School of Computing Science

Real-Time QoS Monitoring and Anomaly Detection on Microservice-based Applications in Cloud-Edge Infrastructure

Ayman Ibrahim Noor

Submitted for the degree of Doctor of Philosophy in the School of Computing Science, Newcastle University

December 2020

Abstract

Microservices have emerged as a new approach for developing and deploying cloud applications that require higher levels of agility, scale, and reliability. A microservice-based cloud application architecture advocates decomposition of monolithic application components into independent software components called "microservices". As the independent microservices can be developed, deployed, and updated independently of each other, it leads to complex run-time performance monitoring and management challenges. The deployment environment for microservices in multi-cloud environments is very complex as there are numerous components running in heterogeneous environments (VM/container) and communicating frequently with each other using REST-based/REST-less APIs. In some cases, multiple components can also be executed inside a VM/container making any failure or anomaly detection very complicated. It is necessary to monitor the performance variation of all the service components to detect any reason for failure.

Microservice and container architecture allows to design loose-coupled services and run them in a lightweight runtime environment for more efficient scaling. Thus, container-based microservice deployment is now the standard model for hosting cloud applications across industries. Despite the strongest scalability characteristic of this model which opens the doors for further optimizations in both application structure and performance, such characteristic adds an additional level of complexity to monitoring application performance. Performance monitoring system can lead to severe application outages if it is not able to successfully and quickly detecting failures and localizing their causes. Machine learning-based techniques have been applied to detect anomalies in microservice-based cloud-based applications. The existing research works used different tracking algorithms to search the root cause if anomaly observed behaviour. However, linking the observed failures of an application with their root causes by the use of these techniques is still an open research problem.

Osmotic computing is a new IoT application programming paradigm that's driven by the significant increase in resource capacity/capability at the network edge, along with support for data transfer protocols that enable such resources to interact more seamlessly with cloud-based services. Much of the difficulty in Quality of Service (QoS) and performance monitoring of IoT applications in an osmotic computing environment is due to the massive scale and heterogeneity (IoT + edge + cloud) of computing environments.

To handle monitoring and anomaly detection of microservices in cloud and edge datacenters, this thesis presents multilateral research towards monitoring and anomaly detection on microservice-based applications performance in cloud-edge infrastructure.

The key contributions of this thesis are as following:

- It introduces a novel system, **M**ulti-microservices **M**ulti-virtualization **M**ulti-cloud monitoring (*M3*) that provides a holistic approach to monitor the performance of microservice-based application stacks deployed across multiple cloud data centers.

- A framework for **M**onitoring, **A**nomaly **D**etection and **L**ocalization **S**ystem (*MADLS*) which utilizes a simplified approach that depends on commonly available metrics offering a simplified deployment environment for the developer.

- Developing a unified monitoring model for cloud-edge that provides an IoT application administrator with detailed QoS information related to microservices deployed across cloud and edge datacenters.

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or any other qualification at Newcastle University.

Ayman Ibrahim Noor

December 2020

Publications

Published

1. U. Demirbaga, Z. Wen, **A. Noor**, K. Mitra, K. Alwasel, S. Garg, A. Zomaya, and R. Ranjan, *"AutoDiagn: An Automated Real-time Diagnosis Framework for Big Data Systems,"* IEEE Transactions on Computers. April 02, 2021. [**ERA A*, ISI Impact Factor 3.131**]

2. G. S. Aujla, M. Barati, O. Rana, S. Dustdar, **A. Noor**, J. Llanos, M. Carr, D. Marikyan, S. Papagiannidis, and R. Ranjan, *"COM-PACE: Compliance-Aware Cloud Application Engineering Using Blockchain,"* Journal of IEEE Internet Computing, Volume 24, Issue 5, Nov. 06, 2020. [**ERA B, ISI Impact Factor: 2.0**]

3. A. Alqahtani, K. Alwasel, **A. Noor**, K. Mitra, E. Solaiman, and R. Ranjan, *"The Integration of Scheduling, Monitoring, and SLA in Cyber Physical Systems,"* Handbook of Integration of Cloud Computing, Cyber Physical Systems and Internet of Things, Springer, Cham, 2020.

4. U. Demirbaga, **A. Noor**, Z. Wen, P. James, K. Mitra, and R. Ranjan, *"SmartMonit: Real-time Big Data Monitoring System,"* The 38th International Symposium on Reliable Distributed Systems (SRDS 2019) Lyon, France, OCT 1-4, 2019. [**Core A Ranking**]

5. **A. Noor**, D. N. Jha, K. Mitra, P. P. Jayaraman, A. Souza, R. Ranjan, and S. Dustdar, *"A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environment,"* In 2019 IEEE 12th International Conference on Cloud Computing (Cloud), IEEE Computer Society, Milan, Italy, July 8-13, 2019. [**Core B Ranking**]

6. **A. Noor**, K. Mitra, A. Souza, D. N. Jha, P. P. Jayaraman, Umit Demirbaga,

Ellis Solaiman, Nelio Cacho, and R. Ranjan, *"Cyber-Physical Application Monitoring across Multiple Clouds,"* Journal of Computer and Electrical Engineering, Elsevier, Volume 77, July, 2019. [**ISI Impact factor: 2.1**]

7. A. Souza, N. Cacho, **A. Noor**, P. P. Jayaraman, A. Romanovsky, and R. Ranjan, *"Osmotic Monitoring of Microservices between the Edge and Cloud,"* The 20th IEEE International Conference on High Performance Computing and Communications (HPCC 2018), Exeter, United Kingdom. [**Core B Ranked**]

8. K. Alwasel, **A. Noor**, Y. Li, E. Solaiman, S. K. Garg, P. P. Jayaraman, R. Ranjan, *"Cloud Resource Scheduling, Monitoring, and Configuration Management in the Software Defined Networking Era,"* Newsletter, IEEE Technical Committee on Cybernetics for Cyber-Physical Systems, Volume 2, Issue 1, February 02, 2017.

Acknowledgements

First, and foremost, I am grateful to Allah for his blessings, the good health, and wellbeing that were necessary to complete this thesis. Praise be to Allah for what he bestowed upon me. He helped me to accomplish this work, which I consider a worship of Allah for His sake. I thank Almighty Allah for his grace and his generosity. All the praises and thanks be to Allah firstly and finally. Allah has commanded us in his great book to thank people, says "So hold that which I have given you and be of the grateful." Surah Al-A'raf:144.

From that respective, I would first like to thank my thesis supervisor Rajiv Ranjan of the chair professor in computing science and Internet of Things at Newcastle University. The door to Prof. Rajiv office was always open whenever I ran into a trouble spot or had a question about my research. He always steered and encouraged me in the right direction whenever he thought I needed it. Without him, this work would not have been completed.

I would also like to acknowledge Dr. Karan Mitra (Luleå University of Technology) as the external supervisor of my PhD, and I am gratefully indebted to his for his very valuable comments on this thesis. In addition, for his guidance, patience, and unflinching support throughout my PhD candidature, especially during hardship times.

I would also like to thank my second external supervisor, Dr. Prem Prakash Jayaraman (Swinburne University of Technology) for his guidance and unflinching support during my first year of my PhD candidature.

I would like to thank my colleagues and lab-mates: Arthur Souza, Bin Qian, Devki Nandan Jha, Duaa AlQattan, Gagangeet Aujla, Jedsada Phengsuwan, Khaled Alwasel, Nipun Balan, Umit Demirbaga, Yinhao Li, and Zhenyu Wen for their help and support.

I would like to extend my gratitude to the government of Saudi Arabia represented by Taibah University (Madinah branch) and the Royal Embassy of Saudi Arabia Cultural Bureau in London for funding my PhD. Without their support, this PhD would not have been possible.

Finally, I must express my very profound gratitude to my parents (Ibrahim Noor and Afaf Khayat). Firstly, my father (Ibrahim Noor) demonstrated what it means to care for his family giving his heart to his children and wife. Not only did it benefit the financial support of the family, but is also related to the spiritual bonds between all of us. My mother (Afaf Khayat) is the very essence of beauty to me. She is a patient, kind, giving, self-sacrificing mother who believes in me with her whole heart.

To my dear sisters (Aziza Noor and Ahlam Noor), who taught me that life without love, bonding and cooperation means nothing.

Last but not least, thank you to my wife, Morooj Samman, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without you. Thank you.

Contents

List of Figures

List of Tables

1

INTRODUCTION

Contents

Introduction

Microservice deployment is now the standard model for hosting cloud applications across industries from streaming media to Big Data [1]. Microservice approach means that an application is split into services. Each microservice is isolated from other services, but it can communicate with them. One key benefit of microservice deployment is that an application can be released faster and managed efficiently [2]. The deployment of microservices is achieved by utilizing virtualization techniques like Virtual Machines (VMs) and containers. Container virtualization technique is much popular nowadays because it consumes less resource and then makes the application more scalable [3].

Cloud DataCentre (CDC): All private, commercial and public providers providing software platform, storage platform, etc. in different form of services such as data analysis services, user interface services based on web server and DB server. Normally such cloud providers are located in different geographical locations and they are considered to have unlimited storage and processing power. These services can be accessed from most of the places of the world. This way they provide on demand service for providing such on demand service the physical datacentre resources are normally virtualized and provide service to the user as per use manner. For doing this virtualization two most commonly used methods by the cloud datacentres are containers (e.g. Docker, LXC etc.) and hypervisors (e.g. Xen, KVM, Virtual Box etc.). These techniques create multiple virtual machines (VMs or containers) within the physical machine and each of these virtual machines are kept isolated from each other. This way different jobs are allocated to different virtual machine [4–6] for providing different services that every cloud service provider different virtual machine configurations with different cost. This way massive data storage and processing activities can also be performed on this datacentre. The cost and selection of virtual machines depends on different QoS parameters e.g. processing power, performance, availability etc.

On the other hand, ”an edge datacenter can be defined as a collection of smart IoT sensors, IoT gateways (raspberry pi 3, UDOO board, esp8266, and so on), and software-defined networking devices solutions (for example, Cisco IOx, HP OpenFlow, and

Middlebox Technologies) at the network edge that can offer computing and storage capabilities on a much smaller scale than cloud datacenters” [7].

The Quality of Service (QoS) parameters, e.g., microservice components usability, microservice load, and microservice throughput are vital to consider. They can change inappropriately based on a few components, e.g., several end-users interfacing with microservice, physical resource, VM/container failure, VM/container overload, etc. In this case, QoS monitoring refer to a continuous view of the status of such parameters, which gives the whole monitoring microservice the needed responsiveness. To meet the QoS focus of cloud-hosted microservices, it is essential to track the system software and hardware resource [8]. Furthermore, these microservices enable various characteristics of the cloud provider. Monitoring in clouds is necessary to keep those services highly accessible and performance, and it is critical for both resource and customer suppliers [9–11]. Monitoring is mainly a primary method for i) management of infrastructure and hardware resources and ii) ongoing knowledge for these resources and cloud-based consumer-hosted microservices. The need for monitoring became more critical with the introduction and widespread adoption of cloud-edge environments. This modern computing model, which requires services to be bought upon demand and paid for their actual use, allows costs to be minimized by effective power utilization and resource planning, as Aceto et al. [12] highlighted. To do so, prices and real resource utilization need to be monitored so that automation can come into action.

1.1 Research Motivation

Monitoring performance is a key research activity in cloud computing, which has seen a significant rise in importance due to the popularity of microservices. Their popularity is primarily based on their lightweight, economical usage of shared resources allowing for greater fiscal reward from service hosting activities while reflecting green agendas. However, this puts significant importance on securing such services as resource misuse, often manifesting as anomalous behaviour in the network activity, can have impacts on distinct hosting provisions of unrelated services

Monitoring is a track of the system and see how the system is been performed. The

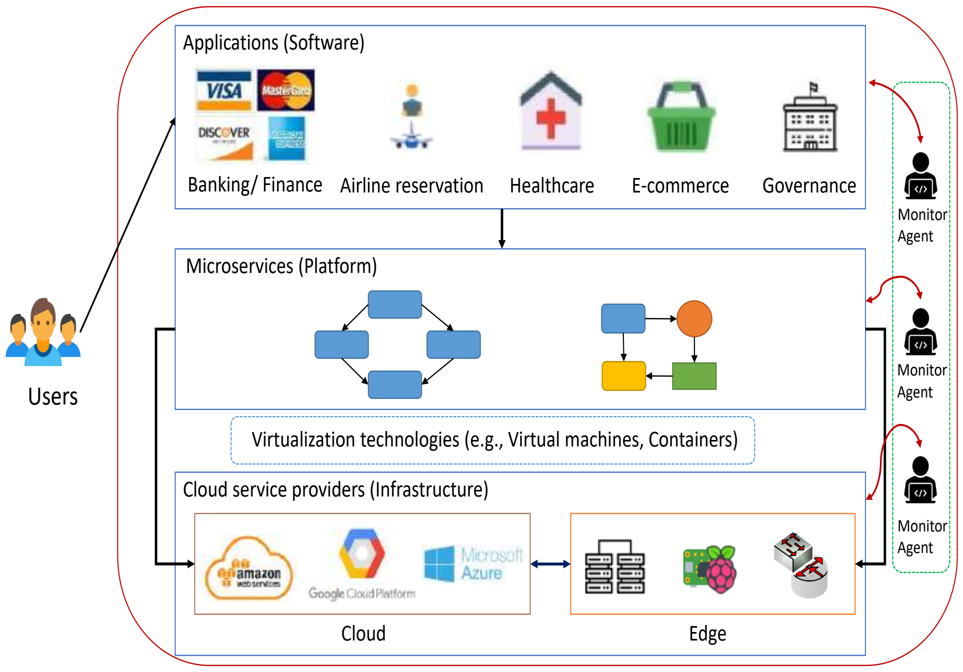


Figure 1.1: Monitoring microservice-based application distributed across cloud and edge datacentres.

monitor is responsible for the monitoring of system data gathered by various process [13]. In the context of service-based systems, Benbernou et al. [14] define the monitoring process as a process for gathering and reporting information on an application execution and evolution. The method of tracking can be used for various purposes. Examples include machine optimization of run-time, identification of essential by detecting changes in run-time, and collecting of system evolution information. It is a highly tricky challenge to track microservice-based systems as services operate in separate processes distributed across multiple hosts [15]. It includes a system of monitoring that gathers, distributes and processes data automatically in a system with a high number of components.

Monitoring [16] plays a central role in identifying "when" a certain microservice should be migrated. For migration to be effective, it is necessary to properly monitor the performance of the microservices. The monitoring of microservices in cloud and edge environment is a recent topic and therefore few works have been carried out in this regard. To explore this topic in the construction of a solution that meets the requirements of monitoring microservices that deployed in the cloud or edge environment as showing in Figure 1.1.

For the same reasons as any distributed system, applications built using the microservice architecture will have to be monitored: that is, all systems inevitably fail. The most apparent explanation why monitoring is important and the reason for that is failure, but it is not the only one. System performance is not binary; systems are not either "up or down." In a degraded state that impacts performance, complex systems, including monoliths, can run. Sometimes these deteriorated states herald impending failures. Until complete failures occur, tracking the actions of systems may alert operators to deteriorated states. Within a Service Level Agreement (SLA), facilities offered internally or to external customers are also provided. Without monitoring, it is difficult to know whether for example, the SLA is being respected or broken.

The failure is that a system cannot execute its requisite functions in compliance with defined performance criteria. Faults (or anomalies) identify an exceptional circumstance that may lead to defects that happen during the system operation. It is an expression of a system error [17]. The optimization of the microservice environment and its constant changes will impact service response times and contribute to the false of anomalies detection. The metrics will be affected by started and stopped services, as well as the continuing deployment of updated services. Besides, where the load and response times differ, it is challenging to establish thresholds. The problems related to scalability and technical diversity are simplified by containers/ VMs, processes, and frameworks. The dynamic environment and the runtime context are other fact. The components are changing or be transferred due to the container management systems and the continuous delivery model.

Scalability of a container-based microservice application allows for more application organization and optimization. But, this strongest characteristic brought new complexities to application performance monitoring. Indeed, inefficient performance monitoring may lead to severe application outages [18] as it is not able to successfully and quickly detect the failures and then localizing their causes. Application failures are observed by the user and are considered as indicators for the problems in application behaviour (also referred to as anomalies). These anomalies can be caused by different faults in the application resources and called root causes. Therefore, anomaly detection and root causes localization is aimed at linking the observed failures with the underlying faults. For instance, Central Processing Unit (CPU) consumption fault may cause a response time delay failure in the application.

Currently, there are multiple monitoring frameworks and anomalies detection have many challenges [19–29], a few of the main challenges considered in this thesis are given below:

- **The deployment environment for microservice-oriented applications in multi-cloud environments** is very complex as there are numerous components running in heterogeneous environments (VM/container) and communicating frequently with each other using REST-based/REST-less APIs. In some cases, multiple components can also be executed inside a container/VM making any failure or anomaly detection very complicated. It is necessary to monitor the performance variation of all the service components to detect any reason for failure.

- **Considering the virtualization environment,** deployment of microservice-oriented applications in containers is very different from that in VM. Containers are defined in terms of namespace and cgroups that share the same host machine whereas each VM is isolated with its own operating system. Also, the resource limitation in containers can be hard or soft as compared to VM which is always strict (hard). A soft limit allows containers to extend beyond their allocated resource limit creating higher chances of interference [30, 31]. Monitoring the performance of microservice-oriented applications in such cross VM/container scenarios is very important to ensure that services are executing in a desirable way.

- **Modern applications can be distributed across multiple cloud environments** including bare metal, public or private cloud depending on several features such as microservice-oriented application component requirements, deployment locations, security concerns, cost, etc. Different cloud providers have their own way of handling deployment and management of microservice-oriented application components. Due to the heterogeneity of cloud providers, it is complex to have holistic management of application components.

- **Application migration in cloud-edge:** Underpin such emerging approaches is the dynamic management of microservices across cloud and edge datacenters. For instance, defining when and how microservices can be migrated from edge resources to cloud-based resources (and vice versa), and characteristics which influence such migration, remains a challenge [32].

- **Infrequent Anomalies**: anomalies are not a common event within cloud environments [33]. This is problematic for those approaches based on supervised machine learning, as there are limited examples of anomalous training data from which to learn. Consequently, the main problem can be either missing anomalies that are not present in the training dataset or identifying normal behaviour as anomalies.

- **Numerous Metrics:** datasets used for supervised learning can consist of many metrics, promoting the use of a variety of different approaches based on metric availability. Such metrics may not be available generally across service providers. This produces datasets for machine learning that tend to be bespoke for given service providers, and sometimes elements of their infrastructures depending on web application domain support, rather than for general usage bringing challenges for localized detection [34].

- **The optimization of the microservice environment** and its constant changes will impact service response times and contribute to the false of anomalies detection. The metrics will be affected by started and stopped services, as well as the continuing deployment of updated services. Besides, where the load and response times differ, it is challenging to establish thresholds. The dynamic environment and the runtime context are other fact. The components are changing or be transferred due to the container management systems and the continuous delivery model.

This PhD research aims to find a monitoring and anomaly detection solution while considering the aforementioned challenges. In particular, this PhD thesis is guided by the following research questions:

- How to monitor the performance of multiple microservice-oriented applications deployed on heterogeneous virtualization platforms distributed across different cloud data centers?

- How to ubiquitously monitoring QoS of microservices mapped to an osmotic computing (cloud+edge) environment?

- How to aggregate QoS measures of microservice-oriented applications running in multiple cloud/ (cloud + edge) environments to give a holistic view of (e.g., Book-shop, highway traffic, smart parking) performance?

- How to detect, identify and locate the anomalies that causes a performance reduction in container-based microservice architecture on cloud environments?

1.2 Research Contributions

There are multiple monitoring tools available to monitor the applications running in the cloud. However, most of the frameworks are either cloud provider specific e.g. [Microsoft Azure Fabric Controller], or virtualization architecture specific e.g. [CAdvisor]. These monitoring tools are not able to satisfy the performance monitoring requirements of complex microservices deployed across multiple cloud data centers. In addition, existing QoS monitoring tools and techniques suffer from serious technical limitations when subjected to osmotic computing (cloud + edge). Furthermore, there are several microservices deployed in multiple containers. A root-cause localization of a detected anomaly can be performed by analysing resource metrics of microservice's containers. Analysis of all observed metric and underlying resourses' metrics from all microservices consists of a large volume of metrics. We use multi-resolution method to analyse the metrics in sequential stages to best formulate a learning environment for machine learning algorithms. The multi-resolution method is achieved by narrowing the metrics' scope and increasing the resolution using classification machine learning algorithms. Lightweight algorithms are used for computational purposes to analyses several representative metrics to detect the observed anomalies. Once the anomalies are detected, a comprehensive drill-down analysis of additional metrics to localize the root cause using complex algorithms. The main contributions of this thesis are as given below:

- Introducing a novel framework: **M**ulti-virtualization, **M**ulti-cloud **M**onitoring in microservice-oriented applications (*M3*) that provides a holistic approach to monitor the performance of microservice-oriented applications composed into multiple applications deployed/running in a multi-cloud and heterogeneous environment (e.g. using different virtualization technologies).

- Developing a **M**onitoring and **A**nomaly **D**etection and **L**ocalization **S**ystem (*MADLS*) which utilizes a simplified approach that depends on commonly available metrics offering a simplified deployment environment for the developer.

- Developing a unified monitoring model for osmotic computing that provides an IoT application administrator with detailed QoS information related to microservices deployed across cloud and edge.

1.3 Thesis Structure

The structure of the thesis will be presented in Figure 1.2 and the arrow here represents the flow of chapters. **Chapter 2** provides background of monitoring, virtualization, microservices, deployment environment, and discusses related work on commercial and open source monitoring tools and anomalies detection. **Chapter 3** presents a framework for monitoring microservice-oriented applications in heterogeneous virtualization environments across multiple cloud. **Chapter 4** provides MADLS: monitoring and anomaly detection and localization system of container-based microservice. **Chapter 5** presents an osmotic monitoring of microservices between the edge and cloud. Finally, **Chapter 6** concluded the thesis by summarizing the work done of the thesis and provides directions for future work.

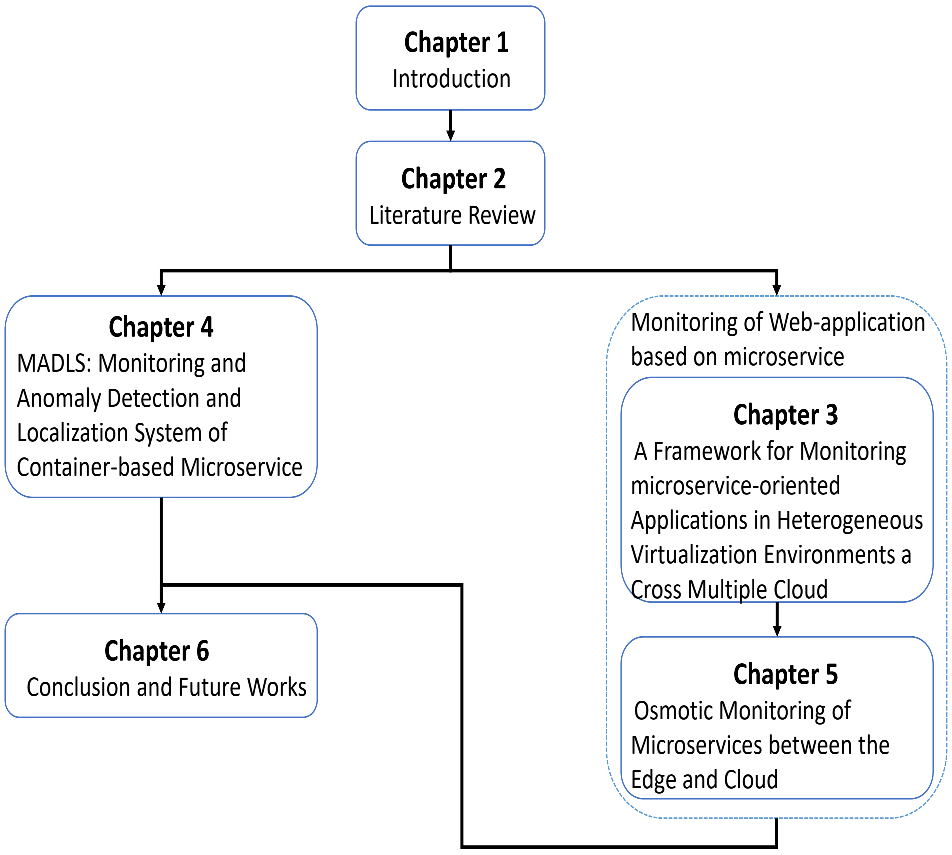

Chapter 1
Introduction
Chapter 2
Literature Review
Chapter 4
MADLS: Monitoring and Anomaly Detection and Localization System of Container-based Microservice
Monitoring of Web-application based on microservice
Chapter 3
A Framework for Monitoring microservice-oriented Applications in Heterogeneous Virtualization Environments a Cross Multiple Cloud
Chapter 5
Osmotic Monitoring of Microservices between the Edge and Cloud
Chapter 6
Conclusion and Future Works

Figure 1.2: Thesis organization.

2

Literature review

Contents

Summary

This chapter presents some background information concerning the overall topic, including a brief description on monitoring, virtualization techniques, microservices, the underlying cloud and edge computing environment, and industry and academic monitoring tools. A major focus of this thesis is to address the challenges of microservice monitoring and anomaly detection in cloud-edge infrastructure. Monitoring regulates the performance of cloud-based microservices in software and hardware resources. It includes information on the monitoring resource's status/health, such as the CPU and memory use for the microservice deployed on the cloud platform. The microservice provisioning mechanism handles the configuration and implementation of microservices in cloud systems successfully. Given the vast number of various cloud resources, the provision of microservices in cloud computing environments is very complicated, including QoS parameters and VM configuration (e.g., CPU, memory, and I/O network). The term resources have historically meant denoting a physical object, such as a device, network, or storage.

2.1 Virtualization

In the process of virtualization, anything such as virtual apps, servers, storage and networks may be viewed on a digital or virtual basis. Software-based virtualization simulates hardware and generates a virtual framework. Virtualization is seen as the core component of cloud and edge computing, enabling many tenants to work in an isolated environment with their heterogeneous applications [4]. It offers several benefits, including heterogeneous workload of the working capacity, simple allocation, decreased risk of failure and improved availability.

The details of two main types of virtualization will be explained below:

2.1.1 Hypervisor-based Virtualization

A virtual machine (VM) is an emulation of an operating system in hypervisor-based virtualization. Virtual machines are built on the architecture of computers and have physical device capabilities. Specialized hardware, software or a mixture can be used for their implementation. Each virtual machine (VM) has its OS, independent of the host on which the hypervisor is operating. The default method of virtualization is known to be virtual machines. They deliver all the benefits of virtualization at the cost of high overheads relative to the performance of bare-metals. Typical examples of high-speed virtualization are Xen [35], VMWare [36], KVM [37], etc.

Virtualization based on hypervisors may be either complete if the guest OS does not know the virtualization or para, if the guest OS is modified it will make unique hypercalls to the hosting system. No matter what kind, hypervisor-based virtualization can be divided into two large architecture groups, namely Type 1 and Type 2 hypervisor. There are fundamental distinctions between Type 1 hypervisors and Type 2 hypervisors because Type 1 hypervisors directly interact with host machines' hardware and manage their virtual hardware resources. In contrast, Type 2 hypervisors operate on the top of the host OS and allow the OS to handle virtual hardware resources. The architectural difference between Type 1 and Type 2 hypervisors is shown in Figure 2.1.

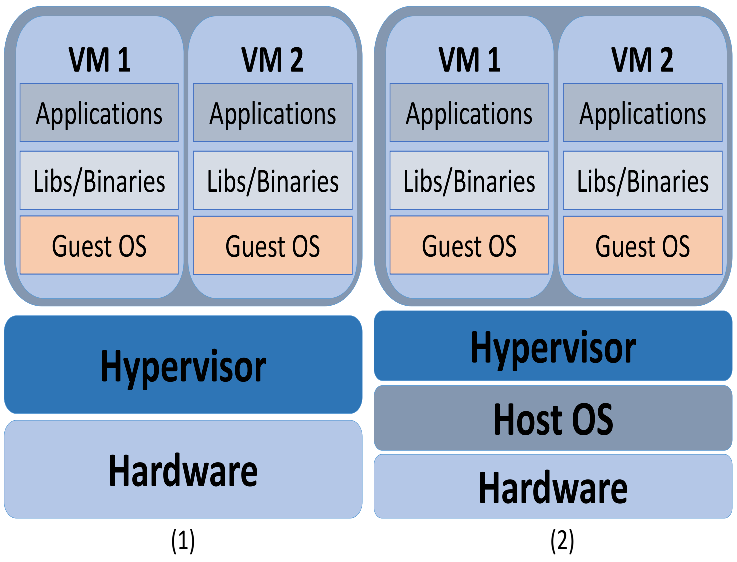


Figure 2.1: (1) Type-1 hypervisor-based virtualization and (2) Type-2 hypervisor-based virtualization.

2.1.2 *Container-based Virtualization*

In comparison, container virtualization is simple to use and the possibility of building microservice environments in contrast to hypervisor-based virtualization. The overhead is not standard when an operating system does not have to be installed. Applications and their dependencies are encapsulated in containers using the same host system and kernel.

Resource isolation is accomplished with features such as kernel namespaces and control groups using Linux kernels. The namespace limits a container's visibility such that it can only control its allocated resources. PID, MNT, NET, IPC are some of the typical container namespace features for process ids, file system mounting points, network functionality and cross-process communications in inter container environments [38] respectively. The clone () method call is used with each new container to build an abstract system in the OS kernel with an existing namespace. Linux cgroups represents additional kernel functions that control the allocation of resources by limiting the system resource utilization for each process group concerning CPU, memory, network and disk I/O. The priority for the use of the resource by the process category is often specified by cgroups. The resource constraints offered by cgroups appear in Figure 2.2.

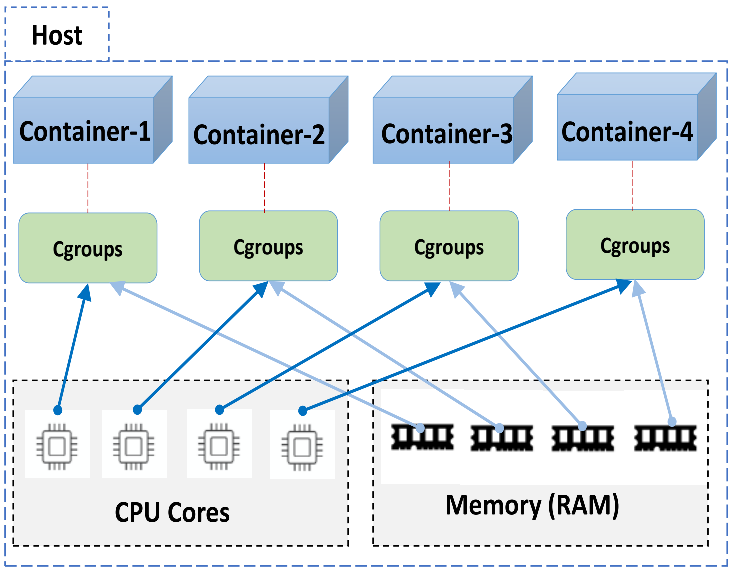


Figure 2.2: Resource limitations provided by Cgroups.

Linux Containers (LXC) [39] are a case in point. This idea is generalized with interfaces by Docker [19] and the ability to construct compact images for instantiation of containers. Images can be generated using a Dockerfile description. For images that contain the application and its dependency, the file works as a blueprint. Docker also offers a platform, comprising of the Docker Engine and the Docker Hub in addition to Docker files. The containers are designed and manufactured by the Docker Engine. The Docker-hub is a cloud infrastructure that enables Docker images to be shared. The container environment architecture, as seen in Figure 2.3. Containers can be used as individual units due to container architecture. Thus, by designing numerous microservices and using container technology, a microservices-oriented framework can be realized.

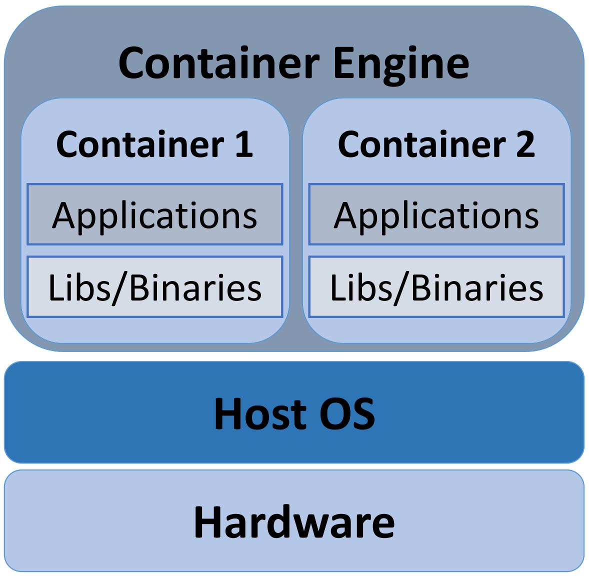


Figure 2.3: Docker architecture.

2.1.3 *Why container?*

The reasons for choosing containers are given below:

1. Lightweight. Containers are regarded as a lightweight alternative for virtualization based on hypervisors, allowing multiple isolated cases of work for customers. Unlike VMs, a container engine parses the equipment, which is essential for running the software inside it, rather than pressing different functionality on the same virtual machine, which simplifies the rapid development of events and the testing of microservices.

2. DevOps support. Recently, DevOps strategies are becoming more common because they promote the continuous supply of software applications and make it easier to work together between various applications development [40]. Although DevOps is not based on containers, the usage of containers gives many advantages to allow DevOps to function. Since the container environment exists regardless of the underlying OS, a stable development, test and output environment is simple to have. DevOps includes constant updates, simple to deploy using containers in an application or application module.

3. Microservice compatibility. Late trends transition towards decoupling application systems into smaller modules (microservices). Each application module can then be designed independently to support heterogeneous development. As containers have a lightweight environment that can isolate microservices to a least of dependence, it is reasonable for rapid development and microservices to be deployment [41, 42]. The convergence of data and microservices is achievable by having well-specified structures that are registered, tested and maintained within the architecture. The goal of the microservice was to make it easier for teams to be decoupled and pushed autonomously.

2.2 Monitoring

2.2.1 What is monitoring?

The topic being monitored, limiting the scope to the engineering sector, is typically a structure composed of components. Monitoring can thus be redefined as the *"action of observing and monitoring a system's behaviour and components over time"* [43]. Several other concepts of monitoring can be found amongst researchers as well as experts in the area of microservice-based applications.

- Fatema et al. [44] define monitoring as a *"process that fully and precisely identifies the root cause of an event by capturing the correct information at the right time and at the lowest cost in order to determine the state of a system and to surface the status in a timely and meaningful manner"*.

It is not useful to monitroing a system without understanding what it is, what the right action is. What these meanings add is then essential, namely to check coherence with pre-defined goals (or specifications). Furthermore, as pointed out in this thesis, monitoring refers to the random actions of a microservice-based application. As Bertolino [45] highlighted, this last feature separates monitoring from research, where instead the system is put in a simulated environment to synthetic a particular non-spontaneous activity. However, as this section aims at seeking an aggregate, universal definition, optional specifications will not be taken into account, the description suggested by Fatema et al. reports advisable properties for a monitoring framework. It is only possible to use a tracking system to verify whether a condition is complied with without providing the root cause. Adjustment, timeliness and cost efficiency are also essential, but not compulsory. Therefore, this thesis reference definition of monitoring will be:

Definition: Monitoring is the action of observing the application's components and the outputs of a system in each layer in cloud-edge infrastructure.

2.2.2 Why monitoring?

Monitoring in clouds is necessary to keep those services highly accessible and performance, and it is critical for both resource and customer suppliers [9–11]. Monitoring is mainly a primary method for i) management of infrastructure and hardware resources and ii) ongoing knowledge for these resources and cloud-based consumer-hosted microservices. In general, the efficiency and seamless processes of all system's resources are monitored for cloud tasks, such as resource management, SLA management, performance management, billing, and security administration [46, 47]. Consequently, the elastic existence of cloud computing is very much to be monitored [48]. Monitoring can be of two types of cloud computing: high and low. The status of virtual platform is related to high-level monitoring [49]. In the sense of low-level monitoring, the state of the underlying physical infrastructure is being collected [50]. A self-adjustable and usually multi-threaded platform is a cloud monitoring framework capable of supporting tracking functionalities. It detects cloud anomalies in-depth for pre-identified instances/resources. The monitors try to restore this instance/resource automatically if the matching monitor has an auto-healing action [51] when the irregular behavior is observed. A supporting team will be alerted if an auto-repair failure or a lack of an auto-healing action happens. Notifications can theoretically be sent in multiple ways, including email or SMS.

Monitoring allows for continued testing as design-time tests are extended following implementation. It is a technique needed to check if problems occur during services following the use of the system. Only then will the requirements on tracks that have not been reviewed in advance be confirmed. Once a problem trace is found, a new test may be generated, and the detected issue is replicated, and subsequent regressions are prevented [52].

The need for monitoring became more critical with the introduction and widespread adoption of cloud computing. This modern computing model, which requires services to be bought upon demand and paid for their actual use, allows costs to be minimized by effective power utilization and resource planning, as Aceto et al. [12] highlighted. To do so, prices and real resource utilization need to be monitored so that automation can come into action. Besides, to verify if Service Level Agreements (SLAs) are satisfied, the efficiency provided by third-party providers needs to be monitored.

2.2.3 Requirements for a Monitoring Platform

- **Portability** [44, 53] Users can deploy apps on different platforms. The same service may be built to work on entirely different hosting solutions in the case of multi-cloud applications. Therefore, monitoring systems should be capable of operating in heterogeneous environments.

- **Interoperability** [44, 53] Monitoring can work through heterogeneous datacenters if you want to give users the ability to run their application on various clouds or hybrid solutions.

- **Elasticity** [12, 44] In the case of complex environments such as the cloud where resources are unpredictable and the architecture subject to abrupt changes, monitoring tools can work continuously, without interruptions and manual reconfiguration.

- **Scalability** [12, 44, 53] The monitoring system should ensure that it works independently of the workload.
- **Multi-tenancy** [44] Cloud services have multifunctional systems in which many users typically use the same physical infrastructure. The cloud provider, as well as its customers, should be granted visibility on a right tracking platform and the necessary isolation should be maintained. Cloud services should be able to collect information about their system and possibly the summary of shared resources for each tenant to assure SLAs.
- **Comprehensiveness** [44, 53] A monitoring system should be able to monitor different physical and virtual infrastructure, multiple levels and platforms, and numerous cloud services.
- **Extensibility** [12, 44, 53] A monitoring framework may be expanded if external systems or cloud services, for example, by plug-ins, can add functionality or support.
- **Accuracy** [44] An exact tracking method can have accurate metrics, ensuring they are as close to the actual value as possible.
- **Resilience** [44] A resilient monitoring system may continue operating or recover those components automatically from a complete or partial failure.
- **Reliability** [44] A reliant monitoring system is able to provide its service for a given period of time under stated conditions.

2.2.4 End-to-end Link Quality Monitoring

Cloud-based applications, such as early warning systems, are time-critical and need sufficient assistance to reach assured, network-based QoS. This is difficult because if the network system's conditions change frequently, the performance is challenging to sustain. The concept of running certain application services on the edge of a network and others on clustered datacenters has raised important questions about the communications' network efficiency across an edge-computing framework between these services. It is a challenging for research because it concerns the live transfer of resources between edge nodes and datacenters and multiple nodes in edge computing frameworks at the same layer. The network performance must be assessed by end-to-end connection content monitoring for all communications across the edge-computing platform. Regardless of the hardware used in the networks, the edge model involves four kinds of network links between the application components:

- **Communications between a cloud datacenter and an edge node:** a foundation for applying innovative concepts for developed, implemented, deployed and operated between the cloud datacentre and the edge nodes within an edge computing environment are new enabling technologies such as SDN and NFV. In recent years, SDN and NFV have opened up new possibilities for virtualizing network work on demand depending on the consistency of the link from start to finish at all times. A complex implementation, migration, and scale-up of network functions, including routing, packet forwarding, and firewall services, can be effectively applied via NFV to network components (routers, bridges, switches). In addition to NFV, the SDN contains an array of APIs and control protocols such as SNMP and OpenFlow to program, managed, and automate network.

- **Communications between edge nodes:** edge nodes locally control a pool of virtualized services at multiple geography locations. It allows the service vendor to enhance the entire applicationperformance by collaborative supply and content distribution between peered edge nodes. Data transmission between these nodes can be done via a centralized approach such as SDN or through standard routing protocols through a full distributed system, e.g., OSPF [54].

- **Cloud-based communications:** the volume of data shared between components in various tiers deployed on cloud datacenters has rapidly increasing with a phenomenal rise in cloud-based applications. Such implementations have become challengingly tricky to ensure that these types of applications is favorable due to variance in running time in network conditions inherent in connections across internally replicated and distributed application components across/within of numerous cloud datacenters.

- **Communications between IoT object/user and edge nodes:** self-adaptive application providers must adapt their networks dynamically and effectively to the IoT device and the customer's network circumstances to provide high performance.

2.3 Microservices

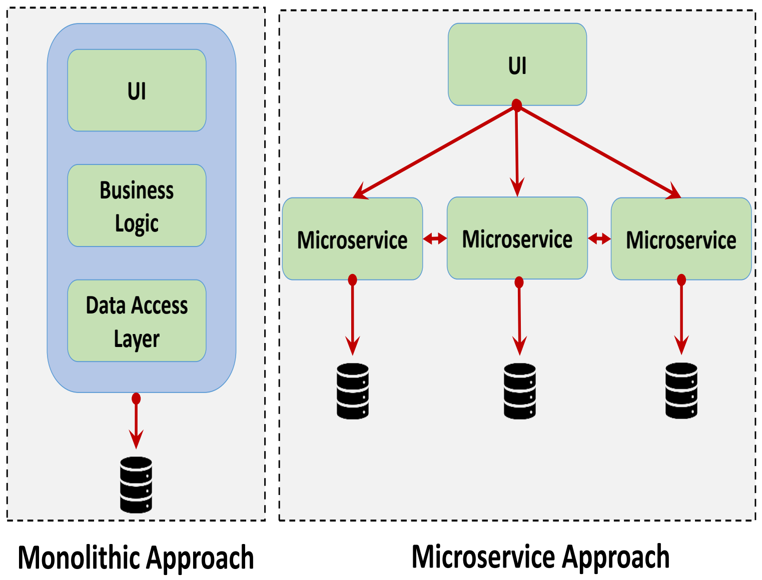


Figure 2.4: Monolithic and microservice architecture.

In the sense of service-oriented architecture, microservices primarily strive to disassemble monolithic systems into small services that connect. In contrast with monolithic structures, Figure 2.4 illustrates the paradigm of microservices. It indicates that little resources that provide their own data storage are microservices. On the opposite, the monolithic Figure simplifies the inclusion of all functional components in one component, including data storage. In effect, the microservice model contributes to independent, self-connected modules. REST is a popular means of interacting with other micro services through today's microservices. With the Hypertext Transfer (HTTP), the underlying data sharing can be completed. Thanks to the flexibility of the microservice, it is usually separate and easy to scale. Microservice may be implemented individually in the sense of continuous deployment. Summary: • Definite graining • Context-bounds • Self-developed • Autonomous • Decentralized [55] describes micro services with following characteristically characteristics.

2.4 Technologies for Diagnosing Anomalies using ML Algorithms

Various detection techniques have been used, they include statistical analysis, adaptive method, machine learning. A comparison between the three main methods is discussed in [56]. In this work, we are interested in detecting anomalies using machine learning. Machine learning is widely applied in many anomaly detection research because it is a simple, effective, and accurate method for detecting and classifying anomalies. There are three methods for machine learning techniques; supervised, semi-supervised, and unsupervised. Choosing one of them depends on the data label existence. The supervised technique is used if labeled data of normal and anomalous are available. A semi-supervised are required when labeled instances for the normal class is available. In contrast, unsupervised techniques do not require any labeled data. Each technique has advantages and disadvantages, see [57].

In this thesis, we applied supervised machine learning methods which are widely used in litterateurs and provide high performance in terms of accuracy and precision [28] [58]. The disadvantage of the supervised method is labeling the anomaly classes due to the fact that anomalies are rare events. This can be tackled by injecting artificial anomalies in a normal data set to obtain a labeled training data of normal and anomaly data. Many algorithms exist for supervised machine learning.

In supervised machine learning, there are different algorithms that can be applied. Sauvanaud et al. [28] conducted anomaly classification with K-Nearest Neighbors (KNN), Random Forest (RF), Neural Network (NN), Naïve Bayes (NB). The results of this research work indicated that RF and KNN achieved higher accuracy compared to NN and NB. Nonetheless, NB performed much faster with a calculation time of milliseconds compared to NN and KNN. In the same content, du et al. [58] conducted a comparative study for anomaly classification with Random Forest (RF), Support Vector Machine (SVM), K-Nearest Neighbors (KNN) and Naïve Bayes (NB). This study reported that RF and KNN achieved higher accuracy, wile NB performed worse followed by SVM, which provide the worst performance. From both studies, the results showed that machine learning classifiers with computational complexity (SVM and NN) are worse compared to traditional supervised methods.

Since anomaly detection is not only an alarm that reaches a threshold, the algorithm used to build the anomaly detection model should provide simple, accurate, low cost, and interpretive methods for anomaly detection. Therefore, the selection of an optimal algorithm is often required. To achieve such goal, we need to compare different supervised machine learning algorithms based on a set of proprieties. Table 2.1 presents a comparison of supervised machine learning algorithms that used for anomaly detection in a microservice-based application. For the purpose of this study, we will consider those algorithms that support the following proprieties: model interpretability, low amount of parameter tuning needed, and low time complexity. Therefore, in this thesis, we will compare four algorithms that have different learning strategies: K-Nearest Neighbors (KNN): instance-based Algorithm, Decision-Tree (DT): rule-based Algorithm, Logistic Regression (LR): Regression Algorithm, and Naïve Bayes (NB): Bayesian algorithm. The next section provides a brief description of each of the selected algorithm.

2.4.1 K-Nearest Neighbors (KNN)

K-Nearest Neighbors is an algorithm that is called a lazy learner because it does not built a model. Instead, each time the KNN algorithm classify new data, it uses all the training set to calculate the distance between the new data point and its Neighbors using some proximity metric like Euclidean distance, Hamming distance, Manhattan distance and Minkowski distance. The class of all k neighbors are determined and the class that most of neighbors belong to is assigned as a class for the new data. Euclidean distance is commonly used as proximity metric. KNN is a very simple algorithm, however, it needs more time to classify the data as it scans all the training

Table 2.1: Comparison of some supervised machine learning algorithms

Algorithm	**ANN**	**SVM**	**RF**	**KNN**	**DT**	**LR**	**NB**
Parametricity: Making an assumption that the dataset belongs to a parametric family of probability distributions	No	No	No	No	No	Yes	Yes
Time complexity: The time an algorithm takes during training and testing.	High	High	High	High	Low	Low	Low
Sample complexity: The amount of training data an algorithm requires and number of features	Large	Large or Small	Large or Small	Large or Small	Large or Small	Small	Small
Interpretability: Understanding the decision-making process of an algorithm	Difficult	Difficult	Difficult	Easy	Easy	Easy	Easy
Tuning the Model: The amount of parameters need to be tuned to avoid over or under fitting	Large	Large	Large	Small	Small	Nothing	Nothing

set [59].

2.4.2 Decision-Tree (DT)

Decision tree is a classification and prediction algorithm. The DT of dataset is constructed and learned by rule-based method. The tree has nodes that represents one of the attributes which are selected in every level of the tree using attribute selection measures (like Information Gain, Gain Ratio, and Gini Index). Each node has a decision rule to split the data into at least two branches. The leaves, that represent different classes, are the bottom nodes. Main advantage of decision tree is that it is easily interpreted and visualized. In addition, the DT has no assumptions about data distribution. However, DT algorithm build unbalanced tree due to majority classes in the dataset. Hence, training dataset for DT should be balanced [60].

2.4.3 Logistic Regression (LR)

Logistic regression is a classification method that using statistical methods. LR calculates the probability that set of attributes (X) belongs to a class and uses the logistic function as stated in the equation:

$$P(X) = \frac{e\hat{}(b0 + b1 * X)}{(1 + e\hat{}(b0 + b1 * X))} \tag{2.1}$$

The coefficients b of the logistic function are estimated from the training data, and this is done using maximum-likelihood estimation. LR is one of the most simple and easy to implement. Moreover, it does make assumptions about data distribution. LR model only needs to stores the coefficients b in the memory [61].

2.4.4 Naïve Bayes (NB)

A Naïve Bayes algorithm is one of supervised learning algorithms. This algorithm constructs a model based on Bayes' theorem. Bayes' theorem (4.1) is stated in equation:

$$P(B|A) = \frac{P(B) * P(A|B)}{P(A)} \tag{2.2}$$

This equation is the probability of attributes (B) belong to class A. For all classes, this probability is calculated and the class with the highest probability will be assigned to the independent attributes (B). This probability is based on the assumption that the attributes (B) are independent from one another. Naïve Bayes is a simple, fast, and very low computation cost algorithm [61].

We trained four supervised ML algorithms by using the labelled database. These algorithms were applied to the two tasks: detection (binary classification) and localization (multi classification).

2.5 Deployment Environment

2.5.1 Cloud Computing

Cloud computing is a crucial technology which provides scalable and versatile IT infrastructure and resources as different internet services. According to the NIST concept of cloud computing, this paradigm is described by Mell et al. [62] as a ”model for allowing omnipresent, easy, on-demand network access to a common pool of configurable computing resources (e.g. networks, servers, storage, software, and services) that can be easily supplied and released with minimal management effort or interference amongst service providers.” In comparison, the underlying cloud-based applications and software processing infrastructure are entirely abstracted from the service users. As a consequence, the cloud-based resource provider is responsible for the performance, stability and scalability of the storage environment and the IT services provided. Software as a Service (SaaS), Platform as a Service (PaaS) and IaaS (Infrastructure as a Service), the offered services can be divided into separate services. SaaS provides the on-demand use of Internet software. In comparison, PaaS provides a deployment ecosystem, including the current cloud infrastructure’s applications development environments and platform layer resources. IaaS is the gateway to the cloud distribution model. This service offers essential services, such as computing power and memory [63]. Besides, virtual computers with operating systems can be built as the frameworks for microservice-based environments.

2.5.2 Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) is an interdisciplinary approach for combining communication devices, computation, and actuation for performing time-constrained actions in a predictive and adaptive manner [64, 65]. This is done using a feedback loop within the physical system, which enables the embedded and network systems to monitor and control the physical processes. In this way the design of a previous model can be modified using feedback from the physical system. This also makes the system more robust, reliable and free from any past errors. According to the National Institute of Information and Communication Technology (NIST) [66], cyber-physical cloud computing is *"a system environment that can rapidly build, modify and provision cyber-physical systems composed of a set of cloud computing based sensors, processing, control, and data services".*

CPS consists of three main elements: cyber, physical, and network components. Each of these components consists of a few other components. For example, the cyber component consists of two components: cloud and IoT devices where the IoT devices work as a bridge between physical and cyber components. The network component is used for interlinking the cyber and physical components and transferring and controlling data. In order to develop a robust architecture for a CPS solution, data needs to be collected from various physical sources (for example traffic, education, and healthcare systems [67]) using IoT devices (e.g. sensor, mobile, and a camera). Every day larger applications with more devices are being connected with CPS, which means that a larger variety of physical conditions need to be considered, and this requires larger volumes of data to be extracted using IoT devices, and filtered and processed using cloud data centres (cloud). Therefore the main components of a CPS can be summarised as follows:

1. Physical Component: This component does not have any computation or communication capability; it only includes biochemical processes, mechanical processes, or humans. Physical components collect and provide data, which is required to be processed in real time for controlling various activities. Such data is usually highly concurrent and dynamic.

2. Cyber Component: is used for collecting, processing, reporting and controlling all the physical components within CPS. As it is challenging to manage the concurrent and dynamic data from the physical component of CPS, the cyber component is divided into two sub-systems. These are cloud data centers, and IoT devices [67].

3. Network Component: is responsible for communication between the physical and cyber components or among the cyber components. The raw data is captured from components such as IoT devices and passed to the cloud. Also, cloud devices send control and feedback to the IoT devices using network components. Main factors that affect network communications are bandwidth, topology, latency, and congestion [68, 69].

2.6 Commercial and open source monitoring tools and anomaly detection

2.6.1 Monitoring Tools

Several works already published have explored topics related to service monitoring, as well as models and metrics for QoS assurance. Whether in the cloud, VMs or even in containers using microservices, or even in monitoring services at the edge, varied solutions and results have been presented.

Docker [19] provides an inbuilt monitoring tool, Docker stats, to examine the resource usage metrics of running containers. The various metrics provided by Docker stats are CPU and memory usage, and actual free memory for each container. However, it does not inspect the performance of individual application running inside a container. Our proposed framework, *M3*, improves on this significantly by monitoring the performance of each application running inside a container. Along with this, *M3* also gathers the monitoring information from containers running in heterogeneous cloud environments (e.g. Amazon, Azure, Openstack, etc.). By aggregating the data collected from multiple containers running across multi-cloud environments, one can perform different types of performance comparisons to assess the performance in containers.

CAdvisor [20] is an open source monitoring framework that displays monitoring performance and resource usage in real time. It provides CPU usage, memory usage, network and throughput information of the running containers. One can access the monitoring information only for two minutes duration, as there is no associated storage mechanism that can retain the data for a longer interval. In contrast *M3* monitors the performance of individual applications that run inside the container/VM and also stores monitoring data in a database shared by both container and VM.

Datadog [21] is a monitoring service that gathers metrics such as CPU utilization, memory, and I/O for all containers. It is an agent-based system that sends data only to the Datadog cloud, making the monitoring job completely dependent on Datadog's cloud. On the other hand *M3* has the ability to store data in any cloud service provider.

CloudWatch [22] is a commercial cloud monitoring tool that tracks CPU, memory usage, and network but cannot monitor application-level QoS metrics. In addition, it is not platform independent (i.e., it works only for Amazon platform and not for Azure). Similarly, **Microsoft Azure Fabric Controller** is limited to work only on the Azure platform [70]. *M3*, on the other hand, has the ability to monitor applications in heterogeneous cloud environments.

In [23] the authors present **CLAMS**, an application monitoring framework for multi-cloud platforms. Moreover, their monitoring framework considers different QoS parameters for web-applications running inside a VM. The model retrieves the QoS performance for different cloud layers. However, the model does not monitor the performance of containers. In addition, the model is constrained to only web applications. This is different to our framework, which monitors cyber-physical applications that run inside containers and VMs.

In [71], the authors present CLAMBS, a framework for monitoring and benchmarking applications in a multi-cloud environment. In addition, a model for multi-layer monitoring in the cloud is presented. In this way, QoS parameters relevant for each cloud service layer are listed. Finally, an experimental evaluation is performed in the IaaS level. The work presented here follows a similar approach for defining and experimenting with QoS parameters, although it is different from the use of the cloud and the edge, besides focusing more on the application level.

The microservice monitoring in the edge environment is reported in the paper presented at [72]. In [72] a state-of-the-art review of self-adaptive applications using edge microservices and services in the cloud are performed. The results observed shows that the main parameters of QoS for virtual machines in the cloud are the usage of: CPU, memory and network.

The monitoring of services deployed in containers is present in the works [24] and [73]. In the work published in [24] the authors present a framework called PyMon that uses the Docker management API to obtain statistics of resources used by containers. Unlike [24], the present study uses libraries to monitor processes inside the containers, thus allowing the effective monitoring of a container that performs a multi-service or multi-process environment. The work presented in [73] brings an assessment of the use of Docker containers versus the use of virtual machines. To verify the QoS parameters to be compared for evaluation, the authors monitored the CPU usage by the installed Docker process, not verifying the parameters of the containers that are being executed or even of the processes internal to the containers.

In [74], the authors present an architecture that collects, analyses and presents the physiological data. Also, it captures data from numerous sensors for further transformation and analysis. This paper is mainly concerned with monitoring particular parameters for performing scheduling in only the cloud environment. In contrast, our framework monitors the performance of an application in a holistic cyber-physical system.

Ganglia is a hierarchical modular device control system with higher performance clusters and grids [75]. It enables users to view both runtime and historical information on all running VMs monitored on their web-interface [76]. Ganglia's simple architecture contributes to a high degree of scalability, robustness, interoperability and low per-node overheads, and fast handling and portability [75]. It is also integrated and extended into cloud-based apps for both private and public cloud monitoring [76]. The insufficient monitoring of containers, restricted choice of data storage options and the low efficiency of the 'gmetad' nodes are some of the drawbacks [77].

Prometheus is the monitoring framework offering a set of metrics that gather PULL metrics, composite, visualization and alerting resources, with the effect shown on the graphs from its web interface [78]. It also offers a modular query language data model for multi-dimensional time series that allows for customization for the monitoring process. While both machine-centered and dynamic service-oriented architectures like microservices function well, there is a lack of precision, since it is not likely to be adequately systematic or precise. Besides, scrape metrics include a mediatory monitoring tool is required. In addition, the container migration monitoring is not even flexible to manage vast quantities of tracked containers [79].

A **Monitis** monitoring framework for a vast array of IT resources, including websites, servers, applications, networks and cloud virtual instances and platforms and mail servers [78] is the only non-open source monitoring tool mentioned. It includes end-user experience, uptime, loading of web pages and purchases for websites. A Monitis monitor agent should be used by networks to gather data on a system of networked splitters, instead of to install a Monitis agent on each device [23]. The framework supports custom features through its API, allows checks for short intervals and alerting [78]. Although Monitis support monitoring applications, it is not clear if container monitoring is possible. However, it offers a wide range of qualities including stability, scalability, ease of use and fast remote access setup.

A **Nagios** has two versions: an open-source version and a commercial version. It is the most popular open-source management platform for IT infrastructures, which offers multiple monitoring supports, including network services, routers, hard-working networks (or switches), and virtual tools and cloud platforms [78]. It also has an alert system and a web-based notification system that allow network administrators to monitor all devices and hosts with a network system's SNMP protocol [76] activities and status. It can also connect to databases such as MySQL from third parties. The downside is that it takes a complex manual setup for Nagios to be deployed [76]. Its architecture includes the use of a central server, making it un-scalable, unsolid and unsuitable for use in dynamic environments [79]. Container or device control is not feasible, while the grid infrastructure and VM monitoring are possible [44].

A **Zabbix** is an enterprise-class management platform used mostly for monitoring network parameters and resources [44] for the network, servers, applications, virtual machines and cloud services. It provides many features such as real-time VM, container and application monitoring, effortless scaling, high interoperability, auto discovery, predictive consistency and efficiency analytics, an alert mechanism, a SQL database and a web interface [44]. Installing and configuring is also convenient. However, its weaknesses include its lack of robustness and inefficient self-discovery, because the time limitation can be a significant issue. It must also be used with other tools to track servers and availability to achieve the required monitoring [44].

A **The Docker Universal Control Plane (DUCP)** is a tool for handling, loading, customizing and monitoring distributed software using Docker containers [76]. This container management solution supports all Docker developer software, such as Docker Compose, in which projects through clusters are multi-container applications. Any of DUCP's main features as a native market solution are high scalability and Web-based GUI.

A lightweight, production-grade application monitoring service equipped for modern development teams is Scout monitoring tool [80]. A **Scout** is another web-based graphical monitoring platform with the ability to store measured metrics at most 30 days. It consists of a sensible engine able to alert based on the metrics and their corresponding preset thresholds. Like Scout, containers with the same characteristics are managed with several commercial solutions.

Docker container efficiency was measured in line using system resource utilization, according to [73]. Authors found from their database that container-based virtualization is comparable with memory, CPU and disks use OS that operate in bare-metal. The performance of Docker is similar to the understanding of the native environment concerning these three metrics. However, the host OS has slightly less network capacity than the Docker container depending on network consumption. For this task, however, only one container was allocated to simplify experiments. Besides, a further supplement to this work may be tested for container performance against other virtualization technologies.

The authors [81] showed an architecture for a dashboard management framework to monitor the use of the source for servers and VMs in real-time. It was observed that the virtual server could not operate normally if CPU, memory, and storage are overloaded. However, the paper did not clarify how the test could be applied, which is why the suggested solution cannot be utilized, as the authors contend, to improve the efficiency of VMs. It is worth noting that only a particular form of virtualization (Xen hypervisors) can be used in the proposed monitoring architecture.

The authors [82] proposed a model for predicting the response time of cloud applications, based on Linux OS counters, for example, LoadAVG. One of the essential explanations for calculating low-level application metrics (e.g. Processor load) is that measuring application-level metrics (e.g. response time) could create overheads both in networks and the re-sources of computation. Besides, monitoring high-level metrics can impact users ' privacy. The findings reveal that LoadAVG's load values complement the response-time behavior, which contributes to a clear positive association between the two behaviors. Their work only takes LoadAVG into account, and thus this model can be further developed by exploring other equivalents such as iostat and netstat.

According to the authors [83], a simple approach for container monitoring uses its docker API for resource aggregation and database storage. The protocol calculates the standard deviation of a metric for resource monitoring. And when the standard deviation reaches any cap will the tracked data be stored in the database. This approach has utility but does not have an alert feature in data storage.

A **Elascale** [84] is a self-scaling and tracking methodology for Docker [19] microservices-based cloud software systems. The tracking is carried out through an ELKB stack that enables performance measurements, e.g. CPU, memory, and networking, to be obtained on a given container through Elasticsearch, Logstash, Kibana, and Beats [85]. Elascale only supports Docker technology; it also provides little detail on execution failures.

A **Sysdig** [86] is a container solution that enables collect resource usage, network statistics, and applications to track within containers, such as microservices. Containers system are recorded using a kernel module, while application monitoring is conducted by instrumentation. Instrumentation consists of written, formatted text strings to /dev/null.converts strings into events to provide the instrumented item with completion time.

A **Netflix Hystrix** [87] is a Java library that seeks to avoid cascade failures with latency and fault tolerance. It facilitates the almost-real-time tracking of resources through the use of source code. However, any call to external systems and resources dependency has to be packaged in a HystrixCommand object, which offers measurements of results and latency.

A **Sensu** [88] is a cloud monitoring and IT infrastructure. It is an open-source management platform that tracks (i) servers bare metal, VMware and AWS servers, (ii) containers such as Docker, RKT and LXC, (iii) database and web server services for instance, and (iv) mobile applications, micro-networks, (v) network devices such as routers, switches, SANs, and (vi) remote resources such as third-party APIs. Some of Sensu's main features are: (i) alerts are generated; (ii) dynamic server registration and de-registration; (iii) plugin monitoring tools like Nagios, Icinga and Zabbix. For exchange of messages, Sensu relies on RabbitMQ and for storage on Redis. If they fail or cause delay, this dependence may become a liability under some circumstances.

2.6.2 Anomaly Detection

There have been a variety of methods applied for detecting anomalous behaviour and analyzing the root causes in microservice-based cloud application.

arla Sauvanaud et al. [28] introduce an anomaly detection system (ADS) that considers performance data for microservices to detect anomalies. ADS can identify the anomalous microservice and the type of anomaly within the observed microservice. This chapter only applies the system on virtual machine-based microservices rather than the containerised approach popular today, which we consider key tackling in our approach. Qingfeng Du t al. [58] also propose an Anomaly Detection System (ADS) but do so for container-based microservices. The proposed ADS has a monitoring module that collects performance data from containers as well as a data processing module based on machine learning. This system can identify the type of anomaly within the anomalous microservice.

Both these works analyse metrics of resource utilization in the infrastructure layer. Still, they do not consider the link of observed failures with underling faults in during anomaly detection as we do. We consider using the multi-resolution technique to analysis metrics in sequence as beneficial, especially for improving the detection accuracy of machine learning algorithms.

Li Wu et al. [27] propose MicroRCA system that correlates failure observation of an application performance corresponding its root cause faults in resource utilization in real-time. The attributed graph is used to analyze the root causes by modelling anomaly propagation among microservices of the application. Juan Qiu et al. [89] also use knowledge graph technology and a causal search algorithm to diagnose the root cause of application performance. Yuan Meng et al. [29] present a framework called MicroCause. Their approach localizes the root cause of low-performance indicators in a microservice. MicroCause useS path condition time series (PCTS) algorithm and temporal cause-oriented random walk(TCORW) method. Jörg Thalheim et al. [26] deploys a platform, called Sieve, to monitor microservice performance metrics and analyze the root causes of observed bad performance. Sieve has two core tasks to apply root-cause analysis: first Sieve filters out metrics that present normal behaviour and keep other metrics. Second, Sieve uses a predictive-causality model to find metrics dependencies of microservices. Wei Cao et al. [90] apply Conditional Random Field(CRF) method for microservice anomaly detection. The method creates the microservice fault matrix by collecting microservice metrics as an observation sequence. Therefore, anomalies of a microservice can be obtained from the microservice fault matrix.

These approaches search the cause root of anomaly observed behaviour using different tracking algorithms (e.g. tree or matrix); however, our approach is to study the feasibility of machine learning algorithms to root cause localization.

As discussed above in **section 2.6.1 and 2.6.2**, existing monitoring solutions do not have the ability to monitor the performance of microservices running inside multi-virtualization heterogeneous cloud environments (container/VM). These monitoring tools are not able to satisfy the performance monitoring requirements of complex microservices deployed across multiple cloud data centers. Also, another challenge that underpins such emerging approaches is the dynamic management of microservices across cloud and edge datacenters. For instance, defining when and how microservices can be migrated from edge resources to cloud-based resources (and vice versa), and characteristics which influence such migration, remains a challenge. Furthermore, scalability of a container-based microservice application allows for more application organization and optimization. But, this strongest characteristic brought new complexities to application performance monitoring. Indeed, inefficient performance monitoring may lead to severe application outages as it is not able to successfully and quickly detect the failures and then localizing their causes.

Based on the aforementioned challenges, this thesis addresses the research gap:

- In chapter 3, our proposed work (*M3*) differs from the aforementioned solutions as it can be used to holistically monitor the performance of microservices / cyber-physical applications running inside containers or VMs distributed across multiple cloud environments.

- In chapter 4, we propose a **M**onitoring, anomaly **D**etection and **L**ocalization **S**ystem (*MDLS*) that follows component-based architecture to provide an efficient approach for detecting and localizing anomalies. This approach can be easily deployed by developers to interpret the causes of performance failures in a container-based microservice application. We add a monitor engine to collect metrices from both microservice and container levels, and utilize multi-resolution technique to improve the accuracy of detecting and localizing the root cause of anomalies significantly.

- In chapter 5, our proposed work differs from the current approaches by presenting an advanced monitoring solution that can be used to monitor microservices deployed in osmotic computing environment i.e. the cloud and/or deployed at the edge.

A summary of literature review with the current challenges is summarized in Table 2.2.

Table 2.2: Comparison of literature review with the major challenges addressed in this thesis

	VM	**C**	**MC**	**EI**	**AD**	**Challenges**
Docker	×	✓	✓	✓	×	• Some of the frameworks are specific for the VM environment only. • Some of the frameworks are specific for the container environment only. • Some of the frameworks are either cloud provider specified. In another word, they are not platform independent. • Some of the frameworks are not monitor the performance of individual applications that run inside the container/VM and also stores monitoring data in a database shared by both container and VM. • Some of the frameworks are not able to store data in any cloud service provider. • Some of the frameworks have lack capability to monitor microservices at the edge. • Some of the frameworks focus on single layer monitoring, i.e., microservices in the cloud or microservices at the edge. • Some of the frameworks are not able to monitor cyber-physical applications that run inside containers and VMs. • Some of the frameworks do not consider the link of observed failures with underling faults in during anomaly detection. • Some of the frameworks do not study the feasibility of machine learning algorithms to root cause localization.
cAdvisor	×	✓	✓	✓	×	
Datadog	✓	✓	✓	✓	×	
CloudWatch	✓	✓	×	✓	×	
Azure Fabric Controller	✓	✓	×	✓	×	
CLAMS	✓	×	✓	×	×	
CLAMBS	✓	×	✓	×	×	
PyMon	×	✓	✓	✓	×	
Ganglia	✓	×	✓	×	×	
Prometheus	✓	✓	✓	✓	×	
Monitis	✓	×	✓	✓	×	
Nagios	✓	×	✓	✓	×	
Zabbix	✓	✓	✓	✓	×	
DUCP	×	✓	✓	✓	×	
Scout	×	✓	✓	✓	×	
Elascale	×	✓	✓	✓	×	
Sysdig	×	✓	✓	✓	×	
Sensu	✓	✓	✓	✓	×	
[72]	✓	×	✓	✓	×	
[74]	✓	✓	✓	×	×	
[81]	✓	×	✓	✓	×	
[83]	×	✓	✓	✓	×	
[26]	✓	×	✓	×	✓	
[27]	×	✓	✓	×	✓	
[28]	×	✓	✓	×	✓	
[89]	×	✓	✓	×	✓	

Abbreviations: ×, No; ✓, Yes; VM, Virtual Machine; C, Container; MC, Multiple Cloud; EI, Edge Infrastructure; AD, Anomaly Detection;

3

A framework for monitoring microservice-oriented applications in heterogeneous virtualization environments across multiple clouds

Contents

Summary

This chapter presents a generic monitoring framework, *Multi-microservices Multi-virtualization Multi-cloud (M3)* that monitors the performance of microservices deployed across heterogeneous virtualization platforms in a multi-cloud environment. We validated the efficacy and efficiency of *M3* using a book-shop application executing across AWS and Azure. In addition, we significantly extended *M3* by implemented of highway traffic monitoring services using a cyber-physical system. So, we propose *M2CPA* - a novel framework for multi-virtualization, and multi-cloud monitoring in cloud-based cyber-physical systems. *M2CPA* monitors the performance of application components running inside multiple virtualization platforms deployed on multiple clouds. *M2CPA* is validated through extensive experimental analysis using a real testbed comprising multiple public clouds and multi-virtualization technologies.

3.1 Introduction

The recent emergence of microservice architecture [91] has made significant changes to the development, deployment, and on-going maintenance of web applications. Compared to the traditional monolithic application architecture, where the whole application is built as a single unified system, the microservice approach decomposes the application into several independently executable software components or units that coherently interoperate to deliver specific application functionality. To enable run-time communication between microservices, approaches such as lightweight REST-based APIs [92–94] have been widely adopted. Microservice-based application architecture has also turbocharged the DevOps [40, 95, 96] design philosophy by minimizing code-base dependencies between software units.

Although decomposing a monolithic application into lightweight microservices eases DevOps processes related to code updates, maintenance, and continuous integration, it does not solve issues related to ongoing performance management and monitoring. To contextualize this, consider the application deployment scenario related to a book-shop application and highway traffic monitoring services as described in **section 3.1.1 and 3.1.2**.

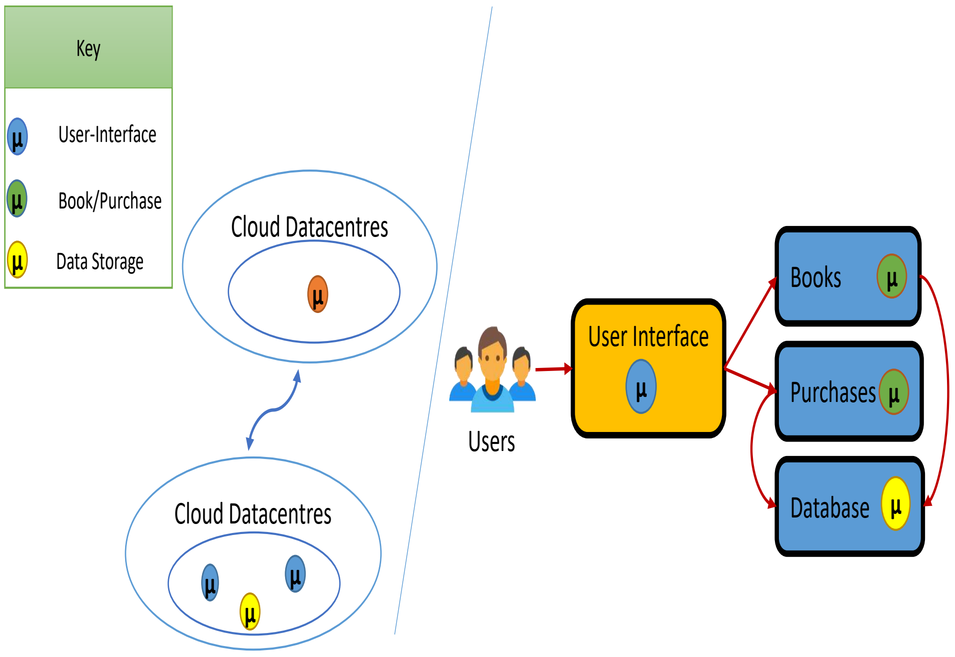


Figure 3.1: Example scenario for microservices distributed across multiple cloud datacentres.

There are two cases of my motivation based on infrastructure that will be explained in detail:

3.1.1 Motivation Example One

Figure 3.1 illustrates a conceptual implementation of a Book-Shop application based on the microservice architecture. The book-shop application is a multi-layer stack which includes, (i) User Interface, (ii) Book Search/Purchase, and (iii) Data Storage. User Interface (UI) is deployed as a web microservice responsible for receiving user requests and returning content to be rendered by the SmartPhone App or browser. Book and purchase layers are deployed as multiple app microservices that implement business logic for searching the inventory and/or processing purchase requests (e.g. credit card transaction management, users' address book management, coordination with distribution and the shipping company). On the other hand, data storage is deployed in multiple database microservices for managing the input and output datasets.

To improve the security of users' data as well as to enforce data privacy regulations such as EU General Data Protection Regulation (GDPR) [97], the owner of the book-shop application may decide to distribute the microservices across multiple private

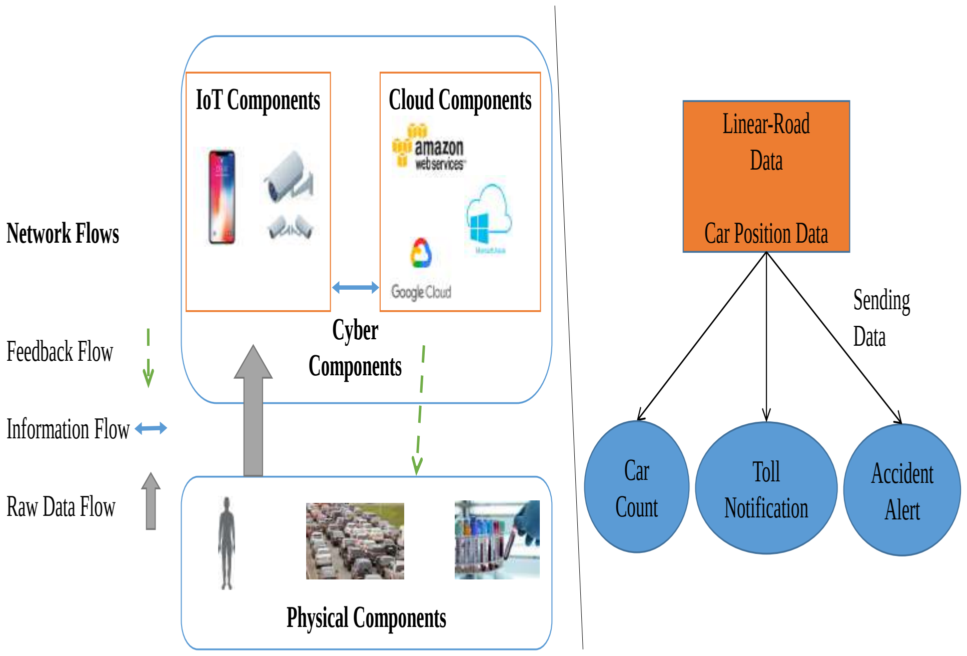


Figure 3.2: Cyber-physical system and an example of stream data management for highway monitoring system.

and/or public cloud environments. For example, the microservices related to credit card transactions and user's address book management, are more likely to be deployed on a secure private cloud data center. On the other hand, microservices related to the current inventory of books are more likely to be deployed on a public cloud data center. Accordingly the database microservices required for provisioning data to the above microservices (address book, inventory, etc.) will also need to be distributed across public and private cloud data centers. Though such wide scale distribution of microservices leads to improved security and privacy, it complicates the ongoing performance management and monitoring as discussed in next **section 3.3**.

3.1.2 Motivation Example Two

Figure 3.2 describes a conceptual implementation of highway traffic monitoring services using a cyber-physical system. The sensed data of highway traffic (for example the position of the cars) is sent as a stream of events that is physically separated and used for problems such as traffic monitoring and management. This requires the processing of huge volumes of data with high efficiency using the capabilities of multi-cloud environments. [65, 98–102]. To effectively explore data processing in a multi-cloud environment, three services for highway traffic are considered. These are: (i) Toll Collection Notification, (ii) Accident Alerts, and (iii) Car Count (a detailed discussion is given in **section 4**). The system will manage its resources in terms of sensor data and other saved data available in the cloud and provide the requested information to the driver. For example, the highway traffic system will send an alert to drivers on their navigation systems to inform them to take appropriate routes (push mode). Also the driver can request information about traffic routes, and then make informed decisions based on that information (pull mode).

The performance of a cyber-physical application in cloud systems may vary considerably due to factors such as application type, interference effect (caused by other applications running in the same or different containers), resource failure and congestion [103–105]. Quality of Service (QoS) denotes the levels of service offered by the cloud provider in terms of service features depending on the user's/application's requirements [106]. QoS is generally defined in terms of application specific features such as availability, pricing, capacity, throughput, latency, and reliability or user dependent features such as certification, reputation, and user experience rating. QoS is essential for both the user who expects the cloud provider to deliver the published services, and the provider who needs to find a balance between the offered service and functional cost. Agreement between the user and the provider on the quality of service offered leads to a Service Level Agreement (SLA). SLA creates transparency between user and cloud provider by defining a common ground, which is agreed by both user and cloud provider [107]. Appropriate penalties are normally associated with the SLA, which are applied in case of SLA violations. Therefore, it is imperative to monitor the QoS provided by the cloud provider to check whether the SLA is satisfied or not [108, 109]. Monitoring is required for different purposes such as resource provisioning [110], scheduling [74, 111, 112], security [113–115], and re-encryption [116, 117]. To detect any performance anomaly or to ensure that SLA requirements are achieved, continuous monitoring is essential [118, 119].

In virtualized environments, an application may be distributed over multiple containers/VMs, each running some services communicating over REST-based APIs [120, 121]. Monitoring is required at both individual container/VM level or at application level to guarantee that the QoS requirements of the application are satisfied. There are some lightweight endpoints available that can easily be plugged in to perform the monitoring operations for a single environment application. However, for complex containerized applications, it is challenging to have a single monitoring end-point, because each container may be hosted on different environments that do not support a common monitoring endpoint [122].

Currently, there are multiple monitoring frameworks e.g. Docker stat [19], CAdvisor [20], DataDog [21], Amazon cloud watch [22], CLAMS [23], available to monitor the applications running in the cloud. However, most of the frameworks are either cloud provider specific e.g. [Microsoft Azure Fabric Controller], or virtualization architecture specific e.g. [CAdvisor]. These monitoring tools are not able to satisfy the complex dependent requirements of microservices/CPS that can provide holistic monitoring across multi-cloud scenarios supporting different types of virtualization. Monitoring the performance of services in such a complex environment is very challenging for the following reasons:

- The deployment environment for microservices/cyber-physical applications in multi-cloud environments is very complex as there are numerous components running in heterogeneous environments (VM/container) and communicating frequently with each other using REST-based/REST-less APIs. Moreover, the performance of such microservice-based applications/cyber-physical applications deployed in a multi-cloud environment can vary considerably due to the heterogeneity such as microservice types (e.g. CPU intensive vs. I/O intensive vs. memory intensive) and resource interference caused by other competing microservices [41, 103, 104, 123–125]. In some case, multiple components can also be executed inside a container/VM making any failure or anomaly detection very complicated. It is necessary to monitor the performance variation of all the service components to detect any reason of failure.

- As different virtualization environments implement different ways to allocate resource limits to microservices, it complicates the performance monitoring problem. Unlike a hypervisor-based Virtual Machine (VM) which has its own guest operating systems, resource allocation for containerized microservices are defined in terms of namespace and cgroups that share the host operating system with other containers. Further, the resource limitation in containers can be hard or soft as compared to VM which is always strict (hard). A soft limit allows containers to extend beyond their allocated resource limit creating higher chances of interference [30, 31]. Monitoring the performance of microservices/cyber-physical applications in such cross VM-container scenarios is very important to ensure that services are executing in a desirable way.

- Modern applications can be distributed across multiple cloud environments including bare metal, public or private cloud depending on several features such as microservices/cyber-physical application component requirements, deployment locations, security concerns, cost, etc. Different cloud providers have their own way of handling deployment and management of microservices/cyber-physical application components. Due to the heterogeneity of cloud providers, it is complex to have holistic management of application components.

Based on the aforementioned challenges, this chapter addresses the following research questions:

- How to monitor the performance of distributed software components of microservice-based applications/cyber-physical applications running on heterogeneous virtualization platforms within the same or different cloud service providers?

- How to aggregate QoS measures of microservice-based applications/cyber-physical applications running in multiple cloud environments to give a holistic view of (e.g., bookshop application or highway traffic) performance?

To answer these questions, this chapter makes following new contributions:

- It introduces a novel system: Multi-virtualization, Multi-cloud Monitoring in multi-microservices/cyber-physical applications that provides a holistic approach to monitor the performance of microservices/CPS applications composed into multiple applications deployed/running in a multi-cloud and heterogeneous environment (e.g. using different virtualization technologies).

- It validates the proposed monitoring system, via a proof of concept implementation that monitors microservices/cyber-physical application performance running across different cloud service providers using different virtualization means. Experimental analysis verifies the efficacy of our proposed monitoring system.

The rest of this chapter is organised as follows: Section 3.2 presents a related work. Section 3.3 presents system design. Section 3.4 presents system implementation. Section 3.5 presents an experimental evaluation. Finally, section 3.6 presents the conclusion.

3.2 Related Work

There are already industry monitoring tools whether in containers [Docker, CAdvisor, Datadog] or in cloud [CloudWatch, Microsoft Azure Fabric]; and academic monitoring tools whether in VMs [23, 74] or even in containers [24, 73].

Docker[19] provides an inbuilt monitoring tool, Docker stats, to examine the resource usage metrics of running containers. The various metrics provided by Docker stats are CPU and memory usage, and actual free memory for each container. However, it does not inspect the performance of individual applications running inside a container. Our proposed system, improves on this significantly by monitoring the performance of each application running inside a container. Along with this, our system also gathers the monitoring information from containers running in heterogeneous cloud environments (e.g. Amazon, Azure, Openstack, etc.). By aggregating the data collected from multiple containers running across multi-cloud environments, one can perform different types of performance comparisons to assess the performance in containers.

CAdvisor [20] is an open source monitoring framework that displays monitoring performance and resource usage in real time. It provides CPU usage, memory usage, network and throughput information of the running containers. One can access the monitoring information only for two minutes duration, as there is no associated storage mechanism that can retain the data for a longer interval. In contrast our system monitors the performance of individual applications that run inside the container/VM and also stores monitoring data in a database shared by both container and VM.

Datadog [21] is a monitoring service that gathers metrics such as CPU utilization, memory, and I/O for all containers. It is an agent-based system that sends data only to the Datadog cloud, making the monitoring job completely dependent on Datadog's cloud. On the other hand our system has the ability to store data in any cloud service provider.

CloudWatch [22] is a commercial cloud monitoring tool that tracks CPU, memory usage, and network but cannot monitor application-level QoS metrics. In addition, it is not platform independent (i.e., it works only for Amazon platform and not for Azure). Similarly, Microsoft Azure Fabric [70] Controller is limited to work only on the Azure platform. Our system, on the other hand, has the ability to monitor applications in heterogeneous cloud environments.

In [23] the authors present CLAMS, an application monitoring framework for multi-cloud platforms. Moreover, their monitoring framework considers different QoS parameters for web-applications running inside a VM. The model retrieves the QoS performance for different cloud layers. However, the model does not monitor the performance of containers. In addition, the model is constrained to only web applications. This is different to our system, which monitors cyber-physical applications that run inside containers and VMs.

In [24] the authors present a framework called PyMon that collects resources like CPU utilization, memory utilization, and network by using Docker container management API. In contrast to [24], our study uses standalone libraries to monitor applications inside the virtualization environment (e.g. containers) and hence can work in heterogeneous environments (e.g. from VM to container). The work published in [73] presents a study between the uses of Virtual Machines and Docker containers comparing the QoS parameters evaluation. They only use Docker containers for their experiments. The authors use the Docker container process to monitor the CPU utilization but they do not validate any application specific parameters of the containers that are being executed.

In [74], the authors present an architecture that collects, analyses and presents the physiological data. Also, it captures data from numerous sensors for further transformation and analysis. This paper is mainly concerned with monitoring particular parameters for performing scheduling in only the cloud environment. In contrast, our system monitors the performance of an application in a holistic cyber-physical system.

As discussed above, existing monitoring solutions do not have the ability to monitor the performance of microservices running inside multi-virtualization heterogeneous cloud environments (container/VM). Our proposed work (*M3*) differs from the aforementioned solutions as it can be used to monitor the performance of microservices running inside containers or VMs distributed across multiple cloud environments.

3.3 System Design

This framework consists of two main components namely a monitoring manager and a monitoring agent. Monitoring agents (represented as $\tilde{A}$) are placed inside containers/VMs that track the performance of underlying applications. A monitoring agent collects the system-level statistics for each service and sends the information to the manager. The manager deployed in a VM/container placed in any cloud, collects the information from different monitoring agents and stores this data in a database for further performance analysis and prediction. The configuration of multi-virtualization (containers/VMs) can be either homogeneous or heterogeneous each of which is provisioned on different cloud providers. Each container/VM may execute one or more services of the same or different types. Figure 3.3 presents a high-level view of the system.

A detailed discussion on the design of the monitoring agent and monitoring manager is given below.

3.3.1 Monitoring Agent

The monitoring Agent is a software component that collects the information from applications running inside (containers/VMs). It has the ability to work in different cloud

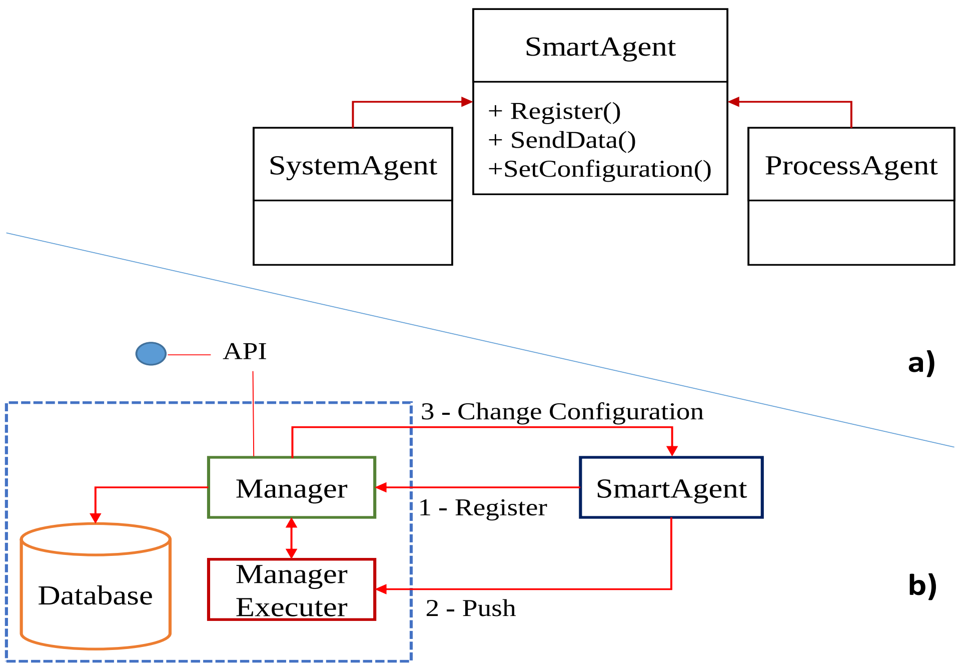


Figure 3.3: Overview of M3 system.

platforms. Agents will wait for requests coming from the manager to push monitoring information to the manager. Our system uses HTTP request for communicating system information between agents and managers. The agents are implemented using the SIGAR (https://github.com/hyperic/sigar/) and RESTLet (https://restlet.com/) libraries that enable them to run on any cloud providers. SIGAR is a multiplatform library (Unix, Windows, Solaris, FreeBSD, Mac OS, etc.) written in Java that provides an API for accessing operating system information while the RESTLet is a Java library that makes it easy to develop HTTP REST APIs.

The system uses SIGAR to obtain the defined system parameters, namely CPU usage, Memory usage, Free Memory, Network usage, etc. RESTLet is used in the development of the services of the monitoring agents that would be accessed by the manager to obtain monitoring data.

The Monitoring Agent is packaged into a jar file and configured to run during the multi-virtualization (container/VM) during boot process. All monitoring agents extend a common agent, called SmartAgent, which consists of two components (*SystemAgent* and *ProcessAgent*) as shown in Figure 3.3 (a). SmartAgent represents a service consisting of three operations: First, agent registration information must be sent to the manger using HTTP PUT request. Second, the agent will send data periodically to the manager using HTTP POST request. Finally, agent configuration will be sent to the manager by using HTTP GET request that can update agent configuration parameters. *SystemAgent* monitors the system as a whole, for example, a container or a virtual machine while *ProcessAgent* monitors the specific process running on that system. The agent utilizes functionalities provided by SIGAR to retrieve the application metrics and other custom built APIs. SIGAR helps in getting the information parameters for the specific application. Using these functionalities, the agent monitors the specified features for each application ID. The agent will start to retrieve the information parameters for this application such as CPU utilization, memory utilization, and so on. The manager utilizes a pull technique that retrieves the information parameters from all the distributed agents and stores them in an SQL database.

3.3.2 Monitoring Manager

The monitoring manager is a software component that receives monitoring information from agents deployed inside (containers/VMs) scattered in the heterogeneous cloud environment, and provides an API for accessing data saved by other services or other applications. Communication between manager and agents is based on pull-based or push-based mechanisms. The manager makes use of the RESTLet library in building the clients accessing the services of the agents. For each registered monitoring agent, the manager starts a thread that coordinates a RESTLet client for access to agent data. Each time the data of a monitor agent is received the manager stores the results in a MySQL database for further access by the graphical management tool.

The sending of information by the monitoring agent to the monitoring manager occurs as a sequence of steps as shown in Figure 3.3 (b): First, agent sends a registration request to the manager, and the manager receives the request and registers the Agent, an access key and an endpoint are sent with the data returning to the agent. Second, the manager executor (uses Push mechanism) is enabled to receive the data sent by the agent using their IP address. Lastly, the agent periodically queries the manager for its configuration (Change Configuration). Dynamic configuration enables real-time agent management.

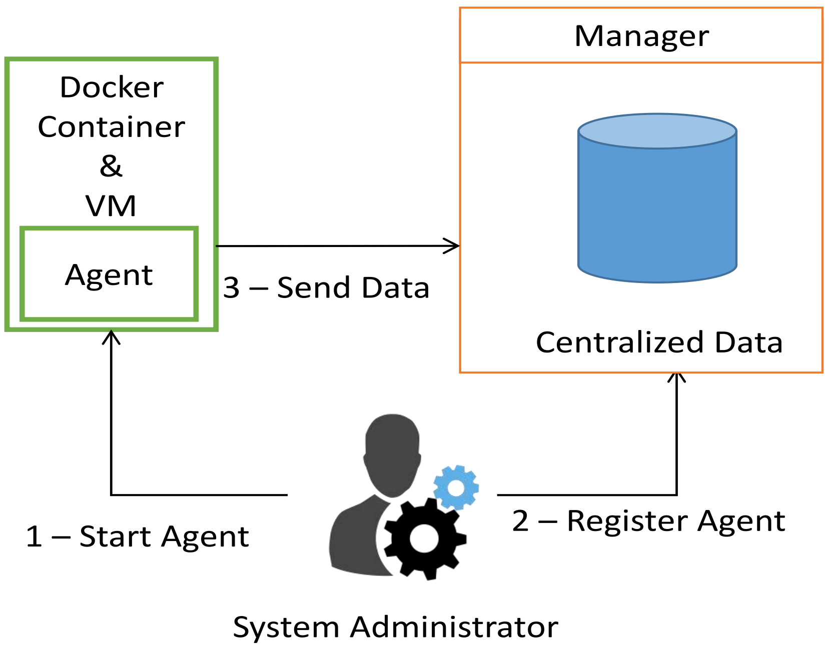


Figure 3.4: *M3* data acquisition model.

The complete monitoring application is represented in the form of a data acquisition model as given in Figure 3.4. It consists of three steps. Initially, the system administrator starts the monitoring agent (Step 1). Subsequently, the administrator registers the agent (An HTTP PUT request registers the agent's IP) to the manager (Step 2). The agent continuously monitors the system (applications, containers, or VMs). Finally, all the monitoring agents send the monitored information periodically to the manager using the HTTP POST request. (Step 3). The manager stores the received data in a shared database and also processes any query (if received) related to the performance of the applications.

3.4 System Implementation

Our proposed monitoring system is implemented in Java and works for both containers and VMs running on any host operating system (Linux, Windows or Mac OS). The agents are implemented using the *SIGAR* (https://github.com/hyperic/sigar/) and *RESTLet* (https://restlet.com/) libraries which enables them to run on any cloud providers. SIGAR is a multiplatform library (Unix, Windows, Solaris, FreeBSD, Mac OS, etc.) written in Java that provides an API for accessing operating system infor-

Table 3.1: List of the configuration parameters and values.

Parameter	**Value**
Cloud service provider	AWS, Azure
Cloud service provider location	USA West, UK South
VM in AWS	t2.micro type
VM in Azure	Standard$_A1_v2$
Operating system	Ubuntu:16.04
Docker platform	Version 17.06.1
Docker-compose	Version 1.18.0
Tomcat	Version 7
Nginx	Version 1.13.7
MySQL	Version 5.7
Java	Version 8
Apache JMeter	Version 5.4.1
Esper	Version 8
SIGAR	Version 1.6

mation while *RESTLet* is a Java library that makes it easy to develop HTTP REST APIs. The system uses SIGAR to obtain various system parameters, namely CPU usage, Memory usage, Free Memory, Network usage, etc. *RESTLet* is used to develop the services for the monitoring agents that allows the manager to access the agents' monitoring data.

There are some strict networking requirements for both manager and agent. The manager must be deployed on a machine with a global IP address, so that the agent can access the manager from any network. Every 30 seconds, the agent queries (GET) the manager to download its configuration file ensuring the dynamic configuration. Communication between the manager and the agent occurs exclusively via HTTP in order to avoid any security or firewall blockages. The manager can dynamically change the agent's data forwarding rate to manage the overload of requests on the network.

List of all the configuration parameters and their values is shown in Table 3.1. We have chosen the latest version available of the selected values at that particular period of time, like MySQL, Tomcat, etc. More details will be discussed in **section 3.4.1 and section 3.4.2**

3.4.1 M3 Implementation

We considered a Book-Shop application as discussed in **section 3.2.1** that is implemented using three types of microservices, namely Tomcat, MySQL and Nginx. These microservices are executed inside either VMs or containers distributed across multiple clouds. The Book-Shop application is distributed into three tiers with User Interface service as the first layer (Web tier), Book and Purchase services in the second layer (Application tier), and finally Storage service (data storage) in the third layer. In User Interface, we considered two microservices (Tomcat, Nginx). In Book and Purchase services, we have either one or two microservices (Tomcat and MySQL). In the Storage service, we have one microservice (MySQL). A User Interface service receives a request from JMeter to communicate with the Book or Purchase services by using HTTP, and the application tier will communicate with the database by using the *JDEC* library (SOCKET network). User Interface receives an HTTP request when selecting a book and forwards this to the Book service. The Book service receives a request, sends a query to MySQL, and returns 500 entities to the User Interface. The Purchase service receives a request from JMeter to save a purchase in MySQL and updates the book entity.

We used Apache JMeter (https://jmeter.apache.org/) to generate HTTP requests to test the capability of *M3*'s system. The test consists of 100 users each having different requests. We generate 900 requests to simulate the users' behaviour. Each request for book select gets 500 entities. We made three tests to capture metrics with the monitoring agent reading data at 1 second, 5 seconds and 10 seconds. The operations and requests made during the experimental evaluation are presented in Figure 5.3. All requests are initiated by Apache JMeter, which simulates the user (req. 1), or simulates another application (req. 4). The choice of the various types of requests is based on the premise of covering the main types of load operations in a data persistence service, namely: data query, insertion and update requests, as well as requests intermediated by a proxy. All requests made by the JMeter are of the "GET" type. The first request flow focuses on query operations and is initiated by request 1 which is directed to the User Interface service. The User Interface receives request 1 through the Nginx web server, which acts as a proxy and forwards it to Tomcat (req. 2). The User

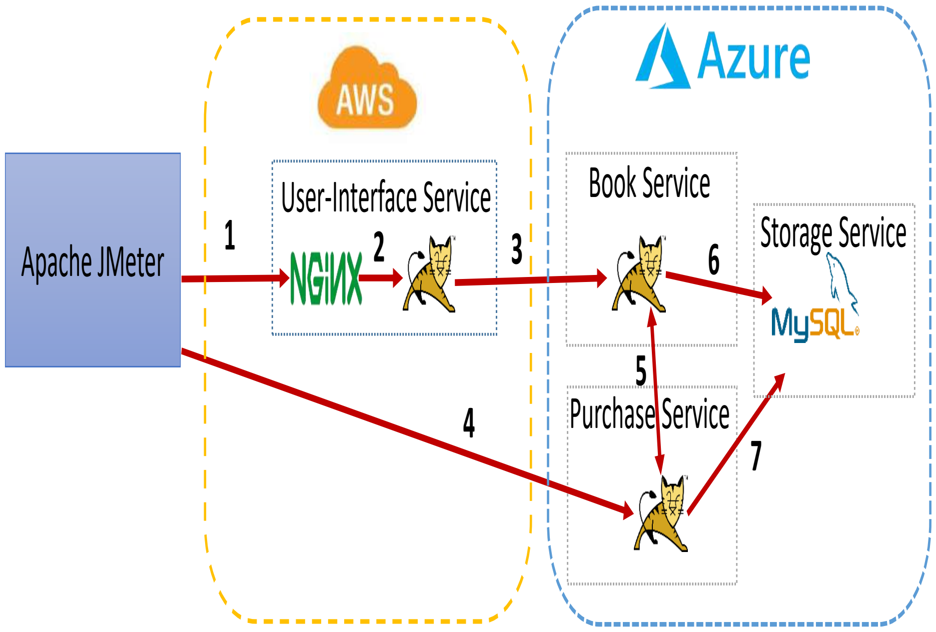


Figure 3.5: Simulation web application pattern.

Interface Tomcat receives the request and creates a new "GET" request to the Book service (req. 3). Request 3 is received by the Tomcat of the Book service that makes a query to MySQL (req. 6). The second request flow is responsible for the insert and update operations. Request 4 is initiated by JMeter and is directed to the Purchase service. Purchase service Tomcat receives the "PUT" request (req. 4) for insertion of a Purchase. This request is decomposed into two others: request 5 and request 7. Request 5 updates the quantity of books in the inventory using the Book service as an intermediary. Request 7 inserts a Purchase into MySQL.

3.4.2 M2CPA Implementation

To validate the M2CPA system, we built a highway monitoring system for automated toll collection, accident alerting, and car counting using a stream data management system. The choice to use these three applications is justified by the need to evaluate the effectiveness of M2CPA in a variety of scenarios running on a distributed and multi-cloud environment and with different virtualisation techniques.

The Highway Monitoring application was built on the basis of work published in [126] which presented the Linear Road Benchmark for evaluating Stream Data Management Systems. Through a simulator called MIcroscopic Traffic SIMulation Laboratory

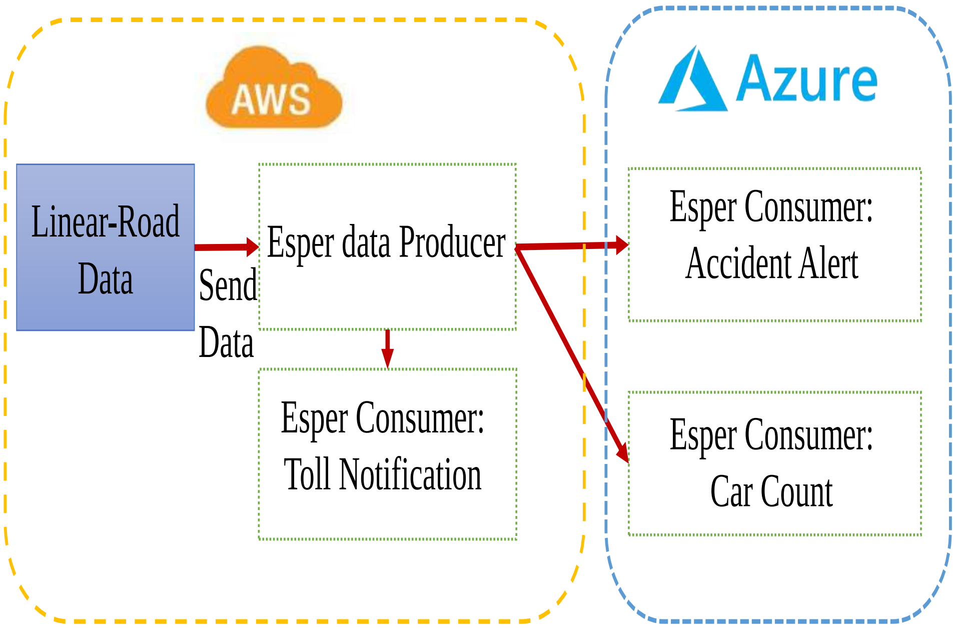

AWS
Azure
Linear-Road Data
Send Data
Esper data Producer
Esper Consumer: Toll Notification
Esper Consumer: Accident Alert
Esper Consumer: Car Count

Figure 3.6: Simulation highway traffic pattern.

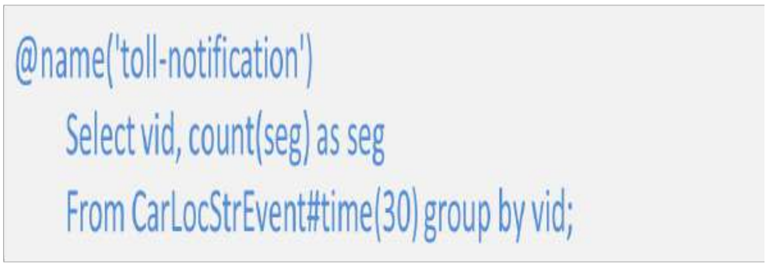
(MITSIMLab) it is possible to construct traffic descriptor files of vehicles that travel on a high-road ((http://www.cs.brandeis.edu/ linearroad/mitsiminstall.html)). The generated data is used as tuples to be sent to the flow processing system. In [126] the authors define some queries that use the data generated in the context of a motorway monitoring application. Following the authors' proposal, we run MITSIMLab and generate a file corresponding to 3 hours of vehicular traffic. We programmed three historical data queries: one for toll billing notification; another to detect accidents; and finally to count the number of cars in each track and segment of the highway in real time. Queries were implemented using Esper ((http://www.espertech.com/esper/)). Esper is a language and an execution machine for processing events and focusses on dealing with high-frequency time-based event data as presented in Figure 3.6.

Queries were built to cover constraints and conditions imposed by the Linear Road Benchmark. Therefore the tuple was used to simulate the position of a car at a certain instant of time. This data was encapsulated in an event composed of the attributes present, and represented the flow of information coming from the positions of the vehicles reported through sensors as shown in Table 3.2.

The data sent is: TIME, VID, SPD, XWAY, LANE, DIR, SEG. TIME represents the

Table 3.2: Input Tuple schemas

Input tuple	**Schema**
Car Position Data	(Time, VID, Spd, XWay, Lane, Dir, Seg)



```
@name('toll-notification')
    Select vid, count(seg) as seg
    From CarLocStrEvent#time(30) group by vid;
```

Figure 3.7: Toll notification query on Esper language.

instant of time in which the information was obtained. VID represents the vehicle identifier. SPD, speed of the vehicle. XWAY on which freeway the car travels. LANE the road strip on which the car is. DIR, the direction, east or west. Finally, SEG represents the segment of the highway from which the position was issued.

The Esper language is based on the data-query pattern defined by SQL-92. For example, to define the toll collection notification (see Figure 3.7), it used a grouping function that counted the number of segments (SEG) reported by the same vehicle in a 30 second time window. In the case of a same vehicle (IE VID) reporting a position of different segments within 30 seconds a toll collection event was triggered.

Following similar concepts, the Car Count query only counts the different VIDs, grouping these results by XWAY, LANE, and SEG. As well, the accident alert query counts the number of vehicles that have zero speed, grouping them by XWAY, LANE and SEG. When the number of vehicles with zero speed in the same tricycle: XWAY, LANE and SEG is greater than two, an accident alert event is generated as presented in Table 3.3.

Table 3.3: Output Tuple schemas: Continuous queries

Query Response	**Schema**
Toll-Notification	(VID, Seg)
Accident-Alert	(Xway, Lane, Seg)
Car-Count	(VID, XWay, Lane, Seg)

3.5 Experimental Evaluation

3.5.1 M3 Experimental

Based on the defined set up as discussed in **Section 3.3.1**, we conducted an experimental evaluation for our proposed monitoring system *M3*. The test application is deployed across Amazon EC2 and Microsoft Azure in both container and VM environments. To demonstrate the effectiveness of the *M3* system, we perform an extensive set of experiments by varying the workload configurations to measure different system parameters, e.g. CPU, memory, latency.

Both Amazon EC2 and Microsoft Azure machines are running Linux Operating System Ubuntu:16.04[1] on which a Docker platform[2] (version $17.06.1 - ee - 1$), was installed to execute the microservices. The VM configuration of Azure is $Standard_A 1_v 2$, with 1 vCPU and 2 GB of memory. We considered four such VMs. The Amazon's VMs were of *t2.micro* type, with 1 VCPU and 1 GB of memory for each machine. Here also we considered two VMs for our experiment.

To emulate the behavior of the Book-Shop application as discussed in the previous section, VMs and containers were installed with different software. For the web server, we chose Tomcat[3] (Version 7) and Nginx[4] (Version 1.13.7) while for Database, we considered MySQL[5] (Version 5.7). All container images used were obtained from the Docker Hub[6] portal.

The machine configurations on which experiments were conducted are as follows: first machine used Java (Version 8) on the virtual machine guest OS, second machine had Docker platform installed and used Docker-Compose file (version 1.18.0) for which we used one image for Tomcat[7] and for MySQL[8], third machine used the same configuration as second machine with different services and the final machine had the Docker

[1]https://www.ubuntu.com/
[2]https://www.docker.com/
[3]http://tomcat.apache.org/
[4]https://nginx.org/en/
[5]https://www.mysql.com/
[6]https://hub.docker.com/
[7]https://hub.docker.com/-/tomcat/
[8]https://hub.docker.com/-/mysql/

Table 3.4: Microservices scenarios deployed at containers and VMs

Environment	Scenario	Containers	VMs
Microsoft Azure Fabric [M] + Amazon Web Services (AWS) [A]	Multi-cloud Virtualization only (S1)		1 - Book/Purchase (Tomcat + MySQL) [M] 1 - User-Interface (Nginx + Tomcat) [A]
Microsoft Azure Fabric [M] + Amazon Web Services (AWS) [A]	Multi-cloud Containers only (S2)	1 - Book (Tomcat + MySQL) [M] 1 - Purchase (Tomcat + MySQL) [M] 1 - User-Interface (Nginx + Tomcat) [A]	
Microsoft Azure Fabric [M] + Amazon Web Services (AWS) [A]	Multi-cloud Cross Containers / VM (S3)	1 - Book/Purchase (Tomcat) [M] 1 - MySQL [M]	1 - User-Interface (Tomcat + Nginx) [A]

platform installed and used Docker-Compose file which consisted of two images: first image for Tomcat and the second image for MySQL. In Amazon, we used two machines, one of them used Java virtual machine, the other installed the Docker platform and the applications using Docker-Compose file which consisted of one image for Tomcat and another for Nginx.

We evaluated the proposed system under the following three scenarios as is shown in Table 3.4:

- **Scenario 01** – Deploying two microservices (Tomcat and MySQL) for Book and Purchase services in one VM deployed in Microsoft Azure (represented as M). In addition, one VM running two microservices (Nginx and Tomcat) for the User Interface service, which is deployed in Amazon Web Services (represented as A). The aim of the scenario was to understand the performance of the application running on multiple clouds with the same virtualization techniques.

- **Scenario 02** – We deployed two microservices (Tomcat and MySQL) for the Book service running in the first container; and two microservices (Tomcat and

MySQL) for the purchase service running in another container; all containers are deployed in Azure (M). In addition to this, we deployed two microservices (Tomcat and Nginx) for the User Interface service in one container which deployed in Amazon Web Services (A). The aim of the scenario was to understand the performance of the application running on multiple clouds with the same virtualization techniques.

- **Scenario 03** – We deployed one microservice (Tomcat) for Book and Purchase services running in the first container and one microservice (MySQL) running Database in another container; all containers are deployed in Azure (M). In addition, one VM running two microservices (Nginx and Tomcat) for the User-Interface service is deployed in Amazon Web Services (represented as (A). The aim of the scenario was to understand the performance of the application running on multiple clouds with multiple virtualization techniques.

We conducted experiments where the manager would push system and process level statistics regarding services running on two public clouds. For results analysis, the metrics obtained for manager were related to all JMeter tests. As mentioned previously, the JMeter tests generate 900 requests to simulate the workload in order to validate the agents' ability to capture performance metrics for all three scenarios.

3.5.1.1 Latency Time Results

M3 measured the average latency time in milliseconds for the workload requests in each scenario (shown in Table 3.5), as well as the agents sending the monitoring information to the manager every 1, 5, 10 seconds respectively. The values captured for latency clearly show the computational difference of multi-virtualization (containers/VMs) case in multi-cloud environments. As we can see, the User Interface service in (S2) has the least average latency (maximal 2.296 for 10 sec) as compared to (S1) and (S3). The reason behind this is that (S2) used container architecture while (S1) and (S3) used VM architecture. Also, for the Book and Purchase service, (S2) gets better performance when compared to (S1) and (S3). Overall, S2 provides the best performance for all

Table 3.5: Request results For all scenarios

Service Name	**Scenario**	**Lat. Average (1 Sec)**	**Lat. Average (5 Sec)**	**Lat. Average (10 Sec)**
User Interface-Amazon	S1	7.984	8.186	8.185
Purchase-Azure	S1	12.65	12.699	12.04
Books-Azure	S1	10.063	10.082	9.381
User Interface-Amazon	S2	1.56	1.567	2.296
Purchase-Azure	S2	16.005	16.107	15.783
Books-Azure	S2	0.152	0.141	0.169
User Interface-Amazon	S3	10.167	10.407	16.868
Purchase-Azure	S3	8.607	7.574	5.088
Books-Azure	S3	16.131	17.097	8.787

scenarios. It shows that the use of container architecture per service in multiple cloud exploits the hardware of the virtual machine more efficiently.

3.5.1.2 CPU Results

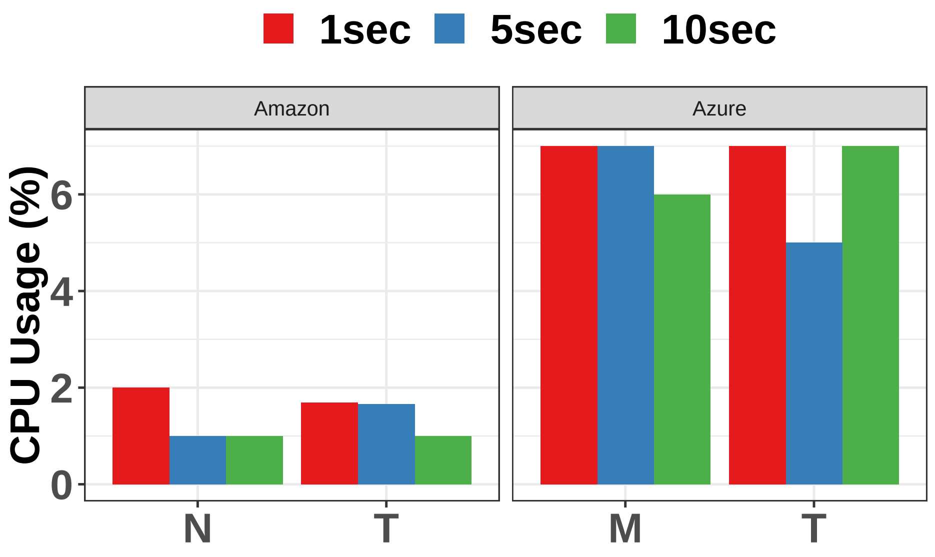


Figure 3.8: CPU usage (percentage) for microservices on: (A) VMs in Amazon and Azure.

The CPU values for all scenarios are shown in Figure 3.8, 3.9, and 3.10. When

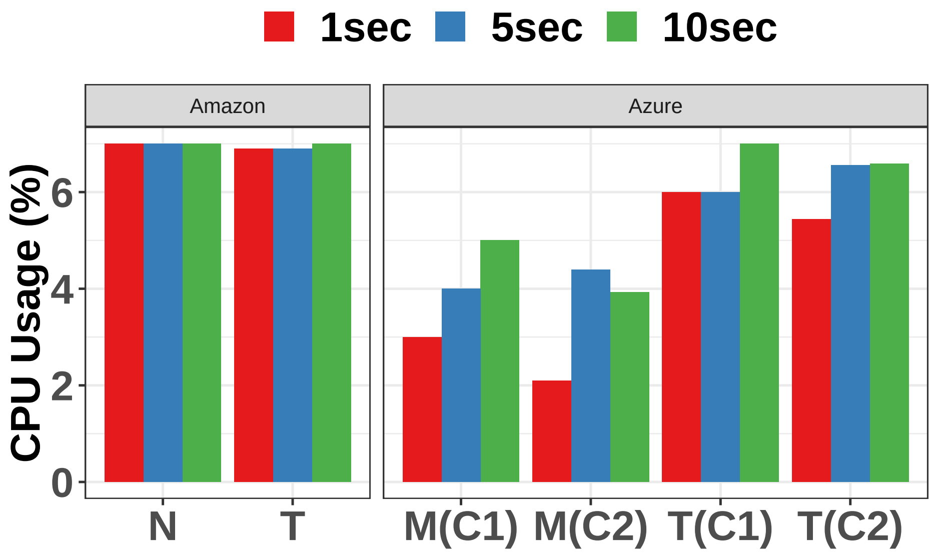


Figure 3.9: CPU usage (percentage) for microservices on: (B) Containers in Amazon and Azure.

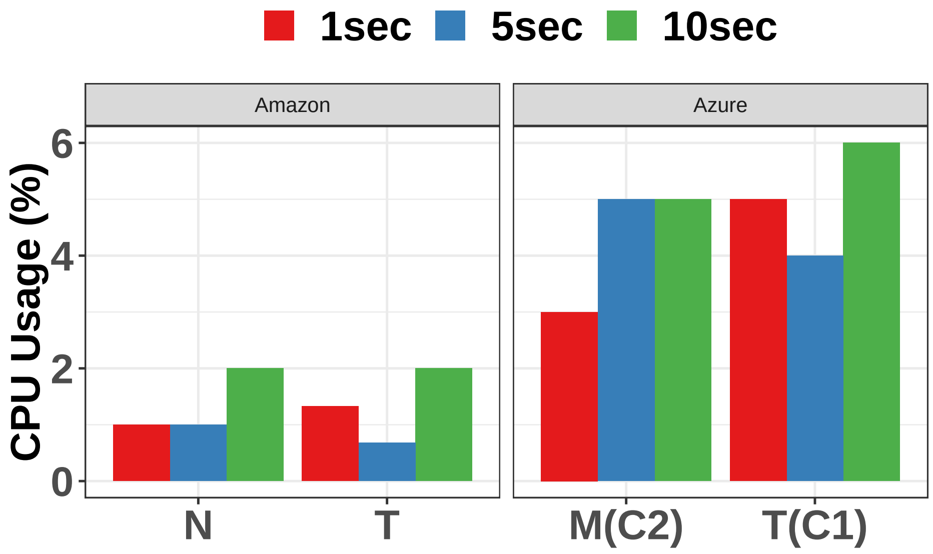


Figure 3.10: CPU usage (percentage) for microservices on: (C) VM in Amazon and two containers in Azure.

analysing S1, for the entire interval of workload test, all microservices were run in VMs, and submitted in Azure and Amazon. The monitoring agents send monitoring information to the manager every 1, 5 and 10 seconds respectively. As shown in Figure 3.8(A), Tomcat microservice of User Interface for 10 sec in Amazon is not affected like that observed during 1 and 5 seconds. The reason behind this is that it has a larger duration as compared to the case when manager sends every 1 or 5 seconds. It shows that the monitoring agents get correct data about CPU for different microservices that reveals the effectiveness of *M3*. The highest average CPU usage is noticed in Amazon for Nginx microservice of User Interface in 1 second with 2.10% and for Tomcat of User Interface in 1 second is 1.80%. In contrast, the highest average CPU usage for all microservices running in Azure is that for MySQL in 5 seconds (7.10%) which is not much different for Tomcat for 1 second duration (7.05%).

For evaluating S2 in Azure containers, the highest average usage of CPU is for Tomcat microservice for the Book service that was running in container (C1) 10 seconds is 6.80%, and not that much different for Tomcat that was running in container (C2) of the Purchase service in 10 seconds which is 6.60%. For MySQL microservice of the Book service running in container (C1) in 10 seconds, the CPU usage is 4.90% and for MySQL that was running in container (C2) for Purchase service in 5 seconds, the CPU usage is 4.10%. However, the average usage of CPU for all microservices running in the Amazon container which consists of Nginx and Tomcat microservices is practically the same in all 1, 5 and 10 second durations with 7.00% as shown in Figure 3.9(B).

For the evaluation of S3, the highest average usage of CPU in Amazon for Nginx microservice of User Interface in 10 seconds is 1.90% and similarly for Tomcat in 10 seconds is 1.90%. However, the highest average usage of CPU for Tomcat microservice for that running in container (C1) in 10 seconds is 5.80% and in 1 second is 5.10%. MySQL microservice running in container (C2) in 10 seconds duration has 5.45% CPU usage and in 5 seconds is 4.50% as shown in Figure 3.10(C).

3.5.1.3 Memory Results

The results obtained for the memory consumption shows the statistics regarding metrics values for the agents monitoring both the public clouds in Figures 3.11, 3.12, and 3.13.

By using *M3*, we can gather fine-grained data from complex multi-tiered applications and can understand the performance of microservices. For instance, in S1 running in Azure VMs as shown in Figure 3.11(A), the highest average memory usage for microservices (Tomcat and MySQL) of Book and Purchase services are practically the

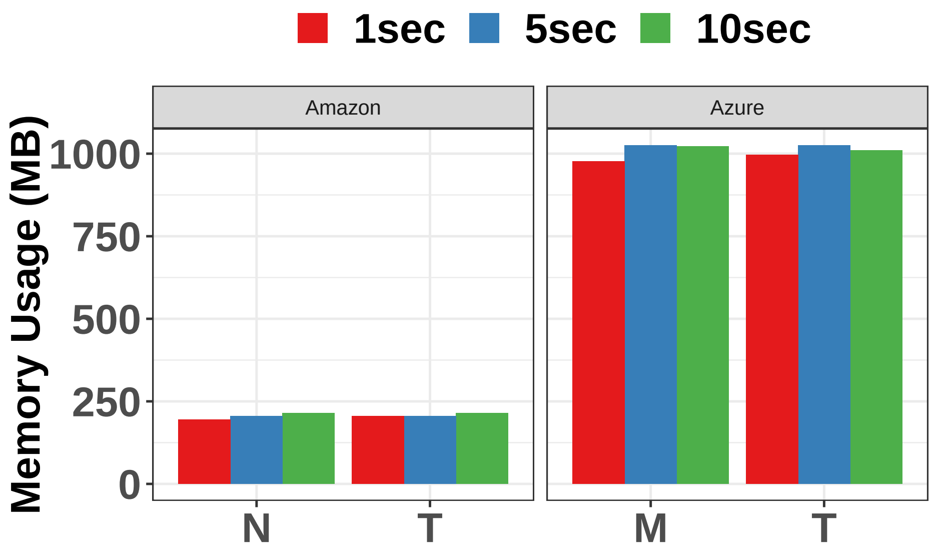


Figure 3.11: Memory usage (MB) for microservices on: (A) VMs in Amazon and Azure.

same in 5 seconds (1025 MB), while in 10 seconds Tomcat uses 1010 MB and MySQL uses 1022 MB from the total memory of the VM which is 1912 MB. Compared to Amazon which is running a VM, the biggest amount of memory used by microservices (Tomcat and Nginx) of User Interface in 10 seconds are the same at a value of 215 MB, while in 5 seconds Nginx used 205 MB and Tomcat used 206 MB from the total allocated memory of the VM which is 992 MB. As shown in Figure 3.11(A), the memory consumption on Azure is larger than Amazon. The larger memory use on Azure could be explained by the difference of virtual hardware configuration between the two clouds. Also, the User Interface service only forwards the requests to the Book service which does more processing because it processes MySQL queries and translates the results to JSON Object which is sent to the underlying services.

For S2 as shown in Figure 3.12(B), running containers in Azure, the highest amount of memory used by MySQL microservice for the Purchase service in container (C2) in 10 seconds is 476 MB and for Tomcat for Purchase service in container (C2) in 10 seconds is 469 MB. Usage for MySQL microservice of Purchase service in container (C1) in 10 seconds is 469 MB and Tomcat for the Purchase service in container (C1) in 10 seconds is the same for Tomcat microservice in (C2) which is 469 MB from the total allocated memory of the container (1920 MB). In contrast to this the highest

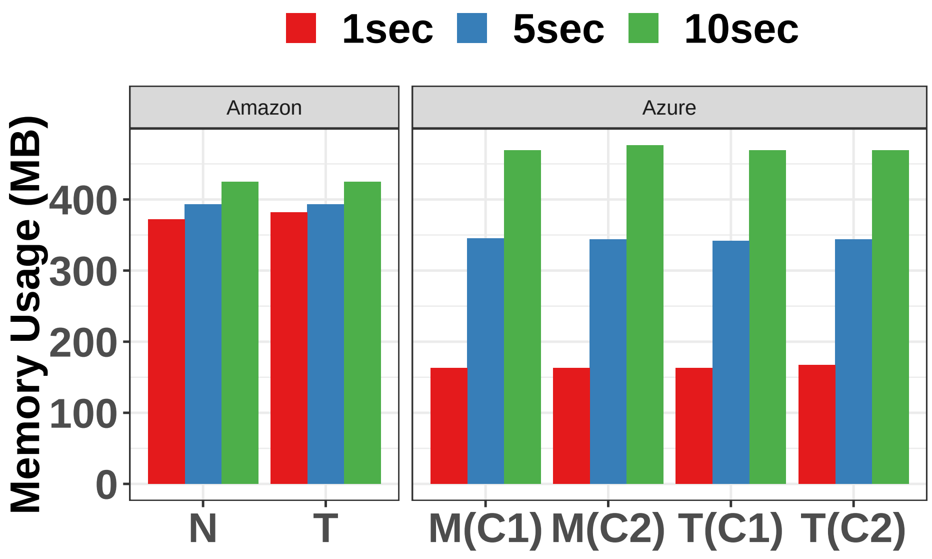


Figure 3.12: Memory usage (MB) for microservices on: (B) Containers in Amazon and Azure.

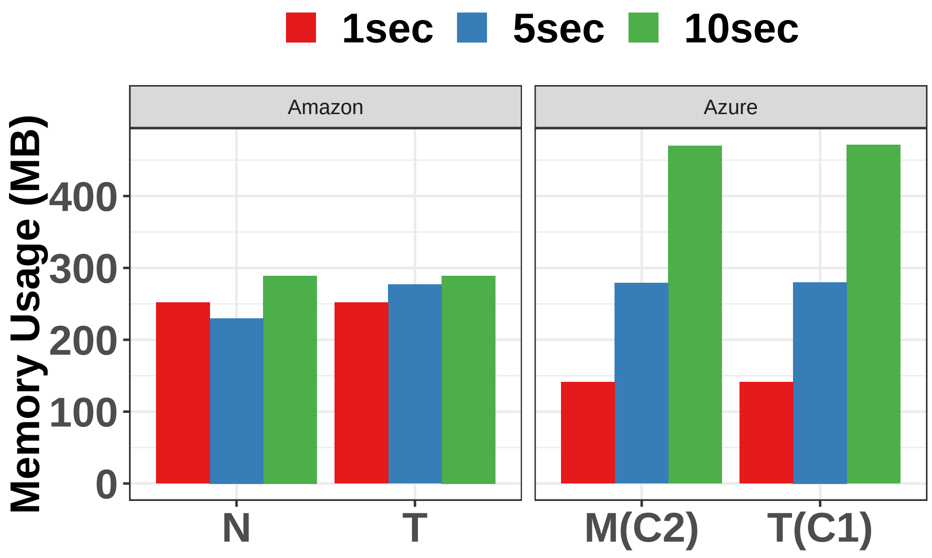


Figure 3.13: Memory usage (MB) for microservices on: (C) VM in Amazon and two containers in Azure.

amount of memory used for all microservices (Nginx and Tomcat) of User Interface running containers in Amazon is the same for both microservices in 10 seconds at 425 MB, which is not much different for 5 seconds (Nginx and Tomcat) where both have the same memory usage which is 393 MB, and in 1 second Tomcat used 382 MB while Nginx used 372 MB from the memory total of the container which is 992 MB.

In S3 as shown in Figure 3.13(C), running in Azure containers, the highest memory usage by the Tomcat microservice for the Book service in container (C1) and MySQL microservice of the Purchase service in container (C2) in 10 seconds is the same (471 MB) while Tomcat in container (C1) in 5 seconds is 280 MB, and MySQL in container (C2) in 5 seconds is 279 MB from the total memory of the container which is 1912 MB. Compared to Amazon running in VM, the average amount of memory used by microservices (Tomcat and Nginx) of User Interface in 10 seconds is the same with the value of 289 MB, Tomcat in 5 seconds is 277 MB, and Nginx in 5 seconds is 230 MB from the total memory size of the VM which is 992 MB.

The collected results show the effectiveness of using the *M3* model in Docker and VM deploying microservices. Our contribution is to validate monitoring multi-virtualization in multi-cloud services as well as the possibility of monitoring individual processes in multi-process containers and VMs running microservices.

3.5.2 M2CPA Experimental

We conducted an experimental evaluation of the M2CPA monitoring system to evaluate its effectiveness and efficiency in monitoring cyber-physical applications running in multi-virtualizations deployed in multi-cloud environments. An application based on a highway data streaming system is deployed in a multi-cloud (Amazon and Azure) environment having both container and VM running it. We test our application by performing an extensive set of experiments using a 3 hour data workload.

We considered both Amazon EC2 and Microsoft Azure clouds where we ran virtual machines using Ubuntu Operating System[9] 16.04 on which the Docker [10] platform, version 17, was installed to execute the application container. The VMs on Azure have the standard A1 configuration, with 1 VCPU and 2 gigabytes (GB) of memory for each machine, which consist of four VMs. The Amazons VMs were t2.micro instance, with 1 VCPU and 1 GB of memory for each machine, which consist of two VMs.

The machine configurations on which experiments were conducted are as follows: first machine used Java (Version 8) virtual machine (used for S1). The second machine

[9]https://www.ubuntu.com/
[10]https://www.docker.com/

Table 3.6: Applications scenarios deployed at containers and VMs

Environment	**Scenario**	**Containers**	**VMs**
Amazon Web Services (AWS) [A]	One-cloud Virtualization only (S1)		1- Linear Road [A] 1- Toll Notification [A]
Microsoft Azure Fabric [M] + Amazon Web Services (AWS) [A]	Multi-cloud Virtualization only (S2)		1- Linear Road [A] 1- Car-Count [M]
Microsoft Azure Fabric [M] + Amazon Web Services (AWS) [A]	Multi-cloud Cross Container / VM (S3)	1- Accident Alert [M]	1- Linear Road [A]

used Java (Version 8) virtual machine (used for S2). The third machine installed the Docker platform (version 1.18.0) and using Docker container that uses one image for Java to run (S3). The final machine used Java virtual machine and used this machine in Linear Road data producer to be consumed by S1, S2 and S3.

The application consisted of a cyber-physical system for monitoring highways. The sensed data (the position of the cars) is sent in a stream of events to be processed by three consumers: Toll Notification, Accident Alert and Car Count. The workload was composed of a file with 3 hours of heavy traffic. Three different scenarios covering different forms of virtualization in a multi-cloud environment (Amazon Web Services A and Microsoft Azure Fabric M) were proposed in order to obtain maximum reach for the various programming models of the cyber-physical system as shown in Table: 3.6:

- Scenario 1 (S1) – A toll Notification Consumer and Linear Road Data producer running on the same cloud service. In our case, this was represented by the deployment of Toll Notification on the same cloud service as the Linear Road Data producer. Both were deployed on Amazon Web Services(A). The aim of the scenario was to understand the performance of applications running on the same cloud service.

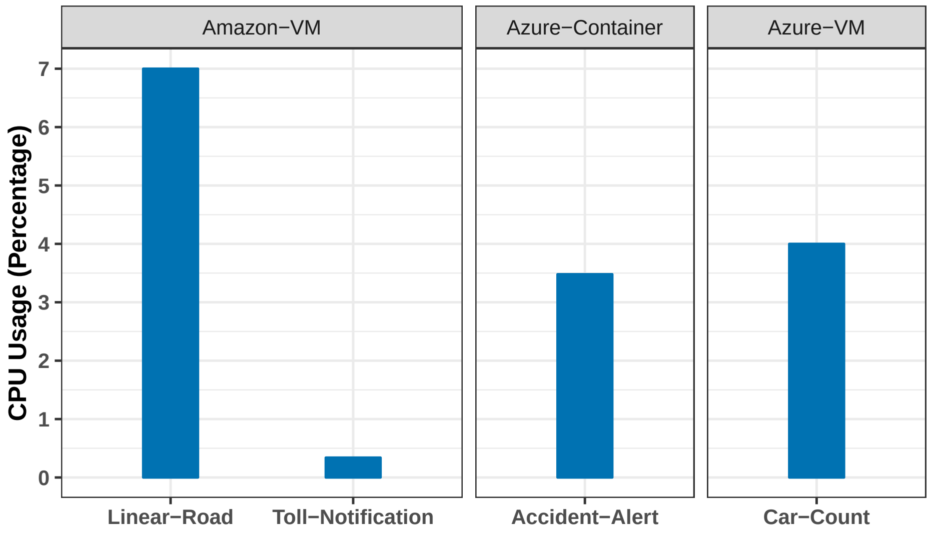


Figure 3.14: CPU usage (percentage) for services on VMs in Amazon, VM in Azure and container in Azure.

- Scenario 2 (S2) – In this scenario we launch two virtual machines, one in each cloud. In Microsoft Azure Fabric (M), we run a car count consumer application. In Amazon Web Services (A), we run a Linear Road Data producer. The aim of the scenario was to understand the performance of the application running on multiple clouds.

- Scenario 3 (S3) – The last scenario serves as an evaluation of the type of virtualization (the data consumer is deployed in a Docker container). Within Microsoft Azure Fabric (M), we run Accident Alert in a container. In Amazon Web Services (A), we run a Linear Road Data producer. The aim of the scenario was to understand the performance of the application running on multiple clouds with multiple virtualization techniques.

We emphasize that the data load generated by the Linear Road Data producer was simultaneously sent to all three consumer applications within scenarios S1, S2 and S3.

3.5.2.1 CPU Results

The CPU values for all scenarios is shown in Figure 3.14. The monitoring agents send monitoring information to the manager every 5 seconds. As shown in Figure 3.14 the average usage of CPU (%) for the Toll Notification service was 0.36%. For the Accident Alert service, in the Azure container, the average usage of CPU was 3.48%. However, the average usage of CPU for the Car Count service running in VM was 4.00%. The Linear Road producer that was run in VM, and submitted in Amazon; had a bigger CPU usage of 7.00% because of continuous reading of 3 days worth of data from file and parsing this data using Esper Event to be sent to all consumers.

3.5.2.2 Memory Results

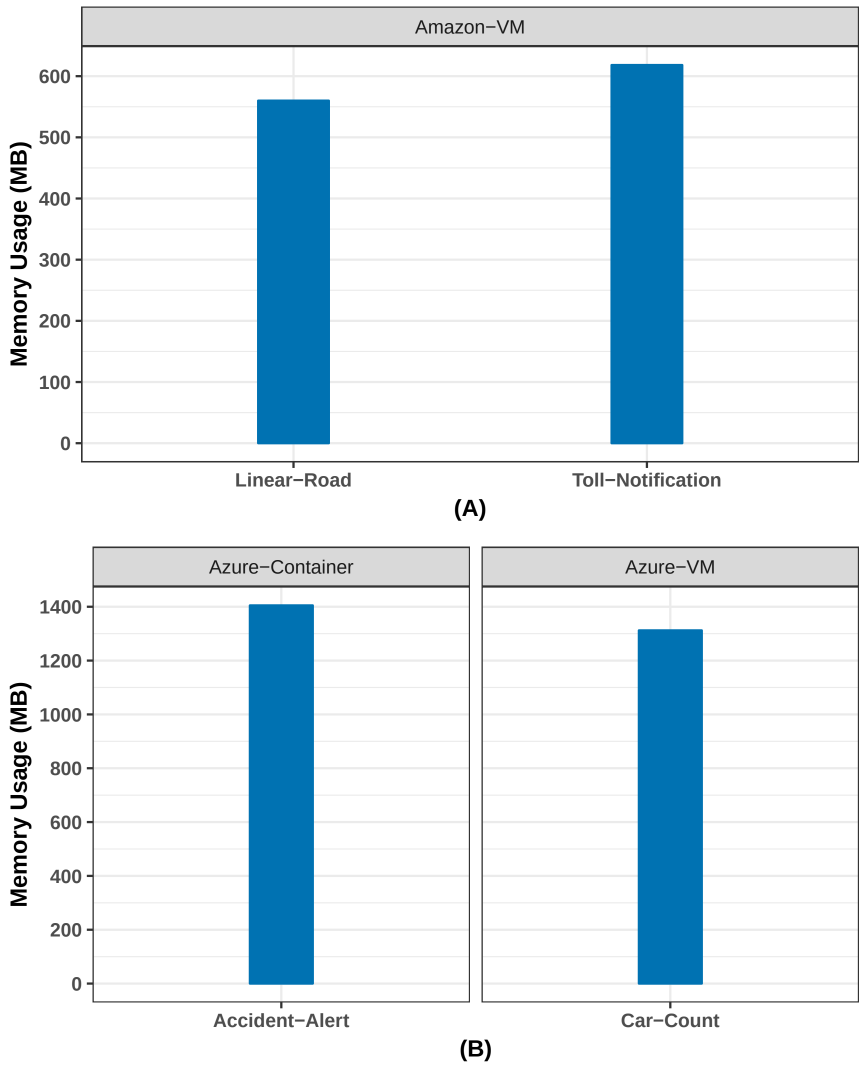


Figure 3.15: Memory usage (MB) for services on VMs in Amazon (A), VM and container in Azure (B).

Figure 3.15 shows memory usage results for agents monitoring services running on both public clouds. The average memory usage for the Toll Notification service that

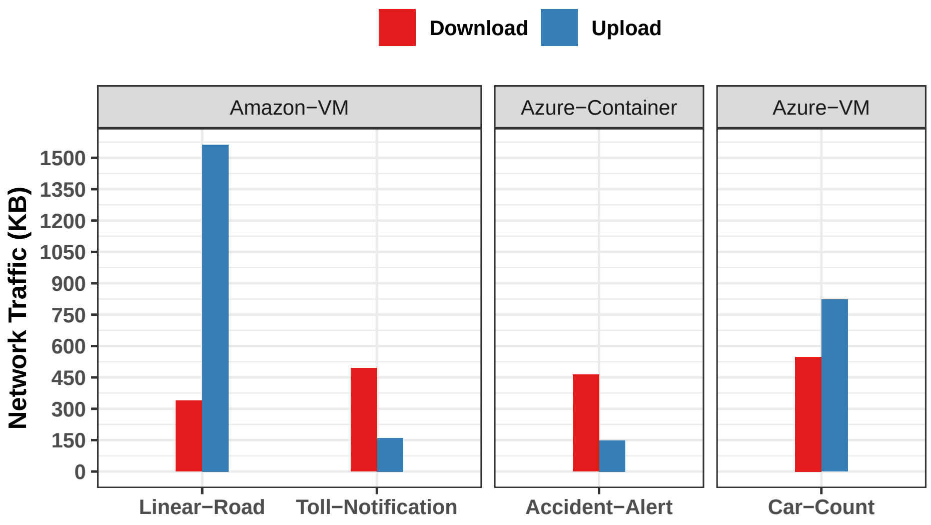


Figure 3.16: Network traffic (KB).

is running in Amazon VM was 618 MB from a memory total of 992 MB as shown in Figure 3.15(A). On the other hand, the memory consummation on Azure is larger than on Amazon. The larger memory use on Azure cloud can be explained by the difference of virtual hardware configuration between the two clouds. When running a container in Azure, the average memory usage for the Accident Alert service was 1405 MB as shown in Figure 3.15(B). This is from a memory total within the container of 1920 MB. Further, the average memory usage for the Car Count service running within a VM in Azure was 1312 MB (as shown in Figure 3.15(B)). This is from a memory total for the VM of 1936 MB. The Linear-Road Data producer was run in VM, and has an average of memory usage of 559 MB as shown in Figure 3.15(A). This is from a memory total for the VM of 992 MB.

3.5.2.3 Network Results

Figure 3.16 shows Network usage results obtained from agents monitoring the network traffic of the services. In the Toll Notification service, Car Count service, and the Linear Road data service (workload of a file with 3 hours of heavy traffic), the download and upload rates of a VM or container are presented. For the the Accident Alert service the download and upload rate of the container are shown. The results show that the traffic caused by using a 3 hours data workload, was detected and verified by the monitoring system. The network traffic of the Toll Notification service running in an Amazon cloud VM, was 495 KB for download and 161 KB for upload. The network traffic of the Accident Alert service running in an Azure cloud container, was 464 KB for download and 149 KB for upload. The network traffic for the Car Count service running on an Azure cloud VM, was 548 KB for download and 823 KB for upload. The Linear Road producer service had a network traffic of 399 KB for download, and 1563 KB for upload. This high upload is expected because it is sending the same data 3 times to all other services running on multiple clouds.

3.5.2.4 Results Summary

In the previous sections we clearly see the effectiveness of our M2CPA framework in accurately monitoring the individual components of a cyber-physical application distributed across multiple clouds using muliple virtualisation means including VMs and containers. The M2CPA framework was able to calculate and report accurate performance metrics of CPU usage, memory usage, and Network usage for 3 scenarios of a traffic monitoring application. Our work improves significantly on current monitoring tools in that it provides a unique combination of features that include a) monitoring the performance of cyber-physical application sub components running inside individual containers and individual VMs, b) gathers monitoring information from applications/sub-applications running inside heterogeneous cloud environments (e.g, Amazon, Azure, Open Stack, etc) and aggregates the results via an agent based system, c) stores monitoring data in a database shared by both containers and VMs, d) monitored data can be stored and accessed on any cloud provider.

3.5.3 Cost Analysis

We performed the experiment to calculate the CPU, memory and other resource consumption of monitoring agents and the manager when the number of microservices increases by a power of 2 (e.g. 1, 2, 4, 8, 16, 32). Also, we tried to capture the behaviour of agents (e.g. how many agents are there).

In order to measure the overhead caused by the manager, an experiment is conducted in which the manager process is monitored for CPU and memory usage while an increasing number of concurrent agents were registered. The amount ranged from 2 to 64 concurrent agents. The results obtained from the performance manager of increasing number of agents are plotted on the CPU and Memory as shown in Figures 3.17

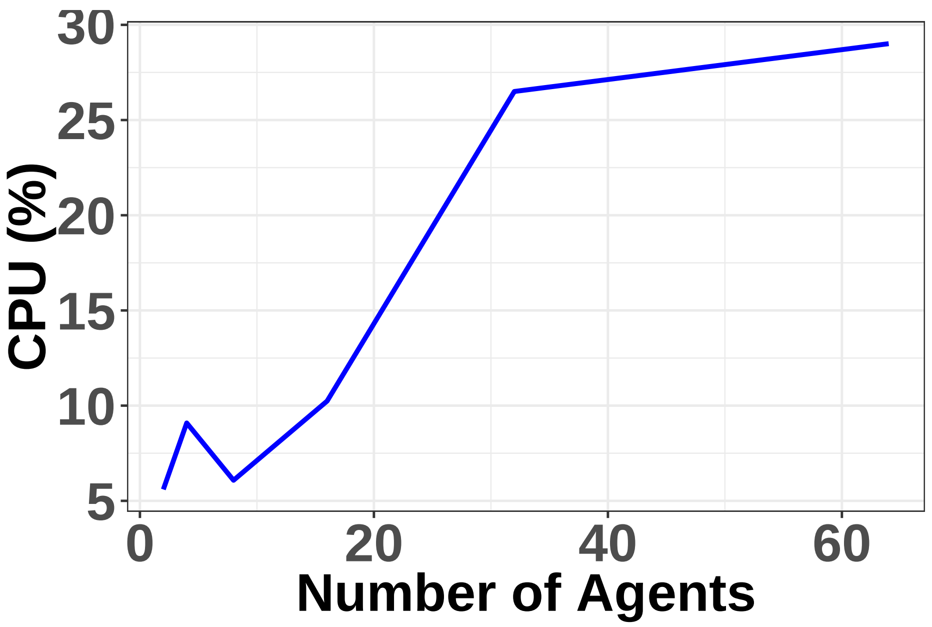


Figure 3.17: CPU usage (percentage) for manager.

and 3.18 . The results show that the increase in the number of agents affects both the

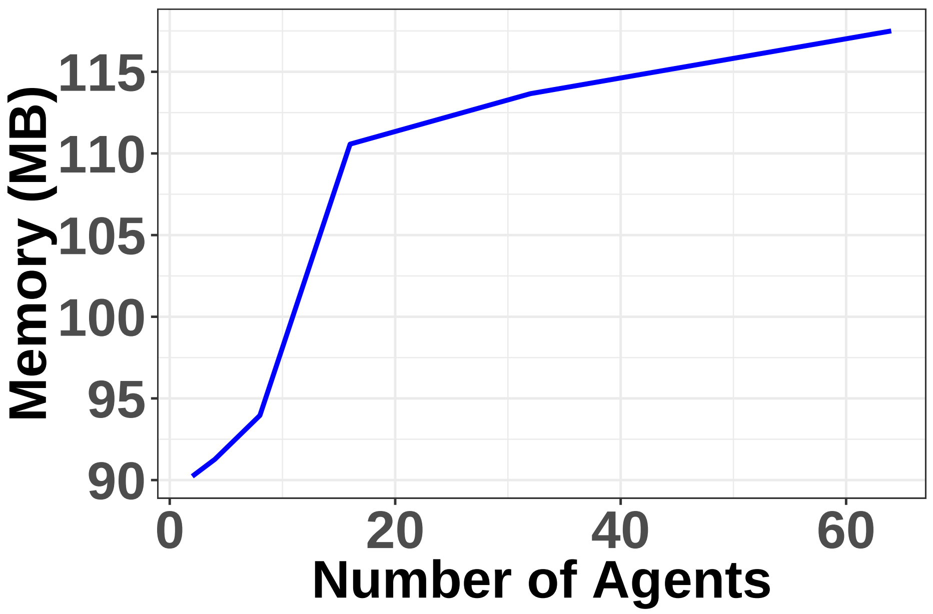


Figure 3.18: Memory usage (MB) for manager.

increase in CPU usage 3.17 and memory usage 3.18. CPU utilization increases by 20% between 2 to 64 concurrent agents, with a more significant increase from 16 to 32 agents. The use of memory has a more linear behaviour presenting a 27 MB increase from 2 to 64 concurrent agents.

As shown in Figures 3.17 and 3.18 manager utilized overhead values in CPU and memory. The reason of that, an increasing number of concurrent agents were registered by manager and it will affect of resource consumption since the manager will collect metrics from each agent that the manager registered it. The values that shown in Figures 3.17 and 3.18 are acceptable, once the number of agents increase; the resource utilization of manager will be increased.

3.6 Conclusion and Future Work

With the anticipated advent of new computing and networking technologies, we can expect to see billions of more devices being connected to the Internet as part of microservice/ cyber-physical systems for critical applications such as smart healthcare, and smart cities. Developing reliable monitoring frameworks that can accurately assess the performance of such critical applications is extremely important. But with the number of components, and complexity of such applications expected to increase, monitoring their performance accurately and efficiently becomes more challenging.

In this chapter, we propose and deploy *M3* – a novel system for efficient and effective monitoring of applications based on multi-virtualization (containers/VMs) multi-microservices deployed in multi-cloud environments. The proposed solution provides users the ability to monitor the performance of microservices that run inside containers and VMs, and report their metrics performance in real-time. The solution uses an agent-based architecture in order to scale from a centralized to a decentralized architecture to suit the demands of monitoring such complex services-based applications. We developed a proof-of-concept implementation of the proposed solution using a Book-Shop application with Docker containers and VMs deployed in Amazon and Azure cloud environments. The proposed system was evaluated under diverse scenarios with evaluation outcomes validating the effectiveness of *M3* in the monitoring of microservices in multi-virtualization multi-cloud environments. In addition, we significantly extended *M3* by propose, develop and validate M2CPA – a novel framework for efficient monitoring of cyber-physical applications based on multi-virtualization (containers/VMs) and multi-cloud environments. The proposed solution provides users the ability to monitor the performance of cyber-physical applications that run inside containers and VMs and report their metrics performance in real-time. We developed a proof-of-concept implementation of the proposed solution using Docker containers and VMs deployed on Amazon and Azure clouds. The proposed system was evaluated using experimental analysis that considered diverse scenarios with evaluation outcomes validating the effectiveness of M2CPA in monitoring the performance of cyber-physical applications in a multi-virtualized and multi-cloud environment.

In the future, we will collect a large set of data using *M3* from production-ready systems to develop efficient deployment and orchestration strategies for microservices. Also, we will expand the framework to monitor physical devices and application container migration to develop efficient deployment and orchestration strategies for cyber-physical applications.

4

MADLS: Monitoring and Anomaly Detection and Localization System of Container-based Microservice

Contents

Summary

This chapter presents a Monitoring and Anomaly Detection and Localization System (MADLS) which utilises a simplified approach that depends on commonly available metrics offering a simplified deployment environment for the developer. Our data collection system uses a monitor engine that monitors response time and throughput in the application layer as well as CPU and memory in the physical layer. We evaluate our approach through a bookstore web application case study. We finally validate *MADLS* to show that *MADLS* can accurately detect anomalies in response time then specify the type and location of fault that exists in microservices.

4.1 Introduction

Microservice deployment is now the standard business model for hosting cloud-based web applications across online industries from streamed media to Big Data [1]. This promotes the risk that wide-scale system disruption may impact many services if anomalous activity detection is not achieved sufficiently early. This problem is compounded as the beneficial aspect of cloud delivery is the ability to scale, in real-time, to satisfy increasing client demand.

Microservice architecture brings many advantages to web applications, most importantly is scalability. However, scaling adds a level of complexity for performance monitoring. Performance monitoring may lead to severe system outages [18] if they are incapable of successfully determining anomalies from actual scaling requests. For example, on November 18th, 2014, the Microsoft Azure Storage service in multiple regions interrupted. As a result, dependent services experienced a connectivity loss with the Azure Storage service, which caused an interruption in these dependent services [70].

Monitoring performance is a key research activity in cloud computing, which has seen a significant rise in importance due to the popularity of microservices. Their popularity is primarily based on their light-weight, economical usage of shared resources allowing for greater fiscal reward from service hosting activities while reflecting green agendas

[127] [128]. However, this puts significant importance on securing such services as resource misuse, often manifesting as anomalous behaviour in the network activity, can have impacts on distinct hosting provisions of unrelated services.

Scalability of a container-based microservice application allows for more application organisation and optimisation. But, this strongest characteristic brought new complexities to application performance monitoring. Indeed, inefficient performance monitoring may lead to severe application outages [18] as it is not able to successfully and quickly detect the failures and then localising their causes. Application failures are observed by the user and are considered as indicators for the problems in application behaviour (also referred to as anomalies). These anomalies can be caused by different faults in the application resources and called root causes. Therefore, anomaly detection and root causes localization is aimed at linking the observed failures with the underlying faults. For instance, Central Processing Unit (CPU) consumption fault may cause a response time delay failure in the application.

Research conducted to study approaches to improve a monitoring system for microservice-based web applications [129–131] is ongoing but has yet to provide a generalised solution that can be widely adopted. As anomalous behaviour is only detectable in its deviation and sometimes complex correlation, expected to scale activities, supervised machine learning techniques are a popular choice for anomaly detection. However, by using this technique, detecting performance anomalies and localizing the anomalous microservice are challenging. Since the supervised machine learning technique needs a large amount of data for training purposes, it will detect only those types of anomalies defined by the dataset. This brings a significant challenge in tackling this research area: a) numerous microservices, b) infrequent anomalies and c) numerous metrics as explained below:

Numerous Microservices, microservice-based web applications consist of a large numbers of microservices deployed in across a number of hosts. Monitoring every individual microservice is challenging and time-consuming due to the intricate, above OS, dependencies in resource sharing [132]. This makes machine learning the only realistic way of achieving autonomy [133]. However, as this has no human-in-loop checks inappropriate actions by this service itself can be destructive to services.

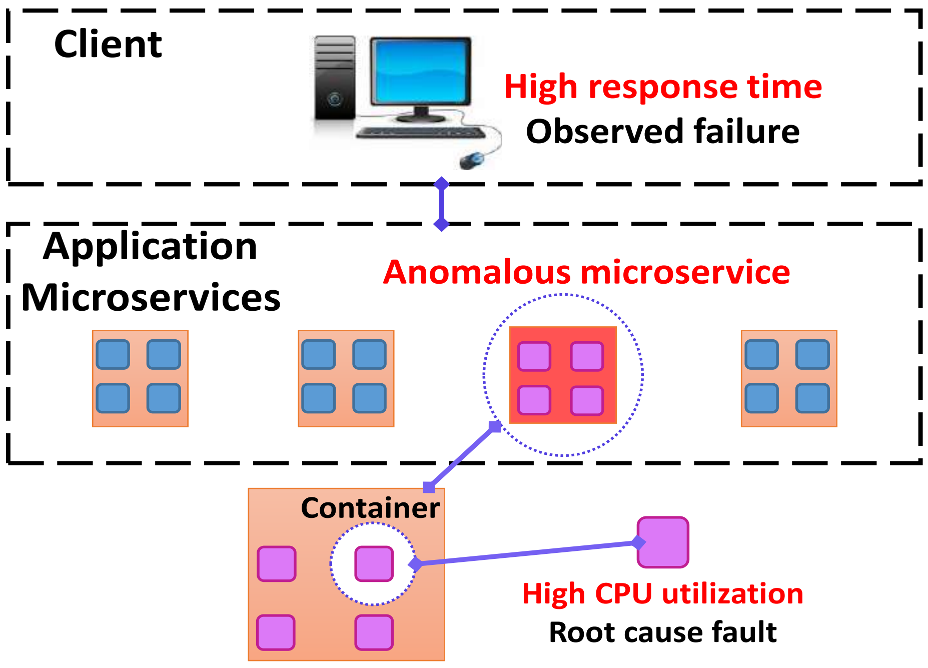


Figure 4.1: Multi-resolution technique

Infrequent Anomalies, anomalies are not a common event within cloud environments [33]. This is problematic for those approaches based on supervised machine learning as there exists limited examples of anomalous training data on which to learn. This poses the main problem of either missing anomalies that are not in the dataset or, if the system is highly tuned, identifying anomalies that do not exist. This causes significant performance disruption to the overall infrastructure as resources are directed to containment activities that do not exist and will disrupt correctly working services.

Numerous Metrics, datasets used for supervised learning can consist of many metrics, promoting the use of a variety of different approaches based on metric availability. Such metrics may not be available generally across service providers. For example, Uber application has 500 million metrics [26]. This may cause a considerable decrease in accuracy of supervised machine learning algorithms bringing challenges for root causes localization [34].

To address the challenges mentioned above, we provide a system for multi-resolution [118] method to track the root cause of observed failure via collecting metrics about the overall performance of a container-based microservices application. Consider, for example, a book-shop application that consists of several microservices deployed in multiple containers. A root-cause localization of a detected anomaly (e.g. observed high response time) can be performed by analysing resource metrics of microservice's containers such as CPU, memory, disk, and network. Analysis of all observed metric and underlying resourses metrics from all microservices consists of a large volume of metrics. We use multi-resolution method (see Figure 4.1) to analyse the metrics in sequential stages to best formulate a learning environment for machine learning algorithms. The multi-resolution method is achieved by narrowing the metrics' scope and increasing the resolution using classification machine learning algorithms. Lightweight algorithms are used for computational purposes to analyses several representative metrics to detect the observed anomalies (e.g. High response time in book microservice). Once the anomalies are detected, a comprehensive drill-down analysis of additional metrics to localize the root cause using complex algorithms (e.g. High CPU utilisation in book microservice container).

Our contribution: This chapter proposes a **M**onitoring and **A**nomaly **D**etection and **L**ocalization **S**ystem (*MADLS*) which offers an efficient approach for detecting and localizing anomalies. This approach can be easily deployed by developers to interpreter the causes of performance failures in a container-based microservice application. We use a monitor engine to collect two sources of metrics: observed metrics (e.g. response time and throughput) from microservice level and resources metrics (e.g. resource utilization) from container level. We use the multi-resolution technique to detect anomalous microservices and localize the root causes. The multi-resolution technique can analyze an application performance in sequential stages using machine learning techniques to narrow the metrics to be analyzed in each stage. As a result, the accuracy of detecting and localizing the root cause of anomalies can be improved significantly. We evaluate our system by conducting several experiments using a bookstore web application case study. We evaluate our system by conducting several experiments using a bookstore web application case study. We finally validate *MADLS* to show that it can accurately detect anomalies in response time then specify the type and location of fault that exists in a microservices with a high accuracy of performance at 90%.

The rest of this chapter is organized as follows: Section 4.2 presents justifies our contribution through a comparison to related work. Section 4.3 presents our Monitoring and Anomaly Detection and Localization System (*MADLS*). Section 4.4 presents a description of how we implemented *MADLS*. Section 4.5 describes our evaluation and results analysis. Finally, section 4.6 presents the conclusion and future work.

4.2 Related Work

There have been a variety of methods applied for detecting anomalous behaviour and analyzing the root causes in microservice-based cloud application.

Carla Sauvanaud et al. [28] introduce an anomaly detection system (ADS) that considers performance data for microservices to detect anomalies. ADS can identify the anomalous microservice and the type of anomaly within the observed microservice. This chapter only applies the system on Virtual Machine-based microservices rather than the containerised approach popular today, which we consider key tackling in our approach. Qingfeng Du t al. [58] also propose an Anomaly Detection System (ADS) but do so for container-based microservices. The proposed ADS has a monitoring module that collects performance data from containers as well as a data processing module based on machine learning. This system can identify the type of anomaly within the anomalous microservice.

Both these works analyse metrics of resource utilization in the infrastructure layer. Still, they do not consider the link of observed failures with underling faults in during anomaly detection as we do. We consider using the multi-resolution technique to analysis metrics in sequence as beneficial, especially for improving the detection accuracy of machine learning algorithms.

Li Wu et al. [27] propose MicroRCA system that correlates failure observation of an application performance corresponding its root cause faults in resource utilization in real-time. The attributed graph is used to analyze the root causes by modelling anomaly propagation among microservices of the application. Juan Qiu et al. [89] also use knowledge graph technology and a causal search algorithm to diagnose the root cause of application performance. Yuan Meng et al. [29] present a framework called MicroCause. Their approach localizes the root cause of low-performance indicators in

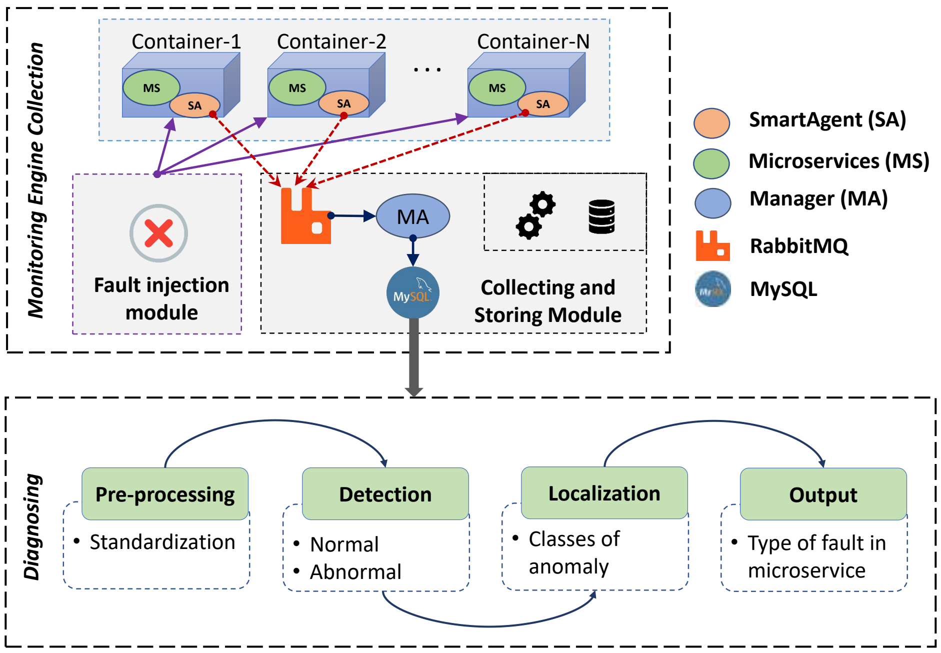


Figure 4.2: Monitoring and Anomaly Detection and Localization System (MADLS).

a microservice. MicroCause useS path condition time series (PCTS) algorithm and temporal cause-oriented random walk(TCORW) method. Jörg Thalheim et al. [26] deploys a platform, called Sieve, to monitor microservice performance metrics and analyze the root causes of observed bad performance. Sieve has two core tasks to apply root-cause analysis: first Sieve filters out metrics that present normal behaviour and keep other metrics. Second, Sieve uses a predictive-causality model to find metrics dependencies of microservices. Wei Cao et al. [90] apply Conditional Random Field(CRF) method for microservice anomaly detection. The method creates the microservice fault matrix by collecting microservice metrics as an observation sequence. Therefore, anomalies of a microservice can be obtained from the microservice fault matrix.

These approaches search the cause root of anomaly observed behaviour using different tracking algorithms (e.g. tree or matrix); however, our approach is to study the feasibility of machine learning algorithms to root cause localization.

4.3 Monitoring and Anomaly Detection and Localization System (*MADLS*)

The underlying intuition behind the MDLS is two-fold. First, monitoring a container-based microservice application and collecting metrics from microservice level and container level. Second, diagnosing performance issues using the multi-resolution technique [118]. The metrics from the two levels continuously collected. Once anomalies are detected in microservice level, MADLS performs root causes localization of the anomalies based on analyzing container level metrics. The Figure 4.2 demonstrates the proposed system and it will elaborately discuss the system components in appreciable details:

4.3.1 Monitoring and Collecting Engine Component

1. Monitoring agents: The capture of the data defined by the metrics is performed through monitoring agents, condensed in the conceptual representation of SmartAgent (SA) in Figure 4.2. There are two types of agents: an agent for monitoring the metrics related to containers and another for monitoring those related to microservices.

- **Microservice monitoring**: A microservice failure is an observable event when the microservice does not work properly. Microservices are also monitored to obtain information on the microservice API, response time, and processing rate, as well as the metrical measurements obtained by tracking containers where operating systems are located.

- **Container monitoring**: An anomalous microservice could be caused by its underlying container faults that are hidden from the user. Container monitoring agent monitors hidden metrics of containers related to containers' resource utilization, e.g. CPU utilization, memory utilization, disk use and network use. Hidden faults can be the root cause of an observed failure.

2. Collecting and storing module: All data captured by the monitoring agents are sent to the messaging service (represented by *RabbitMQ* in Figure 4.2), which transfers: 1) to consumer managers (Manager in Figure 4.2) for storage in the data service (*MySQL* on Figure 4.2)) to interested parties (apps, dashboards, etc.) in monitoring.

- **Message service**: The message service can be implemented by any message-broker, currently in our system *RabbitMQ* was used. *RabbitMQ* is an open-source message-broker software (sometimes called message-oriented middleware) that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and other protocols [1].

- **Manager consumers**: Consuming managers are small processes that act as consumers of the monitored data, performing a parse of the data for the storage service. These agents currently transcribe data in data queries to the *MySQL* database system. Other managers can easily be built in storage in other data services.

- **Database service**: The data service allows storage for later consultation of the monitored data. In the current system, it is represented by the *MySQL* database system. Any storage service such as (Google Cloud Database, AWS RDS) or RRD or a time series storage system like *InfluxDB* could be used with their respective consumer managers.

4.3.2 Diagnosing Component

The diagnosing component in the system is responsible for anomaly detection and root cause localization. The complexity of microservices diagnosing depends on the amount of monitoring metric from different microservices. Therefore, multi-resolution is used to improve the performance of microservices diagnosing. To perform the multi-resolution approach, diagnosing component is segregated into the following modules (see Figure 4.2): pre-processing module, detection module, localization module. These

[1]https://www.rabbitmq.com/

modules are executed in a sequential manner. The description of these modules as follows:

1. The pre-processing data module: The pre-processing module processes the collected data from the monitoring engine stored by the database service. Pre-processing methods include filtering the data, labelling the data and standardization. Details of pre-processing methods and their purposes will be provided in the implementation section.

2. Detection module: This component detects anomalies in response time and throughput metrics collected from the microservice level. A metric threshold was set to determine if the microservice behaviour is abnormal or normal. Therefore the response time, as well as the throughput, can be classified into two groups: abnormal and normal. The abnormal response time that indicates microservice failure varies from 501 milliseconds to one second. Abnormal response time impedes the performance of the throughput. When the response time is shorter than or equal to 500 milliseconds, this is considered to be a normal state. The detection component uses a binary machine learning classification algorithm to categorise response time behaviour of microservices to either normal or abnormal. Thus, anomalous microservices can be detected.

3. Localization module: After anomaly detection, the localization component locates the root causes of an anomalous microservice by analyzing the underlying metrics collected from container level. These metrics include CPU utilization and memory utilization. The fault in these resource utilization considers a root cause of an anomalous microservice. If the resource utilisation rate reaches 50%, the fault exists. A multi-classification machine learning algorithm is deployed to find faults in resource utilisation of microservice's container.

4. The output of anomaly-detection-module: As stated above, the anomaly requires a multiclass algorithm that determines the different classes of resource anomalies of each microservice. Some examples include an anomaly in the CPU of the microservice x or an anomaly in the memory of of the microservice y.

Table 4.1: List of the configuration parameters and values.

Parameter	**Value**
Cloud service provider	AWS
Cloud service provider location	USA West
VM in AWS	t2.micro, t2.medium, t2.xlarge types
Operating system	Ubuntu:16.04
Docker platform	Version 17.06.1
Docker-compose	Version 1.18.0
Javassist	Version 3.26-GA
Java	Version 8
Apache JMeter	Version 5.4.1
RabbitMQ	Version 5.6.0
SIGAR	Version 1.6
Python	Version 3.9.6

4.4 MADLS Implementation

List of all the configuration parameters and their values is shown in Table 4.1. We have chosen the latest version available of the selected values at that particular period of time, like MySQL, Java, etc. The deployment of the *MADLS* modules on the containers is illustrated in Figure 4.3. The modules are described in the following:

4.4.1 Monitoring Engine Module

A monitoring agent is deployed on each of the container. We used *MADLS* monitoring agent, for collecting and storing performance metrics. *MADLS* collects resource usages and performance monitoring data of all the containers and stores in the database as a time series. The implementation of the system was carried out in Java (version 8), consisting of four main parts (see Figure 4.3), namely: construction of monitoring agents external to the code, development of internal agents to the code, and coding of managerial consumers for data storage. The external agents to the code, Outside Agents (OAg), were built using the SIGAR[2] libraries to obtain data about the system and the executing processes or used by the microservices. For example, a microservice that uses Tomcat as a web server and *MySQL* as a database, would need at

[2]https://github.com/hyperic/sigar

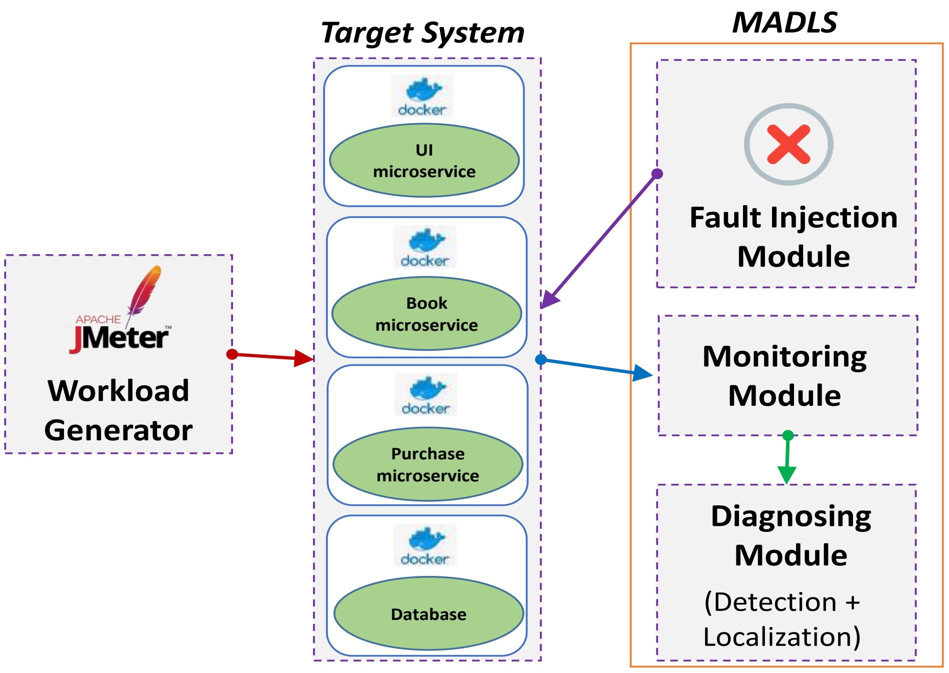


Figure 4.3: MADLS Implementation.

least three external agents: 1) for the system (Disk, Network, System-CPU, etc.), 2) for Tomcat (Tomcat-CPU, Tomcat-Memory, etc.) and 3) for *MySQL* (MySQL-CPU, MySQL-Memory). The encoding of external agents was based on a solution previously published in [132], changing the form of communication from REST to Publish / Subscribe. For this purpose, AMQP producers compatible with *RabbitMQ* were built, through the library *com.rabbitmq.amqp-client* (version 5.6.0), which carried out the publication of the data obtained by monitoring on message topics previously created in *RabbitMQ*. *RabbitMQ* was the AMQP server chosen for the use of the system, exercising the role of message-broker between producer agents and consumer managers. Just to clarify, when we started an agent to monitor the CPU usage of a process, his AMQP client opened a connection to the *RabbitMQ* server and started sending data to the "Process" queue. The data finally sent would be collected from the consumer managers or by the consultation consumers.

The agents inside the code, Inside Agents (IAg) used the same way of communication as the OAg, that is, AMQP clients send data to pre-registered topics in *RabbitMQ*. The difference between the IAg and the OAg in terms of implementation is the use of a library to manipulate the application's Bytecode, the org.javassist.javassist (version 3.26-GA). The manipulation of Bytecode was used in order to facilitate the use of the system by other developers, in addition to allowing a minimal change to the original code of the microservice. Basically, the system development team produced a library based on Java annotations to build the metrics defined in the microservice codes. In other words, currently, two metrics related to the microservice code are defined, namely: Response_Time and Throughput. For the developer to monitor the Response_Time, just insert the Java annotation *@Response_Time* in your code and include the IAg library within your project's classpath. Once the code is noted, the IAg must be started with the JVM to run the microservice, using the *–javaagent* option. In this way, it allows the JVM in the pre-execution process of the main thread of the microservice, to include the initialization execution of the IAg which searches the methods signed with the annotations*@Response_Time* or *@Throughput* to include the specific monitoring code. For example, for the annotation *@Response_Time* a variable is included at the beginning of the method to obtain the time in nanoseconds of the system, and at the end this variable is used in the calculation of the response time of the microservice method.

Managerial consumers also used the *com.rabbitmq.amqp-client* (version 5.6.0) library to subscribe to topics and receive messages sent by agents. The metrics received by them were persisted in a *MySQL* database. The *MySQL* database system represents the long-term storage service of the temporal-series data obtained by the continuous monitoring of microservices.

4.4.2 Target Application

An evaluative experiment of the *MADLS* system was conducted in order to assess the effectiveness in monitoring and alerts of the implemented metrics. In addition, the values obtained in the execution of the analysis constituted a set of 48 hours of data that serve as input for the data processing and fault injection modules. The target application of our evaluation was a composition of three microservices: User Interface (UI), Book Service (BS) and Purchase Service (PS). They integrate an application to control purchases and inventory of an electronic book store. The UI represents the microservice that processes Javascript and HTML content for building the web pages

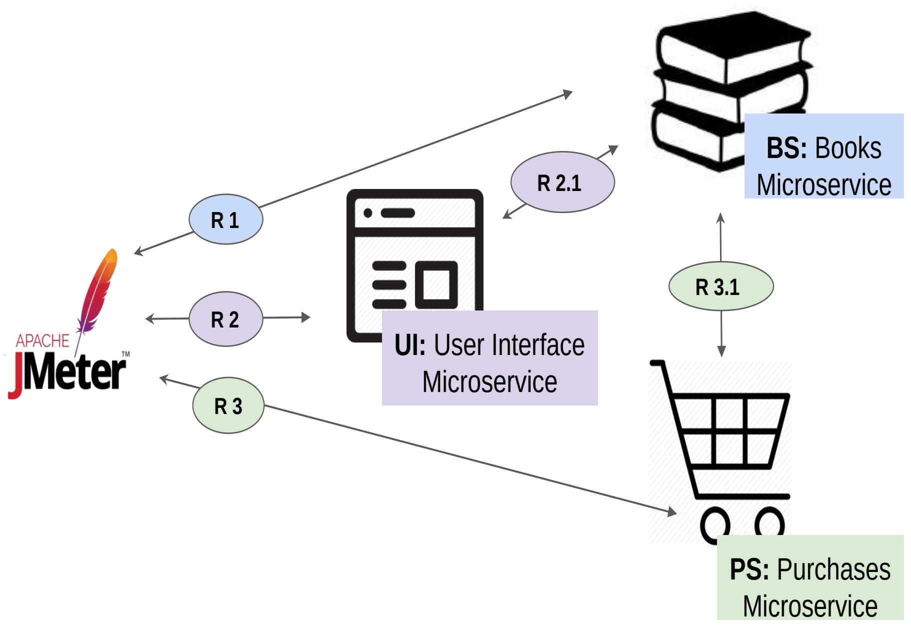


Figure 4.4: **Experiment Requests**: Books-Shop microservices communication flow and JMeter's requests sequence.

of the electronic book store. To simplify application coding, the UI service does not have any database for storing user or shopping cart attributes, which are simulated in memory only for the purposes of the experiment. In turn, the BS represents the service that processes and stores information about books and their respective stocks. Thus, it has a *MySQL* database for exclusive storage for the contextual entity of the books domain. Similar to BS, the PS stores the purchase data and also has its *MySQL* database with the only table, *Purchases*.

We used Apache JMeter (https://jmeter.apache.org/) to generate HTTP requests to test the capability of *MADLS*'s system. The operations and requests made during the experimental evaluation are presented in Figure 4.4, namely: 1) For R2 request, the UI when receiving a book listing request (R2) redirects this request to the BS. The BS upon receiving the book listing requests (R2.1) returns a list of books in JSON format that is converted to HTML format in the UI which finally returns to JMeter; 2) For R1, JMeter sends a detailed information request on 10 random books (R1), BS processes this request with its *MySQL* bank and returns a list in JSON with the requested data; 3) For R3, JMeter sends a request to include a book purchase for the PS. The PS consults the BS (R3.1) to check if the book has available stock and finally saves the purchase in its *MySQL* bank. The three simulated requests are sent continuously starting with 10 users initially and gradually increasing up to 150 simultaneous users in an interval 48 hours.

BookShop microservices were implemented in Java (version 9), making use of the Restlet[3] library to build its APIs. The microservices were implemented using the technology of Docker containers (version 18.09) with the aid of Docker Compose tool (version 1.24.1). We have deploy all Book Shop Application microservices and the framework infrastructure for our experiment in Amazon Virtual Machines cloud EC2 services that running Ubuntu version 18.04). All microservices were encapsulated in Jars files that were used in the construction of Docker Images based on OpenJDK 9[4]. To mitigate the possibility of influencing hardware differences, virtualization in the performance results obtained, all containers had limited resources using directives cgroup[5], namely: *deploy: resources: limits: cpus: '0.50', memory: 256 MB.*

4.4.3 Training ML Models for Diagnosing

4.4.3.1 Data Pre-processing

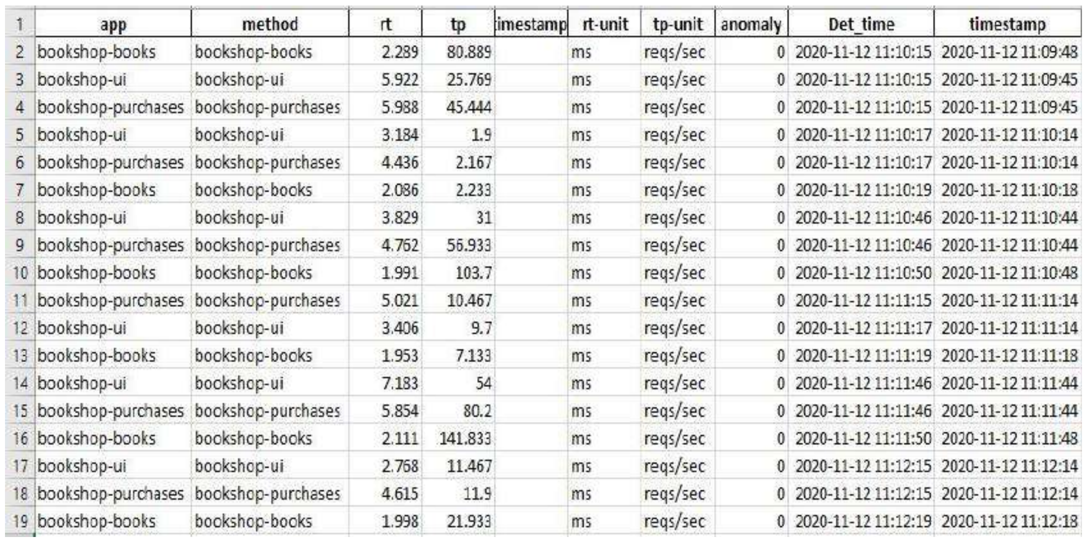
The *MADLS* monitoring engine collects two datasets. The first dataset (DS1) contains both response time (rt) and throughput (tp) for the microservice. In the second dataset (DS2), the resource usage metrics including central processing unit (CPU) and memory (Mem) for the microservice process are collected. To prepare the data, our data pre-processing procedure applies labelling, downsampling and standarization. Figure 4.5 and 4.6 show a snapshot of the dataset (D1 and D2).

Table 4.2: Anomaly detection by detection component (binary classification)

Class label	**Class**	**Description**
0	Normal	Normal response time and throughput
1	Anomaly	High response time and low throughput

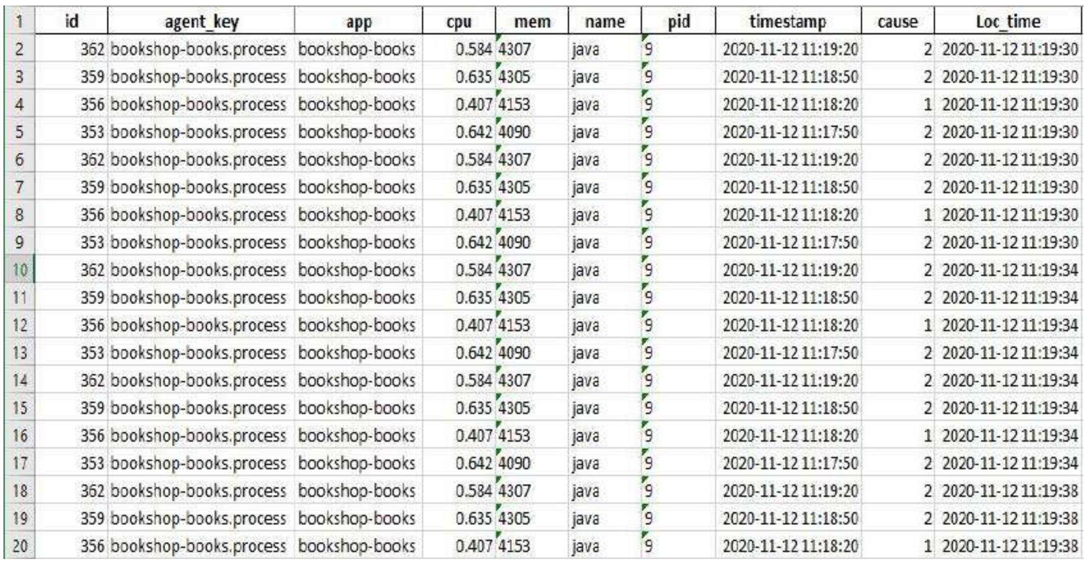
Labelling: for supervised machine learning, the dataset should be labelled first before training. For that reason, the fault injection component allows two fault forms to be

[3]https://restlet.talend.com/ (org.restlet.jse)
[4]https://hub.docker.com/_/openjdk
[5]docker.com/compose/compose-file/



1	app	method	rt	tp	imestamp	rt-unit	tp-unit	anomaly	Det_time	timestamp
2	bookshop-books	bookshop-books	2.289	80.889		ms	reqs/sec	0	2020-11-12 11:10:15	2020-11-12 11:09:48
3	bookshop-ui	bookshop-ui	5.922	25.769		ms	reqs/sec	0	2020-11-12 11:10:15	2020-11-12 11:09:45
4	bookshop-purchases	bookshop-purchases	5.988	45.444		ms	reqs/sec	0	2020-11-12 11:10:15	2020-11-12 11:09:45
5	bookshop-ui	bookshop-ui	3.184	1.9		ms	reqs/sec	0	2020-11-12 11:10:17	2020-11-12 11:10:14
6	bookshop-purchases	bookshop-purchases	4.436	2.167		ms	reqs/sec	0	2020-11-12 11:10:17	2020-11-12 11:10:14
7	bookshop-books	bookshop-books	2.086	2.233		ms	reqs/sec	0	2020-11-12 11:10:19	2020-11-12 11:10:18
8	bookshop-ui	bookshop-ui	3.829	31		ms	reqs/sec	0	2020-11-12 11:10:46	2020-11-12 11:10:44
9	bookshop-purchases	bookshop-purchases	4.762	56.933		ms	reqs/sec	0	2020-11-12 11:10:46	2020-11-12 11:10:44
10	bookshop-books	bookshop-books	1.991	103.7		ms	reqs/sec	0	2020-11-12 11:10:50	2020-11-12 11:10:48
11	bookshop-purchases	bookshop-purchases	5.021	10.467		ms	reqs/sec	0	2020-11-12 11:11:15	2020-11-12 11:11:14
12	bookshop-ui	bookshop-ui	3.406	9.7		ms	reqs/sec	0	2020-11-12 11:11:17	2020-11-12 11:11:14
13	bookshop-books	bookshop-books	1.953	7.133		ms	reqs/sec	0	2020-11-12 11:11:19	2020-11-12 11:11:18
14	bookshop-ui	bookshop-ui	7.183	54		ms	reqs/sec	0	2020-11-12 11:11:46	2020-11-12 11:11:44
15	bookshop-purchases	bookshop-purchases	5.854	80.2		ms	reqs/sec	0	2020-11-12 11:11:46	2020-11-12 11:11:44
16	bookshop-books	bookshop-books	2.111	141.833		ms	reqs/sec	0	2020-11-12 11:11:50	2020-11-12 11:11:48
17	bookshop-ui	bookshop-ui	2.768	11.467		ms	reqs/sec	0	2020-11-12 11:12:15	2020-11-12 11:12:14
18	bookshop-purchases	bookshop-purchases	4.615	11.9		ms	reqs/sec	0	2020-11-12 11:12:15	2020-11-12 11:12:14
19	bookshop-books	bookshop-books	1.998	21.933		ms	reqs/sec	0	2020-11-12 11:12:19	2020-11-12 11:12:18

Figure 4.5: Snapshot of the dataset (D1).



1	id	agent_key	app	cpu	mem	name	pid	timestamp	cause	Loc_time
2	362	bookshop-books.process	bookshop-books	0.584	4307	java	9	2020-11-12 11:19:20	2	2020-11-12 11:19:30
3	359	bookshop-books.process	bookshop-books	0.635	4305	java	9	2020-11-12 11:18:50	2	2020-11-12 11:19:30
4	356	bookshop-books.process	bookshop-books	0.407	4153	java	9	2020-11-12 11:18:20	1	2020-11-12 11:19:30
5	353	bookshop-books.process	bookshop-books	0.642	4090	java	9	2020-11-12 11:17:50	2	2020-11-12 11:19:30
6	362	bookshop-books.process	bookshop-books	0.584	4307	java	9	2020-11-12 11:19:20	2	2020-11-12 11:19:30
7	359	bookshop-books.process	bookshop-books	0.635	4305	java	9	2020-11-12 11:18:50	2	2020-11-12 11:19:30
8	356	bookshop-books.process	bookshop-books	0.407	4153	java	9	2020-11-12 11:18:20	1	2020-11-12 11:19:30
9	353	bookshop-books.process	bookshop-books	0.642	4090	java	9	2020-11-12 11:17:50	2	2020-11-12 11:19:30
10	362	bookshop-books.process	bookshop-books	0.584	4307	java	9	2020-11-12 11:19:20	2	2020-11-12 11:19:34
11	359	bookshop-books.process	bookshop-books	0.635	4305	java	9	2020-11-12 11:18:50	2	2020-11-12 11:19:34
12	356	bookshop-books.process	bookshop-books	0.407	4153	java	9	2020-11-12 11:18:20	1	2020-11-12 11:19:34
13	353	bookshop-books.process	bookshop-books	0.642	4090	java	9	2020-11-12 11:17:50	2	2020-11-12 11:19:34
14	362	bookshop-books.process	bookshop-books	0.584	4307	java	9	2020-11-12 11:19:20	2	2020-11-12 11:19:34
15	359	bookshop-books.process	bookshop-books	0.635	4305	java	9	2020-11-12 11:18:50	2	2020-11-12 11:19:34
16	356	bookshop-books.process	bookshop-books	0.407	4153	java	9	2020-11-12 11:18:20	1	2020-11-12 11:19:34
17	353	bookshop-books.process	bookshop-books	0.642	4090	java	9	2020-11-12 11:17:50	2	2020-11-12 11:19:34
18	362	bookshop-books.process	bookshop-books	0.584	4307	java	9	2020-11-12 11:19:20	2	2020-11-12 11:19:38
19	359	bookshop-books.process	bookshop-books	0.635	4305	java	9	2020-11-12 11:18:50	2	2020-11-12 11:19:38
20	356	bookshop-books.process	bookshop-books	0.407	4153	java	9	2020-11-12 11:18:20	1	2020-11-12 11:19:38

Figure 4.6: Snapshot of the dataset (D2).

injected into three different microservices. Therefore, DS1 has two class labels: normal and anomaly. Table 4.2 shows the description of these labels. Whereas DS2 has five classes include memory fault in book-service, CPU and memory fault in book-service, memory fault in purchase-service, CPU and memory fault in purchase-service, and CPU fault in the user interface. As microservice UI is non-storage microservices we did not consider any anomaly in its memory. Besides, Because Book microservice and Purchase microservice are CPU and memory intensive services, high CPU utilization could cause high memory usage. Therefore, we consider CPU fault associated with memory fault. However, high memory utilization does not necessarily consume the CPU. Table 4.3 provides a detailed summary of these five classes.

Table 4.3: Summary of 5 fault classes to be classified by localization component (Multiclassification)

Class label	**Class**	**Description**
1	Memory in book service	There is high memory consumption in book microservice
2	CPU and Memory in book service	There are high CPU and high memory consumption in book microservice
3	Memory in purchase service	There is high memory consumption in purchase microservice
4	CPU and Memory in purchase service	There are high CPU and high memory consumption in purchase microservice
5	CPU in UI service	There is high CPU consumption in User Interface microservice

Downsampling: The datasets are highly unbalanced because the number of anomalies is very low, resulting in data skewness. THis means the number of anomalies are much less than the number of normal states, where that renders the task of classification extremely difficult and brings many challenges in supervised machine learning. Therefore, Downsampling approach is used to remove samples from the majority class. As a result, we have DS1 that consists of 600 samples, where 300 samples are normal response time, and the other 300 samples are anomalies (High response time). In DS2, we have 300 samples of root causes, where each of the five classes listed in Table 4.3 has 60 samples.

Standardization: The set of values differs significantly for the four features (response time, throughput, CPU, memory). So, by implementing the StandardScalar standardization process, we normalize all of the features. Standardization is a scaling strategy that the values have standard deviation equals to one and mean equals to zero. Standardization is essentially required by algorithms that use the Euclidean distance method to calculate the distance between two points.

4.4.3.2 Anomaly Detection

The anomaly detection is responsible for detecting an anomaly in observed metric (i.e. response time and throughput). It classifies DS1 which is a temporal-series data of response time and throughput obtained by the continuous monitoring of microservices into two classes (Table 4.2): normal and anomaly. This is achieved by binary machine learning classification models.

Binary classification models are trained with four algorithms (i.e. KNN, DT, LR and NB) to detect the high response time and normal response time. The dataset DS1 of 600 samples was split into both training set and test set. Two thirds of the samples represent the training set which contained 400 samples (200 normal and 200 anomaly samples). The last third is the test set, consisting of 200 samples with 100 samples of anomalies and 100 samples are normal.

4.4.3.3 Root Cause Localization

Four algorithms (i.e. KNN, DT, LR and NB) are used to train the multi-classifying machine learning models to identify the anomalous microservice and, thus, to assess the root cause of the anomaly. For example, memory fault in microservice X. For this purpose, the dataset DS2 of 300 samples of anomaly data is divided into training and test sets. The training set contains 2/3 of the samples, 200 (40 samples of each class). The last third of the remaining samples are 100 samples (20 samples for each class).

The four machine learning algorithms were trained using the Scikit-learn[6], which is a free software machine learning library for the Python programming language. All algorithm with the default configuration of the Scikit-learn library.

[6]http://scikit-learn.org

4.4.4 Fault Injecting Module

Fault injection is a method created by researchers and engineers to evaluate the dependability of the hardware or software of computer systems. The fault injection at the software is less expensive in comparison to hardware as it requires changes at the software-state level. For this reason, it is easy to test the software system's higher level mechanisms. Figure 4.3 shows the fault injection environments that consists of two different modules; CPU and memory faults.

- Central processing unit fault: The CPU fault aids errors in the erratic behavior of the container. For example, if the container does not respond adequately or hangs up if heavy loads are run, the performance will be low. It is considered a CPU hog, which can result from increased user demands.

- The memory-fault: In a container, memory use decreases in a short period. Once the memory is thoroughly exhausted, a memory leak can occur. It is called a memory loss. It is called an incident, leading to a response time.

As illustrated in Alg. 1, we implemented these faults into the data collection system one by one to see the behavior of the system in the presence of the faults and to train our model using machine learning techniques in various system conditions. The real anomalies of the system, high CPU and high memory consumption, are simulated after the faults injections. Table 4.4 gives a summary of symbols used in the chapter.

Algorithm 1 demonstrates the proposed fault injection method. The type of fault $\mathcal{F}$ (CPU or Memory), the container in which the error will be injected ($\mathcal{C}$), the duration of the fault ($\mathcal{T}$), and the pause time of the error ($\mathcal{P}$) are defined by the user. Workload ($\mathcal{W}$) is the core component of the algorithm defined before. An iterative process is executed continuously to build a Pascal triangle until the defined time is reached for CPU fault injection. For memory error injection, the algorithm allocates 1 MB byte arrays until memory utilization reaches 90% and occupies the memory with this rate until the time reaches ($\mathcal{T}$). Algorithm, first, generates a workload based on the fault type (see Algorithm 1, Line 5). *Injection* method is run to assign the fault type (see

Algorithm 1: Fault injection execution controller

```
Input:  $\mathcal{C}$ - container id,
        $C_l$ - list of containers,
        $\mathcal{F}$ - fault type,
        $F_l$ - list of fault types,
        $\mathcal{T}$ - injection duration,
        $\mathcal{P}$ - pause time,
        $\mathcal{W}$ - workload,
        inj - created fault.
1  // Start the fault injection process
2  for each $\mathcal{C}$ in $C_l$ do
3      for each $\mathcal{F}$ in $F_l$ do
4          // Generate $\mathcal{W}$
5          $GenerateWorkload$(W)
6          //Run Injection method
7          $inj \leftarrow$ Assing ($\mathcal{F}$, $\mathcal{T}$)
8          //Inject into the $\mathcal{C}$
9          $\mathcal{C} \leftarrow$ Inject ($inj$)
10         //pause the fault injection process
11         $sleep$ ($\mathcal{P}$)
12     end
13 end
```

Algorithm 1, Line 7). Afterwards the generated workload is injected in to the selected container (see Algorithm 1, Line 9). Finally, the injection is suspended within the given time interval $\mathcal{P}$.

4.5 Evaluation and Results Analysis

This section describes and evaluate the classification performance of all algorithms using the test set. The findings of the detection and the description of the localization performance have been summed up using the confusion matrices. Other metrics have been examined in order to perform classification performance comparison. Here we will first define the performance metrics and then discuss and analyze the results.

4.5.1 Performance Measures

Various evaluation measures can be used to test classification results. The most common and suitable approach to measure classification accuracy is confusion-matrix. This

Table 4.4: A summary of symbols used in this chapter

Symbols	**Description**
$\mathcal{C}$	Container id
C_l	List of containers
$\mathcal{F}$	Fault type
F_l	List of fault types
$\mathcal{T}$	Injection duration
$\mathcal{P}$	Pause time
$\mathcal{W}$	Workload
inj	Created fault

approach describes how each element in test dataset has been classified [134]. The element may be graded as either True-Positive (TP), False-Positive (FP), True-Negative (TN), and False-Negative (FN). The TP is an anomaly element that it is predicted correctly as an anomaly. A TN is a normal element that it is predicted correctly as a normal. Whereas, the FP is a normal element that it is predicted incorrectly as an anomaly. The FN is an anomaly element that it is predicted incorrectly as a normal [135]. These four constitute the confusion matrix. The n x n matrix displays n reflecting various class. The row explicitly states the real point, while the column of the matrix shows the predicted class. The accuracy of the classification is seen by the proportion of the number of elements that correctly predicted to the total number of elements in a data collection (N) [134]. There are a few widely employed measures extracted from the confusion matrix [134]: Recall (Sensitivity (S)) means the The proportion of element that correctly predicted. Precision (P) means the proportion of correctly predicted element. F-1 score (1) is represented as the harmonic-mean of precision and recall and gives a better measure than accuracy. It can be determined using:

$$F1 = 2 * (\frac{Precision * Recall}{Precision + Recall}) \tag{4.1}$$

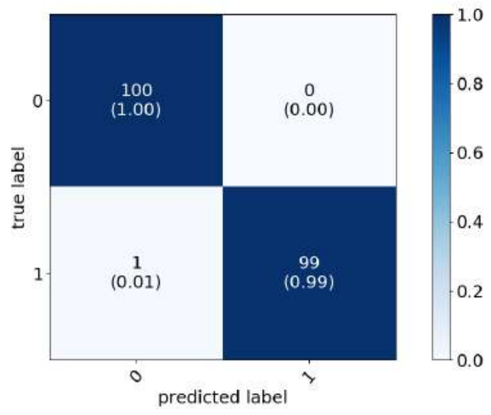


Figure 4.7: Confusion-matrix for anomaly detection using: (A) KNN algorithm.

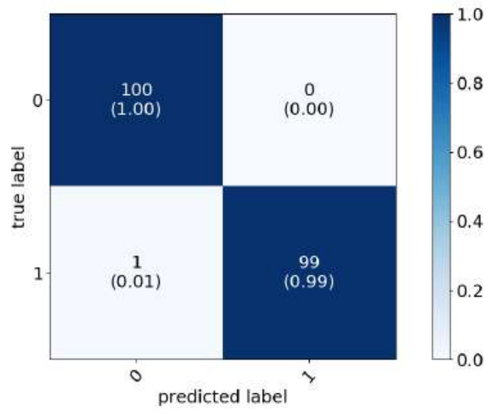


Figure 4.8: Confusion-matrix for anomaly detection using: (B) DT algorithm.

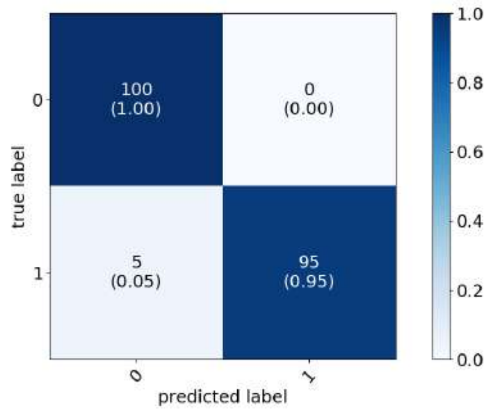


Figure 4.9: Confusion-matrix for anomaly detection using: (C) LR algorithm.

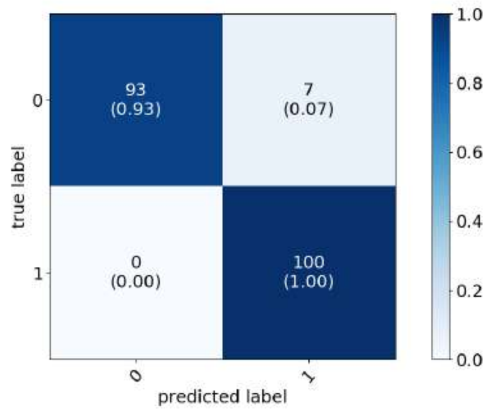


Figure 4.10: Confusion-matrix for anomaly detection using: (D) NB algorithm.

4.5.2 Results Analysis

4.5.2.1 Detection Performance

As a binary classification is a straightforward task, and there is a small amount of features (response time and throughput) to examine in detection phase, the four algorithms perform effectively.

The Figure 4.7, 4.8, 4.9, and 4.10 demonstrates the confusion-matrix for anomaly detection using the four algorithms. It can be observed from the confusion matrices that all algorithms have more TP and TN values.

The Figure 4.11 provides information about performance of detection. It gives Figures for all algorithms and clearly shows that their performance, based on accuracy, precision, recall and F1 metrics, are high and fairly equal. According to the Figure, the most best performance algorithms are KNN and DT, with accuracy rates reaching 99%. Although LR and NB in general achieved high detection performance, their accuracy are lower by only 1% compared to KNN and DT.

4.5.2.2 Localization Performance

A) Per algorithm performance:

We have tested the four algorithms for the fault-localization task using the test set. The localizing confusion matrices for the four algorithms are shown in Figure 4.12, 4.13, 4.14, and 4.15. It shows how the four classification algorithms discriminated over the two

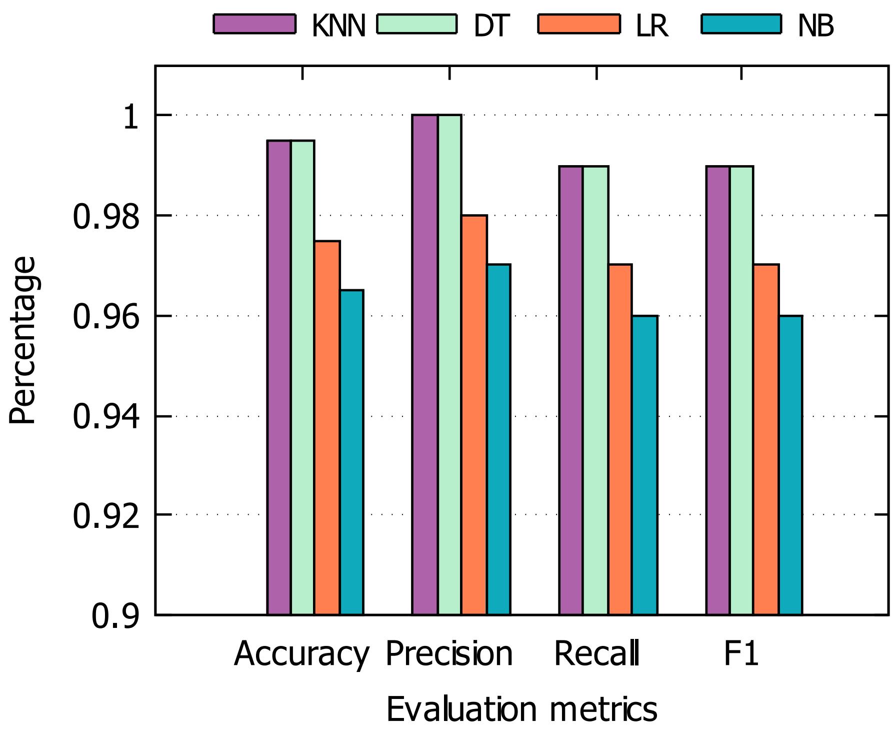


Figure 4.11: Detection

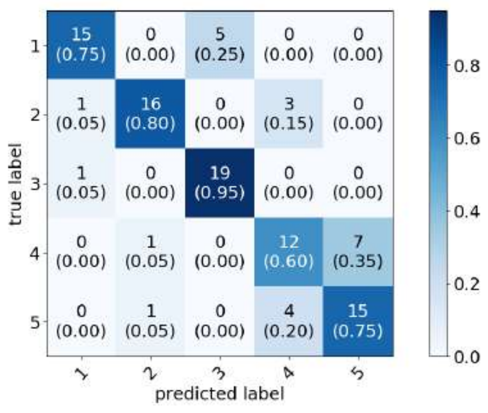


Figure 4.12: Confusion-matrix for anomaly localization using: (A) KNN.

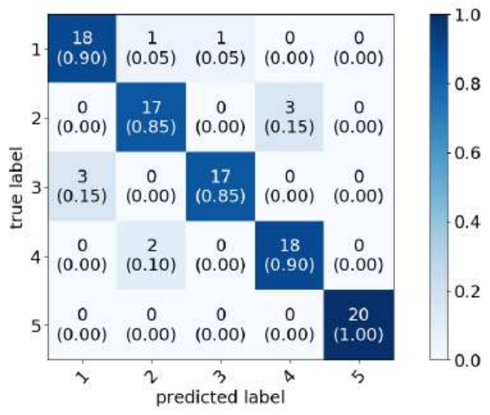


Figure 4.13: Confusion-matrix for anomaly localization using: (B) DT algorithm.

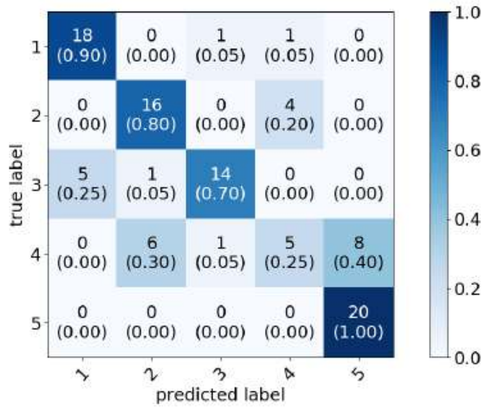


Figure 4.14: Confusion-matrix for anomaly localization using: (C) LR algorithm.

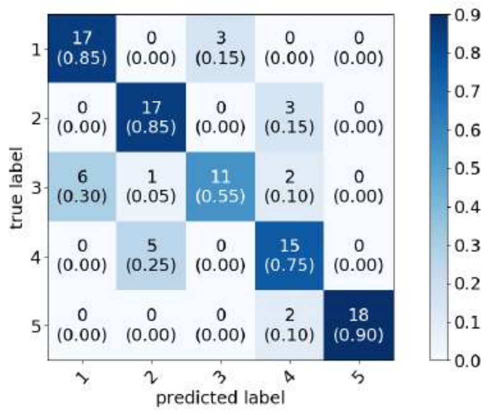


Figure 4.15: Confusion-matrix for anomaly localization using: (D) NB algorithm.

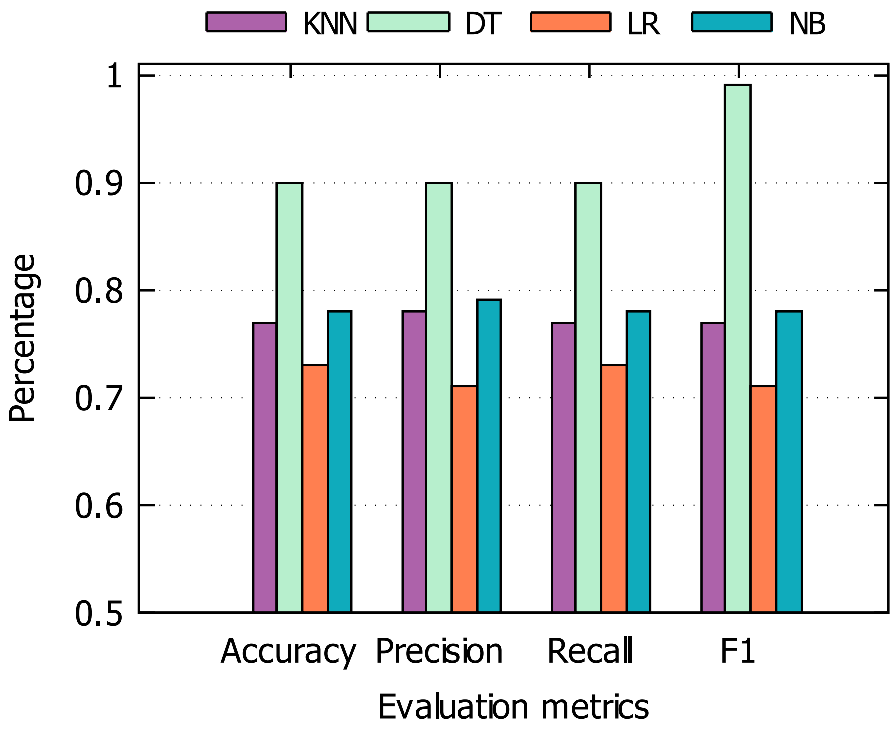


Figure 4.16: Localization.

fault types (CPU fault and memory fault) with their location in different microservices. The diagonal in the matrices represent the ideal case in which the samples were correctly classified. We can see that the confusion matrix of DT algorithm has less non-zero off diagonal cells that represent incorrectly classified samples. A total of 90 out of 100 samples were accurately classified. Whereas there were a total of 78 out of 100, a total of 77 out of 100, a total of 73 out of 100 were accurately classified by NB, KNN and LR respectively.

To compare the localization performance of KNN, DT, LR and NB algorithm, we presented the accuracy, precision, recall, and F-Measure statistics for each classifier. The Figure 4.16 shows the results of these metrics. The results showed that during the classification of classes, the highest accuracy achieved was 90% by DT algorithm. NB and KNN followed, with 78% and 77%, respectively. LR had the lowest accuracy at 73%. Overall, the data illustrates that precision, recall, and F-Measure metrics of DT algorithm were higher followed by NB and KNN. LR achieved the lowest metric values, among the algorithms compared.

B) Per class performance:

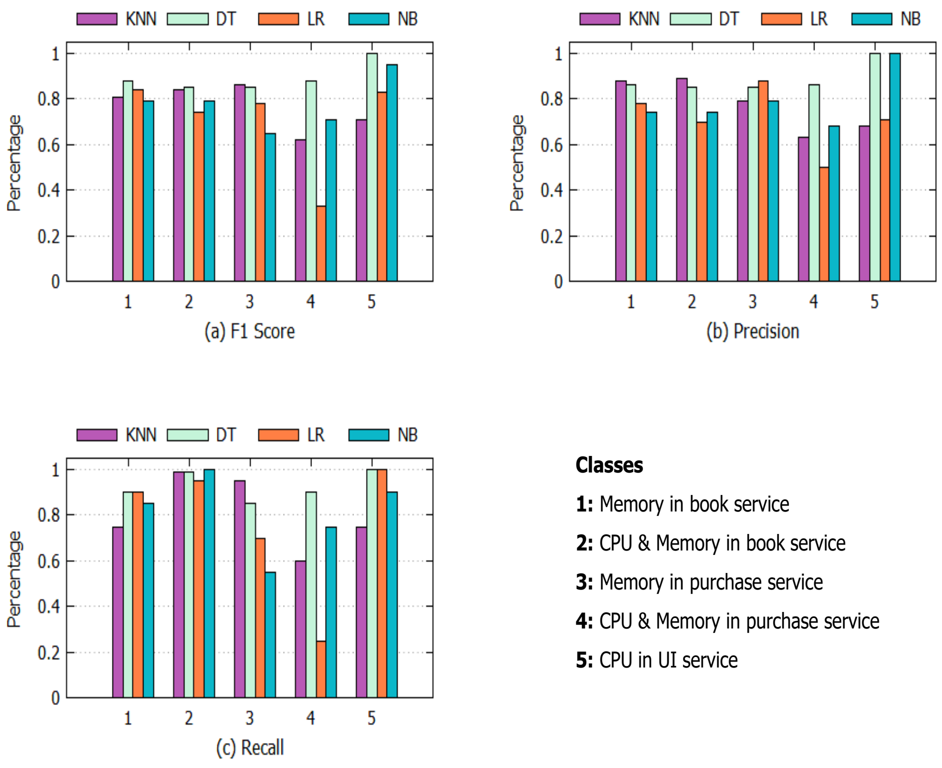


Figure 4.17: Localization per class.

The Figure 4.17 depicts the performance of all algorithms in terms of per class recall, precision, F1 score. As aforementioned, F1 provides a better measure of classified cases and based on that we discuss the performance corresponding to F1 score of all algorithms for each class. What can be deduced from the Figure 4.17 is that overall performance of all algorithms apart from KNN algorithm is highest for class 5 between 100% and 71%. While class 1, 2 and 3 were in the middle with F1-scores in the range of 88% - 65% by all algorithms. It is worth noticing that class 3 scored the highest performance among all classes by KNN algorithm. Class 4 have different trend according to the algorithms applied. LR achieved the significantly lowest score for class 4 at 33%. KNN also provide lowest performance for the same class at 62%. Nevertheless, F1 score of class 4 achieved by DT and NB were 88% and 71% respectively, which is at the same level of performance with other classes.

Since the F1 score is highest for class 5 (CPU fault in UI micorservice) than for others by all algorithms, we can believe that samples belongs to class 5 are easier to identify. The reason could be that it is the only class represents a one fault (CPU fault) in UI microservice. Microservices in our target application are affected by fault injection

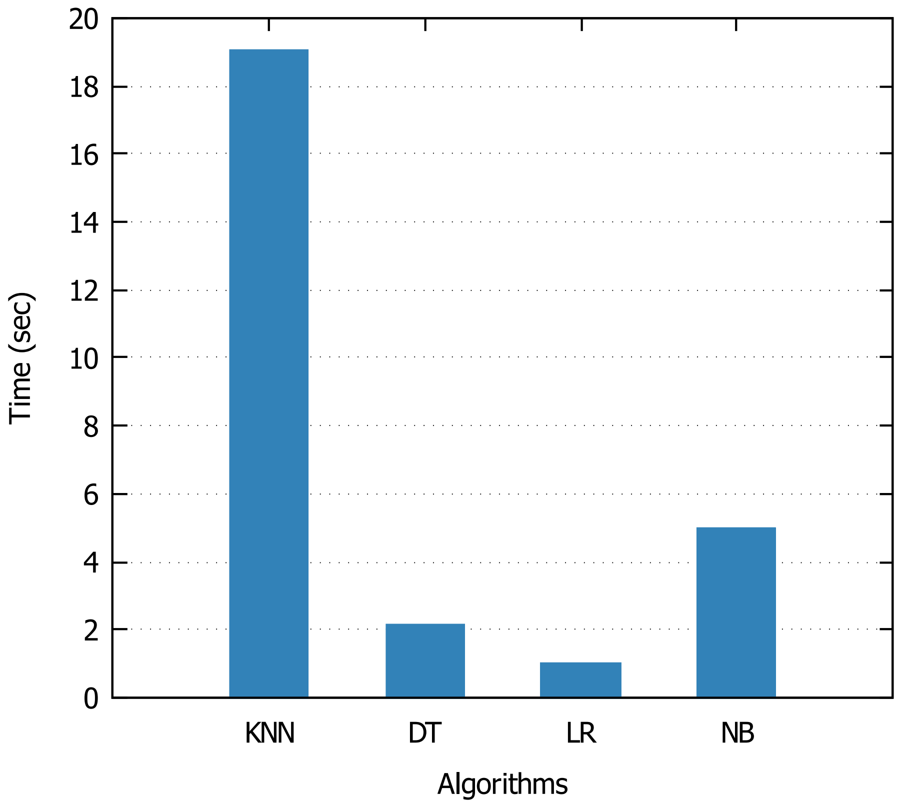


Figure 4.18: Localization time of test set.

differently. When a microservice has more resource faults to be examined, the faults become harder to classify by machine learning models.

4.5.2.3 Prediction Time of Localization Task

The localization performance of the algorithm also examined in terms of the time the algorithm takes to predict the classes of all test set samples. The time was computed on 64-bits intel-core i-7 2.1GHZ processor. Figure 4.18 presents prediction time of the localization task. It is clear from the Figure 4.18 that KNN spent the highest time in prediction. Localization time of KNN was nearly 19.03 sec, However, other algorithms took considerably lower than KNN. NB took 5.01 sec whereas DT spent nearly 2.15 sec, which is twice as many as the localization time spent by LR algorithm.

4.5.2.4 Discussion

Classification results were strongly influenced by the type of classification (binary or multi class). DT algorithm outperformed other algorithms in both type. Other algorithms KNN, NB and LR achieved lower accuracy for multi classification that for binary classification. Therefore, simple algorithm (i.e has less number of tuning parameters) can yield good performance. However, when the sample complexity increases (number of anomaly classes, observed faults and microservices), DT can be better at prediction than other algorithms.

On the other hand, LR and NB are less sensitive algorithms which have a smaller variance so that any small change in training set will not likely to change prediction model and its performance. In contrast, DT and KNN are more sensitive algorithms which have a larger variance, that means any change in training set will change the model, and then, the performance of the model. This because these algorithms do not make any assumption about dataset distribution. That means the distribution of the dataset is also likely to influence detection and localization performance.

We calculated the time required to execute prediction code of localization component that test the accuracy of the algorithm using the test set. This depends on processing Unit, programming language and operating system. KNN spent the longest time in prediction whereas LR spent the shortest time.

In summary, we can conclude that supervised machine learning methods can successfully identify and locate the anomalies. However, multi classification to perform the localization task was much harder than binary classification for detection task. Although KNN produced good results, it has enormous prediction time. In this study, we found it to be the least favourable classifier among the supervised machine learning algorithms that were applied in this study. LR and NB spent negligible times in predication. However, this advantage is outweighed by the poor classification performance. DT has the best performance for both detection and localization tasks. In addition, its prediction time was very short.

4.6 Conclusion and Future Work

We propose **M**onitoring and **A**nomaly **D**etection and **L**ocalization **S**ystem (*MADLS*), to monitor microservices-based application to detect abnormal patterns in metrics and then locate anomalous microservices. The system consists of a monitoring engine and a diagnosing module. The monitoring engine is responsible for monitoring and collecting metrics from the application, whereas the diagnosing module is responsible for detecting and localizing anomalies. Our approach applies a multi-resolution method to track observed failures' root cause by linking them with underlying faults in different layers. Multi-resolution is achieved in stages to narrow the scope of metrics and to increase the resolution.

To implement our system, we first gathered monitoring metrics from the target-application by the monitoring engine. The target-application constituted three microservice, which included; book-service, user-interface, and purchase-service. To simulate anomalies, we injected the CPU faults and, more so, the memory faults into the targeted microservices. Then, we implemented diagnosing by applying a multi-resolution approach in two stages using two components. The first component is detection, where *MADLS* detects abnormal response time and throughput of microservices from the application layer using binary classification Machine Learning (ML) techniques. The second stage is localization, which Multi classification ML techniques to identify the root causes of abnormal response times.

For detection and localization, we examined four supervised machine learning algorithms, namely, K-Nearest Neighbors (KNN), Decision-Tree (DT), Logistic Regression (LR), and Naïve Bayes (NB). For binary classification, we trained models with the training dataset of two features response time and throughput. The samples were labeled as zero (normal) and one (anomaly). On the other hand, the multi-classification models were trained using the training dataset with features CPU consumption and memory consumption. This dataset consists of samples from 5 classes: Memory fault in book service, CPU fault and Memory fault in book service, Memory fault in purchase service, CPU fault and Memory fault in purchase service, CPU fault in UI service.

We finally validate *MADLS* to show that *MADLS* can accurately detect anomalies in response time and then specify the type and location of a fault in a microservices. The detection and localization were tested using the test set. The detection performance using all algorithms was measured based on accuracy, precision, recall, and F1 metrics. All algorithms obtained high and equal performance with more than 98% accuracy. Conversely, The performance of all algorithms for localization task was different. The result showed that accuracy, precision, recall, and F-Measure metrics of the DT algorithm were the highest, followed by NB and KNN. In contrast, the lowest metric values were obtained by LR algorithms. We also examined the time of prediction the algorithms take for the localization task. We found that KNN took a long time compared with other algorithms. NB follows KNN, whereas DT and LR spent a much short time. After evaluating classification accuracy and prediction time, DT provided the best result for detecting and localizing anomalies in a microservices-based application.

Several future works can be considered for our work. First, As the target application used in this work has three microservices, the system can be tested with other applications. Second, the study of microservices dependencies for further root cause analysis using our system can be investigated. Third, we plan to expand the implantation to detect other faults in microservices. Lastly, we also intend to study the influence of microservice inter-dependencies on finding the root cause of failure.

5

Osmotic Monitoring of Microservices between the Edge and Cloud

Contents

Summary

This chapter presents an integrated monitoring system for monitoring IoT applications decomposed as microservices and executed in an osmotic computing environment. A real-world smart parking IoT application is used for an experimental evaluation and for demonstrating the effectiveness of the proposed approach. Through rigorous experimental evaluation, we validate the osmotic monitoring system ability to holistically identify variation in CPU, memory, and network latency of microservices deployed across cloud and edge layers.

5.1 Introduction

The advent of Internet of Things (IoT) [136–138] and Smart City applications created a scenario where billions of users or devices get connected to applications on the Internet, which results in trillions of gigabytes of data being generated and processed in cloud datacenters [32, 72]. The increasing need for supporting interaction between IoT and cloud computing systems has led to the creation of the *Edge*, *Fog* [139] and *Osmotic* Computing [32]. Osmotic computing is a new paradigm to support the efficient execution of Internet of Things (IoT) services (microservices) and applications at the network edge [32] by providing increased resource and management capabilities at the edge of the network. One challenge that underpins such emerging approaches is the dynamic management of microservices across cloud and edge datacenters. For instance, defining when and how microservices can be migrated from edge resources to cloud-based resources (and vice versa), and characteristics which influence such migration, remains a challenge [32].

Monitoring [16] plays a central role in identifying ”when” a certain microservice should be migrated. For migration to be effective, it is necessary to properly monitor the performance of the microservices. The monitoring of microservices in IoT environment is a recent topic and therefore few works have been carried out in this regard. The work presented in the chapter seeks to explore this topic in the construction of a solution that meets the requirements of monitoring microservices as well as IoT and cloud applications.

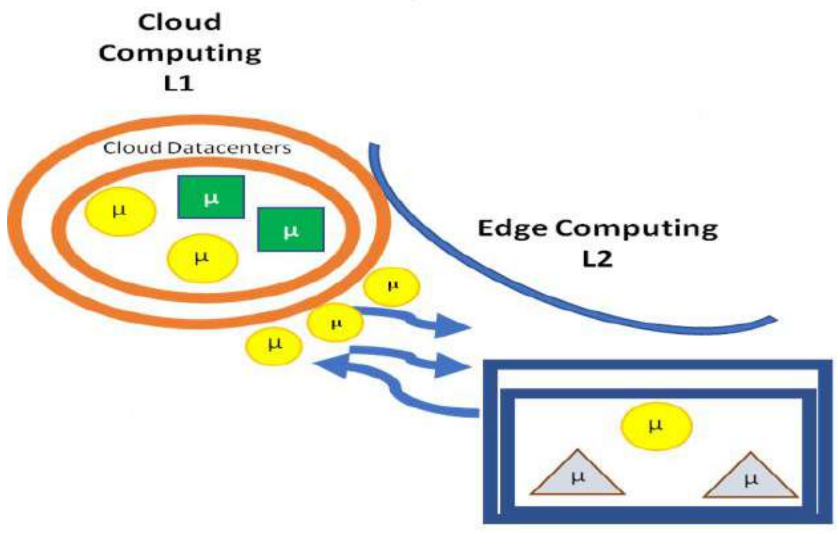


Figure 5.1: Osmotic movement of microservices across cloud and edge.

With osmotic computing, a new IoT application programming paradigm that provides an opportunity to execute multi-service applications between the edge and cloud, the applications need a multiple and dynamic system to properly monitor the osmotic services to promote to make possible dynamic workload balance between the edge network and cloud as showing in Figure 5.1.

Within the scenario of vehicular traffic management in urban centers, the provision and efficient occupation of parking spaces is a common problem to be solved. The intelligent parking application is a multi-layer application widely deployed for such problem [140]. The main purpose of this application is to alert the driver regarding available parking spaces near his/her location. This work leverages the intelligent parking application as a motivation example. Figure 5.2 depicts a conceptual implementation of this application using a microservice architecture. The smart parking application comprises there microservices: (i) *parking management*, (ii) *user data management* and the (iii) *selection of vacancies* according to user's preferences. The parking management deals with the monitoring of the available spaces in the urban space dedicated to parking lot. User management stores the data of locations already used by drivers, as well as their preferences. Finally, the selection of vacancies microservice schedules

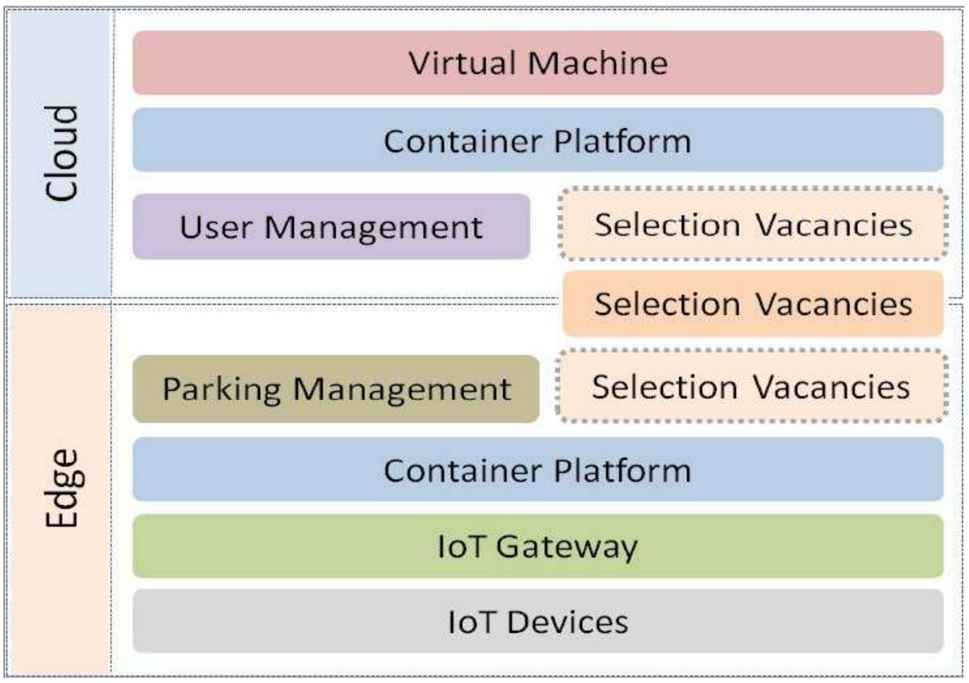


Figure 5.2: Parking management application.

and recommends the possible vacancies available to users.

Parking management is responsible for sensory interfacing and monitoring (with sensors instrumented to indicate the presence of vehicle). Such a microservice is self-contained and deployed at the edge (i.e. at each parking lot). User management is deployed to the cloud, so user preference data is accessible to all city parking lots. The vacancy selection microservice is the most important one. It continuously runs an algorithm for selecting vacancies from the available parking lots according to user preferences. However, this microservice may need to run on the edge or cloud depending on several factors (typical of osmotic computing). For example, during periods of heavy vehicle traffic, and large numbers of vehicles searching for parking spaces (e.g. during weekends), the vacancy selection microservice would run in the cloud where greater processing power and high computing capacity would allow an easy horizontal scalability. However, in periods of low traffic and low demand this microservice could be running on the edge.

Existing Quality of Service (QoS) monitoring tools and techniques suffers from serious technical limitations when subjected to Osmotic Computing. For example, there is an urgent need to find answers to the following research questions:

- How to ubiquitously monitoring QoS of microservices mapped to an Osmotic Computing (Edge+Cloud) environment?
- How to aggregate QoS measures of microservices running in Osmotic Computing environment to give a holistic view of IoT application's (e.g., smart parking) run-time performance?

To address the aforementioned challenges, in this chapter we make following concrete research contributions:

- We develop a unified monitoring model for Osmotic Computing that provides an IoT application administrator with detailed QoS information related to microservices deployed across Cloud and Edge.
- We propose Osmotic Monitoring, a monitoring system for Osmotic computing that implements the proposed unified monitoring model.
- We conduct extensive experimental evaluation of Osmotic Monitoring system in order to study the scalability of the proposed solution.

The rest of this chapter is organised as follows: Section 5.2 presents a related work. Section 5.3 presents monitoring microservices in osmotic computing (edge to cloud). Section 5.4 presents experimental evaluation. Section 5.5 presents the conclusion and future work.

5.2 Related Work

Several works already published have explored topics related to service monitoring, as well as models and metrics for QoS assurance. Whether in the cloud [71], using microservices [25] or even in monitoring services at the edge [24], varied solutions and results have been presented.

The proposed monitoring model is an extension of the CLAMBS [71]. The CLAMBS was selected cause is a multi-cloud monitoring tool that can be used on several operations systems, including container images. The feature of multi environments supported by CLAMBS is very important to monitor osmotic services that will be deployed in a miscellanea of devices and virtualization technologies.

In [71], the authors present CLAMBS, a framework for monitoring and benchmarking applications in a multi-cloud environment. In addition, a model for multi-layer monitoring in the cloud is presented. In this way, QoS parameters relevant for each cloud service layer are listed. Finally, an experimental evaluation is performed in the IaaS level. The work presented here follows a similar approach for defining and experimenting with QoS parameters, although it is different from the use of the cloud and the edge, besides focusing more on the application level. The microservice monitoring in the edge environment is reported in the paper presented at [72]. In [72] a state-of-the-art review of self-adaptive applications using edge microservices and services in the cloud are performed. The results observed shows that the main parameters of QoS for virtual machines in the cloud are the usage of: CPU, memory and network.

Finally, the monitoring of services deployed in containers is present in the works [24] and [73]. In the work published in [24] the authors present a framework called PyMon that uses the Docker management API to obtain statistics of resources used by containers. Unlike [24], the present study uses libraries to monitor processes inside the containers, thus allowing the effective monitoring of a container that performs a multi-service or multi-process environment. The work presented in [73] brings an assessment of the use of Docker containers versus the use of Virtual Machines. To verify the QoS parameters to be compared for evaluation, the authors monitored the CPU usage by the installed Docker process, not verifying the parameters of the containers that are being executed or even of the processes internal to the containers. Moreover, applications built using the microservices architecture obey a set of rules with the purpose of making the microservice self-sufficient and easily scalable.The implementation of microservices can be done in several ways, however, the use of containers in the construction of microservices has attracted significant attention recently. The use of containers is becoming so popular that it is currently possible to run containers on IoT devices, such as IoT Gateways (e.g. RaspberryPi). Thus, the challenge of monitoring containers as well as microservices running within these containers is highly relevant in the context of osmotic environment.

In summary, works such as CLAMBS model [71] enables efficient monitoring of services in a multicloud environment but lacks capability to monitor microservices at the edge. Current works on microservice monitoring [72] usually focus on single layer monitoring, i.e., microservices in the cloud [73] or microservices at the edge [24]. Our proposed work differs from the current approaches by presenting an advanced monitoring solution that can be used to monitor microservices deployed in Osmotic Computing environment i.e. the cloud and/or deployed at the edge.

5.3 Monitoring Microservices in Osmotic Computing (Edge to Cloud)

The proposed monitoring model is an extension of the CLAMBS [71]. In order to support Osmotic computing environment, several extensions to CLAMBS has been proposed and incorporated. First, the PUSH communication model between the agent and the manager was adopted. The choice for this type of communication seeks to meet the restrictions of the IoT devices, as well as the security barriers imposed by IoT device networks for external access. Second, it was necessary to define a generic model of monitoring agents that can be extended to include support for new devices, for this was modeled a SmarAgent that would be extended by the specific agents, namely: ProcessAgent, SystemAgent, NetworkAgent and DeviceAgent. Third, we incorporate the concept of *Smart Agent* for devices that have a permissive computational power. Generally IoT devices have limited computing power and focus on the resolution of the sensing or actuation for which they are intended. However, some devices (sensors/actuators) have a more robust and computationally capable hardware such as they have connectivity through WIFI. In order to allow improved monitoring of these devices, the smart agents have two main abstractions: *Gateway Agents* and *Sensor Agents*.

5.3.1 Monitoring Model

Monitoring systems are commonly composed of *monitoring agents* and *management services*. Normally, monitoring agents are components that only read data from monitored services or machines. The management services store the data collected by the agents and expose this data via API or through graphical interfaces for system administrators.

5.3.1.1 Monitoring Agents

Usually the deployment and configuration of the monitoring agents are performed manually, each agent being specific to the target monitoring architecture. Monitoring agents (OMA) on the other hand are multi-platform monitors agents based on a Multi-cloud monitoring model. OMA supports monitoring of microservices implanted in osmotic environment comprising of heterogeneous cloud and / or edge resources. All monitor agents extend a common agent, called SmartAgent as described earlier. SmartAgent represents a service consisting of three operations: 1 - *register*, 2 - *sendData*, 3 - *setConfiguration*. The *register* operation must make an HTTP PUT request that sends the agent registration information to the management system. The *sendData* operation must periodically perform an HTTP POST request to the management system to send the metrics obtained. *SetConfiguration* must send an HTTP GET request to the manager system to obtain the agent configuration parameters. Figure 5.3 shows the communication model used by the Osmotic monitoring agents. The first action performed by these is the agent registration with the Manager. After this, the manager can receive the data sent by the agent (action 2), as well as (action 3) can modify some agent configuration parameter.

The *SystemAgent* and *NetworkAgent* agents are the most commonly found in the monitoring tools. `SystemAgent` monitors the system as a whole, for example, a virtual machine or a container. *NetworkAgent* is responsible for network monitoring. Although network metrics can be related to a single system, which would lead to the inclusion of these metrics in the *SystemAgent*, the possibility of multiple network interfaces in one system justifies the need for the *NetworkAgent*. ProcessAgent is responsible for collecting metrics related to a specific process running on a system. This type of agent is already present in most virtual machine monitoring tools in a cloud environment. As for the monitoring of processes executed in containers, the current tools focus on the monitoring of the container itself. The execution of only one process per container is the most common scenario in the construction of applications in microservices, however, in an osmotic environment the use of several containers can make it difficult to migrate from the cloud to the edge or vice versa in a way that is monitoring of multiple processes in the same container. Therefore, this work has built a *ProcessAgent* that can run internally to the container. Finally, *DeviceAgent* handles the collection of metrics or data from IoT devices. IoT devices increasingly see improving processing power, so some of these devices need to be monitored. The monitoring of the IoT devices can serve for simple gauging of acquired data, availability, as well as, to prevent failures of misuse of the device.

5.3.1.2 Manager Agent

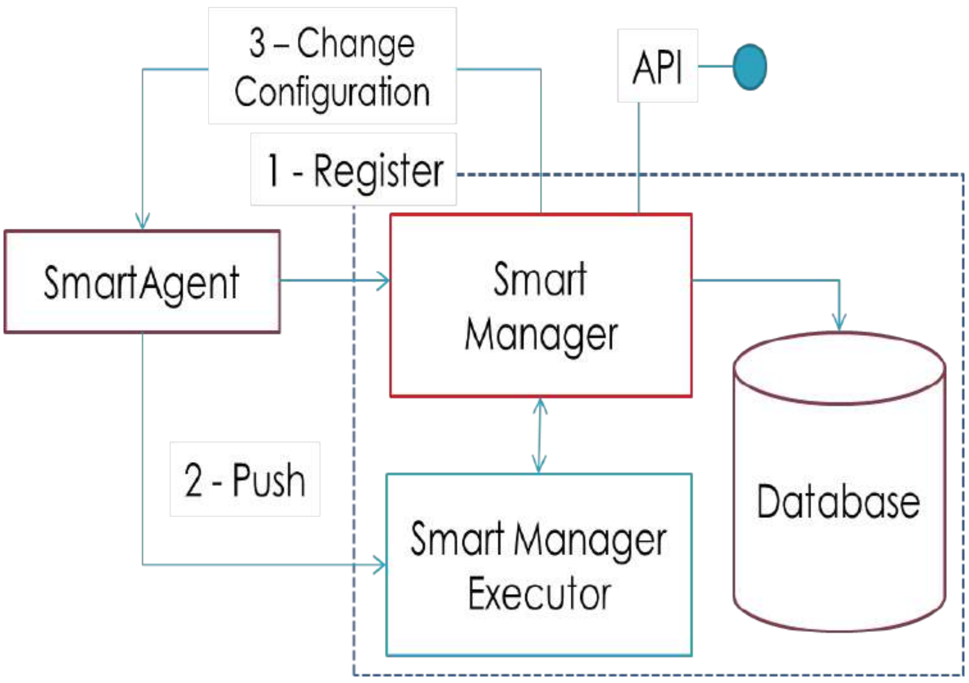


Figure 5.3: Agents to manager communication model.

The Osmotic monitoring data management agent is called *SmartManager*. SmartManager basically performs various services that receive the data from the monitoring agents. The data obtained is persisted in a database or data storage services. SmartManager must also provide an API for accessing data saved by other services or other applications. The sending of data by the monitoring agents to the management system occurs according to a well defined sequence of steps (Figure 5.3). Initially, *SmarAgent* on startup sends a registration request to *SmartManager*. The *SmartMan-*

Table 5.1: List of the configuration parameters and values.

Parameter	**Value**
Cloud service provider	AWS
Cloud service provider location	UK South
VM	t2.medium type
Operating system	Ubuntu:14.03
Docker platform	Version 17.06.1
Tomcat	Version 7
Mongodb	Version 4.3.5
MySQL	Version 5.7
Java	Version 8
Apache JMeter	Version 5.4.1
SIGAR	Version 1.6
RaspberryPI	1 Model B

ager receives the request (*1-Register*) and registers the *SmartAgent*, returning to the *SmartAgent* an access key and an endpoint to send the data. From there, the *SmarManagerExecutor* (*2-Push*) is enabled to receive the data sent by the *SmartAgent*. *SmartAgent* periodically queries *SmartManager* for its configuration parameters (*3 - Change Configuration*). Dynamic configuration enables real-time agent management. It is expected that applications deployed in an osmotic environment will have a degree of self-management. Mainly, in cases of self-managed microservice migration between the cloud and the edge. In this way, the real-time management of the monitor agent is highly relevant, since it allows the application that makes use of the monitored metrics to change the agent at runtime.

5.3.2 Osmotic Monitoring: System Implementation

List of all the configuration parameters and their values is shown in Table 5.1. We have chosen the latest version available of the selected values at that particular period of time, like MySQL, Mongodb, etc. More details will be discussed in **section 5.4.2.**

In order to validate the monitoring model presented previously that underpinned the development of the osmotic monitoring system, was implemented a proof-of-concept solution for monitoring microservices in the osmotic computing environment. The implementation was performed in the Java language, making use of the RESTLet framework and the Hyperic SIGAR library (https://github.com/hyperic/sigar). The use of the Java language allows the construction of a multiplatform solution, easily transferable in multicloud environments, being still compatible with some equipment of edge computing. The RESTLet (https://restlet.com/) is a framework that facilitates the construction of WEB API in Java. RESTLet provides a set of abstractions for the development of REST architectural style APIs. Rest API is a standard between several container monitoring tools [72] and encourages the integration of applications, as well as composite microservices.

The metrics adopted for the monitoring of osmotic microservices followed the same metrics for the SAAS level of microservices in clouds defined in [71, 72]. Although there may be a discussion as to what level, whether SaaS or PaaS, the osmotic microservices are better related, the metrics defined in PaaS level [71], namely: SystemUpTime, SystemServices, SystemDesc, Utilization are already easily obtained by the current monitoring tools containers[72]. The choice of parameters of the level of SaaS is corroborated by the premise that each microservice is directly related to an application that is deployed on a Container platform such as Docker or Linux Container (LXC). Thus, the metrics used for the monitoring model were: *CPU usage*, *memory usage*, *amount of free memory*, *the amount of bytes being downloaded*, *amount of bytes being uploaded*, and *availability*. The availability metrics were applied to two different scopes, the system (microservice or container) and device scopes. Table 5.2 presents the API specification of the smart manager to support monitoring data provided by the monitoring agents.

The metrics have been grouped by Application, that is, any data sent to the Manager API must indicate to which application (parameter *[app]* the URL) that element (process, system, network or device) is bound. In this way, it is possible to provide an overview of the monitoring of the elements used in the execution of a specific application, even though this application has several microservices deployed in various cloud and/or edge environments. All data-sending calls to the Manager API are made up of *HTTP POST* requests to the specific *Path*, for example to send process data the request would be in the *Path [app]/process*. The metrics monitored for the processes were the percentage use of the CPU and the amount of memory used. The

Table 5.2: Monitoring metrics provided by smart manager

Scope	**Metric**	**API Path**
Process	% CPU	[app]/process/
Process	Memory Usage	[app]/process/
System	Memory Usage	[app]/system/
System	Memory Free	[app]/system/
System	Avaliability	[app]/system/
Network	Rx_Bytes	[app]/network/
Network	Tx_Bytes	[app]/network/
Device	Avaliability	[app]/device/
Agents	-	[app]/agents/

first metric captures the CPU usage percentage of a process for a specific application and the second the amount of memory used by the process in *MegaBytes*. Within the monitoring model presented in **section 5.3**, the *SystemAgent* element represents a complete system, that is, it can represent a Virtual Machine (VM), a container or microservice within an osmotic computing environment. This work treated a system as a microservice or a container, when the microservice is fully contained in a container (see Selection Microservice on Figure 5.2).

However, when the microservice is distributed in more than one container (see User Microservice on Figure 5.2), it consists of two systems, one for each container. The data monitored for the system elements were: amount of memory used, amount of free memory and availability. The amount of memory used registers the use of memory in *MegaBytes* for all the processes that are running on that system. The amount of free memory registers the available memory for use by the System. And, finally, availability assesses whether the System is accessible and available. Network usage metrics captured include the amount of bytes in *KiloBytes* downloaded (*RX_Bytes*) or uploaded (*TX_Bytes*) by a system or a network interface in an instant of time. The last monitored data was the availability of a device used by the application. This metric only shows the true or false value and its implementation is very specific for each device. For example, in our evaluation it was necessary to use a specific XloBorg [141] sensor library to check device availability.

The agent running in the container, VM and or any system that hosts the microservice captures the metrics explained above and sends it to the Smart Manager for further processing. The received data is stored in a MongoDB [142] database in the JSON format. The data stored on MongoDB are grouped by application and identified by the type of agent and the unique key of the agent. For example, for any metric (Process, System, Network, etc.) stored in database the recorded data is composed by the identifier of the document, *application*, *agent-key*, *agent-type*, and *timestamp*. The *application* identifies the application. The *key-agent* represents the agent's unique key. The *type-agent* represents the type of agent. At last, the *timestamp* stores the instant the metric was saved. Added to these attributes follows the attributes specific to each type of agent. In other words, if the metric is referring to Proccess beyond the aforementioned attributes, the attributes are added: the *process-id* and *process-name* attributes identify the monitored process, while the *process-CPU* and *memory-used* attributes represent the metrics values.

Each monitoring agent must make an initial registration for sending monitoring data. The agent registration is done through an HTTP PUT request to the *Path [app]/agents*. The configuration attributes required for the registration of an agent are: *agent-type*, *access-key*, *access-password*, and *application*. The *application* is informed directly in the URL. The *agent-type*, *access-key* and *access-password* are informed in the request body. The *agent-type* tells the agent type so that the manager selects the correlated endpoint. The *access-key* and *access-password* attributes are used for agent authentication. Every time that the Manager processes an agent registration request, *an endpoint*, *an agent-key*, and *a read-frequency* are returned to the agent. The *endpoint* identifies the Manager Service that will handle POST sending requests for the agent. The *agent-key* uniquely identifies the agent and ensures that the agent has been correctly registered. The *read-frequency* indicates the time interval the agent should wait for each request to send data. If the Manager deems its necessary to modify any of the returned parameters, it only sends new values when the agent sends an HTTP GET request to the *Path [app]/agents* to verify that the attributes have not been changed.

The specific implementation of agents: *ProcessAgent*, *SystemAgent* and *NetworkAgent* basically made use of the Hyperic SIGAR library. SIGAR is a multiplatform library (Unix, Win, Solaris, FREEBSD, MAC OS, etc) written in Java that provides an functionalities for accessing operating system information. Although the use of the SIGAR library has been presented in the work [72], the present work explores the same library in the monitoring containers as well as virtual machines. The use of SIGAR in the construction of the aforementioned agents is relatively simple, being composed of the instantiation of an object of class *org.Hyperic.sigar.sigar* and the invocation of methods present in this abstraction. For example, for access to CPU usage of a process it is enough to invoke the *getProcCPU* method, informing the process id (*pid*). To access information about the system memory it is necessary to invoke the *getMem* method.

It is important to note that for the measurement of the network traffic rate, it was necessary to deploy timed counters since SIGAR only returns the amount of *RX_Bytes* and *TX_Bytes* of a network interface at a given instant of time. Another change was the addition of all network data of all interfaces to constitute the traffic of a system. Agent-specific settings such as which processes to monitor, which network interfaces to monitor, which the initial endpoint of the manager, and access attributes were informed in an initial agent configuration file. Thus, an agent developed to capture and process metrics can be easily reusable in another system. Agents developed for *ProcessAgent*, *SystemAgent* and *NetworkAgent* can be used on any system that supports JAVA language version 7 or higher. However, the agents developed for the *DevicesAgent* were totally specific to the devices used in the experimental evaluation.

5.4 Experimental Evaluation

An experimental evaluation of the monitoring system described in **section 5.3** was carried out in order to prove the efficiency and efficacy of the monitoring of microservices in cloud and the edge. Thus, an initial version of a microservice-based Smart Parking application (as discussed in **section 5.1.1**) was developed and deployed in an osmotic computing environment (edge and cloud). Subsequently, variable load tests were performed in order to verify if the data monitored reflected the variations introduced by the corresponding load tests. The tests were not intended to measure performance, although they may have presented some data relevant to that scope.

Although a raffle approach is highly recommended in evaluations to avoid cache influence in performance testing of systems in production, the article focuses on evaluation to verify the effectiveness and efficiency of the tool in obtaining the monitored metrics. That said, further work is needed to indicate the impact of using the tool on the quality of the metrics obtained, comparing this with other possible solutions.

5.4.1 Monitored Application

The Smart Parking application (as depicted in Figure 5.2) searches for real-time mapping of parking spaces available in a city. The citizen as a driver accesses the Smart Parking to know the best places available according to his personal preferences. The main use case follows the flow: 1 - the driver travels by a road, 2 - the smart parking application is notified of the position of the driver, 3 - the job selection service searches possible available positions, Smart Parking alerts the driver to the vacancies available. With this scenario in mind, basic versions of the three micro-services specified in the section have been implemented, namely: *User Management, Selection Vacancies and Parking Management.*

User Management (UM) is the service that is deployed to the cloud. It is responsible for storing user data as well as for providing system communication as the user. User interaction can occur via an application deployed on your phone. *Selection Vacancies* (SV) continuously receives UM job requisition notifications. The incoming requisitions are processed through a selection algorithm that continuously consults with Parking Management to check the status of the vacancies. Once the vacancies are defined the SV notifies the UM. *Parking Management* (PM) continuously monitors vacancy status. To do this, it is implanted on the edge and communicates with the IoT sensors that identify the occupation or the release of a vacancy. Theoretically, the PM must be replicated between the various parking lots as many times as necessary.

UM, SV and PM are services deployed on the microservices architecture and were responsible for a very specific functionality as described earlier. Inter-service communication occurs through a REST API to access its functionality. For experimental evaluation, specific API call were implemented for each service, namely: for the UM a call to query the user data; for the PM, a call to consult the vacancies and their states;

and, for the SV a call that returns a vacancy available to a user when it accesses a parking lot. All services were developed in Java, running on an Apache Tomcat server (http://tomcat.apache.org/). For the services UM and PM that require persistence of contextual data of the entities MySQL (https://www.mysql.com/) database was employed. The smart parking application with the three microservices were deployed in containers. The containers were built for execution on the Docker platform [143]. As well, it tried to make the environment of execution of the SV a little more equal, since the same microservice was executed on a virtual machine in the cloud and in a RaspberryPi on Edge. Although the deployment of microservices through Docker Containers is not an unpublished topic, this work explores for the first time the use of Docker Containers for deployment of the same service that runs in the Cloud or the Edge.

The use of Containers Docker allows the use of two possible deployment cases, namely: a container for each microservice or several containers for each microservice. In the first case (F1) in a same image of the container are installed all the components used by the microservice. In the second, each component is installed in its own container, that is, the Tomcat server will compose one container and the MySQL database will be in another. The second case (F2) most commonly used by users of the Docker platform since it does not require the construction of specific images, instead using standard images already available in the Docker HUB catalog of images. Considering the above two cases, as well as, trying to cover most possible scenarios of execution of microservices in an osmotic environment, an explicit plan for experimentation is in Table 5.3. In the cloud environment, the UM micro service can be instantiated by only one container (F1), scenario C1, or for two containers (F2), scenario C2. Similarly, the PM service can also be instantiated in scenarios E1 for F1 and E2 for F2. Finally, the SV service that only requires the use of Tomcat used only the F1 case, although it presents two scenarios: S1 for the cloud environment and S2 for the Edge environment.

5.4.2 Experimental Design

We used Apache JMeter (https://jmeter.apache.org/) to generate HTTP requests to test and validate the Osmotic Monitoring system's capability. The JMeter test cases are presented in Table 5.3. The tests consisted of performing 10, 100, and 500 simultaneous requests to the Osmotic Monitoring system at a fixed interval of 5 minutes. Initially, the idea was to increase the load of the tests by 10 times with each new test battery, however 1000 test requests for the services deployed in RaspberryPi caused a stack overflow and made the service unavailable. Due to this, the last test was performed with 500 requests and even with this number of concurrent requests, some faulty responses were observed in the Edge environment, a fact not observed in the tests of 10 and 100 requests.

The choice of scenarios took into account different types of implantation of osmotic services. To be clearer, usually, osmotic services are pure processing or data processing and storage, thus scenario C1 represents a processing and storage service in the cloud, C2 being the same as the C1 service being deployed in two containers instead of just one. The S1 scenario represents pure processing on the cloud and the S2 on the edge. Finally, scenarios E1 and E2 represent the same as C1 and C2 on edge. The main purpose of the work was not to perform a performance analysis, although the results obtained in the comparison between scenarios C1 and C2 indicate that at edge, for the environment used, the use of Tomcat and MySQL in the same container presented better metrics than the use of separate containers for Tomcat and MySQL.

The containers with the microservices of the Smart Parking Application, were deployed in an Openstack[1] cloud of the Metrópole Digital Institute[2], and in a RaspberryPI[3] 1 Model B, in the Edge. The cloud used a virtual machine that runs a Linux system with Ubuntu[4], version 14.03, on a virtual hardware with configuration of 2 vCPU, 4 GB of memory and 20 GB of disk. On the virtual machine was installed the platform Docker in its version 1.10. The UM service in scenario C1 (as described in Table 5.3) deployment was based on the MySQL 5.7 image[5] obtained via Docker HUB. This image included a version of the Java virtual machine, version 8 and the Tomcat server version 7. The scenario C2 made use of the same image for the container of MySQL

[1]https://www.openstack.org/
[2]https://www.imd.ufrn.br/portal/
[3]https://www.raspberrypi.org/
[4]https://www.ubuntu.com/
[5]https://hub.docker.com/_/mysql/

Table 5.3: Microservice scenarios deployed at Docker

Environment	**Scenario**	**Microservice**	**Containers**
Cloud	C1	User Management	1 - Tomcat + MySQL
Cloud	C2	User Management	1 - Tomcat, 2 - MySQL
Cloud	S1	Selection Vacancies	1 - Tomcat
RaspberryPi	S2	Selection Vacancies	1 - Tomcat
RaspberryPi	E1	Parking Management	1 - Tomcat + Mysql
RaspberryPi	E2	Parking Management	1 - Tomcat, 2 - MySQL

whereas for the container of Tomcat used the image[6] for Java 8 and with Tomcat 7. The same Tomcat image was used in scenario S1 of the SV service. For the Edge environment scenarios (S2, E1, and E2), here simulated by the RaspberryPI 1 Model B that runs a Raspbian system in a configuration from a CPU core to a 700 Mhz clock with 512 MB of RAM and a 4GB SD memory. We used a specific image[7] for MySQL 5.7 and another one[8] to Tomcat 7. The P1 scenario that used an integrated image made use of RPI-MYSQL image on which a Java 8 version and a Tomcat version 7 were installed.

The main objective of the experimental design was to produce a computational load and a network traffic load for the microservices that could be measured by the Osmotic monitoring system in order to prove the effectiveness of the monitoring model. A raffle was not carried out in the order of testing or in the choice of scenarios to be prioritized. The tests were repeated 10 times each to obtain the mean results of the measured metrics.

5.4.3 Latency Time Results

The average latency time results, in milliseconds, obtained for the requests made in each scenarios are shown in Table 5.4, as well as the number of Bytes, in KB, sent by the requests. The values obtained for the latency time clearly reveal the computational power difference of the Cloud and the Edge, as well as a slightly better performance of the C2 scenario in relation to the C1. The best result of scenario C2 indicates that the use of multi container architecture per service exploits the hardware of the virtual

[6]https://hub.docker.com/_/tomcat/
[7]https://hub.docker.com/r/hypriot/rpi-mysql/
[8]https://hub.docker.com/r/dordoka/rpi-tomcat/

Table 5.4: Request results for analyzed scenarios

Number of Requests	**Scenario**	**Latency Avarage (ms)**	**Bytes (KB)**
10	C1	29.93	0.7
100	C1	36.2	7
500	C1	29.3	35
10	C2	30.83	0.7
100	C2	29.89	7
500	C2	29.26	35
10	S1	35.5	0.7
100	S1	36.89	7
500	S1	73.17	35
10	S2	290.3	0.7
100	S2	6966.97	7
500	S2	10535.53	35
10	E1	97.32	0.7
100	E1	575.29	7
500	E1	4623.52	35
10	E2	118.6	0.7
100	E2	843.67	7
500	E2	6075.03	35

machine more efficiently. Another important note, and that the behavior presented in the Cloud environment did not recur in the Edge environment. In fact, in the Edge the behavior was inverse i.e., scenario E1 presented better performance than the E2. Probably, the workload required to execute more than one container on hardware with little computational capacity influenced the performance of the microservices. Although the actual performance observations highlighted here are not the main objective of this work, the proposed Osmotic monitoring system presents a novel way to monitor the performance of such microservices in osmotic computing environments. This provides IoT system administrators who are generally challenged with managing multiple of such microservices deployed across cloud and edge the ability to clearly understand the performance of such microservices.

5.4.4 CPU Results

The CPU usage values for all evaluated scenarios are presented in Figures 5.4, 5.5 ,5.6, 5.7, 5.8, and 5.9. For an analysis of the results obtained for different scenarios in the same environment, for example, for the cloud environment, Figures 5.4 and 5.5 show the percentage of CPU usage for the UM service deployed in scenarios C1 and C2 respectively. Evaluating only the result for scenario C1, for the tests of 10 and 100 requisitions the presented variation was relatively little from 10% to 30%, whereas for the one with 500 requisitions it reached 70% of use. For scenario C2, the results of the variance were similar in behavior, that is, for 10 and 100 the variation was little 2% to 4% compared to the 500 that presented 15% to 25% of use. However, it is important to point out that comparing the results obtained for C1 and C2, the multi container microservice architecture had a lower CPU consumption, that is, it presented a better performance. This performance observation had already been indicated by the analysis of the latency times of the requests (see Table 5.4).

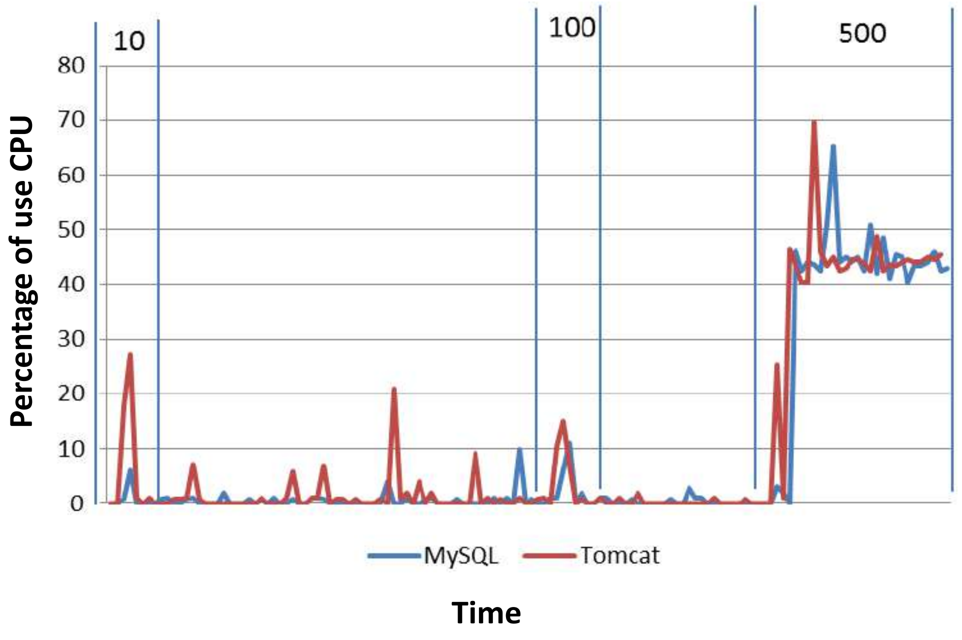


Figure 5.4: % CPU usage for user microservice on one container (C1)

For the scenarios similar to C1 and C2 in the Edge environment, that is, scenarios E1 and E2 (see Figures 5.6 and 5.7), the observed behavior was quite different. Again, the monitoring of the metrics was effective and reflected the increase in CPU usage with the increase in the number of requests. Specifically for the scenario with a container running two processes, E1, there was an expressive increase in CPU usage in relation to the tests with 10 requests, 10% of use, while by 100 the use was 60%. The test use with 500 requisitions ranged from 60% to 80%. For the E2 scenario that explores

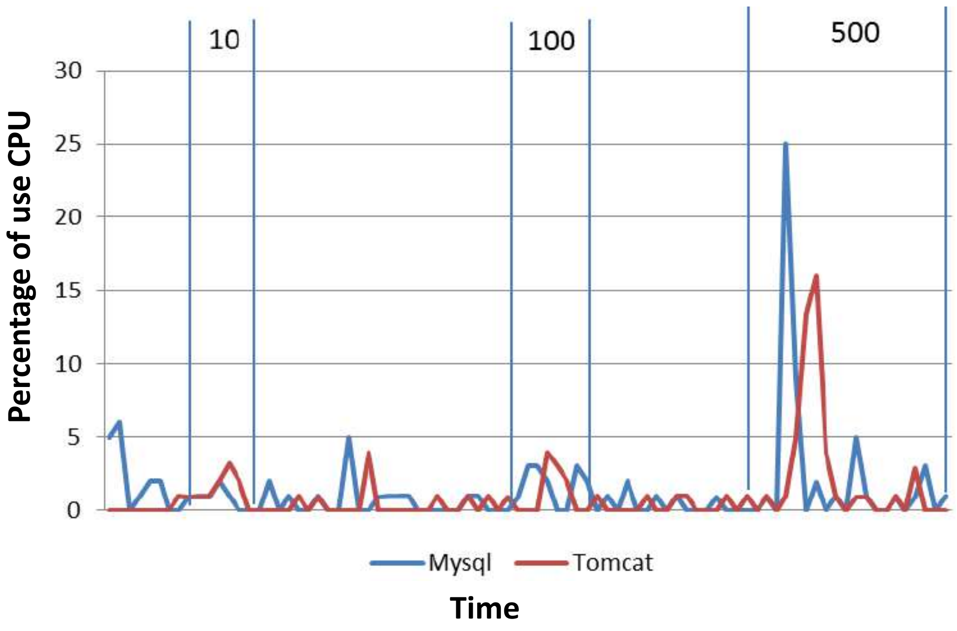


Figure 5.5: % CPU usage for user microservice on two containers (C2)

a single process running per container, a greater variation in CPU consumption was observed in relation to the metrics obtained for scenario E1, as well as a considerable increase of the test of 10, use of 18% for the test of 100, use of 70% to 80%. Still referring to the E2 scenario, the results of the 100 and 500 requisitions tests showed little increase in the variation, 70% to 80% of use for 100 and 80% to 90% of use for 500.

The variation in observed CPU usage for the S1 and S2 scenarios reveals a significant increase in the CPU utilization rate that corresponds directly to the execution period of our tests. For example, as depicted in Figure 5.8 the percentage of CPU usage by the SV service deployed under scenario S1 in the Cloud, shows maximum peaks precisely in the periods that the tests were performed. This same behavior is seen in Figure 5.9 that presents the SV results implanted in scenario S2 in the Edge environment. Although the increase in usage behavior is not as prominent as in Figure 5.8, the variation of CPU usage was again measured by the Osmotic monitoring system.

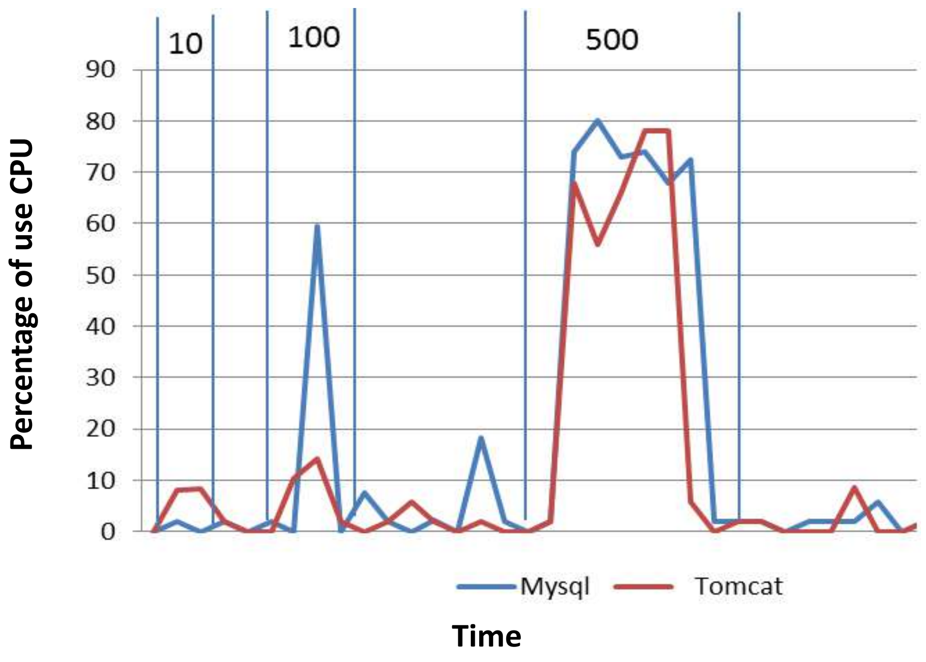


Figure 5.6: % CPU usage for parking microservice on one container (E1)

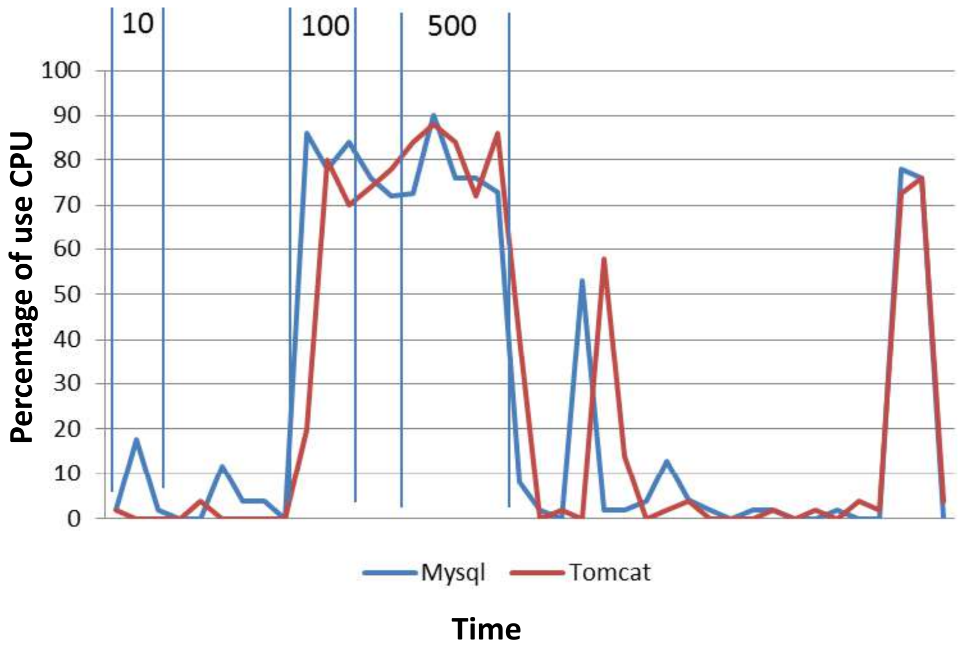


Figure 5.7: % CPU usage for parking microservice on two containers (E2)

5.4.5 Memory Results

The results obtained for the memory consumption (Figures 5.10 to 5.14) although less explicit or as elucidating as the results of CPU consumption reveal some interesting conclusions. For example, for the cloud environment in the UM service both the

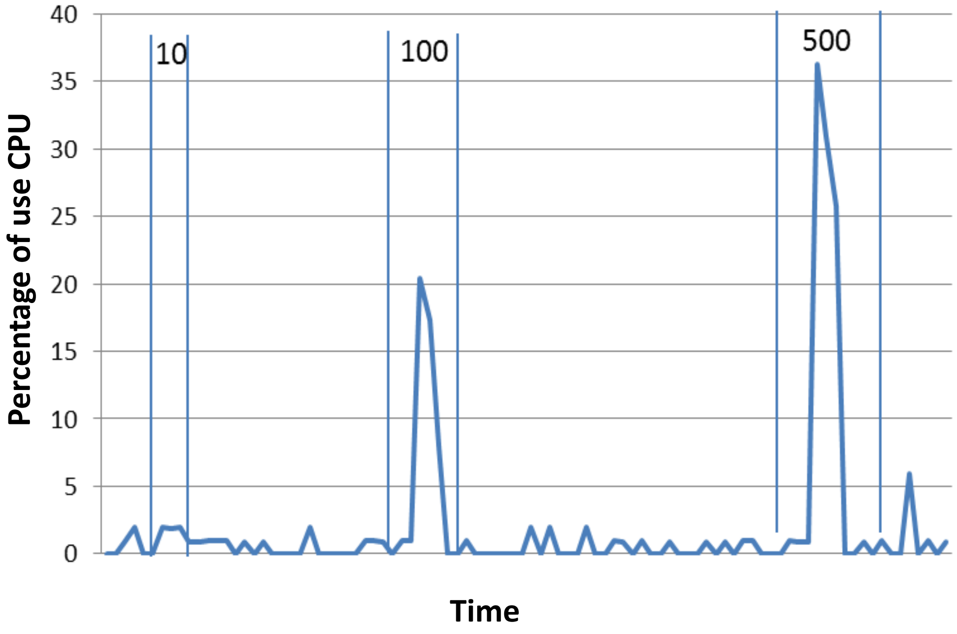


Figure 5.8: % CPU usage for selection microservice on cloud (S1)

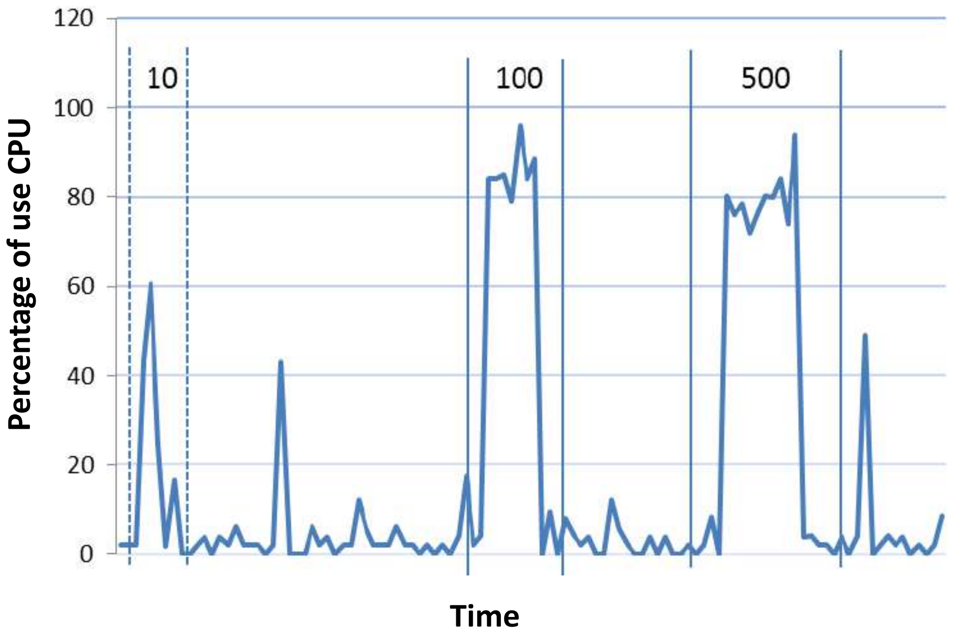


Figure 5.9: % CPU usage for selection microservice on edge (S2)

amount of memory used by the MySQL and Tomcat processes and the system was practically the same in the two scenarios evaluated, scenarios C1 (Figure 5.10) and C2 (Figure 5.11). The only significant increase occurs in Tomcat in scenario C1 in the test of 500 requests where the amount of memory consumed increase 50%, from 100 MB to 200 MB. The MySQL process practically does not suffer memory variation always consuming 200 MB. Also, system memory variation is very low, with values ranging from 1500 MB to 1650 MB. Tomcat's largest change in memory consumption can be explained by the increase in the number of requests since the requests were always of the same type, queries to MySQL were always the same, being easily managed by the database cache.

The graphs shown in Figures 5.12 and 5.13 present results of experimental scenarios E1 and E2 conducted in the Edge environment. The memory consumption of MySQL is practically the same in both scenarios, having a value of approximately 50 MB. The consumption of Tomcat also varied little in the two scenarios, being from 40 MB to 49 MB in E1 and 30 MB to 40 MB in E2. Again, a variation in memory consumption was observed by Tomcat while it did not occur with MySQL. System process consumption, system, experienced a similar variation in the two scenarios from 190 MB to 200 MB in E1 and 260 MB to 270 MB in E2. It is important to note that the same average variation was observed for the two scenarios, that is, for E1 the variation in the three tests (10,100,500) was 10 MB, as was the case for E2, a variation of 10 MB. Thus, the effectiveness of the monitoring system is once again proven, since for the same simulated tests in the two scenarios the monitored values were the same.

Another relevant observation, and the largest memory consumption in scenario E2, that is separate containers. Probably the greatest consumption occurs because of the need to keep the data of specific states of each container in memory. In other words, whereas the E1 scenario uses only one container (Tomcat + MySQL), the E2 scenario uses one container for Tomcat and another for MySQL. In E2, therefore, the memory consumption is specified by the libraries and data container state is duplicated. The memory consumption of the SV microservice (see Figure 5.14) presented a practically linear variation in the two scenarios explored. Differently from the absolute values presented in Figures 5.10, 5.11, 5.12, 5.13, and 5.14 present the results in percent of memory usage so that we can evaluate the variations of the system's performance in both cloud and Edge environments. In the cloud environment, scenario S1, consumption varied from 3% to 5%. On the Edge environment, scenario S2, consumption was between 7% and 10%. The variation measured by the monitoring system again was very similar to the same tests of 10, 100 and 500 requisitions.

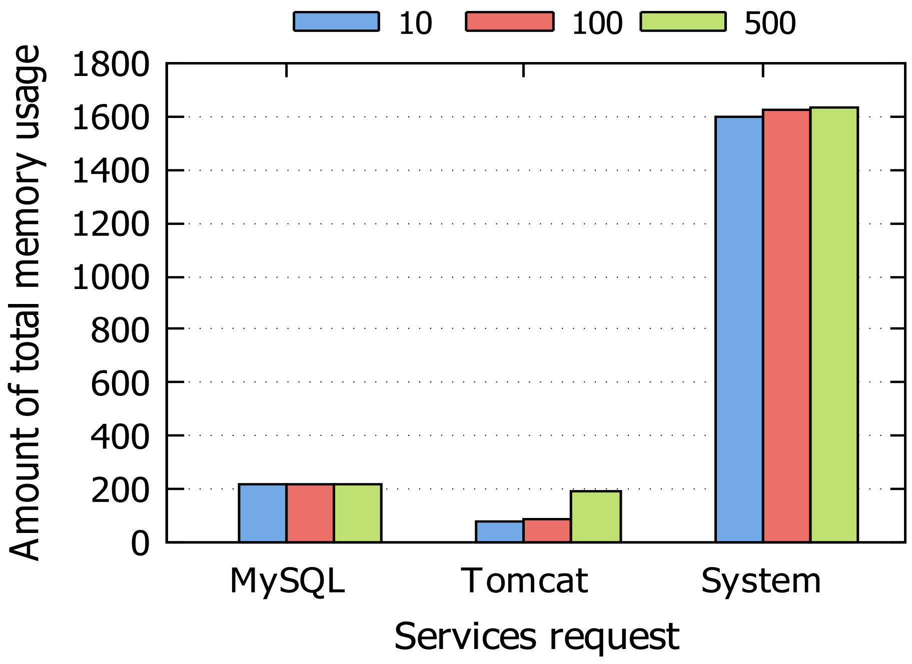


Figure 5.10: Memory usage (MB) for user microservice on one container (C1)

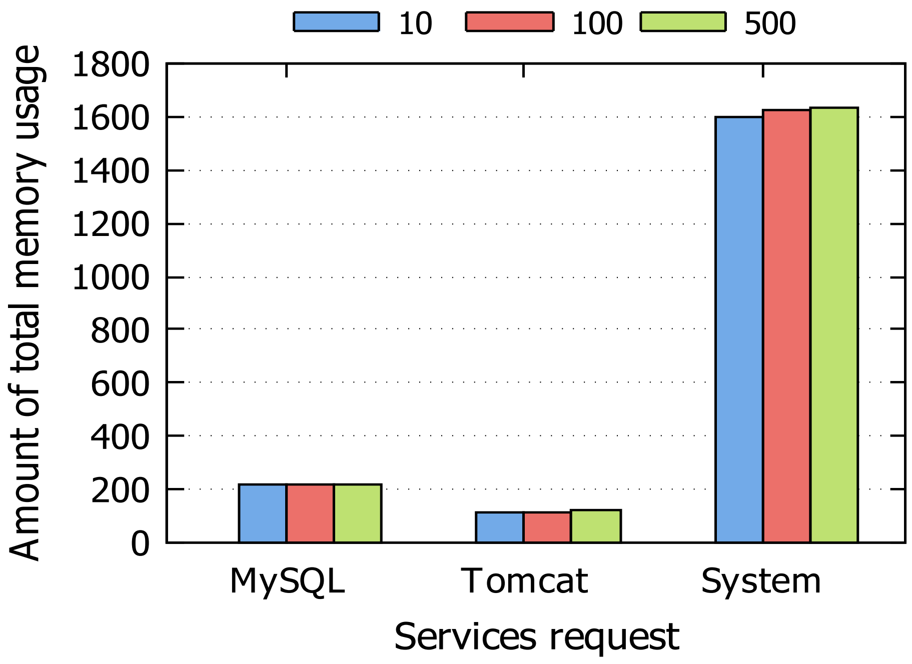


Figure 5.11: Memory usage (MB) for user microservice on two container (C2)

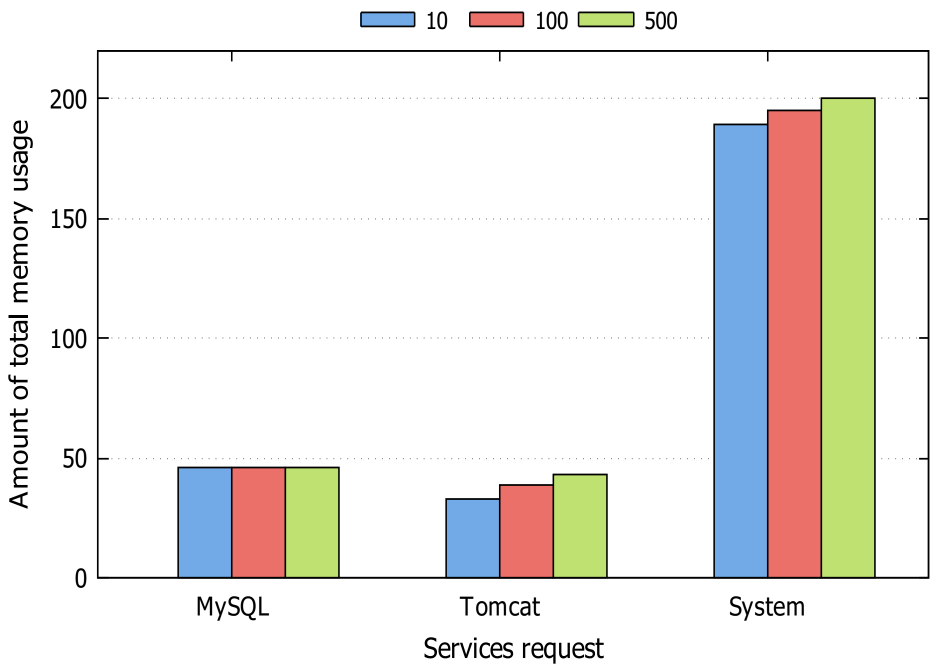


Figure 5.12: Memory usage (MB) for parking microservice on one container (E1)

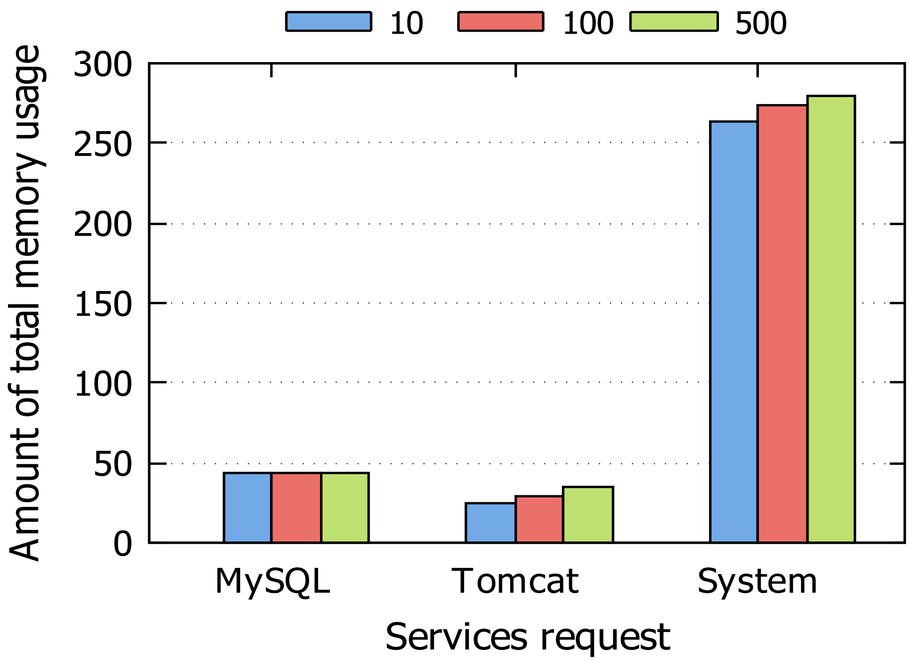


Figure 5.13: Memory usage (MB) for parking microservice on two container (E2)

5.4.6 Network Results

The results obtained in the monitoring of network traffic by the microservices were based on the network traffic of the container or the containers where the microservices were deployed. In scenarios C1, E1, S1 and S2 only the download and upload rate of

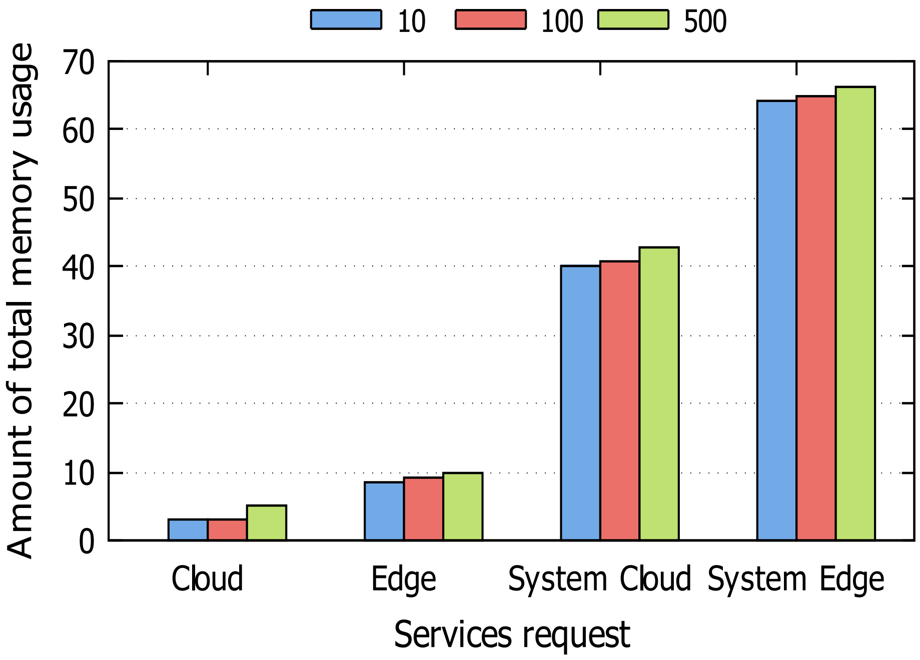


Figure 5.14: % Memory usage for selection microservice (S1 and S2)

a container is presented, while in scenarios C2 and E2 each container is MySQL or Tomcat has its own traffic rate. The first important observation is precisely related to the use of the architecture that uses more than one container for the same microservice because it allows the isolated monitoring of each process of the microservice making the perception of loadings more effective in these scenarios. Regarding the variation in traffic caused by the load tests, it was verified that the variation in the requisition numbers increased the network flow that was correctly registered by the monitoring system. For the cloud environment, scenarios C1 and C2, the metrics obtained are shown in Figure 5.15. The traffic for the 10 and 100 requisitions tests obtained little variation (10 to 80 KB for download or upload) when compared to the 500 test that you get a range of 400 KB to 500 KB. For the Edge environment, scenarios E1 and E2, the results presented in Figure 5.16 presented a similar behavior to that described for the Cloud, where only for the test of 500 requisitions was obtained a download and upload with significant variance rate. It is worth mentioning the similar behavior of the two groups of Figures 5.15 and 5.16 as to the change in the download and upload rates caused by the 10, 100 and 500 tests, especially in relation to the MySQL Container, where the graphs are practically the same in varying. In the osmotic environment, the scenarios S1 and S2, the results can be seen in Figure 5.17. The observed behavior again reflects the variation induced by the increase in the number of requisitions of the tests of 10, 100 and 500.

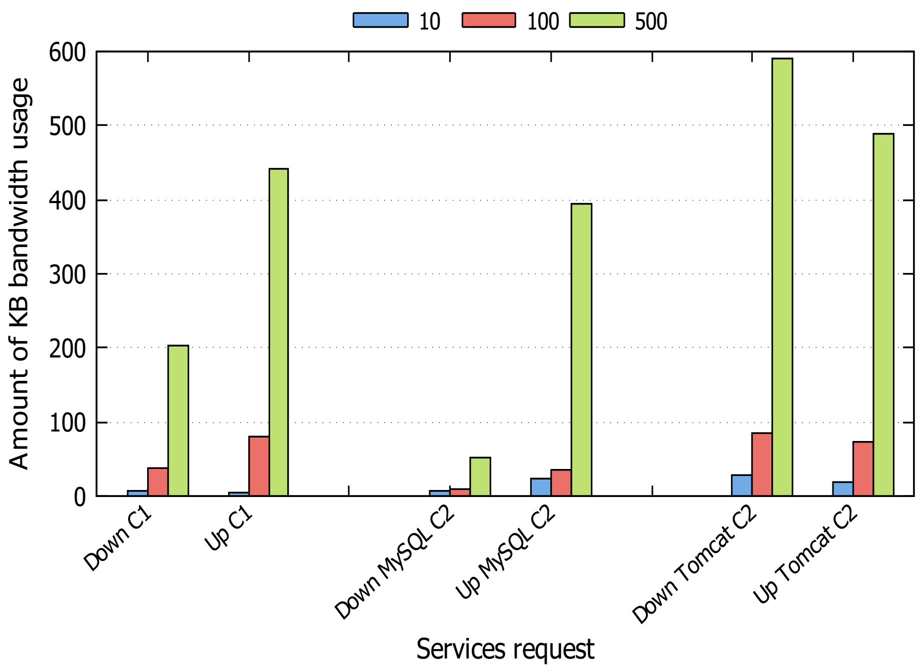


Figure 5.15: Network traffic (KB) for user microservice (C1 and C2)

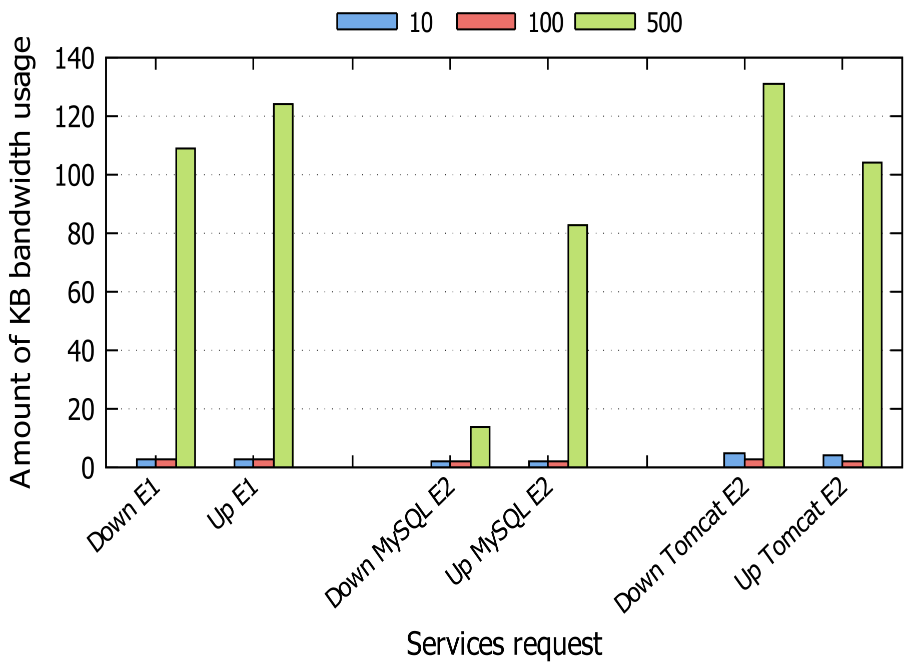


Figure 5.16: Network traffic (KB) for parking microservice (E1 and E2)

Although the results presented above are relevant, the results obtained for the osmotic service of our application more accurately highlight the effectiveness of our monitoring

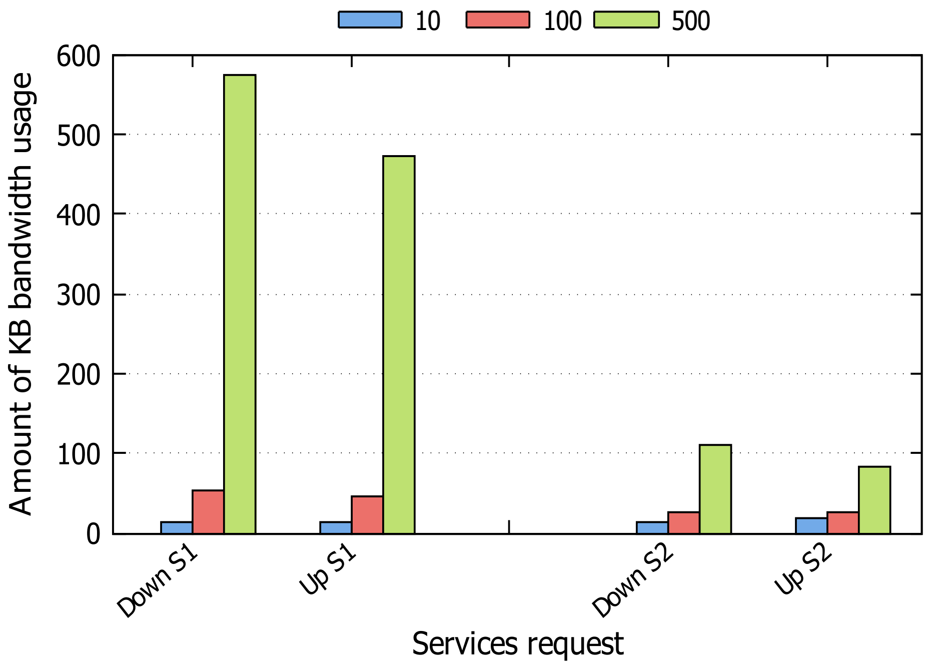


Figure 5.17: Network traffic (KB) for selection microservice (S1 and S2)

system. This behavior, although proven initially by the variation of CPU usage, is even more evident with the analysis of the variation in memory consumption.

5.4.7 Discussion

The above results validates the capability of the proposed Osmotic monitoring system in its ability to capture fine-grained performance of microservice-based IoT Application deployed in osmotic computing environment (cloud and edge), including each individual microservice of the IoT application, each underlying infrastructure e.g. databases and the performance of container/VM hosting the microservice. Moreover, the Osmotic monitoring system accurately captured several variations introduced to impact the performance of the microservices highlighting the effectiveness of our monitoring system. For example, the system was holistically able to identify variation in CPU, memory and network latency across cloud and edge at the application level, microservice level and infrastructure level (e.g. databases, containers, VM) which as identified in Section 5.2 is currently a significantly limitation with other cloud monitoring solutions. The results obtained for memory consumption and network traffic corroborate the observations obtained by the use of the CPU and therefore are not exhaustively explored in this work. However, some relevant observations are made. For example, memory consumption in the cloud was almost unchanged (only by 500 req.), while at the edge it was most successful mainly in the E2 scenario. Already the network traffic only had a big increase in the test of 500 requisitions, although the behavior of the cloud and the edge has been very similar.

5.5 Conclusion and Future Work

This work presents an integrated system for monitoring applications decomposed in microservices and executed in an osmotic environment. In its core there is a model for monitoring the QoS parameters of an application by analyzing microservices executed in containers in a cloud environment and/or on the edge. This chapter introduces an smart parking application that runs in an osmotic environment in the context of smart cities. This osmotic application is used for an experimental evaluation of the monitoring system in order to demonstrate the effectiveness of the proposed approach. This case study shows that it is possible to apply our approach for microservices deployed in osmotic computing environments. Through experimental evaluations we validates the effectiveness and capability of the proposed monitoring system's ability to monitor the performance of microservice deployment using containers and/or VM's through an exhaustive list of scenarios. The osmotic monitoring system was holistically able to identify variation in CPU, memory and network latency across cloud and edge at the application level, microservice level and infrastructure level (e.g. databases, containers, VM). The major limitation that can be explored in future works that the evaluation of the tool's impact on the execution of applications, regarding the use of resources by the tool.

Our future work will expand the model, especially for device monitoring, and ensure an extended evaluation through the execution of new load tests. Also, the raffle approach is recommended to create a comparative isonomy in an environment closer to a production environment.

6

Conclusion

Contents

Summary

In this chapter, we summarize the research work presented in this thesis. Then, we outline the contributions and discuss open research problems in the field that could guide future work.

6.1 Thesis Summary

Modern applications can be distributed across multiple cloud environments including bare metal, public or private cloud depending on several features such as microservices component requirements, deployment locations, security concerns, cost, etc. Different cloud providers have their own way of handling deployment and management of microservices components. Due to the heterogeneity of cloud providers / cloud - edge and heterogeneous environments (VM/container), monitoring microservices is challenging as it requires efficient and scalable techniques that undermine the heterogeneity of underlying infrastructure. Also, despite the strongest scalability characteristic of this model which opens the doors for further optimizations in both application structure and performance, such characteristic adds an additional level of complexity to monitoring application performance. Performance monitoring system can lead to severe application outages if it is not able to successfully and quickly detecting failures and localizing their causes.

In this thesis, we explored numerous challenges for monitoring microservices in multi-cloud environment, cloud - edge, and anomalies detection, and we proposed solutions that ease the monitoring process and detect, identify and locate the anomalies. In particular, this thesis contributes as:

Chapter 2 presents background information concerning the overall topic, including a brief description on monitoring, virtualization techniques, microservices, the underlying cloud and edge computing environment, and industry and academic monitoring tools. A major focus of this thesis is to address the challenges of microservice monitoring and detection in cloud-edge infrastructure. Monitoring regulates the performance of cloud-based microservices in software and hardware resources. It includes information on the monitoring resource's status/health, such as the CPU and memory use for the microservice deployed on the cloud and edge platform.

Chapter 3 presents a generic monitoring framework, *Multi-microservices Multi-virtualization Multi-cloud (M3)* that monitors the performance of microservices deployed across heterogeneous virtualization platforms in a multi-cloud environment. We validated the efficacy and efficiency of *M3* using a Book-Shop application executing across AWS and Azure. In addition, we significantly extended *M3* by implemented of highway traffic monitoring services using a cyber-physical system. So, we propose M2CPA - a novel framework for multi-virtualization, and multi-cloud monitoring in cloud-based cyber-physical systems. M2CPA monitors the performance of application components running inside multiple virtualization platforms deployed on multiple clouds. M2CPA is validated through extensive experimental analysis using a real testbed comprising multiple public clouds and multi-virtualization technologies.

Chapter 4 presents a Monitoring and Anomaly Detection and Localization System (MADLS) which utilises a simplified approach that depends on commonly available metrics offering a simplified deployment environment for the developer. Our data collection system uses a monitor engine that monitors response time and throughput in the application layer as well as CPU and memory in the physical layer. We evaluate our approach through a bookstore web application case study. We finally validate *MADLS* to show that *MADLS* can accurately detect anomalies in response time then specify the type and location of fault that exists in microservices.

Chapter 5 presents an integrated monitoring system for monitoring IoT applications decomposed as microservices and executed in an osmotic computing environment. A real-world smart parking IoT application is used for an experimental evaluation and for demonstrating the effectiveness of the proposed approach. Through rigorous experimental evaluation, we validate the Osmotic monitoring system ability to holistically identify variation in CPU, memory, and network latency of microservices deployed across Cloud and Edge layers.

6.2 Future Research Directions

We provide motivation for a number of areas of future research, which can be inspired by the work done in this PhD thesis.

6.2.1 Fault Injector

Fault injection is a method created by researchers and engineers to evaluate the dependability of the hardware or software of computer systems. The fault injection at the software is less expensive in comparison to hardware as it requires changes at the software-state level. What really happens after a fault is injected and how a fault propagates in a software system are not well understood and that's why we need monitoring to evaluate the loss of performance due to an injected fault. If each containers have circuit breaker and if isolate this container what does it do for my workload. So, we can create circuit breaker design pattern where if component or microservices become unresponsive supplier which can run out of critical resources leading to cascading failures across multiple systems. Future work can provide circuit breaker to isolate the problem.

6.2.2 Monitoring Containerized Big Data Systems

Modern big data process systems are becoming very complex in terms of large-scale, high-concurrency and Multiple-talents. Thus, many failures and performance reductions only happen at run-time and are very difficult to capture. Moreover, some issues may only be triggered when some components are executed. To analyze the root cause of these types of issues, we have to capture the dependencies of each component in real-time. The fault detection in big data systems, however, is very hard due to the considerable scale, the distributed environment and the large number of concurrent jobs. The *emergent failures* happen when the errors exceed the propagation boundaries during the interaction among hardware and software components, and can only be identified at run-time. In order to detect the *emergent failures* or underlying reasons for the performance reduction, we need to have a comprehensive and consistent monitoring plan to collect the information from each individual process job, while storing, maintaining and analyzing very large volumes of the monitoring data. As future work, Spark applications will be monitored in Yarn cluster using Docker technology for big data analytics to collect real-time infrastructure information for each executor, such as the execution time, progress, status, along with computing resource metrics, such as CPU/memory usage, bandwidth and disk availability, etc. This work aims to track end-to-end performance monitoring of big data systems and provide essential information to root cause the reasons for performance reduction in big data systems in a short time and efficiently by taking advantage of containerization, such as immutability, utilization, portability, performance and scalability.

6.2.3 Diagnosis Framework for Container-based Microservices with Performance Monitoring using Deep Learning

In current work, we present Monitoring, Anomaly Detection and Localization System (*MADLS*). This system provides two components: Monitoring component to monitor container-based microservices, and diagnosing component to detect anomalies and localize anomalies' root causes. For detection and localization, the system uses the multi-resolution approach to analyse application performance metrics from the application level and container level in stages using supervised machine learning. Future work can extend our system by using deep learning that is a good at finding those faulty patterns, which is not in the datasets. Because supervised learning can only find the known force, the promise of deep learning can find the unknown force as well. For example, you have an application is running in Amazon and Azure. If something goes wrong in Amazon, your deep learning fix it and, then your deep learning in the Azure can learn from it.

6.2.4 Microservice Migration Method to Balance the Workload of the Microservices by using Osmotic Services Composition

Edge computing brings computation closer to the physical world by moving it away from the Cloud. As a result, the cost of communicating bandwidth between IoT devices and the Cloud is reduced. Edge computing, on the other hand, imposes significant limitations in compute capability due to the devices' limited hardware capacity. This limitation may have a substantial impact on the performance of deployed apps, particularly Smart City applications. This restriction may be exacerbated further by unpredictability in human behaviour, which may quickly overload the Edge computing node. Future research, the concept of osmotic computing is to create a dynamic load balancing platform for Smart City applications. This platform may make use of containerization, which enables developers to quickly relocate or plan the execution of microservices across various computing resources on demand.

References

[1] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "Automap: Diagnose your microservice-based web applications automatically," in *Proceedings of The Web Conference 2020*, 2020, pp. 246–258.

[2] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *Algorithms and Architectures for Parallel Processing*, J. Vaidya and J. Li, Eds. Cham: Springer International Publishing, 2018, pp. 560–572.

[3] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," in *2017 International Conference on Computing, Communication and Automation (ICCCA)*, 2017, pp. 847–852.

[4] Y. Xing and Y. Zhan, "Virtualization and cloud computing," in *Future Wireless Networks and Information Systems.* Springer, 2012, pp. 305–312.

[5] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[6] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 275–287.

[7] D. Puthal, S. Nepal, R. Ranjan, and J. Chen, "Threats to networking cloud and edge datacenters in the internet of things," *IEEE Cloud Computing*, vol. 3, no. 3, pp. 64–71, 2016.

[8] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "{FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 805–825.

[9] K. Zhang, M. Wan, T. Qu, H. Jiang, P. Li, Z. Chen, J. Xiang, X. He, C. Li, and G. Q. Huang, "Production service system enabled by cloud-based smart resource hierarchy for a highly dynamic synchronized production process," *Advanced Engineering Informatics*, vol. 42, p. 100995, 2019.

[10] C. Qian, Y. Zhang, Y. Liu, and Z. Wang, "A cloud service platform integrating additive and subtractive manufacturing with high resource efficiency," *Journal of Cleaner Production*, vol. 241, p. 118379, 2019.

[11] J. Mateo-Fornés, F. Solsona-Tehàs, J. Vilaplana-Mayoral, I. Teixidó-Torrelles, and J. Rius-Torrentó, "Cart, a decision sla model for saas providers to keep qos regarding availability and performance," *IEEE Access*, vol. 7, pp. 38 195–38 204, 2019.

[12] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, “Cloud monitoring: A survey,” *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[13] B. Plattner and J. Nievergelt, “Special feature: Monitoring program execution: A survey,” *Computer*, no. 11, pp. 76–93, 1981.

[14] S. Benbernou, L. Hacid, R. Kazhamiakin, G. Kecskemeti, J. Poizat, F. Silvestri, M. Uhlig, and B. Wetzstein, “State of the art report, gap analysis of knowledge on principles, techniques and methodologies for monitoring and adaptation of sbas,” *S-Cube Consortium, Deliverable PO-JRA-1.2*, vol. 1, 2008.

[15] E. Wolff, *Microservices: flexible software architecture.* Addison-Wesley Professional, 2016.

[16] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[17] A. Samir and C. Pahl, “Anomaly detection and analysis for clustered cloud computing reliability,” *CLOUD COMPUTING 2019*, p. 120, 2019.

[18] U. Demirbaga, A. Noor, Z. Wen, P. James, K. Mitra, and R. Ranjan, “Smartmonit: Real-time big data monitoring system,” in *2019 38th Symposium on Reliable Distributed Systems (SRDS).* IEEE, 2019, pp. 357–3572.

[19] “Docker. [online]. available: https://www.docker.com.”

[20] “cadvisor (container advisor). [online]. available: https://github.com/google/cadvisor.”

[21] “Datadog. [online]. available: https://www.datadoghq.com/.”

[22] “Amazon cloudwatch. [online]. available: https://aws.amazon.com/ cloudwatch.”

[23] K. Alhamazani, R. Ranjan, K. Mitra, P. P. Jayaraman, Z. Huang, L. Wang, and F. Rabhi, “Clams: Cross-layer multi-cloud application monitoring-as-a-service framework,” in *Services Computing (SCC), 2014 IEEE International Conference on.* IEEE, 2014, pp. 283–290.

[24] M. Großmann and C. Klug, “Monitoring container services at the network edge,” in *Teletraffic Congress (ITC 29), 2017 29th International*, vol. 1. IEEE, 2017, pp. 130–133.

[25] L. Knight, P. Štefanic, M. Cigale, A. C. Jones, and I. Taylor, “Towards a methodology for creating time-critical, cloud-based cuda applications.”

[26] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, “Sieve: Actionable insights from monitored metrics in distributed systems,” vol. 14, 2017. [Online]. Available: μhttps://sieve-microservices.github.io/.

[27] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Microrca: Root cause localization of performance issues in microservices," 2020. [Online]. Available: μhttps://hal.inria.fr/hal-02441640

[28] C. Sauvanaud, M. Kaâniche, K. Kanoun, K. Lazri, and G. D. S. Silvestre, "Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned," *Journal of Systems and Software*, vol. 139, pp. 84–106, 2018.

[29] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei, "Localizing failure root causes in a microservice through causality inference," 2020.

[30] D. S. Linthicum, "Practical use of microservices in moving workloads to the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 6–9, 2016.

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

[32] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, Nov 2016.

[33] X. Zeng, S. K. Garg, M. Barika, S. Bista, D. Puthal, A. Zomaya, and R. Ranjan, "Detection of sla violation for big data analytics applications in cloud," *IEEE Transactions on Computers*, 2020.

[34] T. Wang, W. Zhang, C. Ye, J. Wei, H. Zhong, and T. Huang, "Fd4c: Automatic fault diagnosis framework for web applications in cloud computing," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 1, pp. 61–75, 2015.

[35] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[36] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu, "Vmware distributed resource management: Design, implementation, and lessons learned," *VMware Technical Journal*, vol. 1, no. 1, pp. 45–64, 2012.

[37] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: the linux virtual machine monitor, in proceedings of the linux symposium," 2007.

[38] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2006, pp. 101–112.

[39] A. Mouat, *Using Docker: Developing and Deploying Software with Containers.* " O'Reilly Media, Inc.", 2015.

[40] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “Devops,” *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016.

[41] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, “Open issues in scheduling microservices in the cloud,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

[42] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, “Application deployment using microservice and docker containers: Framework and optimization,” *Journal of Network and Computer Applications*, vol. 119, pp. 97–109, 2018.

[43] M. Julian, *Practical Monitoring: Effective Strategies for the Real World.* ” O’Reilly Media, Inc.”, 2017.

[44] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, “A survey of cloud monitoring tools: Taxonomy, capabilities and objectives,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2918–2933, 2014.

[45] A. Bertolino, “Software testing and/or software monitoring: Differences and commonalities,” *Jornadas Sistedes, Cádiz*, 2014.

[46] M. Tavana, V. Hajipour, and S. Oveisi, “Iot-based enterprise resource planning: Challenges, open issues, applications, architecture, and future research directions,” *Internet of Things*, p. 100262, 2020.

[47] X. Liu and R. Buyya, “Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–41, 2020.

[48] B. Sang, P.-L. Roman, P. Eugster, H. Lu, S. Ravi, and G. Petri, “Plasma: programmable elasticity for stateful cloud computing applications,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.

[49] J. Hollender, B. van Bavel, V. Dulio, E. Farmen, K. Furtmann, J. Koschorreck, U. Kunkel, M. Krauss, J. Munthe, M. Schlabach *et al.*, “High resolution mass spectrometry-based non-target screening can support regulatory environmental monitoring and chemicals management,” *Environmental Sciences Europe*, vol. 31, no. 1, p. 42, 2019.

[50] J. E. Rubio, R. Roman, and J. Lopez, “Integration of a threat traceability solution in the industrial internet of things,” *IEEE Transactions on Industrial Informatics*, 2020.

[51] R. Bose, S. Sahana, and D. Sarddar, “An adaptive cloud service observation using billboard manager cloud monitoring tool,” *Int. J. Softw. Eng. Appl.*, vol. 9, no. 7, pp. 159–170, 2015.

[52] A. Orso, “Monitoring, analysis, and testing of deployed software,” in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 263–268.

[53] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, and H. Zeisel, "Reminds: A flexible runtime monitoring framework for systems of systems," *Journal of Systems and Software*, vol. 112, pp. 123–136, 2016.

[54] A. Verma and N. Bhardwaj, "A review on routing information protocol (rip) and open shortest path first (ospf) routing protocol," *International Journal of Future Generation Communication and Networking*, vol. 9, no. 4, pp. 161–170, 2016.

[55] S. Newman, *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[56] A. Sari, "A review of anomaly detection systems in cloud networks and survey of cloud security measures in cloud storage applications," *Journal of Information Security*, vol. 6, pp. 142–154, 04 2015.

[57] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[58] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *International Conference on Algorithms and Architectures for Parallel Processing.* Springer, 2018, pp. 560–572.

[59] H. Zhou, *Learn Data Mining Through Excel.* Apress, 2020.

[60] V. K. Ayyadevara, *Decision Tree.* Berkeley, CA: Apress, 2018.

[61] J. P. Ciaburro, Giuseppe, "2.1 technical requirements," 2019. [Online]. Available: µhttps://app.knovel.com/hotlink/khtml/id:kt011YY4B1/python-machine-learning/constructi-technical-requirements

[62] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011.

[63] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[64] L. T. Yang, X. Wang, X. Chen, L. Wang, R. Ranjan, X. Chen, and M. J. Deen, "A multi-order distributed hosvd with its incremental computing for big services in cyber-physical-social systems," *IEEE Transactions on Big Data*, 2018.

[65] P. Wang, L. T. Yang, J. Li, J. Chen, and S. Hu, "Data fusion in cyber-physical-social systems: State-of-the-art and perspectives," *Information Fusion*, vol. 51, pp. 42–57, 2019.

[66] E. Simmon, K.-S. Kim, E. Subrahmanian, R. Lee, F. De Vaulx, Y. Murakami, K. Zettsu, and R. D. Sriram, *A vision of cyber-physical cloud computing for smart networked systems.* US Department of Commerce, National Institute of Standards and Technology, 2013.

[67] X. Wang, W. Wang, L. T. Yang, S. Liao, D. Yin, and M. J. Deen, "A distributed hosvd method with its incremental computation for big data in cyber-physical-social systems," *IEEE Transactions on Computational Social Systems*, vol. 5, no. 2, pp. 481–492, 2018.

[68] N. Kumar, S. Zeadally, and J. J. Rodrigues, "Vehicular delay-tolerant networks for smart grid data management using mobile edge computing," *IEEE Communications Magazine*, vol. 54, no. 10, pp. 60–66, 2016.

[69] N. Kumar, N. Chilamkurti, and S. C. Misra, "Bayesian coalition game for the internet of things: an ambient intelligence-based evaluation," *IEEE Communications Magazine*, vol. 53, no. 1, pp. 48–55, 2015.

[70] "Microsoft azure. [online]. available: https://azure.microsoft.com/."

[71] K. Alhamazani, R. Ranjan, P. P. Jayaraman, K. Mitra, F. Rabhi, D. Georgakopoulos, and L. Wang, "Cross-layer multi-cloud real-time application qos monitoring and benchmarking as-a-service framework," *IEEE Transactions on Cloud Computing*, 2015.

[72] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, vol. 136, pp. 19 – 38, 2018. [Online]. Available: µhttp://www.sciencedirect.com/science/article/pii/S016412121730256X

[73] E. Preeth, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of docker containers based on hardware utilization," in *Control Communication & Computing India (ICCC), 2015 International Conference on.* IEEE, 2015, pp. 697–700.

[74] K. Lee and K. Gilleade, "Generic processing of real-time physiological data in the cloud," *International Journal of Big Data Intelligence*, vol. 3, no. 4, pp. 215–227, 2016.

[75] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[76] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, vol. 136, pp. 19–38, 2018.

[77] R. Bhatnagar and J. Patel, "Performance analysis of a grid monitoring system-ganglia," *International Journal of Emerging Technology and Advanced Engineering*, vol. 3, no. 8, pp. 362–365, 2013.

[78] H. J. Syed, A. Gani, R. W. Ahmad, M. K. Khan, and A. I. A. Ahmed, "Cloud monitoring: A review, taxonomy, and open research issues," *Journal of Network and Computer Applications*, vol. 98, pp. 11–26, 2017.

[79] G. Iuhasz and I. Dragan, “An overview of monitoring tools for big data and cloud applications,” in *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2015, pp. 363–366.

[80] “Scout. [online]. available: https://scoutapp.com/.”

[81] S.-k. Kwon and J.-h. Noh, “Implementation of monitoring system for cloud computing environments,” *International Journal of Modern Engineering Research (IJMER)*, vol. 3, no. 4, pp. 1916–1918, 2013.

[82] F. D. Rossi, I. C. OLIVEIRA, C. A. F. De Rose, R. N. Calheiros, and R. Buyya, “Non-invasive estimation of cloud applications performance via hypervisor’s operating systems counters,” in *Proceedings of the Fourteenth International Conference on Networking 2015, 2015, Espanha.*, 2015.

[83] Z. Zou, Y. Xie, K. Huang, G. Xu, D. Feng, and D. Long, “A docker container anomaly monitoring system based on optimized isolation forest,” *IEEE Transactions on Cloud Computing*, 2019.

[84] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, “Elascale: autoscaling and monitoring as a service,” *arXiv preprint arXiv:1711.03204*, 2017.

[85] “Elastic. [online]. available: https://www.elastic.co/.”

[86] “Sysdig. [online]. available: https://sysdig.com/opensource/.”

[87] “Netflix hystrix. [online]. available: https://github.com/netflix/hystrix.”

[88] “Sensu. [online]. available: https://sensuapp.org.”

[89] J. Qiu, Q. Du, K. Yin, S.-L. Zhang, and C. Qian, “A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications,” *Applied Sciences*, vol. 10, p. 2166, 3 2020. [Online]. Available: μhttps://www.mdpi.com/2076-3417/10/6/2166

[90] W. Cao, Z. Cao, and X. Zhang, “Research on microservice anomaly detection technology based on conditional random field,” *Journal of Physics*, 2019.

[91] A. Karmel, R. Chandramouli, and M. Iorga, “Nist special publication 800-180: Nist definition of microservices, application containers and virtual machines,” 2016.

[92] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices architecture enables devops: Migration to a cloud-native architecture,” *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[93] O. Barais, J. Bourcier, Y.-D. Bromberg, and C. Dion, “Towards microservices architecture to transcode videos in the large at low costs,” in *Telecommunications and Multimedia (TEMU), 2016 International Conference on*. IEEE, 2016, pp. 1–6.

[94] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon, 2016.* IEEE, 2016, pp. 1–5.

[95] G. Pallis, D. Trihinas, A. Tryfonos, and M. Dikaiakos, "Devops as a service: Pushing the boundaries of microservice adoption," *IEEE Internet Computing*, vol. 22, no. 3, pp. 65–71, 2018.

[96] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *Cloud Engineering (IC2E), 2016 IEEE International Conference on.* IEEE, 2016, pp. 202–211.

[97] G. D. P. Regulation, "Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46," *Official Journal of the European Union (OJ)*, vol. 59, no. 1-88, p. 294, 2016.

[98] Q. Zhang, C. Zhu, L. T. Yang, Z. Chen, L. Zhao, and P. Li, "An incremental cfs algorithm for clustering large data in industrial internet of things," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 3, pp. 1193–1201, 2017.

[99] S. K. Mitra and C. Åhlund, "A mobile cloud computing system for emergency management," *IEEE Cloud Computing*, vol. 1, no. 4, pp. 30–38, 2014.

[100] R. Amin, S. H. Islam, G. Biswas, M. K. Khan, and N. Kumar, "A robust and anonymous patient monitoring system using wireless medical sensor networks," *Future Generation Computer Systems*, vol. 80, pp. 483–495, 2018.

[101] P. Leitão, A. W. Colombo, and S. Karnouskos, "Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges," *Computers in Industry*, vol. 81, pp. 11–25, 2016.

[102] X. Yao, J. Zhou, Y. Lin, Y. Li, H. Yu, and Y. Liu, "Smart manufacturing based on cyber-physical systems and beyond," *Journal of Intelligent Manufacturing*, pp. 1–13, 2017.

[103] B. R. Dawadi, S. Shakya, and R. Paudyal, "Common: The real-time container and migration monitoring as a service in the cloud," *Journal of the Institute of Engineering*, vol. 12, no. 1, pp. 51–62, 2016.

[104] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on.* IEEE, 2017, pp. 243–246.

[105] K. Mitra, S. Saguna, C. Åhlund, and R. Ranjan, "Alpine: A bayesian system for cloud performance diagnosis and prediction," in *2017 IEEE International Conference on Services Computing (SCC)*, June 2017, pp. 281–288.

[106] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-service in cloud computing: modeling techniques and their applications," *Journal of Internet Services and Applications*, vol. 5, no. 1, p. 11, 2014.

[107] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang, "Energy-aware autonomic resource allocation in multitier virtualized environments." *IEEE Trans. Services Computing*, vol. 5, no. 1, pp. 2–19, 2012.

[108] A. A. Ibrahim, S. Varrette, and P. Bouvry, "Presence: toward a novel approach for performance evaluation of mobile cloud saas web services," in *2018 International Conference on Information Networking (ICOIN)*. IEEE, 2018, pp. 50–55.

[109] A. A. Z. A. Ibrahim, S. Varrette, and P. Bouvry, "On verifying and assuring the cloud sla by evaluating the performance of saas web services across multi-cloud providers," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 69–70.

[110] M. Al-Ayyoub, Y. Jararweh, M. Daraghmeh, and Q. Althebyan, "Multi-agent based dynamic resource provisioning and monitoring for cloud computing systems infrastructure," *Cluster Computing*, vol. 18, no. 2, pp. 919–932, 2015.

[111] H. Yan, Y. Zhang, H. Wang, B. Ding, and H. Mi, "S3r: storage-sensitive services redeployment in the cloud." *IJBDI*, vol. 4, no. 4, pp. 250–262, 2017.

[112] J. Li and M. Hou, "Improving data availability for deduplication in cloud storage," *International Journal of Grid and High Performance Computing (IJGHPC)*, vol. 10, no. 2, pp. 70–89, 2018.

[113] M. Al-Shablan, Y. Tian, and M. Al-Rodhaan, "Secure multi-owner-based cloud computing scheme for big data," *International Journal of Big Data Intelligence*, vol. 3, no. 3, pp. 182–189, 2016.

[114] R. Amin, S. H. Islam, G. Biswas, M. K. Khan, and N. Kumar, "An efficient and practical smart card based anonymity preserving user authentication scheme for tmis using elliptic curve cryptography," *Journal of medical systems*, vol. 39, no. 11, p. 180, 2015.

[115] S. Challa, M. Wazid, A. K. Das, N. Kumar, A. G. Reddy, E.-J. Yoon, and K.-Y. Yoo, "Secure signature-based authenticated key establishment scheme for future iot applications," *IEEE Access*, vol. 5, pp. 3028–3043, 2017.

[116] Y. Yang, X. Zheng, V. Chang, and C. Tang, "Semantic keyword searchable proxy re-encryption for postquantum secure cloud storage," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 19, p. e4211, 2017.

[117] Y. Yang, X. Liu, X. Zheng, C. Rong, and W. Guo, "Efficient traceable authorization search system for secure cloud storage," *IEEE Transactions on Cloud Computing*, 2018.

[118] M. Natu, R. K. Ghosh, R. K. Shyamsundar, and R. Ranjan, "Holistic performance monitoring of hybrid clouds: Complexities and future directions," *IEEE Cloud Computing*, vol. 3, no. 1, pp. 72–81, 2016.

[119] J. Pendlebury, V. C. Emeakaroha, D. O'Shea, N. Cafferkey, J. P. Morrison, and T. Lynn, "Somba-automated anomaly detection for cloud quality of service," in *Cloud Computing Technologies and Applications (CloudTech), 2016 2nd International Conference on*. IEEE, 2016, pp. 71–79.

[120] N. Kratzke and R. Peinl, "Clouns-a cloud-native application reference model for enterprise architects," in *Enterprise Distributed Object Computing Workshop (EDOCW), 2016 IEEE 20th International*. IEEE, 2016, pp. 1–10.

[121] L. Li, T. Tang, and W. Chou, "A rest service framework for fine-grained resource management in container-based cloud," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 645–652.

[122] D. Richter, M. Konrad, K. Utecht, and A. Polze, "Highly-available applications on unreliable infrastructure: Microservice architectures in practice," in *Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on*. IEEE, 2017, pp. 130–137.

[123] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*. IEEE, 2015, pp. 27–34.

[124] H. G. Mohammadi, P.-E. Gaillardon, and G. De Micheli, "Efficient statistical parameter selection for nonlinear modeling of process/performance variation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1995–2007, 2016.

[125] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Cloud Engineering (IC2E), 2018 IEEE International Conference on*. IEEE, 2018, pp. 159–169.

[126] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 480–491.

[127] C. Streiffer, R. Raghavendra, T. Benson, and M. Srivatsa, "Learning to simplify distributed systems management," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 1837–1845.

[128] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017.

[129] P. Štefanič, M. Cigale, A. C. Jones, L. Knight, I. Taylor, C. Istrate, G. Suciu, A. Ulisses, V. Stankovski, S. Taherizadeh *et al.*, "Switch workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications," *Future Generation Computer Systems*, vol. 99, pp. 197–212, 2019.

[130] F. Pina, J. Correia, R. Filipe, F. Araujo, and J. Cardroom, "Nonintrusive monitoring of microservice-based systems," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2018, pp. 1–8.

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

[132] A. Noor, D. N. Jha, K. Mitra, P. P. Jayaraman, A. Souza, R. Ranjan, and S. Dustdar, "A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 156–163.

[133] P. Koopman and M. Wagner, "Autonomous vehicle safety: An interdisciplinary challenge," *IEEE Intelligent Transportation Systems Magazine*, vol. 9, no. 1, pp. 90–96, 2017.

[134] L. Rokach and O. Z. Maimon, *Data mining with decision trees: theory and applications*. World scientific, 2008, vol. 69.

[135] S. Brady, D. Magoni, J. Murphy, H. Assem, and A. O. Portillo-Dominguez, "Analysis of machine learning techniques for anomaly detection in the internet of things," in *2018 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*. IEEE, 2018, pp. 1–6.

[136] H. Arasteh, V. Hosseinnezhad, V. Loia, A. Tommasetti, O. Troisi, M. Shafie-khah, and P. Siano, "Iot-based smart cities: a survey," in *Environment and Electrical Engineering (EEEIC), 2016 IEEE 16th International Conference on*. IEEE, 2016, pp. 1–6.

[137] E. F. Z. Santana, A. P. Chaves, M. A. Gerosa, F. Kon, and D. S. Milojicic, "Software platforms for smart cities: Concepts, requirements, challenges, and a unified reference architecture," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, p. 78, 2017.

[138] C. Balakrishna, "Enabling technologies for smart city services and applications," in *Next Generation Mobile Applications, Services and Technologies (NGMAST), 2012 6th International Conference on*. IEEE, 2012, pp. 223–227.

[139] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, *Fog Computing: A Platform for Internet of Things and Analytics*. Cham: Springer International Publishing, 2014, pp. 169–186. [Online]. Available: μhttps://doi.org/10.1007/978-3-319-05029-4_7

[140] W. He, G. Yan, and L. Da Xu, "Developing vehicular data cloud services in the iot environment," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1587–1595, 2014.

[141] "Sensor. [online]. available: https://www.piborg.org/sensors-1136/xloborg."

[142] "Mongod. [online]. available: https://www.mongodb.com/."

[143] "What-docker. [online]. available: https://www.docker.com/what-docker."