A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments

Ayman Noor*†, Devki Nandan Jha*, Karan Mitra‡, Prem Prakash Jayaraman§, Arthur Souza¶, Rajiv Ranjan* and Schahram Dustdar‖

*Newcastle University, Newcastle upon Tyne, UK,
†Taibah University, Madinah, Saudi Arabia,
‡Luleå University of Technology, Skellefteå, Sweden,
§Swinburne University of Technology, Melbourne, Australia,
¶ Federal University of Rio Grande do Norte, Brazil,
‖TU Wien, Austria

***Abstract*—Microservices have emerged as a new approach for developing and deploying cloud applications that require higher levels of agility, scale, and reliability. To this end, 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. To solve this problem, we propose 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.**

***Keywords*-microservices; monitoring; container; VM; cloud computing**

I. Introduction

The recent emergence of microservice architecture [1] 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 [2]–[4] have been widely adopted. Microservice-based application architecture has also turbocharged the DevOps [5]–[7] design philosophy by minimizing code-base dependencies between software units.

A. Research Context

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

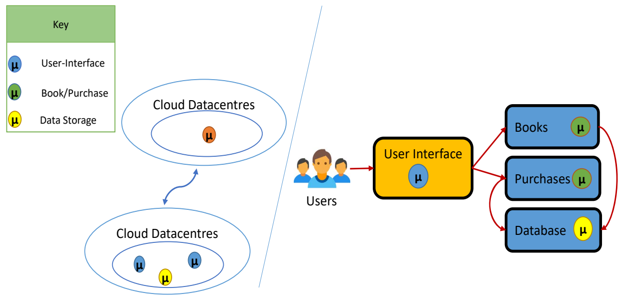


Figure 1. Example Scenario for Microservices Distributed across Multiple Cloud Datacentres.

the application deployment scenario related to a Book-Shop application as described next.

Figure 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) [8], the owner of the Book-Shop application may decide to distribute the microservices across multiple private 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 next.

- 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. Moreover, the performance of such microservice-based 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 [9]–[14].
- 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 [15], [16].

B. Research Contributions

Currently, there are multiple monitoring frameworks e.g. docker stat [17], cAdvisor [18], DataDog [19], Amazon cloud watch [20], CLAMS [21], 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.

Based on the aforementioned challenges, this paper addresses the research question:

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

To answer the above question, this paper makes the following new contributions:

- It introduces a novel system, *Multi-microservices Multi-virtualization Multi-cloud Monitoring (M3)* that provides a holistic approach to monitor the performance of microservice-based application stacks deployed across multiple cloud data centers.
- It implements the *M3* system based on a decentralized agents based approach that provides the ability to independently monitor heterogeneous cloud environments (e.g. different virtualization and cloud service provider platforms). *M3* is cloud agnostic as it implements its own API stack for networking, communication, and managing monitoring data.
- It validates the proposed monitoring system *M3* on a real testbed that includes Amazon Web Services and Microsoft Azure Cloud. Detailed experimental analysis verifies the efficacy of our proposed *M3* monitoring system.

The rest of this paper is organized as follows. Section II discusses recent related work. The system design of *M3* is presented in Section III. Section IV presents the proof of concept implementation of *M3*, and Section V discusses the outcomes of the experimental evaluation. The paper concludes with giving some future work suggestions in Section VI.

II. Related Work

Docker [17] provides inbuilt monitoring tool docker stats, to examine the resource usage metrics of running containers. The various metrics provided by Docker stats are the basic CPU, memory, block I/O and network parameters. Docker stats capture the overall performance of the container, however multiple microservices deployed inside the conatiner (if any) may lead to performance degradation which cannot be captured. An open source framework, cAdvisor [18] also monitors the performance of containers giving different system parameters. However, it needs to execute one more container for each application stack component executing in a distributed cloud environment which consumes additional system resources.

There are some cloud specific monitoring frameworks available such as Amazon CloudWatch [20], Azure Monitor [22]. These frameworks are platform specific and able to monitor only the intended platforms i.e. Amazon CloudWatch is able to monitor only the hosts deployed in Amazon cloud environment and not in Azure environment. Some other commercial monitoring frameworks e.g. DataDog [19], Prometheus [23] are available that are cloud agnostic but provide limited functionality. DataDog is an agent-based system that sends data only to the DataDog server while Prometheus stores the result in a time series database. Both of them support only containers and the support is also limited to some specific cloud providers.

Various monitoring frameworks are also proposed in academia that capture the changing system performance.

Table I
COMPARISON OF RELATED WORK

Monitoring Parameter	Related Work							M3
	[17]	[18]	[19]	[20]	[22]	[21]	[24]	
Virtual Machine (VM)	✗	✗	✗	✓	✓	✓	✗	✓
Container	✓	✓	✓	✓	✓	✗	✓	✓
Multiple Cloud	✓	✓	✓	✗	✗	✓	✓	✓

In [21], Alhamazani et al. present CLAMS, which is an application monitoring framework for multi-cloud platforms. The model retrieves the QoS performance for different cloud layers. However, their monitoring framework is limited to monitor the VM performance only. Moreover, the model is specific for monitoring only web applications. In [24], the authors present the PyMon framework that collects system resources for network edge devices using Docker management API. However, it may not be able to monitor the VM performance. In addition to this, there are some benchmarking framework available that also monitors the specific system parameters [25].

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.

A comparison of different related works with our proposed *M3* is presented in Table I.

III. M3 SYSTEM DESIGN

The proposed *M3* system consists of two main components, namely the monitoring manager and the monitoring agent. Monitoring agents are placed inside each container/VM that tracks the performance of the underlying microservices. Monitoring agents do not care whether the underlying VMs/containers are homogeneous or heterogeneous or are deployed on which cloud infrastructure. The monitoring agents collect the system-level statistics for each service and send the information to the manager. The manager deployed in a distant server collects the information from different monitoring agents and stores this data in the associated database for further performance analysis and management.

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

A. Monitoring Agent

A Monitoring Agent is a software component that collects the information from a microservice running inside containers/VMs. It has the ability to work in different cloud platforms. Agents will wait for the requests coming from the manager to push monitoring information to the manager. *M3* uses HTTP requests as a communicating system for transferring information between agents and managers.

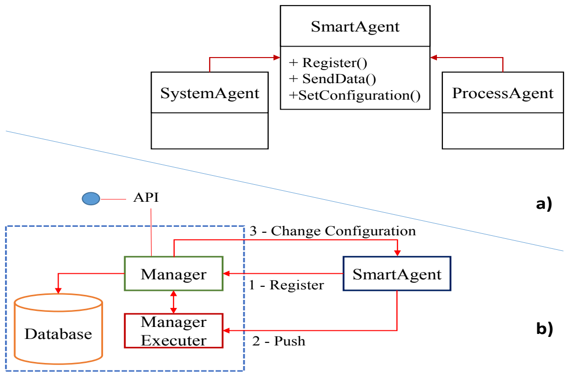


Figure 2. Monitoring Agents Model (a); Communication Model between Manager and Agents (b).

The Monitoring Agent is packaged into a jar file and configured to run during the container/VM boot process. All monitoring agents extend a common agent, called SmartAgent, which consists of two components (*SystemAgent* and *ProcessAgent*) as shown in Figure 2 (a). SmartAgent represents a service consisting of three operations, firstly agent registration information must be sent to the manager using a "PUT" request. Next, the agent will send data periodically to the manager using a "POST" request. Finally, agent configuration will be sent to the manager using "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 the functionalities provided by *SIGAR* to retrieve the microservice metrics and other custom built APIs. *SIGAR* helps in getting the information parameters for the specific microservice. Using these functionalities, the agent monitors the specified features for each microservice. All the information is pushed to the manager after getting a pull request from the manager.

B. Monitoring Manager

The monitoring manager is a software component that receives monitoring information from agents deployed inside containers/VMs scattered in the heterogeneous cloud environments. It also provides an API for accessing data saved by the database and other services or applications. Communication between manager and agents is based on pull- 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 accessing the agent data. Each time the data of a monitor agent is received, the manager stores the results in a MySQL database for further analysis.

The information transfer between the monitoring SmartAgent to the monitoring manager occurs using a sequence of steps as shown in Figure 2 (b). First, the SmartAgent sends a registration request to the manager, and the manager

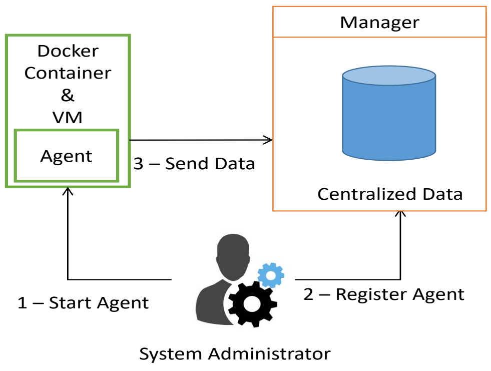


Figure 3. *M3* Data Acquisition Model.

receives the request and registers the SmartAgent, an access key and an endpoint are sent with the data returning to the SmartAgent. Second, the manager executor (uses Push technique) is enabled to receive the data sent by the SmartAgent using their IP address and gets the metrics. Lastly, SmartAgent periodically queries the manager for its configuration (*Change Configuration*). Dynamic configuration enables real-time agent management.

The monitoring agents send the metrics information using RESTful APIs. There are multiple methods to transfer data from agent to manager. The two most common architectures to transfer the data supported by *M3* are (a) centralized architecture and (b) decentralized architecture. In centralized architecture, the manager is located centrally and all the agents are one hop distance to the manager. There is direct communication between the manager and the agent. This is the easiest way of communication, however it may lead to one point failure. To avoid this, M3 also supports decentralized architecture where each cloud has a local monitoring manager that collects information from all the resident containers. There exists a global manager that collects information from all the local managers.

The complete process of microservice monitoring is represented in the form of a data acquisition model as given in Figure 3. It consists of three steps, initially the system administrator starts the monitoring agent (Step 1). Further, the administrator registers the agent to the manager (Step 2). Next, the agent continuously monitors the system (microservices, containers, or VMs). Finally, all the monitoring agents send the monitored information periodically to the manager (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 microservices.

IV. Implementation

Our proposed monitoring system *M3* 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 information while *RESTLet* is a Java library that makes it easy to develop HTTP REST APIs. The *M3* 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.

For our experiment, we considered a Book-Shop application as discussed in Section I 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 4. All requests are

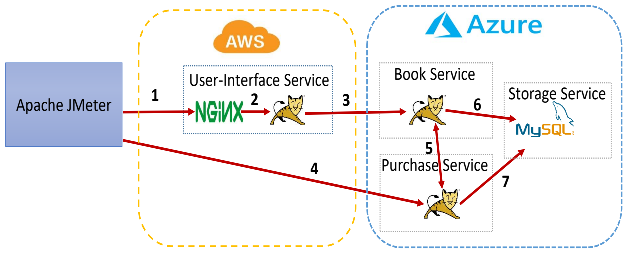


Figure 4. Simulation Web Application Pattern.

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 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.

V. Experimental Evaluation

Based on the defined set up as discussed in Section IV, 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 (https://www.ubuntu.com/) on which a Docker platform (version $17.06.1-ee-1$) (https://www.docker.com/), was installed to execute the microservices. The VM configuration of Azure is $Standard_A1_v2$, with 1 vCPU and 2 GB of memory. We considered four such VMs. The Amazons 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 (Version 7) (http://tomcat.apache.org/) and Nginx (Version 1.13.7) (https://nginx.org/en/) while for Database, we considered MySQL (Version 5.7) (https://www.mysql.com/). All container images used were obtained from the Docker Hub (https://hub.docker.com/) 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 (https://hub.docker.com/-/tomcat/) for Tomcat and (https://hub.docker.com/-/mysql/) for MySQL, third machine used the same configuration as second machine with different services and the final machine had the Docker 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 II:

- **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).
- **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

Table II
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]

Table III
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

which deployed in Amazon Web Services (A).

- **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).

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.

A. *Latency Time Results*

M3 measured the average latency time in milliseconds for the workload requests in each scenario (shown in Table III), 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 scenarios. It shows that the use of container architecture per service in multiple cloud exploits the hardware of the virtual machine more efficiently.

B. *CPU Results*

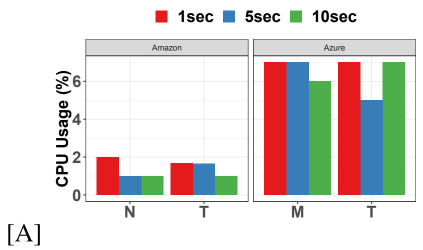
The CPU values for all scenarios are shown in Figure 5. When analysing S1, for the entire interval of workload test, all microservices were run in VMs, and submitted in Azure and Amazon. The monitoring agents send monitoring information to the manager every 1, 5 and 10 seconds respectively. As shown in Figure 5(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%).

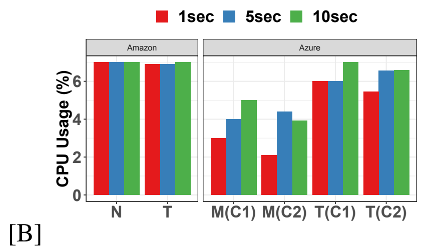
For evaluating S2 in Azure containers, the highest average usage of CPU is for Tomcat microservice for the Book service that was running in container (C1) 10 seconds is 6.80%, and not that much different for Tomcat that was running in container (C2) of the Purchase service in 10 seconds which is 6.60%. 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 5(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 5(C).

C. *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 6. 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 6(A), the highest average memory usage for microservices (Tomcat and MySQL) of Book and Purchase services are practically the 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





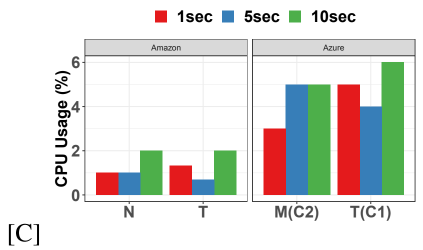
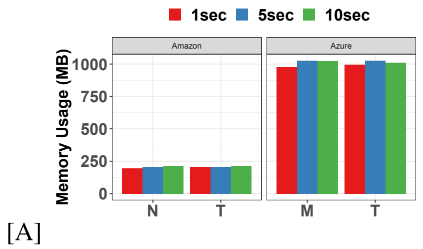
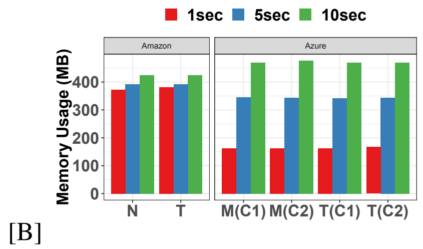


Figure 5. CPU Usage (Percentage) for Microservices on: (A) VMs in Amazon and Azure, (B) Containers in Amazon and Azure, (C) VM in Amazon and Two Containers in Azure. [For the ease of presentation, we used the following abbreviation Nginx (N), Tomcat (T), and MySQL (M)].





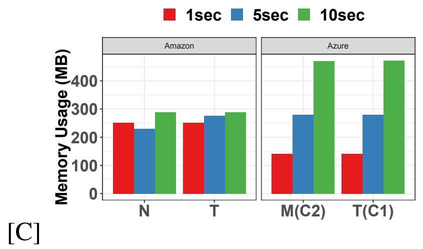
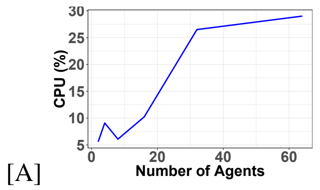


Figure 6. Memory Usage (MB) for Microservices on: (A) VMs in Amazon and Azure, (B) Containers in Amazon and Azure, (C) VM in Amazon and Two Containers in Azure. [For the ease of presentation, we used the following abbreviation Nginx (N), Tomcat (T), and MySQL (M)].



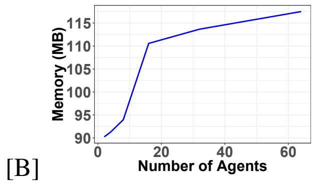


Figure 7. CPU Usage (Percentage) (A) and Memory Usage (MB) (B) for Manager

used 205 MB and Tomcat used 206 MB from the total allocated memory of the VM which is 992 MB. As shown in Figure 6(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 6(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 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 6(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.

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 Figure 7. The results show that the increase in the number of agents affects both the increase in CPU usage (A) and memory usage (B). 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.

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.

VI. Conclusion

In this paper, 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 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.

Acknowledgement

This research is partially supported by Engineering and Physical Sciences Research Council, UK under grant EPSRC - EP/R033293/1.

References

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

[2] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: Migration to a cloud-native architecture, IEEE Software 33 (3) (2016) 42–52.

[3] O. Barais, J. Bourcier, Y.-D. Bromberg, 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.

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

[5] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano, Devops, IEEE Software 33 (3) (2016) 94–100.

[6] G. Pallis, D. Trihinas, A. Tryfonos, M. Dikaiakos, Devops as a service: Pushing the boundaries of microservice adoption, IEEE Internet Computing 22 (3) (2018) 65–71.

[7] H. Kang, M. Le, 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.

[8] 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) 59 (1-88) (2016) 294.

[9] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, 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.

[10] H. G. Mohammadi, P.-E. Gaillardon, 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 35 (12) (2016) 1995–2007.

[11] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, 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.

[12] B. R. Dawadi, S. Shakya, R. Paudyal, Common: The real-time container and migration monitoring as a service in the cloud, Journal of the Institute of Engineering 12 (1) (2016) 51–62.

[13] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, M. Villari, Open issues in scheduling microservices in the cloud, IEEE Cloud Computing 3 (5) (2016) 81–88.

[14] W. Hasselbring, 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.

[15] D. S. Linthicum, Practical use of microservices in moving workloads to the cloud, IEEE Cloud Computing 3 (5) (2016) 6–9.

[16] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, 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.

[17] Docker. [online]. available:https://www.docker.com.

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

[19] Datadog. [online]. available: https://www.datadoghq.com/.

[20] Amazon cloudwatch. [online]. available: https://aws.amazon.com/cloudwatch.

[21] K. Alhamazani, R. Ranjan, K. Mitra, P. P. Jayaraman, Z. Huang, L. Wang, 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.

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

[23] Prometheus. [online]. available: https://prometheus.io/.

[24] M. Großmann, 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] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, G. Morgan, R. Ranjan, A study on the evaluation of hpc microservices in containerized environment, Concurrency and Computation: Practice and Experience (2019) 1–18doi:https://doi.org/10.1002/cpe.5323.