 \dots T\}$
3: Define $DI = []$
-
4: Define $S_{\substack{t,d \\ t\in T \\ d\in D}} = \begin{cases} 1, & \text{if services from node } d \text{ move to node } t. \\ 0, & \text{otherwise.} \end{cases}$
-
5: **for** each node t in T **do**
6: $\quad DI_t = \sum_{d=1}^{D} U_d * S_{t,d}$
7: **end for**
8: **for** each node d in D **do**
9: $\quad DI_d = U_t + \sum_{\substack{p=1 \\ p\neq d}}^{D} U_p * S_{t,d}$
10: **end for**
11: $DI = [DI_t, DI_d]$
12: **return** $ExplorationModel(T \cup D, DI, SLO)$

Algorithm 3 presents the method to obtain the DI for the LAS to explore service consolidation. We define the set of nodes that will be removed, D, and the set of target nodes, T, which are the nodes that will remain after consolidation. We define DI as a set of empty vectors which we have to construct in the algorithm (Line 3). We define the service to be consolidated, $S_{t,d}$ where in each node t in T and in each node d in D, we select the services that will move from node d to node t (Line 4). With the nodes and services defined, we begin the process to construct the DI. For each node t in T, we get the sum of utilization of d nodes, U_d, for the services, $S_{t,d}$ to construct DI_t (Line 6). Since the consolidated services will be deployed from the d nodes to the t nodes,

the DI_t is the distance from the current utilization to the utilization of the consolidated services from each of the nodes d. When the LAS is to be active on the T nodes, it will explore along this distance DI_t. Next for each node in d in D, we construct the DI_d. We get sum of the utilization of all other services moved to t, U_p, and add the current existing utilization of the t nodes, U_t, to get the DI_d. We add DI_t for the target nodes and DI_d for the removed nodes to construct the DI for the LAS to explore the T and D nodes. With the new DI, we call the Exploration Model from Algorithm 2. We input the union of $\mathbf{T} \cup \mathbf{D}$ which equals the current nodes, CU, for the model, the new DI and the SLO. The exploration model returns true if the consolidation is valid, or false if the consolidation will cause a violation of the SLO.

5.3.4 Model Builder and Change Impact Prediction Model

We use performance metrics of the Run-Time Cloud Environment collected by a Cloud Monitor to build a run-time model to predict performance. The Cloud Monitor collects the utilization metrics of the VM(s), the container(s) and the application's end-user metrics such as response time and throughput. The model builder uses the processed dataset of the performance metrics to train a Change Impact Prediction Model that can be used to explore the operational regions of the production application environment and predict performance impact. With only the performance metrics from the observed normal behavior, we can build a performance model at the current operational point. By collecting performance metrics while the LAS is running, we can train the performance impact model with new unobserved data.

We are able to train a valid model for the extended operational region, where we experience change in λ or lower utilization than the current operational region. We are able to train a valid model for the target operational region, where we experience

a change in the run-time environment, such as deploying a co-located application, and causing a large change in ΔU_i on the run-time environment.

In this chapter, we evaluate different performance-specific Machine Learning (ML) models for the Change Impact Prediction. The features used for training are the performance metrics U_l, and the predicted metric is the response time of the production application R_a.

$$R_a = f(U_1..U_K) \tag{5.5}$$

We use an AutoML framework, a popular solution used by many organizations to train well-performing ML models. Several ML models part of the AutoML suite are trained on the same dataset, ranked for performance, and outputs the best ML model.

5.3.5 Provision Controller

When we need to deploy or make new changes in the cloud environment based on the output of the Change Impact Prediction model, we incorporate a Provisioner component for the execution phase of the methodology. Using the output of a model that is now valid for extended and target operational phase, we can apply decision-making that moves the production application to O_t without violating SLO. The Provision Controller has a set of actions that it is able to perform with the results of the Change Impact Prediction Model. The actions can be applied manually or automatically through a set of rules. For an autonomous approach, the Provisioner Controller will make a decision based on the LAS Exploration Scenarios (I) Change In Workload, (II) Co-Location, and (III) Service Consolidation.

We can also use the State Rule Engine discussed in Chapter 2. The State Rule Engine can compare the predicted response time from the LAS-based model with the

SLO thresholds and apply co-location, consolidation and scaling in a proactive manner.

Provision Controller Algorithm

We describe how the Provision Controller will apply adaptive changes during run-time on the cloud environment through the autonomic manager. The motivation for a adaptive and autonomous approach is to keep the production application and environment under the SLO threshold as efficiently and effectively as possible.

Algorithm 4 Provisioner Algorithm (Action, CU, D, T, DI, SLO, Λ_{pred})

```
1:  if Action = ChangeInWorkload then
2:     if Λ_pred > Λ_max then
3:        if ExplorationModel(CU, DI, SLO) = true then
4:           Scale-up production application
5:        end if
6:     end if
7:  end if
8:  if Action = Co − location then
9:     if ExplorationModel(CU, DI, SLO) = true then
10:       Co-locate new application
11:    end if
12: end if
13: if Action = Consolidation then
14:    while ConsolidationModel(D, T, S_t,d, DI, SLO) = true do
15:       D = D + {LeastLoadedNode in T }
16:       T = T - {LeastLoadedNode in T }
17:    end while
18:    T = T + {LastLoadedNode in D }
19:    D = D - {LastLoadedNode in D }
20:    Consolidate containers from D to T
21: end if
```

The process of the Provision Controller is described in Algorithm 4. The Action, along with the corresponding CU, D, T, DI, SLO, and predicted arrival rate, Λ_{pred}, are the parameters to Algorithm 4. The Action is first selected for the Provision Controller based on what operational region that the cloud environment will move towards. These can be the *ChangeInWorkload* for the extended operational region, or $Co-location$ and *Consolidation* for the target operational region (Line 1, 8, 13).

The first scenario is when we have a change in workload from Line 1 to Line 7. The incoming predicted arrival rate, Λ_{pred}, is provided by an external source, e.g. historical workload patterns or a predictive model. We compare the Λ_{pred} with the Λ_{max} of the application, and if the Λ_{pred} is surpasses the Λ_{max}, we call the Algorithm 2 for exploration. The $ExplorationModel$ takes the current number of VM nodes, CU, the distance between the current and extended operational regions, DI and the SLO. If the $ExplorationModel$ from Algorithm 2 outputs **true**, the predicted SLO from the prediction model is less than the SLO and not causing any violations. The Provision Controller then can scale-up the production application (Line 4). If we need to co-locate a new application alongside with the production application, $Co-location$ will be the Action parameter. Similar to $ChangeInWorkload$ action, we call the Algorithm 2 for exploration. The DI parameter will be larger for this scenario, as co-location has a larger distance from the current operational region. If the model returns $true$, then co-locating a new application will be within the SLO requirements. If the action is selected to be consolidation, we call Algorithm 3 (Line 14). Finding the services on nodes D to move to nodes T with the best overall performance becomes an optimization problem that is not in the scope of our paper. We use a simple heuristic method from Line 15 - 19 to demonstrate selecting the D and T nodes for Algorithm 3. However, other optimization algorithms can be used in the place of Line 15 - 19 to select the D, T and $S_{t,d}$. The simple heuristic method checks what the $ConsolidationModel$ returns in a while loop. If **true**, then the Consolidation is within the SLO. We then take the $LeastLoadedNode$, the node with the lowest utilization, in T and move it to D (Line 15, 16). We call the $ConsolidationModel$ again with the parameters of the modified D and T. This process is repeated until the $ConsolidationModel$ returns **false**, and the consolidation is no longer within SLO. The last instance when the $ConsolidationModel$ returns **true**

provides us with all the D nodes and T nodes to be consolidated without violating the SLO. We return the T and D to the last state when the *ConsolidationModel* returned **true** (Line 18, 19). We then consolidate all the containers from D to T (Line 20). This heuristic approach shows how to find the D and T.

5.3.6 Synthetic Data Generation

Synthetic data generation is an important technique that has been used to augment datasets where data availability is a major challenge. Data availability and accessibility for large amounts of performance data can be a major issue due to an organization's security, privacy and the cost of generating new data. Performance models require large amounts of varying data to perform accurately and there are situations where we are unable to run our LAS on the production environment for longer periods of time at higher utilization. We look towards synthetic data generation based off the LAS datasets to assist with producing datasets for performance models.

Generative Adversarial Networks (GAN) is a machine learning solution that is used for data generation. The GAN is composed of two parts, a (1) Generator and a (2) discriminator. The generator learns to generate plausible data, and the discriminator learns to distinguish the generator's false data from the real data, and penalizes the generator for producing false generations [144]. GANs have been used for image generation, however, recent research works have demonstrated that they can be used for tabular data, such as the cloud performance datasets. Tabular Generative Adversarial Networks (TGAN) [145] takes the concept of GANs and applies it to generate tabular data. The generator for TGAN uses a long-short- term memory (LSTM) network as the generator and use Multi-Layer Perceptron (MLP) in the discriminator.

With the LAS methodology, we input the data created by the Exploration and Con-

solidation algorithm into TGAN to generate new synthetic data points at O_t. We then merge the synthetic datasets at O_t with the real datasets at O_t. Since we input the datasets at both O_0 and O_t for the machine learning models built at O_t, using the synthetic datasets allows us to change the balance of data at O_t for training without re-deploying the LAS on the production environment.

5.4 Experimental Validation

In this section, we describe the testbed and experiment setups that answer our research questions. We run the experiments in the AWS Cloud on Amazon Elastic Kubernetes Service (EKS). Amazon EKS is a managed Kubernetes service that allows us to deploy large-scale Kubernetes clusters through Amazon EC2 or Fargate. Through EKS, we can automatically manage scalability, application availability, scheduling containers and other tasks. We deploy a four node EKS cluster that utilizes m5.large VMs running Amazon Linux 2 and Kubernetes 1.30. Each of the cluster nodes are allocated 3 VCPUs, 8 GB of memory and 20 GiB disk size. The co-location experiment runs a large-scale in-production application and a co-located application on the 4 node Amazon EKS cluster. The consolidation experiment consolidates a large-scale in-production application from a 4 node EKS cluster to a 3 node and 2 node EKS cluster. The deployment files and scripts used in the test-bed setup and experimentation are available on Github[1].

[1] https://github.com/yar-yorku/Proactive-Identification-Deployment

5.4.1 Test-Bed Setup

Methodology Implementations

The Look-Ahead Scanner was developed as a lightweight Node.JS application which interacts Stress Tools to induce load to move the operational region. We then created a docker image to containerize the LAS. For the Kubernetes and EKS deployment of the Look-Ahead Scanner, we used Daemonsets on our Kubernetes deployment files for the LAS. Daemonsets enables the ability to ensure that all the cluster nodes must have a copy of the LAS pod. When nodes are added or removed in the cluster, Kubernetes will automatically deploy or remove an LAS on that node. This allows us to automatically have an LAS deployed to probe all the nodes in the cluster.

For the Cloud Monitor, we use Prometheus to monitor each of the cluster nodes. Prometheus uses Exporters to expose performance metrics. To monitor the VM-level metrics, we deploy a NodeExporter pod and to monitor container-level metrics, we deploy a cadvisor pod on each of the nodes. We utilize Kuberenetes daemonsets for the monitoring components such that a NodeExporter and cAdvisor pod will automatically be deployed on all the cluster nodes and monitoring can continue on newly added nodes. We also deploy an exporter for the loadbalancer to export the response time, requests per seconds and other application-specific metrics. For the modeling in our experimentation, we use H20 AutoML framework to train and rank the best machine learning models for co-location and consolidation of a large-scale application.

In-Production Application

The in-production application that we use in the experimentation is Online Boutique, a large 11-service benchmark microservice e-Commerce application developed by Google. Online Boutique is deployed on a 4 node cluster EKS cluster. We follow the Google

specifications and meet the prerequisites with the deployment of Boutique [2]. This includes the resource specifications provided by Google. In addition, we use the Kubernetes deployment files provided by the creators to automate the deployment in our environment.

The Boutique deployment has 4 pods on cluster node 1, 2 and 3, and 2 pods on cluster node 4, with cluster node 4 being the least utilized. The specific pods of Boutique and their placement on the 4 node cluster can be seen in Table 5.1 in the Baseline column.

Co-locating Experiment

The co-locating application we select with Boutique is Acme-Air, a benchmark airline e-Commerce application composed of two containers, the front-end Node.JS server and the back-end MongoDB Database. Acme-Air is deployed on the EKS cluster nodes with Online Boutique. For the actual co-location deployment, the front-end container is deployed on Node 2 and the MongoDB container is deployed on Node 4.

Consolidation Experiment

We use the LAS methodology to explore service consolidation with Boutique and the EKS Cluster. The starting environment is the 4 node cluster with the same pod placement discussed above in Section 5.4.1 and shown in the Baseline column in Table 5.1. We have two different consolidation environments, (1) 4 node to 3 node cluster and (2) 4 node to a 2 node cluster. For the first environment (1), we move the pods from the least utilized 4 node cluster and consolidate the pods in a 3 node cluster. We take the pods on Node 4 and consolidate them to Node 1 and Node 2. For the second environment (2), we take the pods on Node 3 and Node 4, and consolidate them on Node 1

[2] https://github.com/GoogleCloudPlatform/microservices-demo/blob/main/docs/development-guide.md

	Baseline **4 Node EKS Cluster**	**Consolidation** **3 Node EKS Cluster**	**2 Node EKS Cluster**
Node 1	Emailservice	Emailservice	Emailservice
	Frontend	Frontend	Frontend
	Loadgenerator	Loadgenerator	Loadgenerator
	Paymentservice	Paymentservice	Paymentservice
		RecommendationService	**RecommendationService**
			Checkoutservice
			Currencyservice
Node 2	Adservice	Adservice	Adservice
	Cartservice	Cartservice	Cartservice
	Productcataloguservice	Productcataloguservice	Productcataloguservice
	Recommendationservice	Recommendationservice	Recommendationservice
		Shippingservice	**Shippingservice**
			Frontend
			Rediscart
Node 3	Checkoutservice	Checkoutservice	
	Currencyservice	Currencyservice	
	Frontend	Frontend	
	Rediscart	Rediscart	
Node 4	RecommendationService		
	Shippingservice		

Table 5.1: Boutique Consolidation Setup

and Node 2. The specific pod placement is shown in Table 5.1 on the Consolidation columns for the 3 node and 2 node consolidation.

Workload Generator

The workloads will emulate different loads and load mixes for the in-production application, the co-located application and for service consolidation. We use Locust as the workload generator for the Boutique. Locust is a python-based workload generator that allows us to set a number of users to be spawned to generate load. The higher the number of users, the higher the intensity of the workload. Each user has a collection of tasks with random choices that imitate a realistic user shopping flow. We consider three workload types for the in-production application: Light, Medium and Heavy. We set the Locust users to 25 - 100 for the Light workload, 200 - 400 users for the Medium

workload, and 600 - 100 for the Heavy workload. Boutique against the light workload is within 2 - 10 % CPU Utilization, the medium workload is from 15 - 45 % CPU Utilization and the Heavy workload is within 45 - 70 % CPU Utilization.

For the co-located application, Acme-Air, we use Httperf as the workload generator. We run a constant co-location workload for all Light, Medium and Heavy workload types. The Httperf workload is within 10 to 30 % CPU utilization with approximately a 5 % step size. We configure each workload point (e.g. the value set for the Locust number of users) to run a duration of x = 100 seconds and for N = 50 iterations so that we account for cloud variability. The value of x is configurable depending on the type of application and cloud environment. For the consolidated environment, We re-use the Light, Medium and Heavy workload to be constant for the baseline 4 node EKS cluster, and the consolidated 3 node and 2 node EKS cluster.

Data Synthesis

For the synthetic data implementation, we use TensorFlow and TabGan [146]. TensorFlow is a popular python-based machine learning platform which provides tools and libraries to build machine learning models such as neural networks. TabGan is a python library specifically for tabular datasets to build TGAN machine learning models for generating new synthetic datasets. We can adjust multiple hyperparameters of the model through TabGan. Additionally, TabGan provides other models for data generation of tabular data, such as Forest Diffusion and Language Models.

5.4.2 Experiment Set-Up

Co-location experiment. The first experiment evaluates the prediction accuracy of the change models when a new co-located application is deployed on a Kubernetes cluster

with an at-scale application a. We run the first experiment in four steps:

1. We first run the Acme-Air application in isolation and collect the 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 the Acme-Air application DI and run the LAS $ExplorationModel$ algorithm on the Kubernetes Cluster with Boutique to build prediction models $M(O_t)$.

3. We then deploy the Acme-Air Application alongside Boutique and run both the Httperf and Locust workloads simultaneously and measure the response time of Acme-Air and Boutique applications.

4. We compare the metrics predicted by the model prior to deployment with the measured metrics after the deployment

In this experiment, we compare the $M(O_0)$ and $M(O_t)$ models and evaluate the performance of the models in predicting the performance impact of an at-scale deployment. In the previous work, we evaluated the LAS on a 1VM and 2VM environment. We expand our co-location model by evaluating the model on a large-scale Kubernetes cluster and Boutique, to determine if the model can scale accordingly and handle the prediction of more complex large deployments. The $M(O_0)$ model is trained on the current operational region, O_0, and is used to predict response time across O_T. This is representative of the current state-of-the-art models trained with historical data in a model identification adaptive control system. To train the $M(O_0)$ model, we deploy Boutique and the microservice pods on the EKS four node cluster. We run the Locust workload generator against Boutique with the Light, Medium and Heavy workload types. The metrics collected from the Cloud Monitor are used to train three different

$M(O_0)$ models for each workload type to predict the response time of an environment with a co-located application at O_T.

To train the $M(O_t)$ model, we deploy an LAS on each of the nodes. Based on the performance profile of Acme-Air (cf. Step 1), Acme-Air has the highest utilization followed by MongoDB. We deploy an LAS on Node 2 to represent the frontend of Acme-Air, and an LAS on Node 2 to represent the MongoDB of Acme-Air based on the actual co-location deployment described in Section 5.4.1. The LAS on Node 2 starts at a $s1$ = 20 % CPU Stress and the LAS on Node 4 starts at a $s2$ = 10 % CPU Stress with the each LAS starting points $s1, s2...s_n$ is a configurable value based on the performance profile. We increase the perturbation by 10 % for both Node 2 and Node 4, thus $DI_{Node2} = (20, 30, ...60)$ and $DI_{Node4} = (10, 20, ...50)$. These DI values are representative of the Httperf workload for the co-located application. For the experimentation, we keep the DI values constant throughout all workload types. The DI values are configurable based on the performance profile of the application and the requirements of the experimentation. The LAS dataset is then used to build three ML models at O_t of a co-located application for the light, medium and heavy operational regions. We cover a wide range of workload intensities for industrial strength and large-scale cluster deployments to emulate diverse and extreme situations for the evaluation. We compare the performance of $M(O_t)$ model with the $M(O_0)$ model, the results are presented in Section 5.5.1.

Consolidation experiment. The second experiment evaluates the prediction accuracy of the change models when we consolidate the Kubernetes cluster with an at-scale application a from 4 nodes to 3 nodes and from 4 nodes to 2 nodes. The distance between O_0 and O_t is caused by the consolidation of Boutique, where O_0 is the starting 4 node cluster, and O_t is the new consolidated environment and it's performance impact

on Boutique. We run the second experiment in four steps:

1. We first run Boutique in isolation on the four node cluster and collect the performance profile of each Boutique pod, that is, the container utilization for different workloads.

2. We then determine the pods that will be moved from the D Nodes to the T Nodes, and build the DI based on the performance profile of the Boutique Pods in the D and T nodes, as per algorithm 3, $ConsolidationModel$.

3. We use the Consolidation DI and run the LAS algorithm on the Kubernetes Cluster with Boutique to build prediction models $M(O_t)$.

4. We then consolidate the Kubernetes cluster and run the Locust workloads and measure the response time of Boutique

5. We compare the metrics predicted by the model prior to consolidation with the measured metrics after the consolidation.

Similar to the previous experiment, we train the $M(O_0)$ model to be representative of the current state-of-the-art models trained with historical data. This is the environment and performance metrics at O_0, Boutique deployed on a 4 node EKS cluster. We run the Light, Medium, and Heavy workloads on Boutique, collect the performance metrics and build three different models for each workload type. To build the $M(O_t)$ model for consolidation, we first examine the performance profile of Boutique and capture the individual utilization of each service of Boutique at different number of users from the workload generator. We determine the services that will be moved from the D nodes to the remaining T nodes. Generally, we remove the least utilized node in the cluster and move the services from the D nodes to the remaining

T nodes. The pods that are moved can be seen in Table 5.1. As per the algorithm 3, we deploy an LAS on all four nodes in the cluster and set the DI of each LAS based on the algorithm. For the 3 node consolidation, we set the LAS perturbations for each node as $DI_{node1} = (1,1,1,1,2,3,4,5,6)$ to represent the RecommendationService, $DI_{node2} = (1,1,1,1,1,1,1,1,1)$ to represent the ShippingService. DI_t is composed of DI_{node1} and DI_{node2}. For Node 4, the $DI_{node4} = (1,2,4,7,11,14,19,25,29)$ is to represent the existing services on the T nodes that will impact the services to be moved from the D nodes, as $DI_d = DI_{node4}$.

For the 2 node consolidation, the LAS perturbations are larger. For the 2 node consolidation, we have $DI_{node1} = (1,1,2,4,7,8,12,16)$ and $DI_{node2} = (1,1,2,3,4,6,8,1011)$ representative of the services of the D nodes, DI_t. The DI_d is composed of $DI_{node3} = (1,2,4,7,11,14,19,25,29)$ and $DI_{node4} = (1,2,4,7,11,14,19,25,29)$. Each of the perturbations of the set of DI has been derived from the performance profile of Boutique services that are part of DI_t and DI_d at different number of users for the workload. We collect the LAS dataset to build three ML model for each workload type for the prediction of the 3 node consolidation O_t and three ML models for each workload type for the prediction of the 2 consolidation O_t. We evaluate the $M(O_t)$ model with the $M(O_0)$ model for consolidation in Section 5.5.2. The datasets for the Co-location and Consolidation experiments are available on Github[3].

5.5 Experimental Results

In this section, we evaluate the accuracy of the $M(O_0, O_t)$ and $M(O_t, O_t)$ performance model(s) by comparing it with large at-scale deployment for both co-location and consolidation. The goal of the evaluation is to observe the prediction accuracy of the

[3]https://github.com/yar-yorku/Proactive-Identification-Datasets/

models on Co-Location and Consolidation use cases and to evaluate scalability by using large-scale applications deployed on multi-node clusters. The addition of the co-location and consolidation algorithms and the run-time datasets at the target operations regions will improve the performance of the models for large-scale cloud applications.

We evaluated the effectiveness and accuracy of the models by using Mean Absolute Percentage Error (MAPE). We compare the $R_{measured}$ of the application a, Boutique, with the $R_{predict}$ outputted by the $M(O_0)$ and $M(O_t)$ models when we co-locate with application b, Acme-Air, and when we consolidate the cluster. 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

$$MAPE = \frac{1}{n} * \sum_{i=1}^{n} \frac{|R_{measured,i} - R_{predicted,i}|}{|R_{measured,i}|} * 100 \tag{5.6}$$

Each record i is the combination of application a and application b or the LAS with their respective workload(s), workload intensities and step size within the workload(s). We evaluate an ML models using historical data at the normal operational point, $M(O_0)$, and LAS data at the target operational point, $M(O_t)$, and compare the MAPE between the models.

5.5.1 Co-location Scenario

Figure 30 presents the MAPE box plots for models built with O_0 and O_t data at different workload types. For a large-scale at-scale deployment, we see that at the lower intensity light workload that the $M(O_0)$ model slightly outperforms the $M(O_t)$ model. At

this part of the operational region, the workload intensity has very low utilization that the introduction of a co-located application has minimum effect on the performance impact. However, we move to the medium and heavy workloads, we can observe that the $M(O_t)$ model outperforms the $M(O_0)$ model.

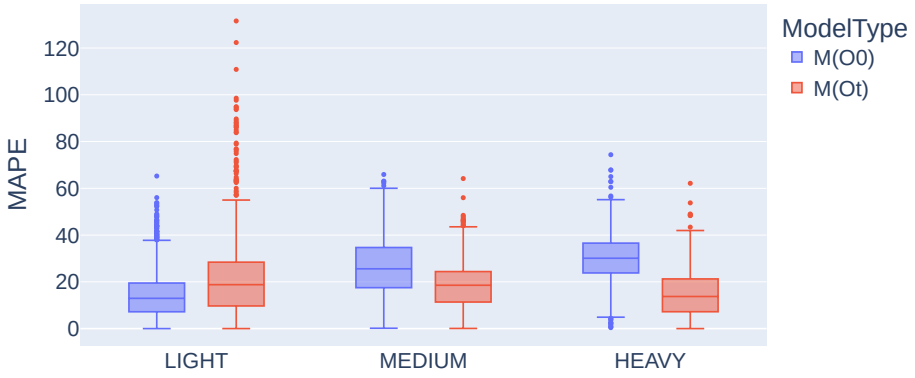

Figure 30: Prediction of Error Improvement of Co-locating Applications

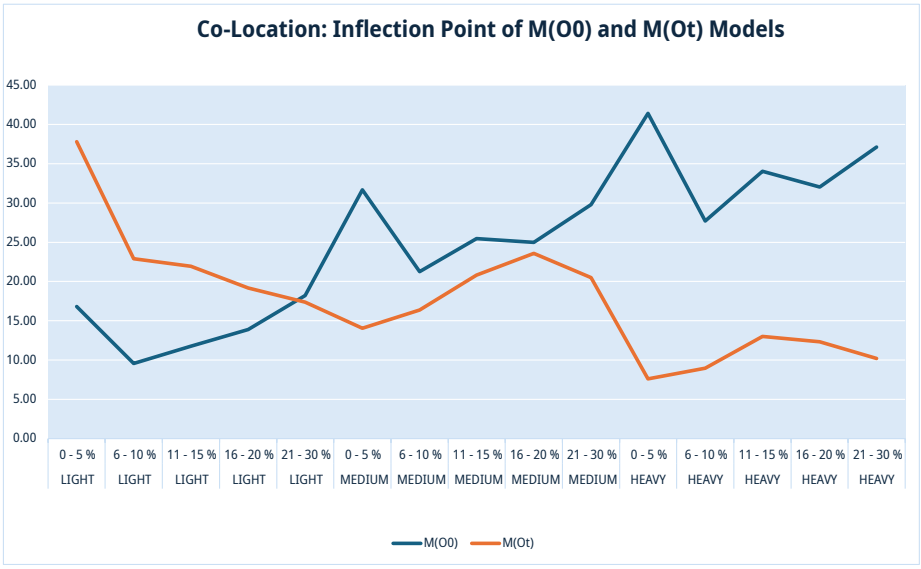

Figure 31: Inflection Point for Models of Co-locating Applications

We can see the change in the overall MAPE for $M(O_0)$ and $M(O_t)$ as we move across the workload intensity of Boutique in Figure 31. In light workloads, the $M(O_0)$

model outperforms the $M(O_t)$ model in the lowest utilization. However, we the utilization of Boutique increases, the $M(O_0)$ model performance begins to degrade while the $M(O_t)$ model performance improves. The inflection point of both models is at the light workload where the co-located application is at around 21 - 30 %. The $M(O_t)$ model starts to outperform the $M(O_0)$ model beyond this point for the different workload intensities. We can see the largest performance gap between the models during the Heavy workload, where the $M(O_t)$ model strongly outperforms the $M(O_0)$ model.

Table 5.2 presents a closer look at the MAPE as we move further away from O_0 across the distance DI towards O_t for each workload type. At the light workloads, the impact of the DI is minimal on the measured response time, causing the MAPE to be similar between the $M(O_0)$ and $M(O_t)$ models, or at times, the $M(O_0)$ outperforms the LAS-based model. We can observe during the light workload, the performance of the $M(O_t)$ model increases the further we move across the distance DI. When the utilization of the co-located application reaches 21 - 30 %, the $M(O_t)$ model begins to perform just as well as the $M(O_0)$ model. As we move to the medium and heavy workloads, the LAS model starts to outperform the $M(O_0)$ model with a greater difference. At the heavy workload, we show the most consistent improvement of MAPE where the $M(O_t)$ outperforms throughout the entire DI space. In general, the farther across the DI we need to predict the response time, the stronger the $M(O_t)$ model tends to perform compared to the $M(O_0)$ model.

			$M(O_0)$	$M(O_t)$
Workload	Utilization of Target Application	Utilization of Co-located Application	MAPE	MAPE
LIGHT	0 - 10 %	0 - 5 %	16.82	37.82
		6 - 10 %	9.59	22.89
		11 - 15 %	11.77	21.93
		16 - 20 %	13.91	19.17
		21 - 30 %	18.22	17.37
MEDIUM	15 - 40 %	0 - 5 %	31.68	14.05
		6 - 10 %	21.27	16.36
		11 - 15 %	25.46	20.83
		16 - 20 %	25.00	23.59
		21 - 30 %	29.80	20.50
HEAVY	50 - 70 %	0 - 5 %	41.39	7.62
		6 - 10 %	27.70	8.98
		11 - 15 %	34.05	13.02
		16 - 20 %	32.04	12.32
		21 - 30 %	37.12	10.23

Table 5.2: Prediction of Error Improvement of Co-locating Application on Kubernetes Cluster

RQ-1: The proactive model consistently outperforms the standard model when the distance from the current operational point passes a noticeable performance threshold seen in the Inflection Point in Figure 31. We show that it is feasible for the LAS methodology to build a proactive performance model consistently more accurate in predicting the performance for co-located at-scale deployments on large multi-node clusters. The proactive model outperforms the standard models built with historical data up to 33.77 %.

5.5.2 Consolidation Scenario

Figure 32 compares the MAPE of the models built in O_0 with the LAS models built in O_t when we consolidate the cluster from 4 nodes to 2 nodes, where we see the most significant difference between the MAPE. For the lower intensity light and medium workloads, there is not a significant difference in accuracy for predicting the response

time. The response time of Boutique is similar on both a four node and two node cluster for a lower intensity workload, as the resource utilization is the same between the environments. At higher intensity workloads, the difference in prediction error is significant. The $M(O_0)$ model struggles with predicting the response time with MAPE values over 100 %. The LAS Model can predict the response time with a significantly lower MAPE for consolidation, and outperforms the $M(O_0)$ model. Figure 33 shows the inflection point of the $M(O_0)$ and $M(O_t)$ models for consolidation. As discussed, the $M(O_0)$ and $M(O_t)$ model has similar prediction error across the light and medium workload for both 3 node and 2 node consolidation. However, as we move to the heavy workload for both 3 and 2 node consolidation, the inflection point between the model becomes clear. The $M(O_t)$ model starts outperforming and the $M(O_0)$ model degrading significantly in prediction error.

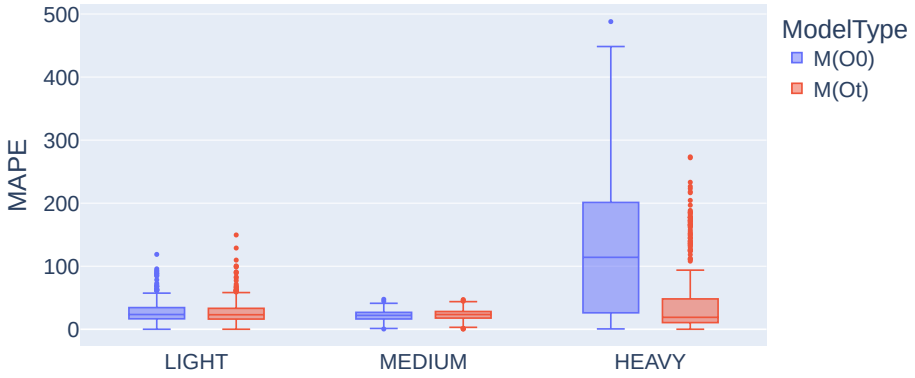

Figure 32: Prediction of Error Improvement of Consolidation Applications

Table 5.3 present a closer look at the MAPE at different workload intensities and at different node consolidations. At the light and medium workload when we consolidate 4 nodes to 2 nodes, there is almost no difference between $M(O_0)$ and $M(O_t)$ models, with only a 1 % difference at different workload iterations. During the heavy work-

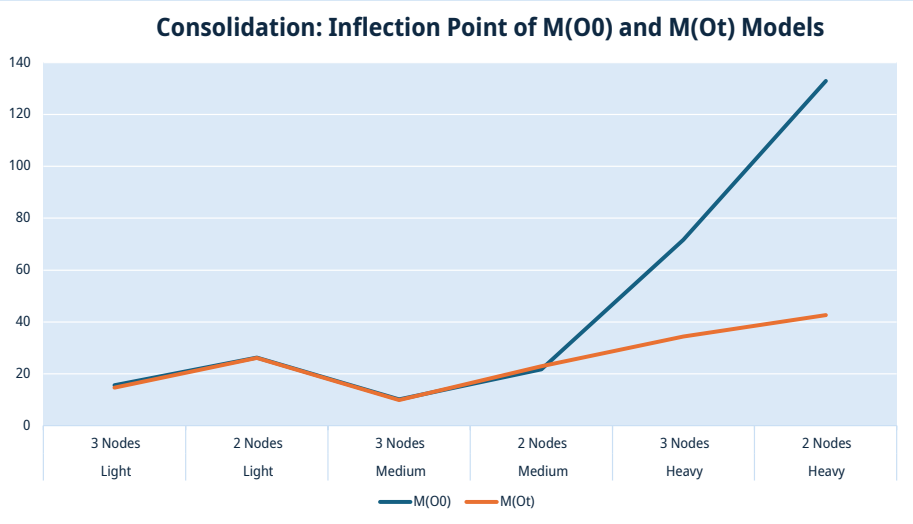

Figure 33: Inflection Point for Models of Consolidation

loads, the MAPE improves up to 67.37 % when we use the LAS-based model, $M(O_t)$. For the results of the models when we consolidate the cluster from 4 nodes to 2 nodes, there is around a 1 % difference in MAPE between the models. However, the $M(O_0)$ is unable to predict the response time with a MAPE from 61.59 % up to 204.38 %. With the LAS model, we can significantly reduce the MAPE at different iterations for the high intensity workload.

			$M(O_0)$	$M(O_t)$
Workload	Consolidation	Num of Users	MAPE	MAPE
LIGHT	3 Nodes	25	17.23	16.27
		50	15.95	14.9
		100	13.75	12.62
	2 Nodes	25	35.73	33.01
		50	24.33	25.83
		100	18.58	19.56
MEDIUM	3 Nodes	200	12.11	11.3
		300	10.9	10.69
		400	7.5	7.78
	2 Nodes	200	25.05	26.27
		300	24.26	25.64
		400	15.68	16.94
HEAVY	3 Nodes	800	86.29	18.92
		1000	56.98	49.91
	2 Nodes	600	61.59	22.23
		800	204.38	63.1

Table 5.3: Prediction of Error Improvement of Consolidation of Kubernetes Cluster

RQ-2: The experiment results shows that by utilizing the exploration and consolidation Look-Ahead Scanner approaches, we can build a more accurate performance model for consolidation. The proactive model works for at-scale deployments as we use our approach to predict the performance of consolidating large-scale Kubernetes cluster. While in light to medium intensity workloads and with a shorter distance DI traversed, the $M(O_0)$ model is similar to the performance of the $M(O_t)$. However once we run higher intensity workloads and move further across the DI, the $M(O_t)$ model significantly outperforms the standard $M(O_0)$ models (As much as 61.59 % up to 204.38 % seen in Table 5.3).

5.5.3 Data Augmentation

To improve the MAPE of the LAS-based model, $M(O_t)$, we combine synthetic data generated through a GAN model with the LAS dataset for the $M(O_t)$ model. When we generate the synthetic data, we visualize the difference between the original data and

synthetic data in Figure 34 to examine the data quality. We look at the visual similarity of the generated data compared with the actual data. While the shape is similar between the two datasets, the synthetic dataset has its utilization datapoints limited to under 50 %, while the original data has utilization data points that go beyond 50 %. Otherwise, the spread of data points between the original data and synthetic data has a similar shape.

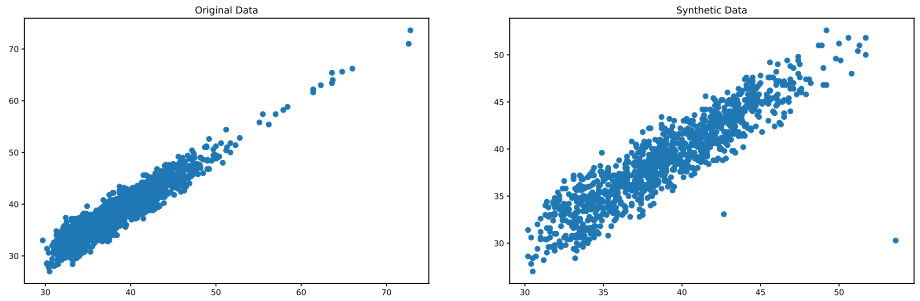

Figure 34: Synthetic LAS Dataset Quality W/ Generative Adversarial Networks (GAN)

		MO_0	MO_t	$M(O_t)$ W/ 3000 Synthetic Data Points
Workload	Num of Users	MAPE	MAPE	MAPE
Light	25	13.18	24.61	25.77
	50	14.25	22.56	20.64
	100	15.40	15.71	14.48
Medium	200	13.83	19.84	18.94
	300	30.55	19.86	19.63
	400	35.11	15.45	14.31
Heavy	600	33.79	14.97	9.20
	800	29.85	9.31	9.15
	1000	25.96	20.49	18.96

Table 5.4: Prediction of Error Improvement with Synthetic Data

Table 5.4 compare the results of the $M(O_0)$ model, the $M(O_t)$ model and the $M(O_t)$ with additional synthetic data points. We see that at lower intensity workloads, we see an improvement from 1 to 2 % between the $M(O_t)$ and $M(O_t)$ model with synthetic data. At higher intensity workloads where the response time is significantly impacted as we move within the *DI* distance, we can see the GAN synthetic data improves the

model from 1.5 % to 5.77 %. In majority of the cases, adding GAN synthetic data does not deteriorate the model and can help with the robustness of the model.

	% Fraction of DI	$M(O_t)$ MAPE
Base Response Time: 71.4 ms	0%	35.02
	5%	20.75
	10%	17.72
	20%	12.29
	40%	10.04
	60%	8.45
	80%	8.36
Max DI Response Time: 193 ms	100%	8.70

Table 5.5: Sensitivity Distance Analysis of $M(O_t)$ Model

When we build the LAS dataset, we may not be able to explore the entire *DI* to O_t due to SLO violation and performance overhead on the runtime application. It is important to know how the ML model performs when we train it with a dataset at different *DI* levels. We evaluate the sensitivity of the distance covered in LAS dataset and its effect on the prediction error in Table 5.5. We build 8 ML models at different *DI* levels. The utilization of operational region O_t is around 80 % and the maximum response time is 193 ms. We build an ML model that covers 5 % fraction of that operational region, and we repeat this with increasing % fractions, with a total of 8 ML models to evaluate the distance sensitivity. We observe when we do not use the LAS at 0 %, the prediction error is 35.02 %. When we cover 5 % of the *DI* with the LAS, the performance of the ML model begins to improve. We can see the largest leap in performance when we only explore 20 % of *DI* with the LAS. Beyond this point, we only see a small increase in performance as we move from 40 % to 100 %. This demonstrates that the LAS does not need to explore the entirety of the *DI*, and can still build robust and better performing $M(O_t)$ models. Thus, the LAS can be dynamically deployed on a runtime environment without violating SLO thresholds

or without causing large performance degradation to build a dataset at O_t for the ML models.

RQ-3: We demonstrate that we are able to augment GAN synthetic data that is representative of target operational points into existing proactive datasets for training the $M(O_t)$ performance model. The addition of synthetic data does not deteriorate the proactive model in the experimentation and improves the model from 1.5 % to 5.77 %.

5.6 Threats to Validity

Provisioner Scenario	Model	Type	Num of Samples	Avg Num of Data Points	Num of Features
Co-Location	$M(O_0)$	ML	N/A	3000 - 4000	50
	$M(O_t)$	ML	K*DI*v / ΔU	+ 1500 - 2000	50
Consolidation 3 Nodes	$M(O_0)$	ML	N/A	1500 - 2000	47
	$M(O_t)$	ML	K*DI*v / ΔU	+ 800 - 900	47
Consolidation 2 Nodes	$M(O_0)$	ML	N/A	1500 - 2000	46
	$M(O_t)$	ML	K*DI*v / ΔU	+ 600 - 700	46

Table 5.6: Complexity Comparison

An internal threat to validity is that the performance of the $M(O_t)$ is dependent on the application and cloud cluster setup. We note the complexity addition when incorporating the LAS method with a performance model. In Table 3.4, The number of samples required for the ML model can be represented by $K * DI * v$ / ΔU where we collect points for all K VMs / nodes, ΔU is the chosen increment, and to account for the variability of the cloud, the sampling can be repeated v times. In the experimentation, the LAS collected an additional 600 - 2000 data points to build the $M(O_t)$ model from the $M(O_0)$ model depending on the workload and scenario. From the metrics collected from the Cloud Monitoring tools, the ML model has a feature set of 46 - 50 attributes.

An additional internal threat to validity is that the performance of the model is also proportional to the number of LAS data points added to the base dataset. For example, we added an additional 2000 data points to the base model of 4000 data points for the co-location experimentation, which is an additional 50 % of data points for the LAS-based $M(O_t)$ model. However, in the experimentation of the sensitivity distance from Table 5.5, we can observe that the $M(O_t)$ model begins to show better accuracy at 20 % towards *DI*. For the ML $M(O_t)$ model in 20 % *DI* row, 258 data points were added to the base $M(O_0)$ model of 1023 data points, which is an additional 20 % of LAS data points. This demonstrates that the LAS approach can show good performance even at smaller proportions of LAS data points. An external threat to validity is that the 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.

5.7 Summary

In this chapter, we presented a controllable Look-Ahead Scanner (LAS) that induces perturbation signals to explore the target operational of an in-production application to proactively build predictive performance models for consolidation and co-location. We validated the approach on a large-scale Kubernetes application demonstrating the scalability of the approach. The LAS method produced an ML model that outperformed traditional ML models. Future works include discovering and testing more use cases for the LAS-approach.

Chapter 6

Conclusion

We explore the current state-of-the-art and practice of self-adaptive systems for cloud-native applications and cloud infastructures. Our findings show that there is a need to support DevOps operations by building an autonomic framework through COTS and executing adaptive actions with high-level language tools. Multi and hybrid cloud applications have become widely-used in industry, and need to be supported by MAPE-K frameworks for enabling automation. Performance modeling is an important aspect in the analysis and planning phases of a self-adaptive framework however, the run-time models used by self-adaptive managers utilize historical data and are unable to extrapolate to unexplored points for performance. There is a need to be able to predict performance changes based on cloud operations such as co-location and consolidation, and a need to classify and disambiguate performance interference for runtime self-management.

In this thesis, we proposed an industrial MAPE-K framework that can support the automation of services on the cloud. The architecture follows the MAPE-K framework, utilizing COTS and open-source services performing the Monitor, Analysis,

Planning and Execution stages to autonomously manage Multi and Hybrid cloud applications. We detailed the three main components of our framework, the Cloud Monitor, State Rule Engine and the Workflow Engine. These components interact with each other in our framework implementation based on the MAPE-K feedback loop. Each of our components in our architecture adequately satisfies the 5 requirements for a DevOps Autonomic Framework. We ran our experiments for three use-cases, Self-configuration, Self-Healing and Hybrid Cloud Scaling. We tested our framework's self-configuration ability to see if it can keep the average CPU utilization of the Acme-Web and MongoDB containers within an acceptable threshold range. We used our framework for self-healing Acme-Air by automating the redeployment of a MongoDB container when it goes down instead of redeploying MongoDB manually. The framework enables the capability of Cloud Automation Manager to allow for automatically managing services in a Hybrid Cloud setup. In the use case, we scale a MongoDB instance in EC2, while simultaneously scaling an Acme-Web container in our SAVI cloud.

We proposed a 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. 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). Our first experiment compares 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. Our second experiment evaluates 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 application.

We presented an interference identification method that generalizes across interference scenarios and using a performance model to detect interference against a large at-scale microservice application deployed on a multi-node cluster. We validated the method on four industrial strength applications and show that it scales to large deployments and works in public clouds.

We proposed a Look-Ahead Scanner Controller to dynamically inject controlled load perturbations and build a proactive performance model to explore the operational regions of consolidation and co-location for an at-scale deployment. We proposed three different algorithms for the LAS approach for CloudOps. The first algorithm is the exploration method that probes the environment for scaling up and consolidation. The second algorithm uses the LAS exploration methodology to proactively sample and model consolidation. The third algorithm is a Provisioning Controller that uses the proactive models to apply adaptive changes during run-time.

6.1 Future Works

We plan on investigating our framework for more Hybrid Clouds scenarios as various large industry partners are moving towards the domain of Hybrid Clouds. Future works include discovering and testing more use cases for the LAS-approach and applying the LAS-approach with machine learning models outside of an AutoML framework.

With the LAS-Methodology, we also need to optimize the data collection process and consider more features (e.g. i/o, memory utilization).

Bibliography

[1] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, July 2018.

[2] I. Zliobaite, A. Bifet, M. Gaber, B. Gabrys, J. Gama, L. Minku, and K. Musial, "Next challenges for adaptive learning systems," *SIGKDD Explor. Newsl.*, vol. 14, p. 48–55, Dec. 2012.

[3] 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*, ICPE '22, (New York, NY, USA), p. 125–132, Association for Computing Machinery, 2022.

[4] "Microsoft azure https://azure.microsoft.com/en-ca/overview/what-is-a-container/," [Online].

[5] "Ibm cloud https://www.ibm.com/cloud/learn/containerization," [Online].

[6] S. Shrivastava, N. Srivastav, A. Artasanchez, I. Sayed, and D. S. C. Ph.D, *AWS for Solutions Architects: The definitive guide to AWS Solutions Architecture for migrating to, building, scaling, and succeeding in the cloud*. Packt Publishing, 2023.

[7] “”vmware cloud ops https://www.vmware.com/topics/cloud-operations”,” [Online].

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

[9] “What is multicloud? https://www.redhat.com/en/topics/cloud-computing/what-is-multicloud,” [Online].

[10] “An architectural blueprint for autonomic computing,” tech. rep., IBM, June 2005.

[11] J. Kephart and D. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, pp. 41 – 50, 02 2003.

[12] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, *Engineering Self-Adaptive Systems through Feedback Loops*, pp. 48–70. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[13] T. Chen, R. Bahsoon, and X. Yao, “A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems,” *ACM Computing Surveys*, vol. 51, 06 2018.

[14] “Redhat devops https://www.redhat.com/en/topics/devops/what-is-ci-cd,” [Online].

[15] “Ibm continuous-deployment: https://www.ibm.com/cloud/learn/continuous-deployment,” [Online].

[16] D. Weyns, *Basic Principles of Self-Adaptation and Conceptual Model*, pp. 1–15. John Wiley & Sons, Ltd, 2020.

[17] P. Arcaini, E. Riccobene, and P. Scandurra, "Modeling and analyzing mape-k feedback loops for self-adaptation," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 13–23, 2015.

[18] robvet, "What is Cloud Native? - .NET https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/definition," 12 2023.

[19] CNCF, "Cloud Native Computing Foundation https://www.cncf.io/."

[20] D. Cukier, "Devops patterns to scale web applications using cloud services," in *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, (New York, NY, USA), p. 143–152, Association for Computing Machinery, 2013.

[21] M. Guerriero, M. Ciavotta, G. P. Gibilisco, and D. Ardagna, "A model-driven devops framework for qos-aware cloud applications," in *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 345–351, 2015.

[22] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering elastic containerized cloud applications to enable devops," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 65–75, 2017.

[23] Y. Wadia, R. Gaonkar, and J. Namjoshi, "Portable autoscaler for managing multi-cloud elasticity," in *2013 International Conference on Cloud Ubiquitous Computing Emerging Technologies*, pp. 48–51, 2013.

[24] M. Miglierina, G. P. Gibilisco, D. Ardagna, and E. Di Nitto, "Model based control for multi-cloud applications," in *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, MiSE '13, p. 37–43, IEEE Press, 2013.

[25] D. Loreti and A. Ciampolini, "Mapreduce over the hybrid cloud: A novel infrastructure management policy," in *Proceedings of the 8th International Conference on Utility and Cloud Computing*, UCC '15, p. 174–178, IEEE Press, 2015.

[26] Hyejeong Kang, Jung-in Koh, Yoonhee Kim, and Jaegyoon Hahm, "A sla driven vm auto-scaling method in hybrid cloud environment," in *2013 15th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pp. 1–6, 2013.

[27] Y. Ahn, J. Choi, S. Jeong, and Y. Kim, "Auto-scaling method in hybrid cloud for scientific applications," in *The 16th Asia-Pacific Network Operations and Management Symposium*, pp. 1–4, 2014.

[28] Y. Ahn and Y. Kim, "Vm auto-scaling for workflows in hybrid cloud computing," in *2014 International Conference on Cloud and Autonomic Computing*, pp. 237–240, 2014.

[29] Y. Li and Y. Xia, "Auto-scaling web applications in hybrid cloud based on docker," in *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, pp. 75–79, 2016.

[30] A. Drumea and C. Popescu, "Finite state machines and their applications in software for industrial control," in *27th International Spring Seminar on Electronics Technology: Meeting the Challenges of Electronics Technology Progress, 2004.*, vol. 1, pp. 25–29 vol.1, 2004.

[31] J. Boyer and H. Mili, *Agile Business Rule Development: Process, Architecture, and JRules Examples*. Springer Berlin, 2014.

[32] D. Mor´n, L. M. Vaquero, and F. Gal´n, "Elastically ruling the cloud: Specifying application's behavior in federated clouds," in *2011 IEEE 4th International Conference on Cloud Computing*, pp. 89–96, 2011.

[33] Y. Rouf, J. Mukherjee, M. Fokaefs, M. Shtern, J. Le, and M. Litoiu, "Rule-based security management system for data-intensive applications," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, (USA), p. 254–263, IBM Corp., 2019.

[34] "Amazon web services," [Online].

[35] "Google cloud platform," [Online].

[36] 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, pp. 662–667, July 2011.

[37] 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*, pp. 166–177, 2016.

[38] 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), pp. 1–6, 2017.

[39] 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*, ICPE '19, (New York, NY, USA), p. 25–32, Association for Computing Machinery, 2019.

[40] 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.

[41] 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)*, pp. 261–268, 2016.

[42] 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.

[43] 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*, pp. 941–949, 2015.

[44] 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*, Middleware '14, (New York, NY, USA), p. 301–312, Association for Computing Machinery, 2014.

[45] 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)*, pp. 200–210, June 2019.

[46] 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)*, pp. 522–530, Dec 2017.

[47] 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)*, pp. 1528–1529, Dec 2019.

[48] 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)*, pp. 72–81, June 2019.

[49] T. Chen and R. Bahsoon, "Self-adaptive and online qos modeling for cloud-based software services," *IEEE Transactions on Software Engineering*, vol. 43, pp. 453–475, May 2017.

[50] A. O. Strube, D. Rexachs, and E. Luque, "Software probes: A method for quickly characterizing applications' performance on heterogeneous environ-

ments," in *2009 International Conference on Parallel Processing Workshops*, pp. 262–269, Sep. 2009.

[51] 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)*, pp. 638–645, Aug 2019.

[52] 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*, ANCS '14, (New York, NY, USA), p. 237–238, Association for Computing Machinery, 2014.

[53] 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*, pp. 2258–2266, 2009.

[54] 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.

[55] 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.

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

[57] E. LeDell and S. Poirier, "H2O AutoML: Scalable automatic machine learning," *7th ICML Workshop on Automated Machine Learning (AutoML)*, July 2020.

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

[59] S. Edelkamp and S. Schrödl, *Heuristic Search - Theory and Applications*. Academic Press, 2012.

[60] "Acme air," [Online].

[61] "Mosquitto," [Online].

[62] "Nodered," [Online].

[63] "Influxdb," [Online].

[64] P. D'silva, "Deploying acme air microservices application on red hat openshift container platform," Sep 2020.

[65] J. Bagur and T. Chung, "Data logging with mqtt, node-red, influxdb and grafana: Arduino documentation," Mar 2023.

[66] "Node.js," [Online].

[67] "Stress-ng https://kernel.ubuntu.com/ cking/stress-ng/," [Online].

[68] "Prometheus," [Online].

[69] 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.

[70] "Opera," [Online].

[71] "Jmeter," [Online].

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

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

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

[75] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, (USA), p. 219–230, USENIX Association, 2013.

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

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

[78] 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)*, pp. 1–6, IEEE, 2017.

[79] 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)*, pp. 522–530, IEEE, 2017.

[80] F. Voutsas, J. Violos, and A. Leivadeas, "Filtering alerts on cloud monitoring systems," in *2023 IEEE International Conference on Joint Cloud Computing (JCC)*, pp. 34–37, IEEE, 2023.

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

[82] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, (Republic and Canton of Geneva, CHE), p. 469–478, International World Wide Web Conferences Steering Committee, 2017.

[83] V. Horký, J. Kotrč, P. Libič, and P. Tůma, "Analysis of overhead in dynamic java performance monitoring," in *Proceedings of the 7th ACM/SPEC on Inter-*

national Conference on Performance Engineering, ICPE '16, (New York, NY, USA), p. 275–286, Association for Computing Machinery, 2016.

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

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

[86] 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*, pp. 125–132, 2022.

[87] R. Boucherie and N. van Dijk, *Queueing Networks: A Fundamental Approach*. International Series in Operations Research & Management Science, Springer US, 2011.

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

[89] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, (New York, NY, USA), p. 847–855, Association for Computing Machinery, 2013.

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

[91] "Online boutique," [Online].

[92] "Air quality monitor," [Online].

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

[94] Y. Ueda and M. Ohara, "Performance competitiveness of a statically compiled language for server-side web applications," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 13–22, IEEE, 2017.

[95] R. Gao and Z. M. Jiang, "An exploratory study on assessing the impact of environment variations on the results of load tests," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 379–390, IEEE, 2017.

[96] "Locust," [Online].

[97] D. Mosberger and T. Jin, "Httperf—a tool for measuring web server performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, p. 31–37, Dec. 1998.

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

[99] M. Firdhous, "A comprehensive taxonomy for the infrastructure as a service in cloud computing," in *2014 Fourth International Conference on Advances in Computing and Communications*, pp. 158–161, 2014.

[100] Y. Rouf, J. Mukherjee, and M. Litoiu, "Towards a robust on-line performance model identification for change impact prediction," in *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 68–78, 2023.

[101] M. Filho, E. Pimentel, W. Pereira, P. H. M. Maia, and M. I. Cortés, "Self-adaptive microservice-based systems - landscape and research opportunities," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 167–178, 2021.

[102] S. Zhang, M. Zhang, L. Ni, and P. Liu, "A multi-level self-adaptation approach for microservice systems," in *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pp. 498–502, 2019.

[103] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, "An architecture for self-managing microservices," in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, AIMC '15, (New York, NY, USA), p. 19–24, Association for Computing Machinery, 2015.

[104] M. Xu, A. N. Toosi, and R. Buyya, "A self-adaptive approach for managing applications and harnessing renewable energy for sustainable cloud computing," *IEEE Transactions on Sustainable Computing*, vol. 6, no. 4, pp. 544–558, 2021.

[105] S. R. Boyapati and C. Szabo, "Self-adaptation in microservice architectures: A case study," in *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 42–51, 2022.

[106] A. Bucchiarone, C. Guidi, I. Lanese, N. Bencomo, and J. Spillner, "A mape-k approach to autonomic microservices," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pp. 100–103, 2022.

[107] O. Iraqi and H. El Bakkali, "Immunizer: A scalable loosely-coupled self-protecting software framework using adaptive microagents and parallelized microservices," in *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pp. 24–27, 2020.

[108] Y. Wang, "Towards service discovery and autonomic version management in self-healing microservices architecture," in *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, (New York, NY, USA), p. 63–66, Association for Computing Machinery, 2019.

[109] A. Arulappan, A. Mahanti, K. Passi, T. Srinivasan, R. Naha, and G. Raja, "Dqn approach for adaptive self-healing of vnfs in cloud-native network," *IEEE Access*, vol. 12, pp. 34489–34504, 2024.

[110] P. Haindl, P. Kochberger, and M. Sveggen, "A systematic literature review of inter-service security threats and mitigation strategies in microservice architectures," *IEEE Access*, pp. 1–1, 2024.

[111] R. Alboqmi, S. Jahan, and R. F. Gamble, "Toward enabling self-protection in the service mesh of the microservice architecture," in *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pp. 133–138, 2022.

[112] V. Ishakian, R. Sweha, J. Londoño, and A. Bestavros, "Colocation as a service: Strategic and operational services for cloud colocation," in *2010 Ninth IEEE*

International Symposium on Network Computing and Applications, pp. 76–83, 2010.

[113] A. Ashraf, B. Byholm, J. Lehtinen, and I. Porres, "Feedback control algorithms to deploy and scale multiple web applications per virtual machine," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 431–438, 2012.

[114] M. Bano, U. A. Qureshi, R. N. B. Rais, M. Tufail, and A. Qayyum, "Miracle: An agile colocation platform for enabling xaas cloud architecture," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 604–610, 2019.

[115] W. Chen, K. Ye, and C.-Z. Xu, "Co-locating online workload and offline workload in the cloud: An interference analysis," 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)*, pp. 2278–2283, 2019.

[116] X. Li, L. Wen, M. Xu, and K. Ye, "An interference-aware approach for co-located container orchestration with novel metric," in *2023 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, pp. 600–607, 2023.

[117] R. Gao and K. Ye, "Interference detection of colocation workloads in cloud sidecar clusters," in *2023 International Conference on High Performance Big Data and Intelligent Systems (HDIS)*, pp. 29–33, 2023.

[118] L. Cao and P. Sharma, "Co-locating containerized workload using service mesh telemetry," in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, (New York, NY, USA), p. 168–174, Association for Computing Machinery, 2021.

[119] D. Deng, K. He, and Y. Chen, "Dynamic virtual machine consolidation for improving energy efficiency in cloud data centers," in *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*, pp. 366–370, 2016.

[120] Q. Chou, W. Fan, and J. Zhang, "A reinforcement learning model for virtual machines consolidation in cloud data center," in *2021 6th International Conference on Automation, Control and Robotics Engineering (CACRE)*, pp. 16–21, 2021.

[121] S. B. Nath, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "Green containerized service consolidation in cloud," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pp. 1–6, 2020.

[122] S. Nanda and T. J. Hacker, "Racc: Resource-aware container consolidation using a deep learning approach," in *Proceedings of the First Workshop on Machine Learning for Computing Systems*, MLCS'18, (New York, NY, USA), Association for Computing Machinery, 2018.

[123] Y. Li, Y. Xu, and X. Zhou, "Cvfcc: Cv-based framework for container consolidation in cloud data centers," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 795–796, 2021.

[124] C. Chen, K. He, and Q. Guan, "Minimum migration time selection algorithm for container consolidation," in *2018 IEEE International Conference on Information and Automation (ICIA)*, pp. 1664–1668, 2018.

[125] N. Hamdi and W. Chainbi, "A multi-weight strategy for container consolidation," in *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, pp. 11–18, 2020.

[126] Q. Wu, J. Yu, L. Lu, S. Qian, and G. Xue, "Dynamically adjusting scale of a kubernetes cluster under qos guarantee," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 193–200, 2019.

[127] S. Imai, S. Patterson, and C. A. Varela, "Cost-efficient elastic stream processing using application-agnostic performance prediction," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 604–607, 2016.

[128] A. F. M. Hani, I. V. Paputungan, M. F. H, and V. S. A, "Manifold learning in sla violation detection and prediction for cloud-based system," in *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ICC '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[129] H. Zhang, P. Li, and Z. Zhou, "Performance difference prediction in cloud services for sla-based auditing," in *2015 IEEE Symposium on Service-Oriented System Engineering*, pp. 253–258, 2015.

[130] R. Anitha and C. Vidyaraj, "Workload and sla violation prediction in cloud computing," in *2019 Third International Conference on Inventive Systems and Control (ICISC)*, pp. 582–587, 2019.

[131] 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.

[132] A. Di Stefano, A. Di Stefano, G. Morana, and D. Zito, "Prometheus and aiops for the orchestration of cloud-native applications in ananke," in *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pp. 27–32, 2021.

[133] A. A. Pramesti and A. I. Kistijantoro, "Autoscaling based on response time prediction for microservice application in kubernetes," in *2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)*, pp. 1–6, 2022.

[134] J. Scheuner and P. Leitner, "Estimating cloud application performance based on micro-benchmark profiling," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 90–97, 2018.

[135] J. Zhan, X. Xie, J. Mao, Y. Liu, J. Guo, M. Zhang, and S. Ma, "Evaluating interpolation and extrapolation performance of neural retrieval models," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, (New York, NY, USA), p. 2486–2496, Association for Computing Machinery, 2022.

[136] L. Kegel, M. Hahmann, and W. Lehner, "Generating what-if scenarios for time series data," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[137] F. Ventura, Z. Kaoudi, J. A. Quiané-Ruiz, and V. Markl, "Expand your training limits! generating training data for ml-based data management," in *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21,

(New York, NY, USA), p. 1865–1878, Association for Computing Machinery, 2021.

[138] J. Will, D. Scheinert, S. Zunzer, J. Bode, C. Kring, and L. Thamsen, "Privacy-preserving sharing of data analytics runtime metrics for performance modeling," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, (New York, NY, USA), p. 269–272, Association for Computing Machinery, 2024.

[139] H. Ping, J. Stoyanovich, and B. Howe, "Datasynthesizer: Privacy-preserving synthetic datasets," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[140] N. Sharifisadr, D. Krishnamurthy, and Y. Amannejad, "A comparative analysis of generative adversarial networks for generating cloud workloads," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pp. 279–290, 2024.

[141] M. Su, S. Du, J. Hu, and T. Li, "A generative adversarial network with attention mechanism for time series forecasting," in *2023 8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pp. 197–202, 2023.

[142] W. Yang, G. Zhu, Q. Chen, and N. Liu, "Computing power scheduling method based on long-term workload prediction for computing power platform," in *2024 6th International Conference on Communications, Information System and Computer Engineering (CISCE)*, pp. 1475–1479, May 2024.

[143] Y. Guo, Z. Wang, J. Xue, and Z. Shao, "A spatio-temporal series data model with efficient indexing and layout for cloud-based trajectory data management,"

in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 1171–1184, 2024.

[144] "Google GAN https://developers.google.com/machine-learning/gan/gan_structure," [Online].

[145] L. Xu and K. Veeramachaneni, "Synthesizing tabular data using generative adversarial networks," 2018.

[146] I. Ashrapov, "Tabular gans for uneven distribution," 2020.