School of Computing Science

Innovative Techniques for Deployment of Microservices in Cloud-Edge Environment

Devki Nandan Jha

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

September 2020

Abstract

The evolution of microservice architecture allows complex applications to be structured into independent modular components (microservices) making them easier to develop and manage. Complemented with containers, microservices can be deployed across any cloud and edge environment. Although containerized microservices are getting popular in industry, less research is available specially in the area of performance characterization and optimized deployment of microservices.

Depending on the application type (e.g. web, streaming) and the provided functionalities (e.g. filtering, encryption/decryption, storage), microservices are heterogeneous with specific functional and Quality of Service (QoS) requirements. Further, cloud and edge environments are also complex with a huge number of cloud providers and edge devices along with their host configurations. Due to these complexities, finding a suitable deployment solution for microservices becomes challenging.

To handle the deployment of microservices in cloud and edge environments, this thesis presents multilateral research towards microservice performance characterization, run-time evaluation and system orchestration. Considering a variety of applications, numerous algorithms and policies have been proposed, implemented and prototyped.

The main contributions of this thesis are given below:

- Characterizes the performance of containerized microservices considering various types of interference in the cloud environment.
- Proposes and models an orchestrator, *SDBO* for benchmarking simple web-application microservices in a multi-cloud environment. *SDBO* is validated using an e-commerce test web-application.
- Proposes and models an advanced orchestrator, *GeoBench* for the deployment of complex web-application microservices in a multi-cloud environment. *GeoBench* is validated using a geo-distributed test web-application.

- Proposes and models a run-time deployment framework for distributed streaming application microservices in a hybrid cloud-edge environment. The model is validated using a real-world healthcare analytics use case for human activity recognition.

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 or any other institution.

Devki Nandan Jha

September 2020

Publications

Portions of the work within this thesis have been documented in the following publications:

Published

1. **D. N. Jha**, P. Michalak, Z. Wen, R. Ranjan, and P. Watson, "Multi-objective Deployment of Data Analysis Operations in Heterogeneous IoT Infrastructure", *IEEE Transactions of Industrial Informatics*, vol. 16, no. 11, pp. 7014-7024, 2020, IEEE, https://doi.org/10.1109/TII.2019.2961676.

2. **D. N. Jha**, K. Alwasel, A. Alshoshan, X. Huang, R. K. Naha, S. K. Battula, et al., "IoTSim-Edge: A simulation framework for modeling the behavior of Internet of Things and edge computing environments", *Software: Practice and Experience*, vol. 50, no. 6, pp. 844-867, 2020, Wiley, https://doi.org/10.1002/spe .2787.

3. Y. Li, **D. N. Jha**, G. S. Aujla, G. Morgan, A. Zomaya, and R. Ranjan, "IoTWC: Analytic Hierarchy Process Based Internet of Things Workflow Composition System", In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 1-10. IEEE, 2020, https://doi.org/10.1109/IC2E48712.2020.00007.

4. K. Alwasel, **D. N. Jha**, E. Hernandez, D. Puthal, M. Barika, B. Varghese, et al., "IoTSim-SDWAN: A simulation framework for interconnecting distributed datacenters over Software-Defined Wide Area Network (SD-WAN)", *Journal of Parallel and Distributed Computing*, vol. 143, pp. 17-35, 2020, Elsevier, https://doi.org/10.1016/j.jpdc.2020.04.006.

5. B. Qian, J. Su, Z. Wen, **D. N. Jha**, Y. Li, Y. Guan, et al., "Orchestrating the Development Lifecycle of Machine Learning-based IoT Applications: A Taxon-

omy and Survey", *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1-47, 2020, ACM, https://doi.org/10.1145/3398020.

6. **D. N. Jha**, Z. Wen, Y. Li, M. Nee, M. Koutny, and R. Ranjan, "A Cost-Efficient Multi-cloud Orchestrator for Benchmarking Containerized Web-Applications", In *International Conference on Web Information Systems Engineering*, pp. 407-423. Springer, 2019, https://doi.org/10.1007/ 978-3-030-34223-4_26.

7. M. Villari, M. Fazio, S. Dustdar, O. Rana, **D. N. Jha**, and R. Ranjan, "OSMOSIS: OSmotic computing platform for MicrOelementS in cloud, edge and Internet of thingS", *Computer*, vol. 52, no. 8, pp. 14-26, 2019, IEEE, https://doi.org/10.1109/MC.2018.2888767.

8. **D. N. Jha**, M. Nee, Z. Wen, A. Zomaya and R. Ranjan, "SmartDBO: Smart Docker Benchmarking Orchestrator for Web-application", In *The World Wide Web Conference (www2019)*, pp. 3555-3559. ACM, 2019, https://doi.org/10.1145/3308558.3314137.

9. **D. N. Jha**, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A Study on the Evaluation of HPC Microservices in Containerized Environment", *Concurrency and Computation: Practice and Experience*, e5323, 2019, Wiley, https://doi.org/10.1002/cpe.5323.

10. 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," *2019 IEEE International Conference on Cloud Computing (CLOUD 2019)*, pp. 156-163, IEEE, https://doi.org/10.1109/CLOUD.2019.00035.

11. **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 *IEEE International Conference on Services Computing (SCC2018)*, pp. 33-40, IEEE, 2018, https://doi.org/10.1109/SCC.20 18.00012.

12. G. Kecskemeti, G. Casale, **D. N. Jha**, J. Lyon, and R. Ranjan. "Modelling and

simulation challenges in internet of things", *IEEE Cloud Computing*, vol. 4, no. 1, pp. 62-69, 2017, IEEE, https://doi.org/10.1109/ MCC.2017.18.

Under Review

1. **D. N. Jha**, Y. Li, Z. Wen, G. Morgan, P. P. Jayaraman, A. Zomaya, and R. Ranjan, "GeoBench: A User-Centric Cost-Efficient Geo-Distributed Web-Applications Deployment via Automatic Benchmarking", *IEEE Transactions on Computers*, IEEE.

2. **D. N. Jha**, Z. Chen, Sh. Liu, M. Wu, R. Ranjan and X. Li, "Accuracy and Energy Aware Activity Recognition In IoT Environment", *IEEE Transactions on Sustaianable Computing*, IEEE.

3. K. Alwasel, **D. N. Jha**, F. Habeeb, O. Rana, T. Baker, S. Dustdar, et al., "IoTSim-Osmosis: a Framework for Modeling and Simulating Smart IoT Applications in Edge-Cloud Continuum", *Journal of Systems Architecture*, Elsevier.

Acknowledgements

This thesis would not have been possible without the guidance and support of many people over many years. My foremost gratitude extends to my advisor, Prof. Rajiv Ranjan for believing in my abilities and guiding me with innovative ideas, timely suggestions, feedbacks and unflinching support throughout my PhD candidature. Raj has taught me about the various aspects of research: finding good problems, exploring the solution space, keeping the bigger picture in mind, and the importance of clear research communication. Furthermore, I wish to thank Prof. Paul Watson who has been demonstrative and generous in his guidance and support throughout my PhD study. I am also very grateful to Dr. Xiaoli Li for his guidance and valuable suggestions during my visit to A* Star, Singapore.

I would also like to thank my examiners, Prof. Joanna Kolodziej and Prof Aad van Moorsel, who provided encouraging and constructive feedbacks. It is not an easy task, reviewing a thesis, and I am grateful for their thoughtful and detailed comments.

The work presented in this thesis is a result of many collaborations. I am grateful to Prof. Albert Zomaya (Sydney University), Prof. Rajkumar Buyya (University of Melbourne) and Prof. Schahram Dustdar (TU, Wien), who provided close collaboration and mentorship throughout my PhD. Special thanks to Dr. Prem Prakash Jayaraman (Swinburne University) and Dr. Saurabh Garg (University of Tasmania) for guiding me at different stages of my research.

I would like to thank all my colleagues and lab-mates: Areeb Alsoshan, Ayman Noor, Bin Qian, Deepak Puthal, Fawzy Habeeb, Jiahan Zhang, Jie Su, Gagangeet Aujla, Khaled Alwasel, Michael Nee, Nipun Balan, Osama Alrajeh, Peter Michalak, Top Phengsuwan, Umit Demirbaga, Xianghua Huang and Yinhao Li. I'm thankful that our centre brought us together so we can learn from each other. Special mention to Zhenyu Wen for being a great friend and brilliant collaborator. Your insightful comments and constructive criticisms at different stages of my research were thought-provoking and helped me to focus on my ideas. Thank you all for bringing the energy

that kept me going forward.

I would like to thank Billy Lau, Min Wu, Shudong Liu, Uzair Javaid and Zhenghua Chen for all your help and support during my study trip to Singapore.

I wish to thank all the people whose works have been utilized for developing and implementing ideas outlined in the thesis. I am also grateful to the Newcastle-Singapore scholarship committee for funding my PhD candidature. I would like to thank the administrative staff and IT support members in the School of Computing for their support and help during the last four years. Special thanks to Helen Munday for making everything run smoothly, always happy to help, and having genuine care for our wellbeing.

This thesis would never have been possible if it were not for the support of family and friends, especially my mother, Nilam Ranjan Jha, father, Anil Kumar Jha, sister, Dipti Kumari and brother in law, Nandan Jha. Above all, I am indebted to the almighty for making my life meaningful even outside the confines of this PhD candidature.

Contents

List of Figures

List of Tables

1

Introduction

Contents

This chapter introduces the context of the research themes explored in this thesis. It starts with a high-level overview of the microservice architecture for application design and analyzes the deployment environment for microservices. Next, it discusses the fundamental challenges for optimized deployment of microservices in the cloud-edge environment which is the motivation behind this research. The chapter thereafter outlines the contribution of this thesis and finally concludes with the thesis structure.

Introduction

With the advent of technology, different applications are developed to transform aspects of our everyday lives. These applications can vary from a simple calculator to a complex smart city. Depending on the data generation, computing requirements and user interaction, these applications can be categorized into different types such as High-Performance Computing (HPC), Web, Streaming, Machine Learning or a combination of these.

The emergence of microservice architecture has revolutionized the application design and development. By adopting the microservice architecture, developers can engineer applications that are composed of multiple self-contained components communicating with each other using lightweight APIs [80]. Each application microservice can be deployed, updated and redeployed independently without compromising the integrity of the application's ecosystem [85, 105]. Updating only one or few microservices instead of the entire application stack increases the application scalability and availability.

Currently, *cloud computing* is the most commonly used deployment environment for a variety of application microservices. It provides seemingly unlimited resource access on a pay-per-use basis which enables applications to customize their deployment plan in a highly elastic manner [106]. These cloud providers are distributed in different geographical regions and can be accessed ubiquitously on demand. Numerous cloud providers (e.g. Amazon, Google, Microsoft, IBM) are available in the marketplace to offer a variety of configurations.

The emergence of *edge computing*, which provides cloud-like services but at the edge of the network, facilitates the processing of microservices without sending the Internet

of Things (IoT) data to the cloud [126, 131]. This is necessary for certain applications where the delay incurred by the centralized cloud-based deployment is unacceptable. It is also effective for the environment where the network is unstable with a higher chance of failure or data loss. Smart edge devices such as Smartphone, Raspberry Pi, UDOO board support local processing and storage of data on a widespread but smaller scale. Unlike the cloud where the location of a datacenter is fixed, edge devices can be mobile and change location frequently. Combining edge and cloud datacenters creates a complex IoT infrastructure.

Cloud and edge resources are virtualized to deploy the application microservices. Although hypervisor- and container-based virtualization are available [62], microservices can be efficiently deployed on containers. Container as a virtual runtime environment executing on the top of the host operating system supports the design principle of microservices by wrapping up all their requirements and dependencies into a lightweight single unit with the desired level of isolation. Containers uses Linux namespace and cgroups features to provide isolation and abstraction [136]. For the deployment of these containerized microservices, three main Deployment Infrastructures (DI) are provided by cloud and edge computing environments: Bare Metal (BM), Virtual Machine (VM) and Container Service (CS) [61]. Each type of DI has the variety of hosts either predefined by the providers or user-customized. For example, Amazon EC2 provides different types of pre-customized hosts (VMs) for their customers along with the self-customized hosts. In the rest of the thesis, we use the host to represent an instance of the DI. The containerized microservices can be deployed and executed on any type of host.

1.1 Project motivation

The fundamental objective behind the emergence of microservice is to ease the design and development of various applications, however, deployment of containerized microservices brings new challenges. Depending on the application type (e.g. web, HPC, streaming) and the provided functionality (e.g. filtering, storage, encryption), microservices can be heterogeneous with specific requirements in terms of hardware

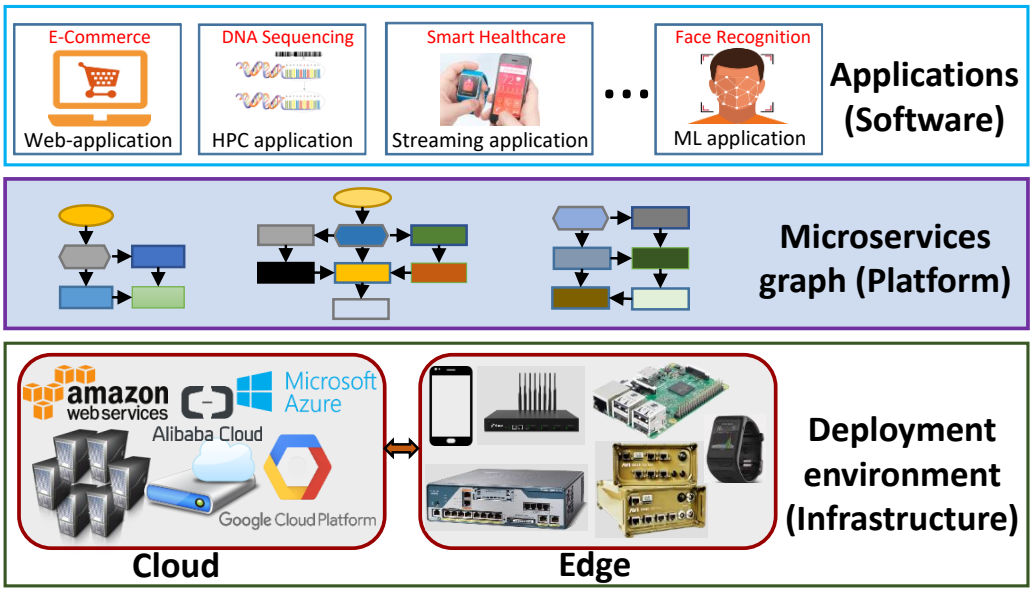


Figure 1.1: Microservice-based application deployment

and software specification. In addition to this, there is a strict data and control flow dependency between different microservices. As shown in Figure 1.1, an application generates a graph of microservices with strict data and control flow dependency. It is important to maintain the dependency and satisfy the requirements and constraints while deploying in the cloud or edge environment.

Due to the rising popularity of cloud and edge computing, the number of cloud providers along with their host configurations and type of edge devices continues to grow at a rapid pace. For the deployment of microservices, it is necessary to find a suitable host configuration that satisfies the requirements and constraints in an optimal manner.

Finding a suitable deployment solution for microservices has many challenges [38, 61], a few of the main challenges considered in this thesis are given below:

Deployment environment heterogeneity. Cloud and edge environments are heterogeneous with a variety of host configurations available. The hardware and software support provided by each host is also different. In addition to this, the internal modeling and provisioning mechanism for each cloud provider is unique e.g. AWS resource can be managed using Amazon CloudFormation which cannot be used for any other environment.

Communication protocols. To offer a unified deployment of the microservices, IoT, edge and cloud infrastructure components need to communicate with each other. Since the capabilities of different infrastructure components are not the same, a single communication mechanism is not an option. Protocols for *Application layer* (e.g. Constrained Application Protocol (CoAP), Message Queuing Telemetry Transport (MQTT)), *Infrastructure layer* (e.g. Datagram Transport Layer Security (DTLS), IPv6 over Low power Wireless Personal Area Networks (6LoWPAN), Bluetooth Low Energy (BLE), IEEE 802.11), *Discovery process* (e.g. Universal Plug and Play (uPnP), Apache River) and *Semantic protocols* (e.g. RDF/XML, SensorML) are available which must be selected depending on the requirements and constraints defined for the application [32]. In addition to this, adoption of NFV (Network Function Virtualization) and SDN (Software-Defined network) technology [41] requires some alternative protocols that support flexibility and usability in network management and communication.

Network imbalance. The underlying cloud and edge network is unpredictable as it relies not only on the local area network (LAN) but also on wide area network (WAN) using the Internet. Since the application microservice can be distributed across different cloud and edge datacenters, communication through the Internet may lead to jitter, data loss or unordered data delivery which leads to performance loss.

Interference. The performance of containerized microservices not only depends on its own characteristics but is also affected by other microservices running on the same host/server leading to interference. Executing an application microservices in cloud where the resources are virtualized, interference is inevitable. The performance of containerized microservice might be affected by other microservices running inside the same container causing *intra-container* interference. However, the performance of microservices running in separate containers may also get affected because of *inter-container* interference as the containers share the same host machine. The effect of interference multiplies if all the collocated microservices have similar resource requirements. It is necessary to visualize the performance variation before making the deployment decision.

Monetary cost. Deployment in the cloud and edge environment incurs a cost and the

user always tries to minimize the cost without losing the performance. A cheaper host may not always satisfy the requirements which lead to testing numerous host configurations before making a deployment decision. Analyzing all the possible options before making the deployment decision is not always feasible in polynomial time. Nevertheless, host performance that fluctuates due to unforeseen reasons requires long-duration analysis which again increases the cost.

Energy. Executing the microservices in the cloud or edge host consumes energy. Energy consumption depends on various factors such as functionality support, load state and network connectivity in a non-linear way and is complex to manage, especially, for edge and IoT devices which are powered by a battery with limited capacity and need to be recharged after a particular time period. These devices need to sustain the battery for longer durations, especially for the case where it is not easy to recharge e.g. sensor in the river or at a disaster site (earthquake, landslip, etc.). Saving energy impacts on cost-saving which is vital for every application, however energy saving may lead to performance degradation or delay.

This PhD project aims to find an optimal deployment solution while considering the aforementioned challenges. In particular, this PhD thesis is guided by the following research questions:

1. *How does the interference of microservices executing in the cloud-edge environment affect the overall application performance?* As different microservices may have similar resource requirements which may lead to interference and resource contention. It is important to recognize the performance variation of microservices executing together so that the interference can be minimized and resources can be utilized in an optimal manner.

2. *How can we identify a suitable deployment environment for application microservices belonging to different applications in a cloud-edge environment?* Each application microservice has numerous Quality of Service (QoS) objectives which can be conflicting. The obtained solution need to find a trade-off between various objectives depending on the application requirements and available cloud-edge deployment environment.

3. *How can we deploy the application microservices for the solution obtained by* 2 *in the selected cloud-edge environment?* There is a complex data and control flow dependency between different microservices. In addition to this, cloud and edge environment have heterogeneous infrastructure. Deployment of microservices should be orchestrated in such a way that maintains the dependency between microservices irrespective of the underlying infrastructure heterogeneity.

1.2 Contributions

Since the deployment of microservices is dependent on application type and the underlying infrastructure, this thesis focus only on three three types of application: (a) HPC application, (b) web-application and (c) streaming application with HPC and web application executing only in cloud environment while streaming application is executing in cloud-edge environment. The main contributions of this thesis are as given below:

- *A survey of research issues and research opportunities in microservice deployment:* An overview of existing work on microservice performance evaluation in a containerized environment is discussed. In addition to this, a detailed discussion about the available deployment approach for microservices in cloud-edge environment is presented.

- *Performance evaluation of containerized microservices in the interfering environment:* A detailed evaluation of application microservice in the cloud environment is performed. The work is particularly concerned with how intra-container and inter-container interference influences the performance of microservices. Moreover, it also investigates the performance variations of microservices when control groups (cgroups) are used for resource limitation. For ease of presentation and reproducibility, it uses Cloud Evaluation Experiment Methodology (CEEM) [101] to conduct the comprehensive analysis. The analysis is performed using HPC microservices.

- *An orchestrator for benchmarking web-microservices in multi-cloud environments:*

To find a suitable deployment option for containerized microservices, we considered the approach of benchmarking the host configuration. To achieve this, a novel orchestrator is developed and implemented for defining and executing the benchmark application microservices in a multi-cloud environment. In particular, the orchestrator allows users to choose the benchmark application and host configurations across different cloud providers. It also permits the user to choose a maximum budget and execution time for benchmarking. The orchestrator computes the performance metrics of different host configurations which can be used to choose the optimal one. The orchestrator is evaluated for an e-commerce web-application benchmark executing on AWS and Azure cloud environments.

- *A deployment framework for complex geo-distributed web-application microservices using benchmarking in multi-cloud environment:* To find a suitable deployment option for the complex and distributed web-application microservices, a model is developed and implemented. Based on the user's requirements, the model first generates a set of candidate deployment solutions using K-means clustering [82] and Adaptive Particle Swarm Optimization (APSO). Next, these solutions are executed over the real cloud environments and finally, the resulting metrics are evaluated to find a suitable deployment solution. The model is validated using a customized geo-distributed web-application executing on AWS, Azure and Google cloud environments.

- *A multi-objective deployment framework for a complex, distributed streaming application microservices in cloud-edge environments:* Combining the edge and IoT in the cloud environment increases the complexity of deployment. To solve this, a multi-objective optimization model is developed and implemented for distributed streaming application microservices. First, a formal model is developed and the deployment problem is proved to be NP-hard. Next, using a well-known heuristic method, Analytic Hierarchical Process (AHP) [124], an optimal deployment solution is found. Finally, the application is deployed on the selected deployment solution using the Path2iot model. The model is validated using a healthcare-based streaming application.

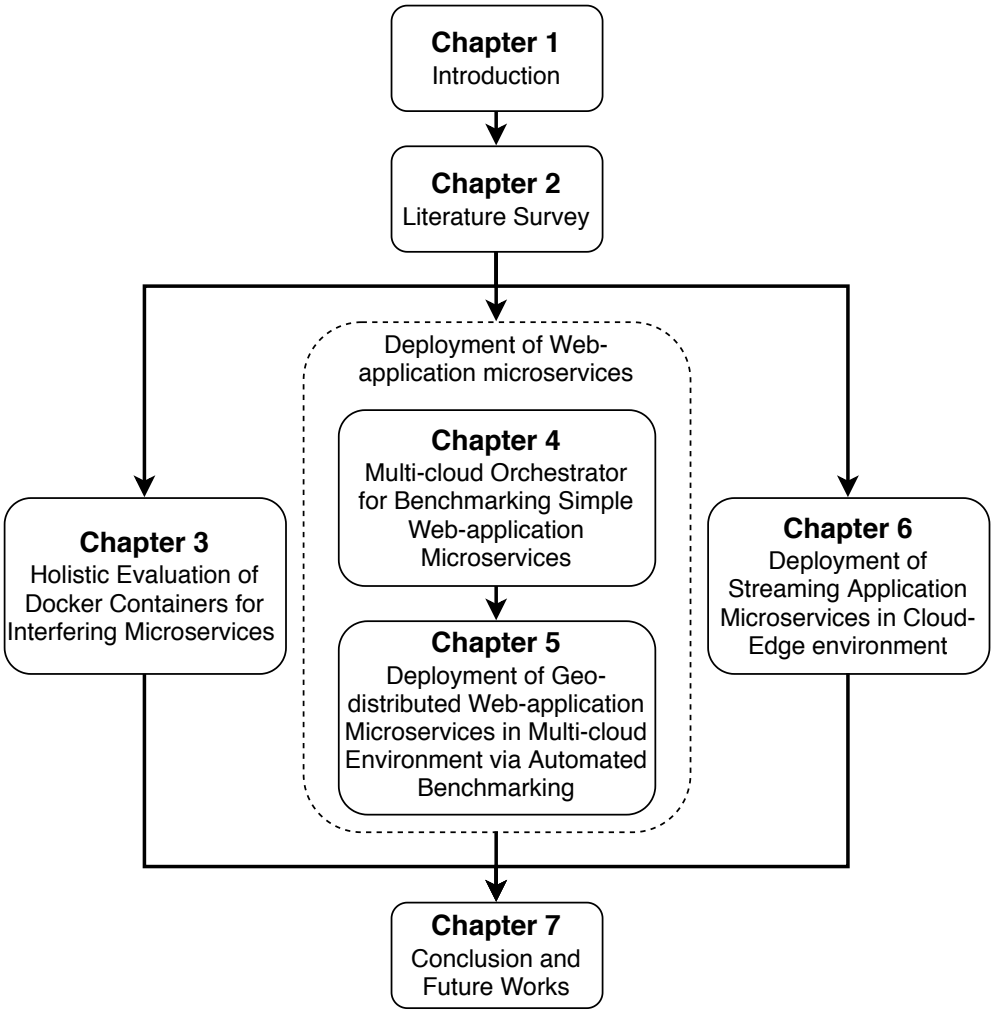


Figure 1.2: Thesis organization

1.3 Thesis outline

The remainder of this thesis is structured as shown in Figure 1.2. The arrow here represents the flow of chapters i.e. order in which one should read as the former chapters have prerequisite information for later chapters. **Chapter 2** provides the background of virtualization, microservices and deployment infrastructure and discusses related work on addressing the deployment challenges in cloud and edge environments. **Chapter 3** describes a detailed performance evaluation of microservices in the interfering cloud environment. **Chapter 4** presents an orchestrator for benchmarking a simple application microservice while **Chapter 5** discuss the orchestrator for complex geo-distributed applications. Both these works are validated by a benchmark web-application executing in a multi-cloud environment. **Chapter 6** develops and

presents a deployment framework for distributed application microservices with multiple conflicting objectives in the cloud-edge environment. The validation is performed using a real healthcare based streaming application. Finally, **Chapter 7** concludes the thesis by summarizing the work done in this thesis and presents some directions for future work.

2

Literature review

Contents

Summary

This chapter starts by describing some background information concerning the overall topic, including a brief primer on virtualization techniques, microservices and the underlying cloud and edge computing environment. Section 2.4 discusses the research in the area of microservice deployment which is the main focus of this thesis. At the same time, current research gaps are highlighted and it is briefly illustrated how this thesis fills these gaps.

2.1 Virtualization

Virtualization is considered to be the core component of cloud and edge computing that allows multiple tenants to run their heterogeneous applications in an isolated environment [154]. It provides numerous advantages including heterogeneous workload consolidation, easy allocation, reduced failure probability and increased availability that makes virtualization user amenable whilst increasing hardware utilization.

Virtualization is not a new concept. The term "virtualization" was first coined by IBM in 1960 to represent a time-sharing system which allowed multiple users to concurrently use the system emulating the hardware. The first hypervisor which allowed multiple operating systems (OS) to run simultaneously was IBM's $CP-40$ developed in 1964. In 1965, $CP-40$ was replaced by $CP-67$ with new memory sharing features and is considered to be first fully virtualized virtual machine (VM). Later in the 1970s para-virtualization was developed in *Denali* virtual machine manager which can update the guest OS with hypervisor code to make specific system calls. In 1979 a new concept *Chroot* was introduced for creating and hosting a separate virtualized copy of the software system which was considered to be the start of OS virtualization. The idea of hypervisors was abandoned due to inexpensive x86 servers during the 1980s. In the late 1990s, hardware performance increased but server under-utilization became apparent as multiple applications could not be executed together. To overcome this, $VMWare$ released the first virtualization software for $x86$ systems in 1999 and is considered as the rebirth of virtualization. Later on other hypervisors such as Xen (2003) and KVM (2006) were developed. In 2006, application virtualization evolved

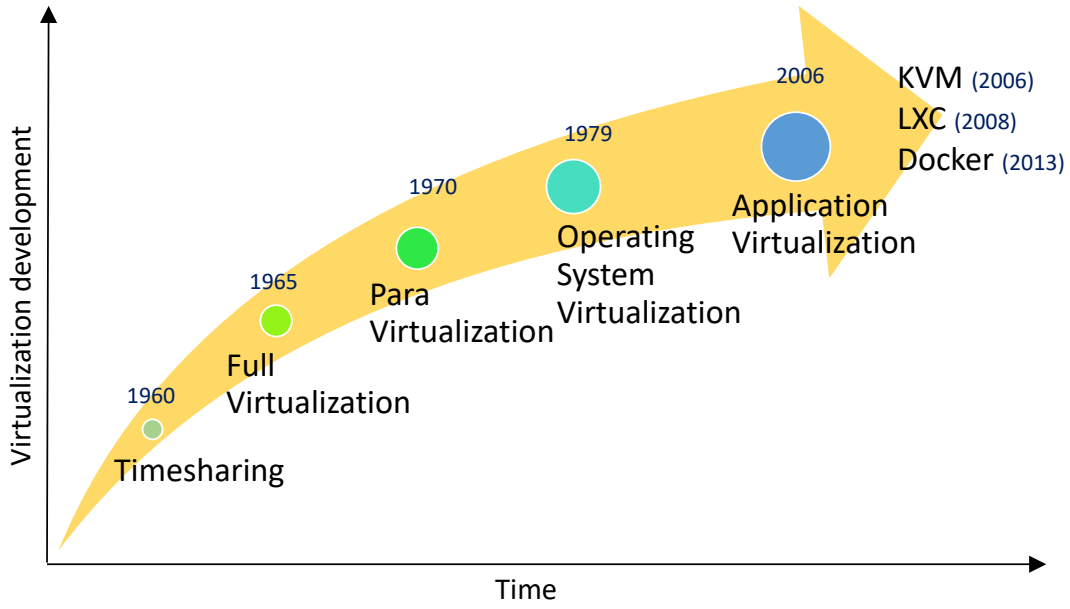


Figure 2.1: Virtualization evolution

which separates the application from the host OS thus allowing them to execute in environments that do not suit the native application. Development of LXC 2008 and Docker 2013 providing isolated user-space (containers) with the help of namespace and cgroups changes the means of virtualization specially for application virtualization by providing a lightweight and isolated option. Figure 2.1 depicts the basic evolution timeline for virtualization.

The details of two main types of virtualization, namely, hypervisor-based and OS-based/container-based virtualization, are given below:

2.1.1 Hypervisor-based virtualization

In hypervisor-based virtualization, each virtual machine (VM) has its own OS irrespective of the host machine running on a hypervisor. Virtual machines are considered to be the default method of virtualization. They provide all the advantages of virtualization but at the cost of high overhead as compared to the bare-metal performance. Xen [37], VMWare [86], KVM [66], etc. are typical examples of hypervisor-based virtualization.

Hypervisor-based virtualization can be either Full, where the guest OS is unaware about the virtualization or Para, where the guest OS is modified so that it can make

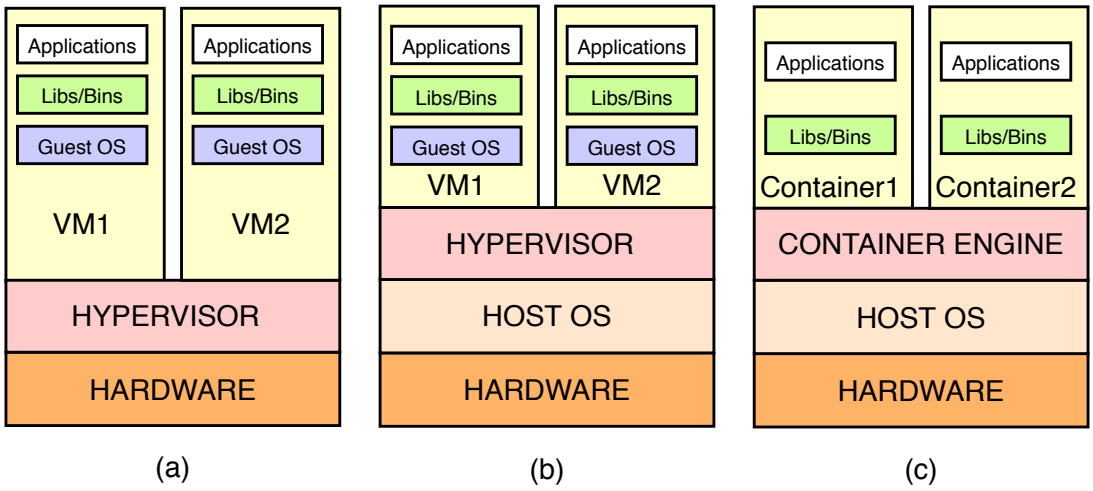


Figure 2.2: Virtualization types (a) Type-1 hypervisor-based virtualization (b) Type-2 hypervisor-based virtualization and (c) Container-based virtualization

specific hyper-calls to the hosting system. Regardless of the type, hypervisor-based virtualization can be categorized into two broad classes on the basis of architecture namely Type 1 and Type 2 hypervisor. There is a basic difference between Type 1 and Type 2 hypervisors, as a Type 1 hypervisor communicates directly with the hardware of the host machine and controls the virtual machine hardware resources itself whereas a Type 2 hypervisor runs on the top of the host OS and lets the OS handle the virtual hardware resources. The architectural difference between Type 1 and Type 2 hypervisors is shown in Figure 2.2.

2.1.2 Container-based virtualization

Container-based virtualization utilizes the services provided by the host OS using a container engine. Different containers can share the same physical resources but from a hosted application's point of view, each container has its autonomous OS running independently. Docker[11], LXC[19], etc. are typical examples of container-based virtualization.

The isolation and abstraction in a container are provided by the Linux feature, namespace and cgroups [136]. The namespace feature restricts the visibility of a container so that it can only access the resources allocated to it. PID, MNT, NET, IPC are some common namespace features used by containers to provide the abstraction for process ids, file system mount points, network features and inter-process communi-

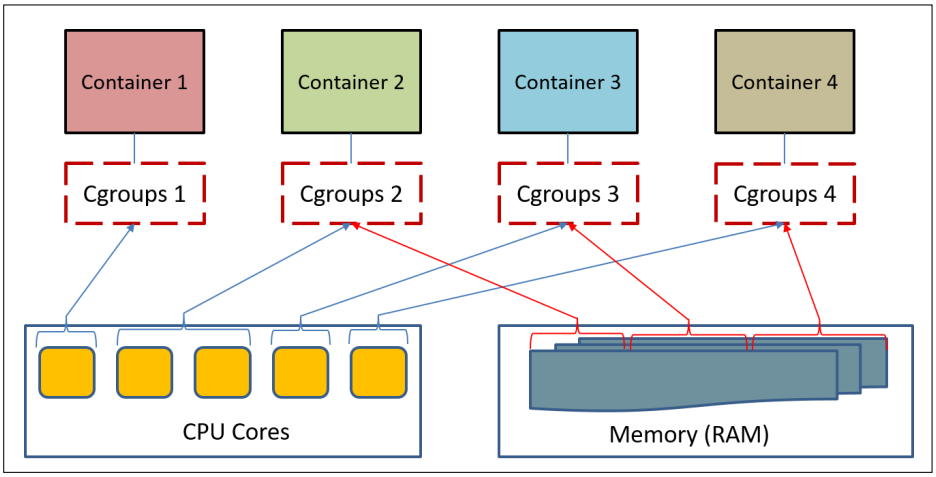


Figure 2.3: Resource restrictions provided by the cgroups

cations respectively in an inter-container environment [40]. Each new container uses the clone() system call to create an abstract system of an existing namespace in the OS kernel. Linux cgroups are additional kernel mechanisms that control the resource allocation by restricting the system resource consumption in terms of CPU, memory, network and disk I/O for each process group. cgroups also determine the priority of resource usage by a process group. Figure 2.3 shows the resource limitations provided by cgroups.

2.1.3 Why container?

The remainder of this thesis uses container-based development and deployment. The reasons for choosing containers are given below:

a. Lightweight. Containers are considered to be a lightweight alternative to hypervisor based virtualization, which creates multiple isolated user-space working instances. Unlike VMs, a container engine parses down the equipment necessary to run the software inside it, rather than packing multiple functions into the same virtual machine, which makes the rapid development and testing of microservices easier. While hypervisors provide an abstraction for the full guest OS (one per VM), the container based virtualization works at the OS level. Figure 2.2 shows the main difference between container and hypervisor-based virtualization.

b. DevOps support. Recently, DevOps practices are gaining popularity as they

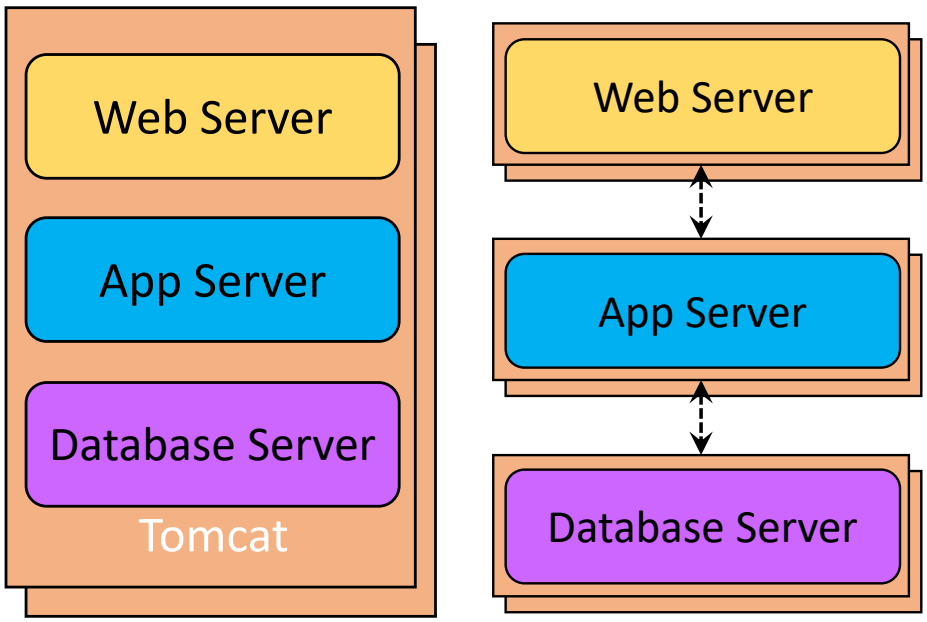
support the continuous delivery of software application and allow easy collaboration among different phases of application development [59]. Although DevOps is not dependent on containers, use of containers provides several benefits to enable DevOps workflow. As the container environment is persistent irrespective of the underlying OS, it is easy to provide a consistent environment for development, testing and production phases. In DevOps, application or application module requires continuous updates which is easy to implement with the help of containers.

c. Microservice compatibility. Recent trends move towards the decoupling of application systems into smaller modules (microservices) so that each application module can be developed independently supporting heterogeneous technology. Since containers provide a lightweight environment which can isolate microservices with minimal dependency, it is suitable for the fast development and deployment of microservices [61, 147].

d. Limited research work. Although container-based virtualization was proposed long ago, containers gained popularity only after the development of Docker in 2013. With the numerous advantages provided by containers, it also brings some new challenges such as adapting applications to support containers, increasing complexity, maintaining isolation, handling host resource utilization and guaranteeing security. Compared with the VM-based application development and deployment, container-based development is still emerging. For the progression of containers, new research is required which addresses these challenges with the adoption of containers.

2.2 Microservices

Traditional representation of an application follows monolithic representation with each application being represented as a single autonomous unit. Consider an example of a standard web-application. For designing such applications, we need a Web server for providing access to users, an App server (Application server) for handling all the business logic and a Database server for providing the access and retrieving data from a database. Now, in order to run the entire application, we will create either a WAR or an EAR package and deploy it on an application server (like Tomcat, JBoss or WebLogic).



(a) Monolithic representation (b) Microservice representation

Figure 2.4: Monolithic vs. Microservice representation of the example web-application

Now, because we have packaged everything as an EAR/WAR, it becomes monolithic, which means that, even though we have separate and distinguishable components, everything is wrapped under one roof. Figure 2.4a shows the example monolithic application.

The problem with such monolithic architecture is that even a small modification of the application requires the deployment of a new running version of the codebase. Failure of one component leads to the breakdown of the entire application which is problematic. In the DevOps environment, where multiple components can follow different technology, it cannot be handled using monolithic architecture. The adoption of microservice architecture is transforming the way to design future applications by providing the flexibility to change and redeploy the modules without worrying about the rest of the components. Figure 2.4b shows how the web-application is decomposed into multiple independent microservice components. The arrow in Figure 2.4b shows REST-based communication.

Microservices as a new architectural pattern attract attention from both industry and academia, and a report shows that 91% of industries either use or have plans to use microservices [104]. According to NIST, a microservice is defined as "*a basic element*

that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology" [83].

Modern applications either can be entirely composed of microservices or use microservices as auxiliary support for a monolithic application. Since the microservice architecture is lightweight and can easily be shipped and updated, it is ideal for engineering applications where we cannot fully anticipate functionalities in advance.

2.2.1 Internal structure of microservices

Microservices are formed as a result of decomposing an application over the functional boundaries. Generally, they are designed according to Domain-Driven Design (DDD) principles and each unit exposes their functionalities in the form of interfaces. Regardless of the application type, each microservice unit is considered to have a layered structure as shown in Figure 2.5 [80, 94]. There are four main components as described below:

1. Resources. This layer is involved in translating service requests from a user into the domain objects. It performs the validation of each request before transferring them to the domain layer and also sends the output back to the user in the desired protocol-specific format.

2. Domain model layer. This layer has three components and is mainly involved in performing service logic. The *services* perform co-ordination across multiple *domains* where each *domain* is involved in performing business logic implementation. A *domain* contains all the entities and objects to process the required implementation. The *repositories* stores the collection of domain entities.

3. Data mappers. This layer is involved in providing persistence access to the objects between domains. This is usually achieved with an object-relation mapping which can be directly stored in an external datastore.

4. Gateways. Gateway is involved in communicating with other collaborator services.

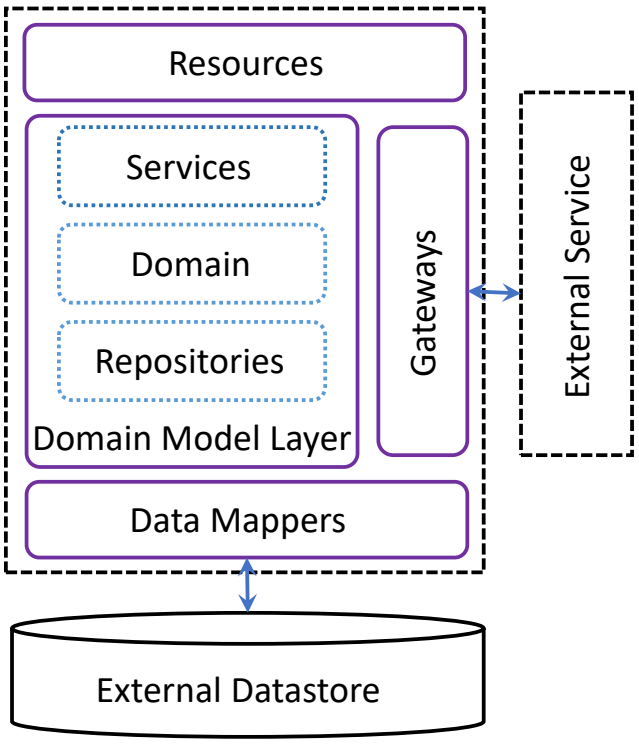


Figure 2.5: Layered structure of a microservice unit

It usually maps the messages (requests/responses) coming to or from the domain objects.

2.3 Deployment environment

2.3.1 Cloud computing

Cloud computing is the most popular deployment environment for a variety of applications/microservices. According to NIST, "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [106]. It provides computing as a utility which can be provisioned on a pay-per-use basis with the users' applications demand ubiquitously. The computing requirements are provided by public cloud (e.g. Amazon AWS [1], Microsoft Azure [20], Google Cloud [14]), private cloud (e.g. Newcastle University datacentre) or a combination of these cloud platforms. Availability of the cloud services across the globe has been possible because of the proliferation of the Internet,

though private cloud also provides services through Intranet.

To handle the increasing diversity and scalability of current applications, cloud environments offer resources with different characteristics and abstraction. The resources are usually provided in terms of infrastructure, platform and software as-a-service. The infrastructure resources are virtualized to provide different hardware and software configurations. Public cloud providers offer multiple choice with different dimensions such as Amazon AWS gives more than 275 pre-defined host configurations, over 4 different OSs distributed across more than 10 different geographically distributed datacenters. Applications are deployed on the cloud environment at a minimal cost.

2.3.2 Edge computing and Internet of Things

With the gaining popularity of Internet of Things (IoT) applications that create huge data and require real-time processing, a traditional cloud environment may not always be a suitable environment. For data processing in the cloud, all the data need to be sent to cloud which delays the processing. Also, it consumes a lot of network bandwidth. This may not be optimal for certain applications where the requirement is (a) close coupling between the data generators and actions taken based on the analysis of the data [155], (b) data transfer bandwidth is limited [132] and (c) data-generating devices are battery-operated [110]. To addresses these challenges, an alternative approach is to execute some parts of the application close to the data generating device.

This approach has been made possible by the introduction of what has become known as *edge computing*. Development of smarter IoT and edge devices, with some storage and processing, opens up a tremendous opportunity for local analytics. Smartphones and field gateways can perform local analytic operations on the data, as well as acting as a network bridge between IoT devices and the cloud [42, 139].

2.4 Microservices deployment

As discussed in Section 1.1, deployment of microservice is challenging. Depending on the type of application and the supported functionality, microservices can have different requirements. The requirements can be categorized in to two types, functional

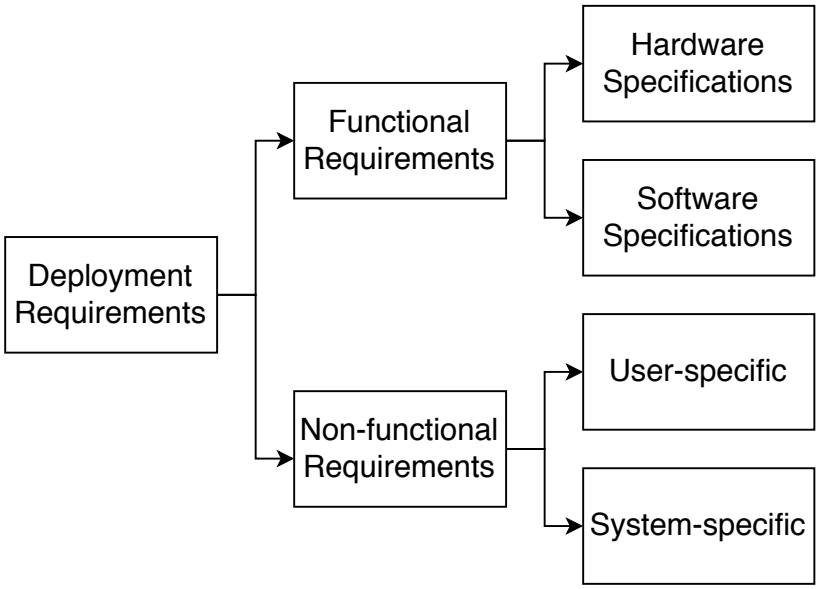


Figure 2.6: Deployment requirements taxonomy

and non-functional as shown in Figure 2.6. The details of each requirement is discussed as follows:

Functional requirements. These define the specific behaviours or functions required by that specific application. The infrastructure must satisfy these requirements for deploying the application. It can be partitioned into two categories as given below.

1. **Hardware Specification** – generally consists of CPU, memory, storage and network requirements. These requirements can be either known apriori or can be predicted/detected at the run time. Since the heterogeneous cloud infrastructure is not stable and the requirements may vary at run time, infrastructure with extra resources is allocated. Different elasticity and scalablility mechanisms are imposed to increase the utilization and reduce the resource wastage.

2. **Software Specification** – indicates the OSs, programming languages and software tools required for the application. Because of the heterogeneity of infrastructure, an application microservice can not be deployed everywhere. It is necessary to analyze the infrastructure before deployment.

Non-functional requirements. Non-functional requirement, also known as Quality of Service (QoS) requirements, outlines the entire qualities and characteristics of the

resulting system. It specifies criteria that can be used to judge the operation of a system, rather than specific behaviours [64]. To provide a holistic view of QoS requirements needed to select a host configuration, Service Measurement Index (SMI) attributes are designed based on International Organization for Standardization (ISO) standards [133]. It is partitioned in two categories as given below.

1. **System-specific** requirements that are affected by the performance of the deployment system. Throughput, response time, reliability, scalability, security, etc. are some common system-specific metrics.

2. **User-specific** requirements that specify the user dependent criteria such as budget, reputation, client interface and user experience.

2.5 Thesis scope in terms of microservice deployment

Containerized microservices are getting popular in industry however, there is very little research available in the area of deployment optimization in cloud and edge environments. Although there are numerous problems to be solved for the deployment of containerized microservices, this thesis is bounded by the workplan given in Figure 2.7. We consider the microservices belonging to three application types namely HPC application, web-application and streaming application. Since the HPC and web applications are mainly cloud-based, therefore, we consider the cloud environment for their deployment. However, streaming applications are now deployed in a hybrid cloud-edge environment, therefore we also considered a similar environment.

To cover multiple aspects of the deployment, we started with the performance characterization of microservices. We utilized HPC application execution in cloud environments for this purpose. We then address the challenges for the deployment of web-application microservices. Since benchmarking the host configuration beforehand leads to finding a suitable deployment solution, we first proposed and implemented a benchmarking orchestrator for a simple web-application in the multi-cloud environment. Later, we proposed a deployment framework utilizing a run-time benchmark

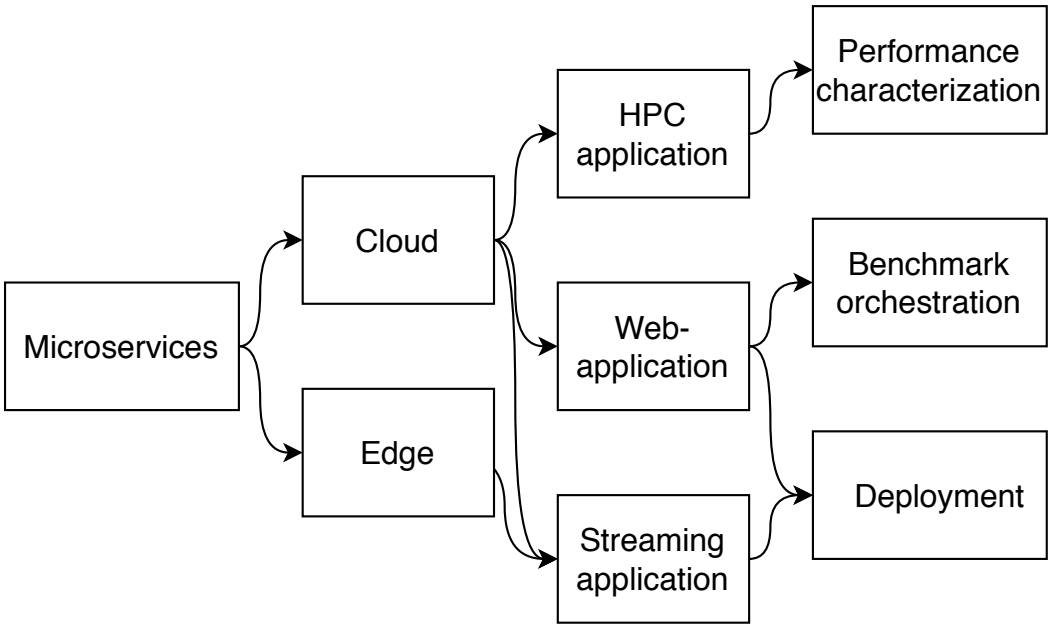


Figure 2.7: Workplan undertaken by this thesis

for a complex geo-distributed web-application. Finally, we address the deployment of streaming applications with multiple, conflicting objectives in a hybrid cloud-edge environment. The section below briefly outlines each problem with the current research gaps.

2.5.1 Evaluation of containers for interfering HPC microservices

Numerous efforts [18, 36, 62, 112, 153] show that containerizing the cloud infrastructure leads to highly efficient and agile solutions. It is evident from the previous work that containers can reduce the execution overhead while increasing the overall performance. These studies compare the performance of containers with respect to VMs for different benchmarks and show that the performance of the container is better than or almost equal to the performance of the VM.

A few of the works also specify running HPC applications in a Docker environment. Jacobsen et al. [75] advocates the use of containers for HPC environments. The work in [68] shows how to orchestrate multiple containers on a physical node. The study confirms that a job can be transparently executed inside a Docker container without having any knowledge about the underlying host configuration.

Most of the above studies do not consider the effect of interference in containerized environments. Ruiz et al. [123] evaluated the performance of LXC containers in different configurations e.g. isolated, inter-container and multi-node inter-container using NAS parallel benchmark. The results conclude that inter-container communication is faster than physical machine communication but there is a degradation of CPU performance for memory-intensive operations. The result also shows that for a multi-node inter-container communication, the performance of network-intensive applications degrades. Sharma et al. [130] also compares the performance of collocated applications on a common host but only one application is running in a container/VM. They show the effects of interference caused by noisy neighbour containers running competing, orthogonal or adversarial applications. All the experiments are done on the LXC container.

Ye et al. [156] also considers the inter-container interference for big data application (Spark). Similar work is done by Kejiang et al. [159] that evaluates the performance of big data applications by changing the system configurations and also considering the interference between containers. Using different Spark applications (K-Means, Page rank, etc.) and changing the Docker system configurations using cgroups, the performance is evaluated.

None of the existing works considers the performance evaluation of heterogeneous microservices executing inside a container and compares the interference impact with the microservices running in separate containers. In this thesis, we have demonstrated the performance evaluation of HPC micro-benchmarks intended towards specific resource type (CPU, memory, disk and network) in the form of microservices running inside the Docker container. The obtained result presents the performance variation of containers while running single or multiple co-allocated (competing or independent) microservices. More details are presented in Chapter 3.

2.5.2 Orchestration for benchmarking containerized web application in multi-cloud environment

Nowadays, web-applications are becoming complex and may have multiple tiers and each tier may be developed using a wide range of technology stacks such as Java, .NET, PHP, Neo4j. Evaluating the real web-application in the cloud environment

gives the best performance behaviour, however, it is very hard if not impossible to test it on all the available cloud host configurations. This is due to the complexity of the web-application which is not easy to set-up and control.

Benchmark applications which are simple but mimic the exact behaviour of a web-application are developed. The emulated benchmark has mainly three features server-side – to emulate the domains, associated tiers and interactivity support; workload generation – to capture the user interaction behavior; and workload injection – to maintain a continuous stream of load [55]. Regardless of the complexity to capture the idiosyncrasies of every domain, it is necessary to consider as many domains as possible to make the benchmark generic. Numerous benchmarks are proposed in the literature. TPC-W [27] and RUBiS [23] (for e-commerce), Wikibooks [28] (for Wikipedia) and SPECWeb2009 [25] (for banking and e-commerce) are some common web-application benchmarks. However, most of these benchmarks are old and may not be suitable for web 2.0 due to the lack of interactive content and support for mobile users. Few of the benchmarks are modified (e.g. TPC-W [116]) while many new benchmarks are proposed to tackle the complexity of web 2.0.

CloudStone [135] and WPress [43] are multi-platform benchmarsk for web 2.0 which are composed of a load injection framework and a test application. BenchLab [49] is another realistic cloud benchmark that uses Wikibooks [28] and Cloudstone [135] as the backend server and a browser-based load generator to replay the workload trace. However, most of the above research in benchmarking is focused on developing a better benchmark application in terms of the reality of emulation, efficiency and scalability. Also, the benchmarks configuration, host configuration and set up, benchmark execution and result collection are performed manually. It is not convenient to perform all these operations ourselves in a multi-cloud environment as each cloud provider has specific APIs/SDKs for interaction. Also, benchmarking across a large number of hosts increases the complexity.

To compare the performance of web-applications on various cloud environments, CloudGuide [103] is proposed which compares the performance of the cloud environment using a legacy web-application. However, the profiling of the cloud host is generic without considering the application complexity and dependency. CloudBench [134]

and Smart CloudBench [50] offers a framework that automates the benchmarking and evaluation in a multi-cloud environment. Another model, Okta [57], provides a generic framework for the execution of complex multi-tier benchmarks on the real cloud environment. It is dependent on Cloud WorkBench [128] for the basic benchmarking, Apache Jmeter [17] for load generation and Chef [7] for provisioning and automation.

These frameworks are specific for the virtual machine environment and may not be applicable for the containerized microservices. In addition to this, defining the benchmarks using these frameworks is not easy and requires specific knowledge. Performing benchmark is very expensive but none of the available work considers the cost of benchmarking. Also, testing the host for longer duration gives more accurate evaluation but imposes high cost and is not always suggested. Since the number of cloud hosts is very large and the benchmark application needed to execute for a longer duration, it is not possible to benchmark all the available hosts. In Chapter 4 of this thesis, we propose an orchestrator for automating the benchmarking of web-application microservices that allows a user to choose a set of cloud hosts from the available host list. It then finds a subset of hosts which maximizes the diversity and execution time within the defined budget. The orchestrator is based on Infrastructure as a code that allows users to reuse the available code.

2.5.3 Deployment of geo-distributed web-application microservices via automated benchmarking in a multi-cloud environment

Current web-applications deliver personalized services to their users distributed across the globe. To handle the varying user requests, current WA require a sophisticated architecture that supports distributed databases, geographical replication of contents, temporal and spatial caching mechanisms along with fast prefetching. Deployment of applications in a geo-distributed manner has been well studied in the literature [74, 96, 107, 150–152], however, most of the available work is on the scientific workflows. Compared to web-application, these systems are not affected by the location of users accessing the application thus affecting the response time. Various benchmark studies and orchestrators are proposed for web-application as discussed in §2.5.2 however, none

of them are able to handle the complexity of geo-distributed web-application.

To overcome these challenges, we propose an orchestrator for the deployment of geo-distributed web-applications. It first finds a set of deployment solutions with maximum host and location diversity in a defined budget for benchmarking. Next, it executes all those solutions with a test web-application and captures various metrics. Finally, it evaluates the collected metrics to find an optimal solution for the deployment of the geo-distributed web-application. More details are given in Chapter 5.

2.5.4 Deployment of streaming application microservices in cloud-edge environments

Compared to web-applications, the current streaming application is distributed and deployed across the edge and cloud environment. As discussed in §1.1 the inclusion of edge devices brings additional challenges as compared to cloud-native environments. The deployment problem of stream processing in the cloud environment has been extensively studied in the literature. Frameworks such as Stream [35], Flextream [70], Naiad [113], Cayuga [46] from academia and Apache Storm [3], Amazon Kinesis [2], Google MillWheel [31], Time-Stream [119] from industry are common examples of stream processing frameworks. However, these approaches are limited only to the cloud environment.

Very few models are available in the literature that considers the deployment of streaming applications across edge and cloud environments. Work in [69, 125] presents a framework for large-scale distributed streaming applications in the cloud-edge environment, however the model is theoretical and very simple.

[127] proposes a programming infrastructure, Foglets, for distributing the deployment across edge and cloud environments. However, the model is evaluated using simulation. Another model LEONORE [144] is presented that provisions applications on resource-constrained edge devices for flexible application deployment. The model is scalable but it does not consider the application's QoS requirements in the deployment process.

Kea [56] is another framework for offloading the sensor data computation for processing on edge or cloud. Similar work is done in [81] which considers the problem of dynamic

Table 2.1: A summary of literature review with the major challenges addressed in this thesis.

Problem	**Related Works**	**Challenges**
Evaluation of containers for interfering microservices in cloud environment	[62], [153], [112], [18], [36], [75], [68], [123], [130], [156], [159]	- no intra-container performance evaluation. - no cgroups enabled or disabled performance evaluation.
Orchestration for benchmarking containerized web application in multi-cloud environment	[27], [23], [28], [25], [116], [135], [43], [49], [135], [103], [134], [50], [57],	- most frameworks are specific for the VM environment only. - defining the benchmarks is not easy - benchmarking a large number of host for longer duration is very expensive
Deployment of geo-distributed web-application microservices in a multi-cloud environment	[96], [74], [107], [150], [152], [151], [65], [57], [65], [109], [54], [128], [51], [115], [48], [30]	- not able to handle the complexity of geo-distributed web-application - most of the work ignores the geo-location of client for making the deployment decision
Deployment of streaming application microservices in cloud-edge environments	[2], [3], [31], [35], [46], [69], [70], [113], [119], [125], [127], [144], [56], [81], [47]	- most of the work are theoretical - not considered automatic computation partitioning and deployment - not considered conflicting non-functional requirements

computation offloading on wearable healthcare devices. The main focus of both these work is to make a decision about offloading the data processing to cloud or not. An abstract model to support QoS-aware deployment is presented in [47], where a multi-component IoT application is deployed across fog infrastructure. A simple Java-based prototype, FogTorch is presented to illustrate the proposed model however, it does not address how to optimize the deployment solution.

In addition to these frameworks, various simulation environments are proposed for modelling the application deployment in an edge-cloud environment. iFogSim [67], EdgeCloudSim [137] and IoTSim-Edge [76] are some popular simulators, however, these environments are very generic and are not able to give infrastructure-specific performance evaluation.

Early efforts centred on the deployment of IoT applications across cloud and edge datacentres are mostly theoretical. Moreover, these solutions have not considered automatic computation partitioning and deployment. Nor have they considered the optimization of multiple conflicting non-functional requirements during the deploy-

ment process. In Chapter 6 of this thesis, we propose, implement and evaluate an optimized framework to find a suitable deployment solution for the distributed stream processing application. The framework incorporates the user preferences along with a high-level computation description to generate the deployment solution which optimizes the conflicting non-functional requirements.

A summary of research problems, related works and the current challenges is summarized in Table 2.1.

3

Holistic evaluation of Docker containers for interfering microservices

Contents

Summary

To investigate the performance evaluation of containerized microservices, this chapter presents an extensive experimental evaluation of microservices executing in an interfering cloud environment. Specifically, we have considered the performance variation of heterogeneous High-Performance Computing (HPC) microservices executing together inside a container or in different containers. Moreover, we have also investigated the performance variation due to the explicit definition of cgroups. The experiment results can be used in understanding the behavior of HPC microservices in the interfering environment and can further also be used to make smart deployment decisions for microservices.

3.1 Introduction

Virtualization is the key concept of cloud computing that separates the computation infrastructure from the core physical infrastructure. There are numerous benefits of virtualization such as: (a) it supports heterogeneous applications to run on one physical environment which is not otherwise possible, (b) it allows multiple tenants to share the physical resources which increases the overall resource utilization, (c) multiple tenants are isolated from each other using virtualization abstraction that maintains the performance of each virtualized application guaranteeing the QoS requirements, (d) it also helps in easy allocation and maintenance of resources for each tenant, (e) it helps in easy resource scale up or scale down depending on the dynamically changing process requirements and (f) it increases the service availability and reduces the failure probability. Applications leverage the advantages of virtualization for cloud services in the form of software, platform or infrastructure [154].

There are two types of virtualization practices common in cloud environments namely, hypervisor-based virtualization and container-based virtualization. Hypervisor-based virtualization represents the de-facto method of virtualization where, the hypervisor partitions the computing resources in terms of virtual machines (VMs) e.g. KVM [86], VMWare [66]. Each VM possesses an isolated operating system allowing heterogeneous consolidation. However, the advantages of virtualization are provided at

a cost of additional overhead as compared to the non-virtualized system. Since there are two levels of abstraction, top-level by VM operating system and bottom level by the physical host machine, any delay incurred by the VM layer can not be removed. Current research trends concentrate on reducing the degree of performance variation between virtualized and bare-metal systems [136]. Containers provide the virtualization advantages by exploiting the services provided by the host operating system e.g. LXC[19], Docker[11]. Except for applications that require strict security requirements, it becomes a viable alternative for virtual machines.

Different features provided by the containers (e.g. light-weight, self-contained, fast start-up and shut down) makes it a popular choice for virtualization. Recent research findings [62, 68, 123] verifies the suitability of the containers as an alternative deployment infrastructure for the high performance computing (HPC) applications. Most of the HPC applications have strict software requirements including system libraries and support software which are closely dependent on the operating system version, underlying compilers and particular environment variables. Containers can easily embed these functionalities in an image that can run on heterogeneous host platforms. These features allow containers to perform repeatable and reproducible experiments on different host environment without considering the system heterogeneity and platform configurations. Container images also are very flexible as it can be easily customized to add or remove any additional functionality. A recent study [61] also shows that multiple microservices can be executed inside a container.

Executing different microservices together can have many benefits such as dropping off any inter-container data transfer delay, efficient utilization of the resources, avoiding any dependency, etc. Using this scenario is suitable for HPC workloads where the resource requirements for each component/microservices are fixed and known apriori. However, the performance of containerized microservice might be affected by other microservices running inside the same container causing intra-container interference. The performance of microservices running in separate containers may also get affected because of inter-container interference as the containers share the same host machine. The effect of interference is higher if both the microservices are having similar resource requirements (competing). To make an optimal decision about the deployment of

microservices requires extensive performance evaluation of both intra-container and inter-container interference.

Despite the increased interest in container technology, there is a lack of detailed study that evaluates the performance of containerized microservices considering different interference effects. Many research studies are available for HPC micro-benchmarks running in containerized environment [62, 68, 112, 123] but they normally consider micro-benchmarks running in an isolated environment. Our work is built on the existing works by evaluating the performance variation of containerized microservices while considering the interference effects. In a nutshell, this chapter is intended to answer the following research questions:

RQ 1. How does the performance of containers vary while running in the intra-container and the inter-container environment?

RQ 2. Is it suitable to deploy multiple microservices inside a container? If yes, which type of microservices should be deployed together?

The motivation of this chapter is based on the above two research questions. This chapter answers these questions and provides an understanding of performance variation for HPC microservices. The most common way to evaluate the performance of a system is to benchmark the system parameters. To represent the behavior of the HPC application, we considered a set of micro-benchmarks where each micro-benchmark is specific for a particular resource type. Here, the micro-benchmarks are considered as microservices. For evaluating the performance of common system parameters, namely CPU, memory, disk and network, we consider Linpack, STREAM, Bonnie++ and Netperf (TCP Stream and TCP RR) micro-benchmarks respectively. We also considered another micro-benchmark Y-Cruncher which, has an affinity towards both CPU and memory. All these micro-benchmarks are evaluated in the Docker container environment under real-world conditions. To ease the performance evaluation of containerized microservices, we employed Cloud Evaluation Experiment Methodology (CEEM) [101]. In particular, the main contributions of this chapter are as follows:

- We evaluate the performance of containers running collocated microservices causing intra-container interference and compare it with the baseline container that

runs only one microservice in an isolated environment. This helps us to identify the interference effect of varying microservices, each intended towards specific resource types, running inside a container (intra-container interference). This also gives an idea about mixing different microservices inside a container with minimal performance degradation.

- We also evaluate the performance of containers running in an inter-container environment. Two containers running in parallel can cause interference and the effect of interference depends on the type of microservice the containers are executing. If both the containers are executing microservices having similar resource requirements, the interference effect may be higher. Our result compares the performance of this interference with the baseline performance and intra-container performance. The result can also be used for modeling smart container resource provisioning techniques to minimize the interference effect.

Outline. The rest of this chapter is organized as follows. The basic concepts of evaluation methodology, CEEM is presented in §3.2 followed by the application of CEEM for the evaluation of Docker container in §3.3. §3.4 presents the experimental results with the detailed inference effect. §3.5 gives some relevant related work while §3.6 gives some insights about the results. Finally, a detailed discussion along with the conclusion is presented in §3.7.

3.2 Evaluation methodology

In order to investigate the performance of heterogeneous HPC microservices running in a container (such as Docker), we followed the Cloud Evaluation Experiment Methodology [101]. CEEM is a well-established performance evaluation methodology for cloud service evaluation and provides a systematic framework to perform evaluation studies that can easily be reproduced or extended for any environment. Due to similar guiding principles of VMs and containers, we argue by using CEEM, we will achieve rational and accurate experimental results [100]. The steps of CEEM is briefly illustrated as follows [98, 101]:

1. **Requirement Recognition**: Identify the problem and state the purpose of the proposed evaluation.

2. **Service Feature Identification**: Identify cloud services and their features to be evaluated.

3. **Metrics and Benchmarks Listing**: List all the metrics and benchmarks that may be used for the proposed evaluation.

4. **Metrics and Benchmarks Selection**: Select suitable metrics and benchmarks for the proposed evaluation.

5. **Experimental Factors Listing**: List all the factors that may be involved in the evaluation experiments.

6. **Experimental Factors Selection**: Select limited factors to study, and also choose levels/ranges of these factors.

7. **Experimental Design**: Design experiments based on the above work. Pilot experiments may also be done in advance to facilitate the experimental design.

8. **Experimental Implementation**: Prepare experimental environment and perform the designed experiments.

9. **Experimental Analysis**: Statistically analyze and interpret the experimental results.

10. **Conclusion and Reporting**: Draw conclusions and report the overall evaluation procedure and results.

To represent our evaluation in a better-structured way, we divide the CEEM methodology into two major steps namely Experimental design and Experimental evaluation as given in §3.3 and §3.4 respectively.

3.3 Performance evaluation: Experimental design

3.3.1 Requirement recognition and service feature identification

Following the CEEM methodology, the whole study is completely based on explicitly defined requirements. In this chapter, our main aim is to evaluate the performance variation of containerized microservices executing in the interfering environment and compare it with the baseline performance. The requirement is defined in terms of two research questions as given in §3.1. The evaluation is mainly driven under the following three scenarios:

Case 1. *Single container running one microservice.* The resources are constrained by defining the strict cgroups for different resource types. This performance acts as a baseline for the remaining experimental comparisons.

Case 2. *Single container running multiple microservices (either competing or independent).* No cgroups restrictions are enforced so, containers can share the host machine resources in a fair-share manner. We call this set-up as intra-container configuration.

Case 3. *Multiple containers each running one microservices.* We specified two subcase:

a. *No cgroups:* no cgroups restrictions are defined so containers can compete for the resources in a fair-share manner.

b. *With cgroups:* the maximum resource a container can use is limited by specifying the cgroups restrictions.

We call this set-up as inter-container configuration.

For the sake of experimental validity and fair performance comparison, the resources allocated to each container depends on the number of microservices executed by that particular container. For instance, the resource allocated to a container deploying

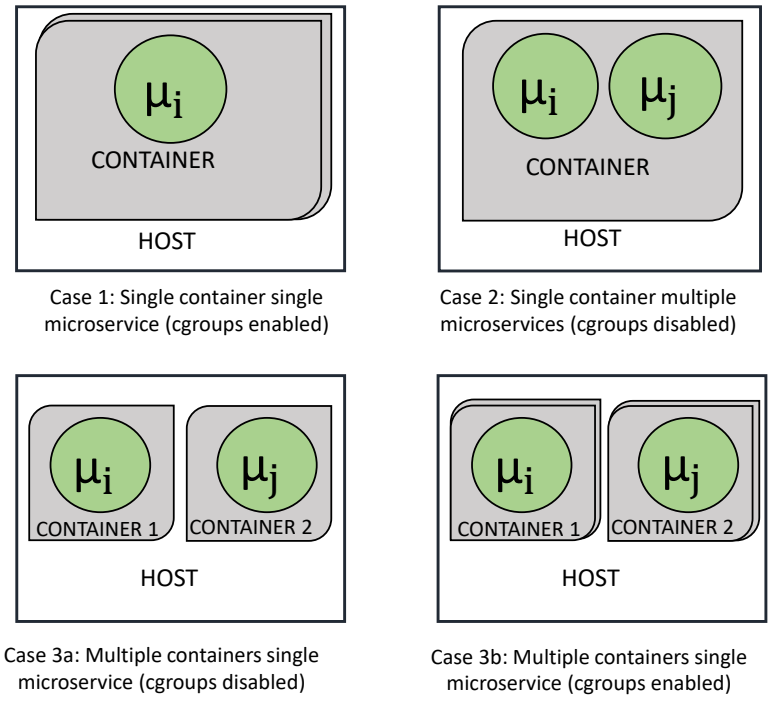


Figure 3.1: Resource restrictions provided by the cgroups

two microservices is double the resource allocated to a container running only one microservice. Figure 3.1 depicts the different scenarios explained here.

In this study, we view containers as an alternative to virtual machines. Following the cloud service evaluation strategy [99], we examine the fundamental resource parameters e.g. CPU computation, memory, I/O and network to evaluate.

3.3.2 Metrics and benchmarks listings and selection

For measuring the performance of containerized microservices, we need to consider the metrics that represent the exact system behavior. The selection of benchmarks depends on the chosen metrics, again, it is also required to be easily configurable and customizable to adapt to different system configurations. The metrics and benchmarks selected for the fundamental resource parameters are discussed below:

1. *CPU Computation Performance:* CPU is the system component responsible for all the processing operations happening in the system. To measure the CPU computation performance, we considered measuring FLOPS (Floating Point Operations Per Seconds), Total Computation Time and Total Turnaround Time.

Table 3.1: STREAM benchmark operations

Operation	**Kernel**	**Flops per Iteration**	**Bytes per Iteration**
COPY	A[i] = B[i]	0	16
SCALE	A[i] = n × B[i]	1	16
ADD	A[i] = B[i] + C[i]	1	24
TRIAD	A[i] = B[i] + n × C[i]	2	24

To check the FLOPS, we used the HPC benchmark, Linpack [15]. It measures the CPU computation performance by solving a set of linear algebra equations of defined order (N) using partial pivoting and Lower-Upper (LU) factorization. It estimates the highest CPU performance.

To evaluate the Total Computation and Turnaround Time, we considered Y-Cruncher. Y-Cruncher [29] is a CPU+memory benchmark that stresses the CPU by computing the value of Pi for large decimal digits. It is also dependent on the disk memory for swapping the content at run time when the available memory is not enough. It measures the performance for single-core as well as multi-core systems. Y-cruncher is very flexible as it allows us to set different run-time parameters.

2. *Memory Performance:* To measure the memory performance, we considered STREAM [26] micro-benchmark that measures the data throughput for different memory operations. We choose STREAM benchmark for analyzing the memory performance. The system performance is measured by performing different operations (COPY, SCALE, ADD and TRIAD) on the memory system. Table 3.1 explains the kernel operations and FLOPS used by the STREAM operations. The result of STREAM is presented in terms of MB/sec.

3. *Disk I/O Performance:* We considered disk throughput and random seeks to measure the disk I/O performance. To measure the disk throughput, we used Bonnie++ [6] micro-benchmark, which allows us to measure the I/O file system performance with respect to data read/write speed. The output represents different performance parameters in terms of data read/write, data rewrite and random seeks per second.

Table 3.2: Metrics and Benchmarks for selected resource types

Resource Type	Selected Metrics	Selected Benchmarks	Version
CPU	FLOPS (Floating Point Operations Per Sec)	Linpack	Mklb_p_2018.0.006
	Total Computation Time Total Turnaround Time	Y-Cruncher	0.7.5
Memory	Data Throughput	STREAM	5.10
Disk I/O	Data Throughput Random Seeks	Bonnie++	1.03e
Network	Network Throughput	Netperf	2.7.0

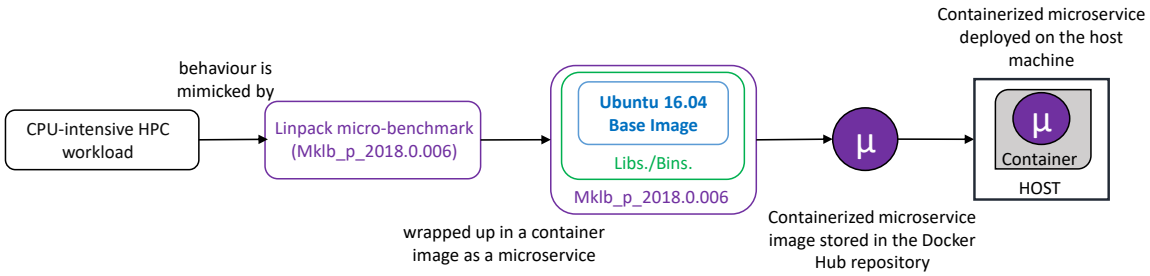


Figure 3.2: Steps for Linpack HPC microservice construction

4. *Network Performance:* To measure the network performance, we considered round-trip network throughput. We choose Netperf [21] for measuring network throughput. It is a request-response benchmark that measures the network performance between two hosts. We identified the bidirectional network traffic using the TCP-Stream test. We also used to show the round trip network performance by using the TCP-RR test. To maintain integrity, no external traffic is placed during the test duration. The results are given in terms of Mbps.

Table 3.2 summarizes the selected metrics and benchmarks for different resource types. For deployment, the micro-benchmarks are first containerized by wrapping up in the form of a container image and then initialized to evaluate the performance. Figure 3.2 shows the whole process of composing a Linpack microservice benchmark and deploying on a host machine. A similar process is used for other micro-benchmarks. Finally, the container image is stored in the Docker Hub [12] repository so that it can be easily downloaded and deployed anytime.

3.3.3 Experimental factors listings and selection

The performance of designed experiment is entirely driven by the experimental factor selection. Following the experimental factor framework for cloud service evaluation [102], we identify the various factors as given below:

- *Resource Type:* We considered Docker container (version: $17.05.0 - ce$, API version: 1.29, Go version: $go1.7.5$) for our evaluation. The selection of Docker containers is because of its popularity and uniformity across the cloud environments. Docker can easily wrap up the application along with its dependency on one image that can be deployed in different environments without having any prior knowledge of the underlying infrastructure environment.

- *CPU Index:* The CPU configuration of the host machine running Docker container is $X64$ bit CPU @ 2.30 GHz processor with 2 cores. For Case 1 and Case 3b, each container can use only 1 CPU core as specified by the cgroups while for Case 2 and 3a, both the available cores are shared by the two containers in a fair share manner.

- *Memory and Storage Size:* The host memory and storage configuration is 4 GB DDR3 RAM and 50 GB respectively. Similar to CPU configuration, containers in Case 1 and 3b can use 2 GB and 25 GB of RAM and storage respectively while the configuration is fairly shared for Case 2 and 3a.

- *Operating System:* The operating system employed for all the experiments is Ubuntu:16.04. Docker also uses Ubuntu:16.04 as a base image for all the containers.

- *Workload Size and Configuration:* For each micro-benchmark, we specified a particular configuration. For Linpack, the problem size i.e. the number of equations to solve is considered to be 15000. Also, the leading dimensions of the array and data alignment value are set to 15000 and 4 Kbytes respectively. For Y-Cruncher, the decimal digit is set to $100M$. We are only concerned about the single-core performance so parallelism is disabled (-PF: none). We set the STREAM benchmark by configuring DSTREAM_ARRAY value as $60M$ and DNTIMES value

as 200. The file size for Bonnie++ is set to 8192 MB while the uid is set as root. Finally, for Netperf, we specified TCP as the selected protocol. To check the network streaming and round trip performance, we choose TCP-Stream and TCP-RR benchmark. Also, we set the testlen at 120 seconds.

3.3.4 Experimental design

Our aim is to evaluate the performance of individual microservice running in the containerized environment. We used **docker run** command to start a new container instance. The container is removed (using —**rm** instruction) after finishing the execution and a new container instance is started. For Case 1 where only one microservice performance is evaluated at an instance, we simply run the containers and collect the results. To validate the results and normalize for any variations, we repeated our experiments for 50 times.

For running multiple microservices together, we considered all combinations as discussed in §3.3.1 with different cases of independent and competing microservices e.g. CPU-intensive with other CPU-intensive or with memory-intensive and so on. Since the average running time of different microservices are not identical, running the experiments for Case 2 and 3 for a particular number of iterations is not suitable. Therefore, we repeat the experiments for an interval of two hours and compute the average performance. For Case 2, both the microservices are executing in parallel in an infinite loop while for Case 3, both the containers are running in parallel.

3.4 Performance evaluation: Experimental results

This section describes the experimental evaluations illustrating the effect of interference for containerized microservices executing in different cases as given in §3.3.1. For the easy representation of the results, the following abbreviations are used for the microservices, Bonnie++: B, Linpack: L, Netperf TCP-Stream: NS, Netperf TCP-RR: NR, STREAM: S and Y-Cruncher: Y.

For each experimental outcome, we compute different statistics e.g. Mean, Trimmed Mean, Median, Maximum, Minimum, Standard Deviation (SD), Coefficient of Variance

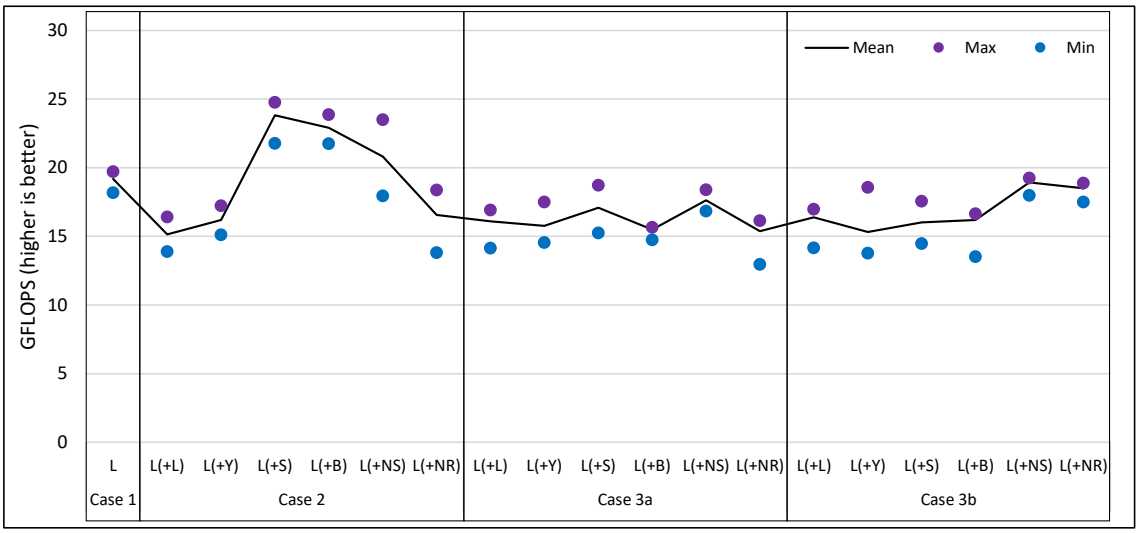


Figure 3.3: Linpack performance results

(CV) and Interference Ratio (IR). These statistics are categorized into three types. The first category consists of Mean, Trimmed Mean and Median that represents the average result. Mean is the most commonly used parameter to represent the average result, however, in some situations where there is a large variation in the result, mean does not provide the exact average. Hence, we also selected Trimmed Mean and Median values. Trimmed Mean simply discards the 10% extreme value (that may represent the error spikes) while calculating the total average. The second category of statistics consists of Maximum, Minimum, SD and CV. Maximum, minimum and SD represent the variations of the result but are not able to give a clear comparison for the different range of values. To compare the degree of variation between the different range of values, we chose CV. Finally, IR composes the third category of statistics which explains the effect of interference as compared to the baseline performance. IR is calculated using the following equation:

$$IR = \begin{cases} (\mu_i - \mu)/\mu, & \text{if } \textit{higher is better} \\ (\mu - \mu_i)/\mu, & \text{if } \textit{lower is better} \end{cases} \tag{3.1}$$

where, μ_i is the mean value for the particular set of microservices and μ is the baseline mean. The positive value of IR represents the performance enhancement while negative IR value represents the performance degradation.

Table 3.3: Linpack result (GFLOPS)

		Mean	Median	Tr. Mean	SD	CV
Case1	L	19.17	19.37	19.19	0.5	0.026
Case 2	L(+L)	15.14	15.14	15.14	0.641	0.042
	L(+Y)	16.18	16.48	16.43	0.94	0.058
	L(+S)	23.81	23.99	23.87	0.6	0.025
	L(+B)	22.92	23.06	22.93	0.586	0.026
	L(+NS)	20.81	20.55	20.82	1.753	0.084
	L(+NR)	16.57	16.88	16.62	1.193	0.072
Case 3a	L(+L)	16.09	16.47	16.16	0.871	0.054
	L(+Y)	15.76	15.68	15.73	0.961	0.061
	L(+S)	17.09	17.5	17.1	1.403	0.082
	L(+B)	15.49	15.81	15.58	1.177	0.057
	L(+NS)	17.63	17.71	17.63	0.496	0.028
	L(+NR)	15.38	15.68	15.47	0.874	0.057
Case 3b	L(+L)	16.39	16.75	16.48	0.824	0.05
	L(+Y)	15.32	15.4	15.22	1.03	0.067
	L(+S)	16.02	15.75	16.02	1.104	0.069
	L(+B)	14.18	14.24	14.19	0.314	0.022
	L(+NS)	18.93	19.01	18.97	0.284	0.015
	L(+NR)	18.49	18.61	18.53	0.359	0.019

1. *CPU Computation Performance Evaluation and Analysis*

   To evaluate the CPU performance, we implemented Linpack and Y-Cruncher microservice in Docker container. Figure 3.3 shows the arithmetic Mean with Maximum and Minimum value for the performance of Linpack in different scenarios. Other statistics are presented in Table 3.3. The result shows that the performance of Linpack is highest in Case 2 L(+S) with a value of 23.81 GFLOPS, which is 24% higher than the baseline performance. The next highest performance is for Case 2 L(+B) followed by Case 2 L(+NS) with a performance gain of 19% and 8% respectively. The performance gain is achieved because of the availability of extra computation resources not used by other microservices (non-CPU intensive) thus, increasing the performance of Linpack.

   For all the other cases, considerable performance interference is noticed. The worst performance is observed in Case 2 L(+L) where, two instances of Linpack are competing in the same container with a performance degradation of 21%. This is because of the lack of resource pinning which causes both the microser-

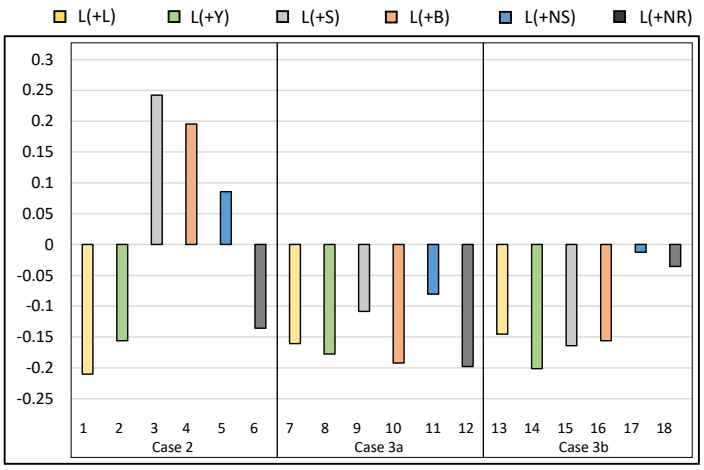


Figure 3.4: Linpack Interference Ratio (IR) values. Horizontal axis labels represent various cases. 1 – 6 represents L(+L), L(+Y), L(+S), L+(B), L(+NS) and L(+NR) for Case 2. Similarly, 7 – 12 and 13 – 18 is used to represent different scenarios for Case 3a and 3b respectively.

vices to compete for the same core at a time even though multiple cores are available. For two Linpack instances, the best performance is observed for Case 3b where microservices are running in separate containers with cgroups enabled with the performance degradation of only 14%. The remaining performances are comparable with the baseline performance. The effect of interference is clearly observed in Figure 3.4.

The result in Figure 3.3 and Table 3.3 also shows that the results do not deviate too much from the Mean value. The maximum deviation is noticed in Case 2 L+(NS) followed by Case 3a L+(S) with the SD of 1.753 and 1.403 and CV of 8.4% and 8.2% respectively. Also, the difference between the Mean and Median is very small with the highest difference of 0.41 for Case 3a L(+S) which is much smaller than the SD value (1.403).

Y-Cruncher is a CPU as well as memory-intensive microservice. The average performance of Computation Time (CT) and Total Time (TT) evaluated by Y-Cruncher in different scenarios is presented in Figure 3.5. The result shows that the performance of Y-Cruncher is worst for Case 2 Y(+L) with a performance degradation of almost 46% as compared to the baseline performance. This is because of the fact that both Linpack and Y-Cruncher are CPU-intensive microservices, they both compete for CPU resources inside a container and therefore lead

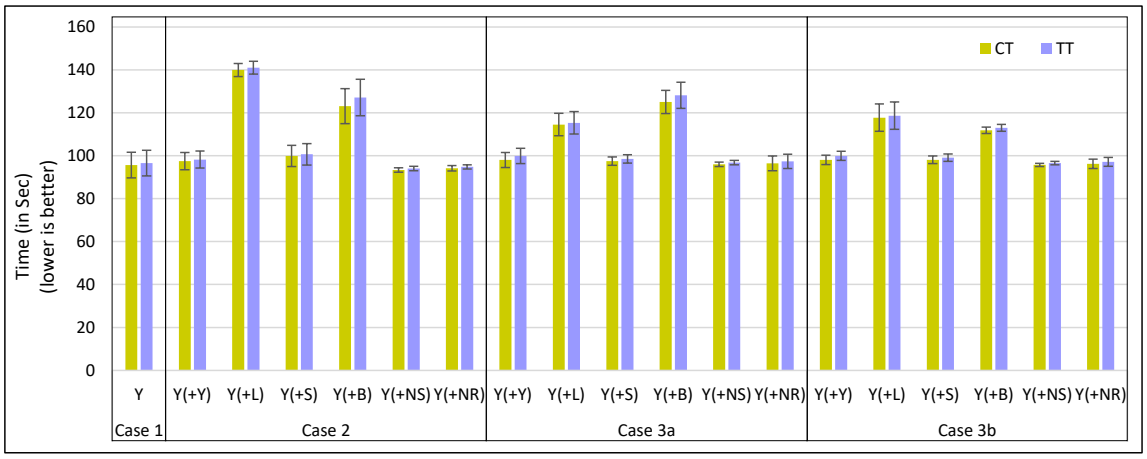


Figure 3.5: Y-Cruncher performance result for Computation Time (CT) and Total Time (TT). Black bars on the top represents the SD.

to performance degradation. Since the operations in Y-Cruncher are highly parallelized using multi-threading which fully utilizes the multi-core processor, there is very small performance degradation ($< 2\%$) for Case 2 Y(+Y) as in this scenario, two cores available for the execution of two instances of Y-Cruncher. For a similar reason, the performance degradation for Case 3a and Case 3b Y(+Y) is only 2.3% and 2.4% respectively. The next worst performance is observed for the collocated execution of Y-Cruncher and Bonnie++ with a performance degradation of 28.7%, 30.6% and 21.4% for Case 2, Case 3a and Case 3b respectively. This is because of the constrained disk size. Since Y-Cruncher uses continuous swapping from main memory to disk while Bonnie++ also accesses the disk for performing different operations, only one process can access the disk memory to perform the I/O leading to the higher completion time for Y-Cruncher.

Even though both Y-Cruncher and STREAM are memory-intensive microservice, for the collocated execution of Y-Cruncher and STREAM, there is only a slight degradation of 4% for Case 2 and 1.9% and 2.5% for Case 3a and Case 3b respectively. The reason behind this is the availability of enough memory to run the experiment without any performance degradation. The best performance is observed for Case 2 Y(+NS) followed again by Case 2 Y(+NR) with a performance gain of 2.4% and 1.5% respectively as they are not directly interfering for any resources. The effect of interference can be easily observed in Figure 3.6.

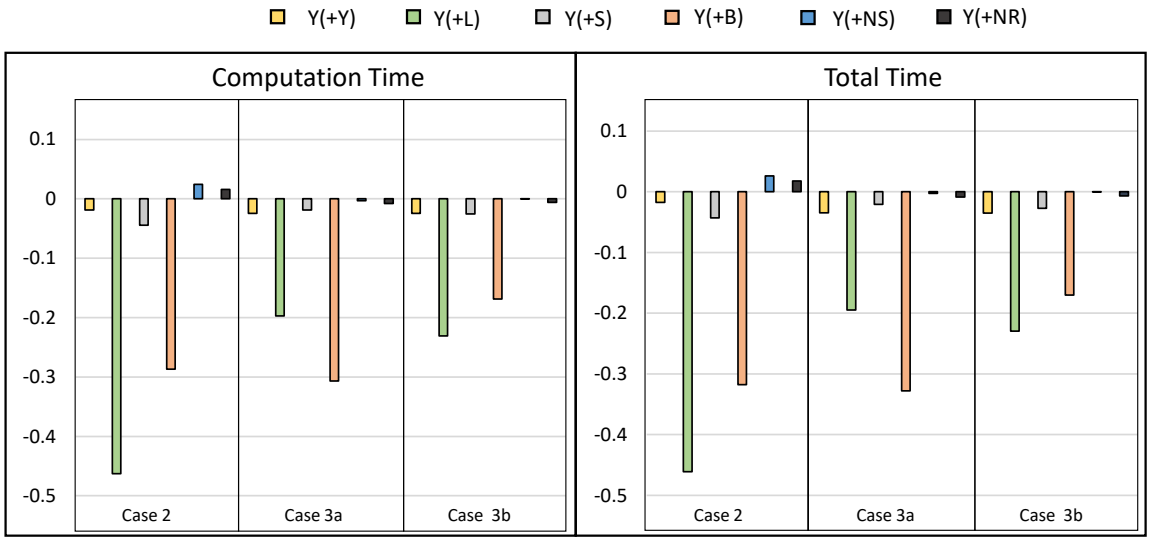


Figure 3.6: Y-Cruncher Interference Ratio (IR) values. Horizontal axis labels represent various cases.

Overall, the result infers that inter-container interference is lesser than intra-container interference while considering a similar type of CPU-intensive microservices. Another important point to notice is that the performance of microservices is comparable for the case with having cgroups enabled or disabled for our defined scenarios.

2. *Memory Performance Evaluation and Analysis*

To evaluate the memory performance, we used STREAM microservice benchmark. Different statistics for the four vector operations namely, COPY, SCALE, ADD and TRAID are presented in Figure 3.7. For COPY operation, degradation of 14%, 15% and 16% is observed for collocated execution of two STREAM microservices for Case 2, Case 3a and Case 3b respectively. This is because of the interference caused by the other memory-intensive operation executing together. The next worst-case performance is observed for Case 2 S(+Y) as Y-Cruncher is also intended to share the available memory with degradation of 3%. For other combinations in Case 2, a slight performance gain is noticed with a maximum of 4.8% gain for S(+L) followed by 3.9% for S(+NS) due to non-strict dependency of these microservices on memory. The result also shows that there is a slight deviation from the Mean value as the Median and Trimmed Mean are almost

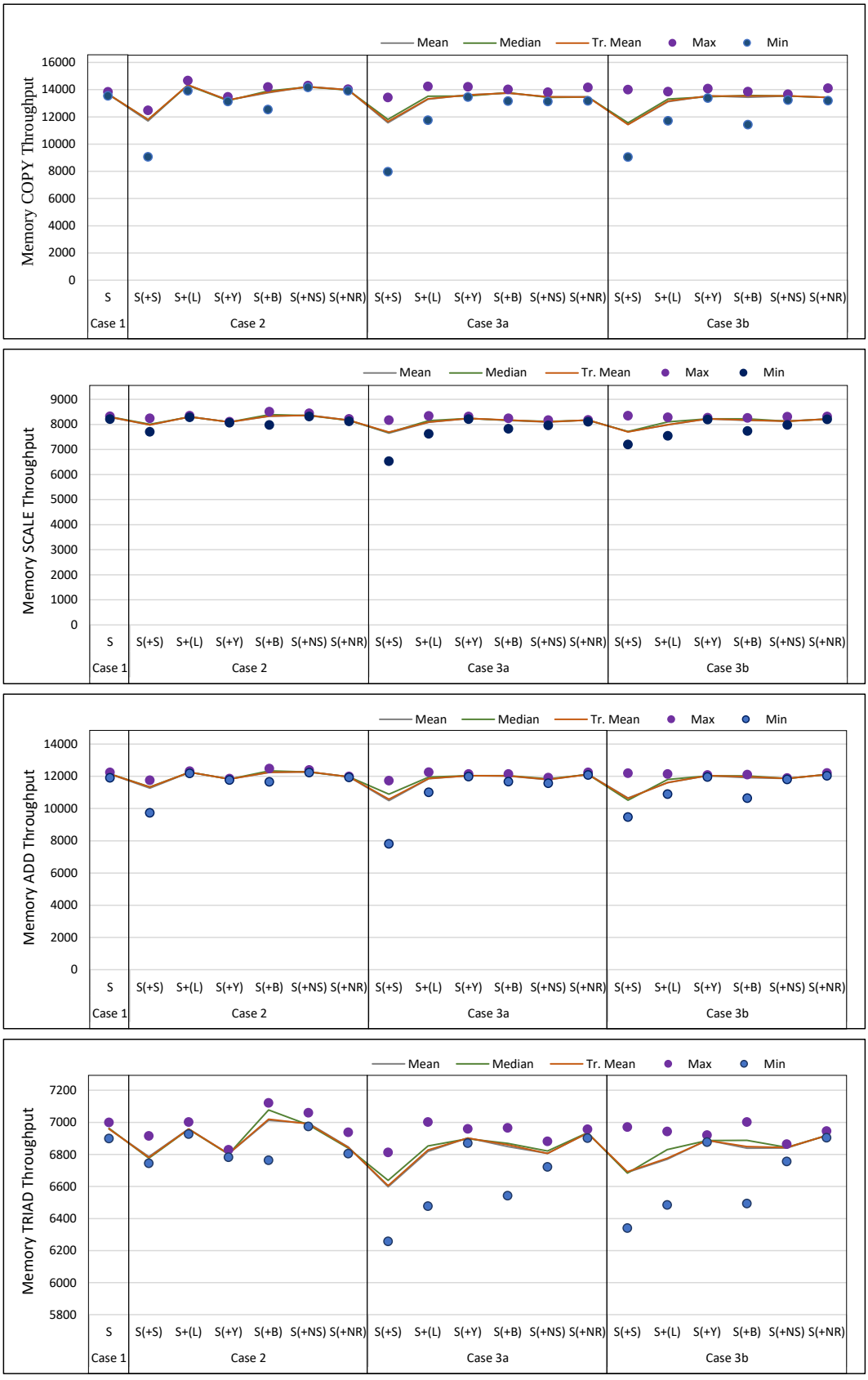


Figure 3.7: STREAM performance result (in GB/sec)

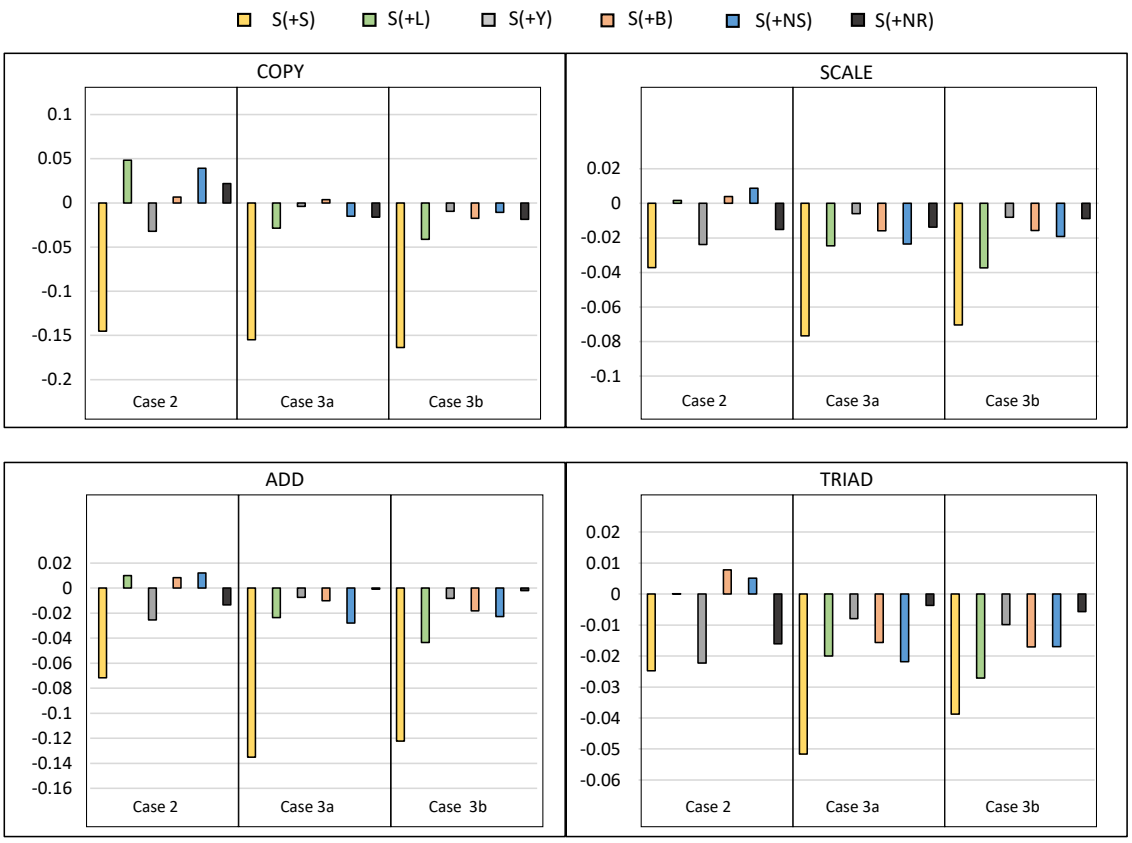


Figure 3.8: STREAM Interference Ratio (IR) values. Horizontal axis labels represent various cases.

the same as the Mean. The Maximum and Minimum values are also very close to the Mean value except for the case of collocated execution of two STREAM instance.

For the SCALE and ADD operation, there is a slight difference in various scenarios. For SCALE operation, the worst performance is for S(+S) with 7.6% and 7% for Case 3a and Case 3b followed by 3.7% for Case 2. The remaining performance is comparable with a maximum gain of 1% for Case 2 S(+L). Similarly, for the ADD operation, the worst performance is noticed for collocated execution of STREAM with degradation of 13%, 12% and 7% for Case 3a, Case 3b and Case 2 respectively. The maximum performance gain is observed for S(+NS) followed by S(+L) with an increment of 12% and 10% respectively. However, for TRIAD operation, a large performance deviation is observed but follows the same trend of performance degradation for collocated execution of the same type

Table 3.4: Bonnie++ Block Input result (in /sec).

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	348948.7	346912.5	349092.2	366590	328725	8816.5	0.025
Case 2	B(+B)	150318.6	151020.0	150499.3	153749	143634	2707.9	0.018
	B(+L)	280510.3	279708.0	280276.2	300307	264927	9073.6	0.032
	B(+Y)	310682.1	306304.0	307027.5	401019	286128	25319.8	0.081
	B(+S)	293854.5	294489.5	294033.4	304766	279722	7891.1	0.027
	B(+NS)	321789.2	322759.5	322253.7	333090	302127	8111.1	0.025
	B(+NR)	345039.6	345520.0	344307.0	379975	323291	12765.5	0.037
Case 3a	B(+B)	152934.3	153021.5	152967.7	155528	149739	1549.4	0.010
	B(+L)	268383.2	271084.0	268774.8	291607	238110	16004.1	0.060
	B(+Y)	292321.3	292458.0	292357.1	306960	277038	7988.0	0.027
	B(+S)	291840.9	293848.5	292551.0	303054	267845	8129.0	0.028
	B(+NS)	330460.5	329437.0	330246.0	367948	296834	14219.2	0.043
	B(+NR)	326684.5	326248.0	326754.7	343819	308286	11378.8	0.035
Case 3b	B(+B)	150259.9	149811.0	150222.7	159286	141903	5348.0	0.036
	B(+L)	292410.9	292421.5	292703.9	301705	277842	6226.1	0.021
	B(+Y)	313616.1	296750.5	297370.1	639995	279666	77770.2	0.248
	B(+S)	294275.9	295652.0	295235.1	302914	268373	7984.3	0.027
	B(+NS)	263074.5	266438.0	264483.2	285272	215519	17366.2	0.066
	B(+NR)	288610.8	293233.0	289713.6	310925	246446	16459.5	0.057

of microservices. The maximum performance deprivation is observed in S(+S) for Case 3a (5%) followed by Case 3b (4%) and Case 2 (2.5%). The effect of interference in terms of IR is given in Figure 3.8.

Overall, the execution of STREAM microservice in different scenarios does not show a large variation from the baseline performance. A small performance gain is achieved when STREAM is collocated with different microservice inside a container. Also, the performances are comparable in Case 3a and Case 3b for different scenarios.

3. *Disk I/O Performance Evaluation and Analysis*

The I/O performance is represented using Bonnie++ microservice which generates a dataset of size at least twice the size of available memory (RAM). The performance for Sequential Block Input, Block Output, Block Rewrite and Random Seeks is presented in Table 3.4, Table 3.5, Table 3.6 and Table 3.7 respectively. For Block Input, the performance is always affected by the collocated execution of other microservices. The maximum performance degradation is observed for

Table 3.5: Bonnie++ Block Output result (in /sec).

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	278362.4	277073.0	278099.4	294574	266885	8425.81	0.030
Case 2	B(+B)	159530.5	152429.5	152711.6	294041	147760	31809.07	0.199
	B(+L)	283667.1	281342.5	281533.3	332479	273264	12189.84	0.043
	B(+Y)	281957.0	283641.0	281708.0	303000	265396	8224.12	0.029
	B(+S)	289314.7	289074.0	289009.1	305278	278851	8192.07	0.028
	B(+NS)	310772.1	309600.5	309897.6	350216	287068	14772.71	0.048
	B(+NR)	283923.6	285201.5	283682.2	297530	274662	6981.28	0.025
Case 3a	B(+B)	148960.7	148394.0	148732.7	161455	140569	4818.72	0.032
	B(+L)	264280.2	263157.0	262325.1	290663	243089	11910.16	0.045
	B(+Y)	252390.3	252859.5	252346.7	293046	232520	4665.51	0.018
	B(+S)	270180.9	279268.5	269561.9	286893	254610	8232.98	0.030
	B(+NS)	262805.1	268557.0	262225.2	273535	242514	12189.49	0.046
	B(+NR)	255982.2	256025.0	256512.7	288801	233615	10050.62	0.039
Case 3b	B(+B)	154354.5	153829.0	153333.8	177870	149210	6037.06	0.039
	B(+L)	269236.3	268707.5	268775.0	286873	249902	13364.55	0.050
	B(+Y)	252809.9	251839.0	252907.3	264715	239152	5977.67	0.024
	B(+S)	270158.8	268596.5	269212.7	291482	265866	9354.95	0.035
	B(+NS)	269616.7	270105.5	269896.2	283752	250450	9592.73	0.036
	B(+NR)	257024.6	257307.5	257125.4	266154	246080	5303.87	0.021

two instances of Bonnie++ with a loss of 56.92%, 56.17% and 56.13% for Case 2, 3a and 3b respectively. The high degradation is occurred because of the common disk which is shared by all the microservices thus, leading to performance degradation. The least interference is noticed for the collocated execution of Bonnie++ with Netperf (NS, NR) with performance loss of only (7%, 1%), (5%, 6%) and (24%, 17%) for Case 2, Case 3a and Case 3b respectively. Table 3.4 also shows that the results are consistent as there is a slight difference between Mean, Median and Trimmed Mean values except for Case 3b B(+Y) and Case 2 B(+Y) with SD value of 77770.2 and 25319.8 and with CV of 24.8% and 8.1% respectively. In these situations, Median and Trimmed Mean are the more appropriate measures to represent the average values. The interference effect is presented in Figure 3.9.

For Block Output operation, a slight performance gain is observed for heterogeneous execution of microservices for Case 2 with a maximum performance gain of 11.6% for B(+NS) followed by 3.9% for B(+S). As usual, the performance of multiple instances of Bonnie++ is worst with a maximum loss of 46.48% for

Table 3.6: Bonnie++ Block Rewrite result (in /sec).

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	149159.4	149094.0	149202.7	153877	143663	2841.94	0.019
Case 2	B(+B)	66082.6	66379.5	66575.9	67674	55611	2546.87	0.039
	B(+L)	129841.8	129942.0	129935.8	134415	123577	3087.65	0.024
	B(+Y)	125820.3	125511.0	125729.9	131053	122215	2721.36	0.022
	B(+S)	131675.6	131611.0	131756.6	137909	123983	4193.53	0.032
	B(+NS)	131091.2	131262.0	131171.2	136593	124149	3128.53	0.024
	B(+NR)	127168.5	126673.5	127022.2	136821	120149	4287.56	0.034
Case 3a	B(+B)	66617.6	66792.5	66614.1	68768	64529	1223.94	0.018
	B(+L)	132061.0	132441.5	132218.2	139880	121412	4558.59	0.035
	B(+Y)	126750.6	126659.0	126806.7	131635	120855	2589.28	0.020
	B(+S)	132399.4	131666.5	132319.7	139712	126521	3440.37	0.026
	B(+NS)	134379.0	133777.5	133984.6	147012	128845	4553.22	0.034
	B(+NR)	111762.8	123709.5	116469.7	127234	11566	34380.04	0.308
Case 3b	B(+B)	67366.5	67807.5	67362.8	70339	64459	1627.47	0.024
	B(+L)	137653.4	137873.5	137813.6	142397	130027	3186.39	0.023
	B(+Y)	133628.2	132893.0	133397.9	142822	128580	3435.94	0.026
	B(+S)	136741.4	137559.5	136850.7	144602	126913	3577.91	0.026
	B(+NS)	126108.8	127890.5	127169.7	131055	102065	6403.13	0.051
	B(+NR)	128480.1	129910.0	128663.6	134240	119417	4381.08	0.034

Table 3.7: Bonnie++ Random Seeks result (in /sec).

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	10801.3	10807.4	10817.4	11890.7	9422.4	628.77	0.058
Case 2	B(+B)	3576.1	3538.0	3572.6	3901.1	3314.5	156.69	0.044
	B(+L)	9974.5	9918.3	9940.6	11877.4	8681.6	813.41	0.082
	B(+Y)	8895.7	8947.2	8915.1	9452.9	7988.8	370.89	0.042
	B(+S)	10274.4	10427.5	10330.5	11516.4	8022.4	779.58	0.076
	B(+NS)	10849.3	11048.3	10878.1	12582.4	8596.7	1224.63	0.113
	B(+NR)	9607.7	9925.2	9626.5	10670.6	8205.9	715.84	0.075
Case 3a	B(+B)	3912.7	3866.1	3905.5	4191.9	3763.3	131.87	0.034
	B(+L)	9290.7	9480.5	9447.6	11660.3	4096.4	1782.48	0.192
	B(+Y)	8779.4	8803.5	8822.2	9550.6	7239.3	488.66	0.056
	B(+S)	10401.3	10583.1	10409.4	11882.7	8773.5	766.95	0.074
	B(+NS)	8198.8	8130.5	81711.2	9231.5	7243.3	785.97	0.096
	B(+NR)	8652.5	8335.0	8720.6	9861.5	6218.2	844.55	0.098
Case 3b	B(+B)	4877.5	4745.0	4896.1	6149.2	3270.8	926.19	0.190
	B(+L)	9821.2	9619.9	9785.0	11232.2	9061.9	614.84	0.063
	B(+Y)	8234.6	8245.3	8249.7	8713.6	7483.6	293.17	0.036
	B(+S)	9803.9	10233.4	9948.3	11260.3	5748.7	1397.07	0.143
	B(+NS)	8180.1	8592.0	8282.0	9531.7	4994.1	1290.04	0.158
	B(+NR)	7642.7	7671.3	7664.9	8224.5	6660.5	451.41	0.059

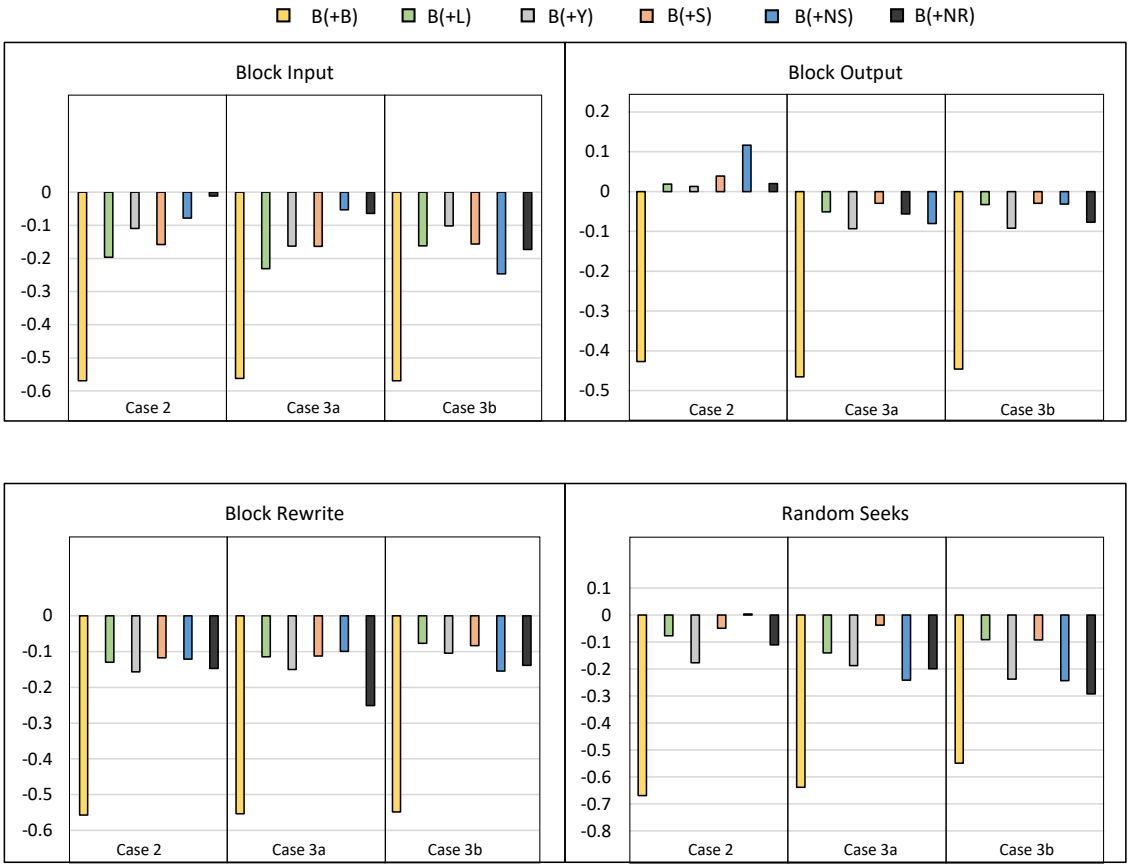


Figure 3.9: Bonnie++ Interference Ratio (IR) values. Horizontal axis labels represent various cases.

Case 3a followed by 44.5% and 42.6% for Case 3b and Case 2 respectively. The remaining performances are comparable to the baseline performance.

The result of Block Rewrite follows the trend of Block Input as is clear from Table 3.6. The worst performance is observed for Case 2 B(+B) with 55.7% followed by Case 3a B(+B) with 55.3% performance loss. The least performance loss is noticed for Case 3a B(+L) with degradation of only 7.7%. A similar performance is visualized for Random Seeks with only a small performance gain of 0.4% for Case 2 Y(+NS). For all other scenarios, there is a performance loss with the maximum of 66.9% for Case 2 B(+B). There is one important point to notice here is that there is a large variation in the result as shown by the SD (CV) values for example in Case 3a B(+L), the SD (CV) is 1782.48 (19.2%).

4. *Network Performance Evaluation and Analysis*

Netperf microservice is used to analyze the system network performance. It

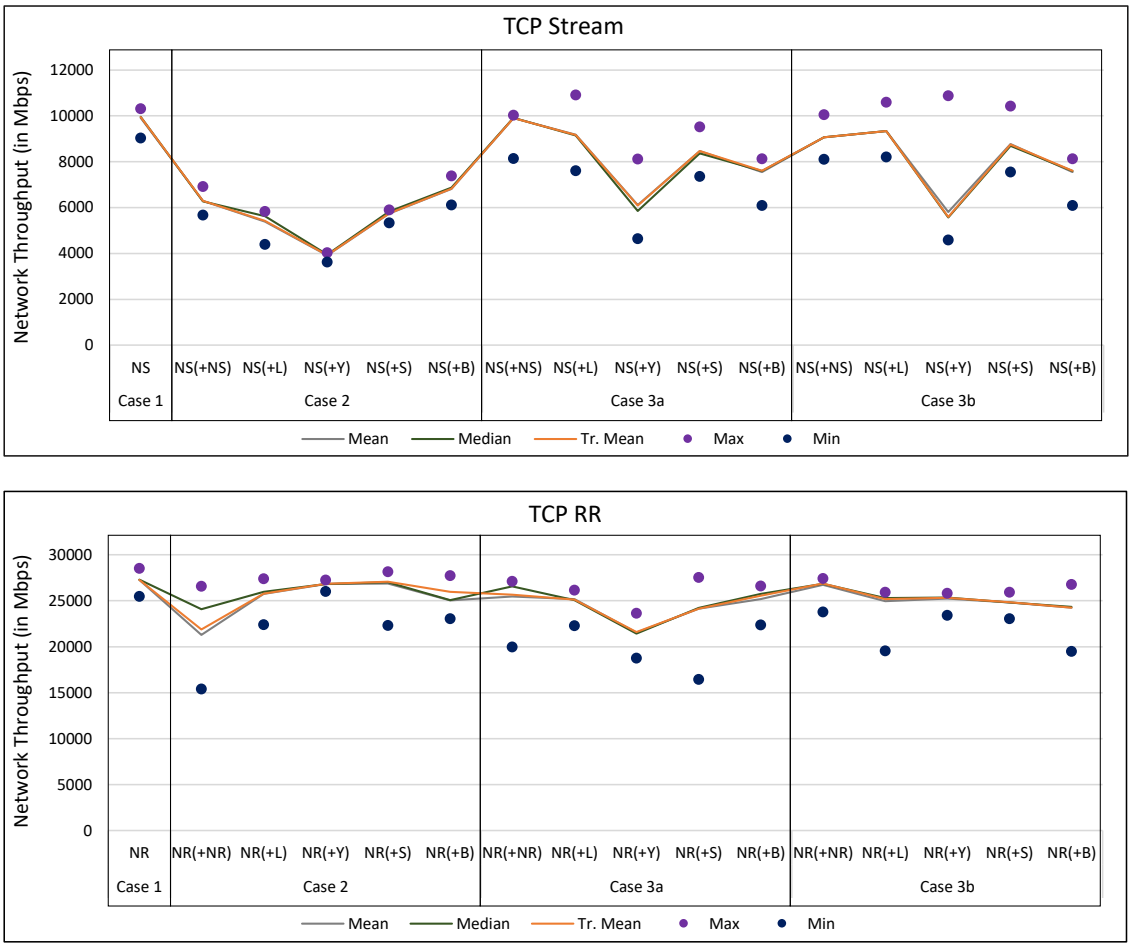


Figure 3.10: Netperf Performance result

uses client-server architecture for data transfer. In our test case, one container serves as a server that runs the *netserver* application of Netperf while the other container serves as the client running the *netperf* application. Data stream is transferred from the client to the server for a defined duration of 120 seconds using TCP protocol and the network performance is analyzed. The throughput of Request-Response is also analyzed for the defined configuration. The experiment results showing the performance of TCP Stream and TCP RR is presented in Figure 3.10.

The result in Figure 3.10 shows that the average throughput for Netperf TCP-Stream is always affected by the co-execution of other microservice. On average the maximum deprivation is observed for multiple microservices executing inside a container (Case 2) with an average performance loss of 42.8%. The worst

performance is noticed for NS(+Y) with degradation of 60.4% as compared to the baseline. For other cases also, there is a large performance degradation for co-execution of TCP Stream with Y-Cruncher with a loss of 38.3% and 41.4% for Case 3a and Case 3b respectively. This is due to the fact that Y-Cruncher is a stress testing tool that stresses both CPU and memory together leading to performance interference. TCP Stream also accesses the memory and CPU for transferring continuous data stream. For the execution of two instances of TCP Stream in different containers, the performance is comparable with the baseline performance with only a small degradation of 1% and 8% for Case 3a and 3b respectively, however, a large degradation of 36% is observed for collocated execution inside a container (Case 2). The result also shows a huge performance difference for other scenarios for Case 2.

For TCP RR, the result is somewhat different from that of the TCP Stream. Most of the performance is comparable with the baseline performance with an exception of two instances of TCP Stream executing inside a container (Case 2 NR(+NR)) with a performance degradation of 22% from the baseline. For this scenario, the result also shows a large variation with Mean and Median values far apart. In other cases, these values are almost the same. The best performance is noticed for the execution of two instances of TCP RR for Case 3a and 3b with a performance loss of only 6% and 2% respectively. The overall interference effect is visualized from Figure 3.11.

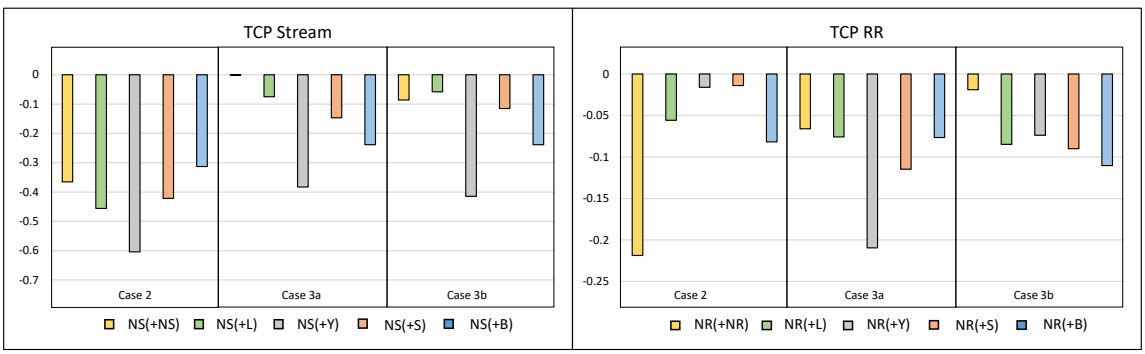


Figure 3.11: Netperf Interference Ratio (IR) result. Horizontal axis labels represents various cases.

3.5 Related work

Numerous efforts [18, 36, 62, 112] show that containerizing the cloud infrastructure leads to highly efficient and agile solutions. It is evident from the previous work that containers can reduce the overhead while increasing the overall performance. These studies compare the performance of containers with respect to VMs for different benchmarks and show that the performance of the container is better than or almost equal to the performance of the VM. Xavier et al. [153] compared the performance of VM with container-based virtualization for HPC environments. The experiments are performed on Linux VServer, OpenVZ and LXC comparing with Xen and bare-metal performance using NAS Parallel Benchmark (NPB). The result shows that container-based virtualization has near-native performance for different fundamental components (CPU, memory, disk and network).

Felter et al. [62] performed similar experiments but with Docker in comparison to KVM using different benchmarks. The result shows that for CPU and memory the performance of Docker container is comparable to KVM but for I/O and network-intensive applications, the Docker's performance is better than KVM. A similar study is performed by Morabito et al. [112], but here LXC and OSv are also compared with Docker and KVM. They conclude that LXC outperforms KVM and Docker in almost all cases. A similar study is given by Li et al. [97] that uses DoKnowMe evaluation strategy to compare the performance of KVM and Docker and illustrates that the effect of virtualization depends not only on features but also on job types. The result shows that the average performance of container is almost better than the VM but it also shows a lot of performance variation in the case of containers.

Similar comparative study between bare-metal, VM and container performed on OpenStack is presented in [89]. The experimental result shows that Docker container has the fastest boot-up time. Also, the performance is comparable with the bare-metal except for network performance. The VM has a high overhead that increases with the workload size and the assigned resources.

A study by Kozhirbayev et al. [90] evaluates the performance of two-container technology Docker and Flockport running different benchmarks and shows that Flockport

outperforms Docker in almost all case. Study in [111] compares the power consumption of container and VM and shows that both types of virtualization has similar power consumption for the idle situation and CPU/memory operations, but containers consume less power for network-intensive operations.

Cuadrado-Cordero [53] compared the QoS and energy performance of Docker containers and KVM for different services. The experimental result shows that Docker allows more services to run as compared to KVM. The result also shows that the Docker consumes less energy than KVM allowing more energy savings.

Few of the works also specify running HPC workloads in Docker containers. Jacobsen et al. [75] advocates the use of containers for HPC environments. The work in [68] shows how to orchestrate multiple containers on a physical node. The study confirms that a job can be transparently executed inside a Docker container without having any knowledge about the underlying host configuration. The study is validated by running Linpack inside the container.

Few of the studies also consider big data applications for comparing the performance of containers. Bhimani et al. [39] compares the performance of VMWare and Docker for different big data applications using Spark. The experimental results show that Docker achieves a speed-up for map- and reduce- intensive applications. Zhang et al. [159] also presented a similar study where an extensive comparison between VM and Docker container is presented for different big data applications. The result shows that Docker containers are more convenient, highly scalable and achieves higher system utilization as compared to VMs.

Most of the above study does not consider the effect of interference in containerized environments. Ruiz et al. [123] evaluate the performance of LXC containers in different configurations e.g. isolated, inter-container and multi-node inter-container using NAS parallel benchmark. The results conclude that inter-container communication is faster than physical machine communication but there is a degradation of CPU performance for memory-intensive operations. The result also shows that for a multi-node inter-container communication, the performance of network-intensive applications degrades. Sharma et al. [130] also compares the performance of collocated applications on a common host but only one application is running in a container/VM. They show

the effects of interference caused by noisy neighbour containers running competing, orthogonal or adversarial applications. All the experiments are done on the LXC container.

Ye et al. [156] also considers the inter-container interference for big data application (Spark). Similar work is done by Kejiang et al. [159] that evaluates the performance of big data applications by changing the system configurations and also considering the interference between containers. Using different Spark applications (K-Means, Page rank, etc.) and changing the Docker system configurations using cgroups, the performance is evaluated.

From the best of our knowledge, none of the existing works consider the performance evaluation of heterogeneous microservices executing inside a container and compares the interference impact with the microservices running in separate containers. In this chapter, we have demonstrated the performance evaluation of HPC micro-benchmarks intended towards specific resource types (CPU, memory, disk and network) in the form of microservices running inside the Docker container.

3.6 Discussion

Extensive experimental evaluation results highlight the performance of microservices executing in the interfering cloud environment. For the combined CPU + memory intensive operations, network intensive operations cause the least performance interference while the core CPU intensive operations cause the highest performance degradation. The result shows that memory-intensive operations are least affected by any other microservices. Memory- and network-intensive operations can be combined together with the least interference effect. I/O intensive operations shows a similar performance for for all the cases. One important point for network-intensive operation is that interference effect is less when the microservices are executed in separate containers.

The interference caused by microservices with similar resource requirements is always higher as compared to the microservices with different resource requirements. The re-

sult also shows that the effect of interference for similar resource-intensive microservice is higher for intra-container scenarios than inter-container scenarios. Also, the performance of containerized microservices are comparable with either cgroups enabled or disabled if the system resources in both the cases are exactly the same.

Limitations. Although the benchmark experiments are extensive, it is performed in a controlled private cloud environment. Executing in public cloud environment may leads to different performance however, we agree that the performance pattern must follow similar pattern. The benchmarking experiments here are performed manually using script which is not feasible for benchmarking large number of systems.

3.7 Conclusion

With the combination of virtualization advantages and bare-metal performance, containers are treated as a feasible alternative for VM in the cloud environment. They bind all the required environment variables along with the application in a container image that can easily be executed in different environments. These advantages can be easily utilized to package HPC microservices, which usually have complex software and hardware requirements. However, the execution of microservices in a containerized environment may cause interference that leads to performance degradation. Therefore, it is necessary to understand the behavior of microservices executing in a containerized environment.

In this chapter, we investigated the performance of HPC microservices in the Docker container environment. The result presents comprehensive details about the performance variation of containerized microservices. It shows that executing multiple microservices inside a container is also a feasible deployment option as the result indicates that for some cases the achieved performance is better than the baseline performance. The obtained result can be used for making further deployment decision.

4

Multi-cloud orchestrator for benchmarking containerized web-application microservices

Contents

Summary

The previous chapter evaluated the performance of containerized microservices in an interfering environment. However, that evaluation alone cannot be used for making the deployment decision. To facilitate the microservice deployment decision, this chapter presents an orchestrator, **S**mart **D**ocker **B**enchmarking **O**rchestrator (**SDBO**). SDBO is a general orchestrator that automatically benchmarks containerized applications in a multi-cloud environment. At the same time, SDBO is able to maximize the numbers of evaluated cloud providers and type of hosts without exceeding users' budgets. Moreover, a *flexible execution* module is proposed which enhances SDBO's ability to capture the performance variation of benchmark web-application for a longer period in the defined users' budgets. Although SDBO is generic enough for a different type of applications, the current chapter focuses only on the web-application benchmarking orchestration.

4.1 Introduction

In the past three decades, web-applications have transformed from serving only static content to a complex multi-tier Web 2.0 system. Current web-applications provide extensive interactivity and personalization support on different devices (smartwatch, mobile phone, tablet, laptop, etc.) for geographically distributed clients in real-time. To handle the varying user's requests, current web-applications require a sophisticated architecture that supports distributed databases, geographical replication of contents, temporal and spatial caching mechanisms along with fast prefetching. With the increasing complexity of web-applications, it is not easy to handle the development and deployment of web-applications in a monolithic way [88, 121]. The evolution of microservice architecture that modularizes the application into smaller independent components gives the flexibility for developers to implement each component as a standalone service. Note that many cloud providers such as Amazon and Microsoft offer containers virtualized at the operating system level which facilitates the deployment of microservices i.e. each component of the web-application can be encapsulated into a container. Since containers have many advantages including light-weight, fast

start-up/shut down, packaged; as a result, users can move their web-applications fast and deploy them efficiently.

However, the multi-cloud environment provides diverse options for users to deploy their web-applications, which means users have more chance to find a cheaper host which still meets their deployment requirements such as cost, throughput and latency. To this end, the users need to test the performance in these hosts before actually deploying and publishing their web-applications. The common practice is to use the standard benchmarking applications to test the hosts instead of using users' own applications. This is because these benchmarking applications have the standard procedures to evaluate the performance of the host, thereby obtaining more comprehensive results. Moreover, benchmarking all the hosts from different cloud providers is very challenging as each provider has its own architecture and programming interface [120]. Existing research [128, 135] focuses mainly on evaluating the benchmarking web-application on different host configurations alone. However, [50, 134] discuss some frameworks that provide automatic systems to perform the benchmark across multiple clouds.

Web-application is a long running system and its performance must be guaranteed all the time. On the other hand, the underlying cloud environment is very dynamic and resource preemption happens frequently in the virtualized environment [92]. The performance is also affected by the interference caused by other applications deployed on the same server [77]. Observing the performance variation for a longer duration is an important task for benchmarking web-application. Unfortunately, running the benchmark applications in various hosts over different clouds for a longer duration (say at least 24 hours) is very costly. To the best of our knowledge, we could not find any study that considers *cost efficiency* for benchmarking i.e. maximize the number of evaluated hosts and benchmarking time within a defined budget.

In this chapter, we aim to build a smart orchestrator for benchmarking containerized web-applications in a multi-cloud environment. SDBO is designed to solve the complexity of deploying benchmark applications in multi-cloud environment that have different programming APIs and numerous ways to interact. To achieve the *cost efficiency*, first, we develop an algorithm that maximizes the number of evaluation hosts based on users' budgets and pre-defined benchmarking time. Then, the *flexible execu-*

tion module is designed to capture the performance variation of cloud environment by partitioning the pre-defined benchmarking time into a set of slots. In summary, this chapter makes the following contributions:

- We developed a novel orchestrator, SDBO that automates the definition and execution of benchmarks for containerized web-applications. In particular, the orchestrator allows the user to choose the benchmark applications and hosts across different cloud providers.
- SDBO has a native feature of optimization that maximizes the utility of user's budget by maximizing the number of cloud providers and the hosts for benchmarking.
- Based on the optimized execution plans, we interact the plans with the *flexible execution* module to run the benchmarks in a set of time intervals thereby capturing the performance variation for a longer duration.

Outline. We illustrate the system design of SDBO in § 4.2, following which we show the execution workflow of our system in § 4.3. Next § 4.4 show the basic features that can be collected by our system and other advanced measurements that can be obtained based on the basic metrics and in § 4.5, we evaluate the experimental results. In § 4.6, we outline the related work and highlight the contributions of this chapter after which, we presents a discussion about the results obtained and the limitation of this chapter in § 4.7. Finally, we draw a conclusion in § 4.8.

4.2 System overview

This section discusses the architecture and system design details of SDBO.

4.2.1 SDBO *architecture*

Figure 5.3 illustrates the architecture of SDBO and the dependencies of each component. SDBO is implemented as a web-application that provides a *User Interface* for users to interact, explore and manage their benchmarking experiments. The *User*

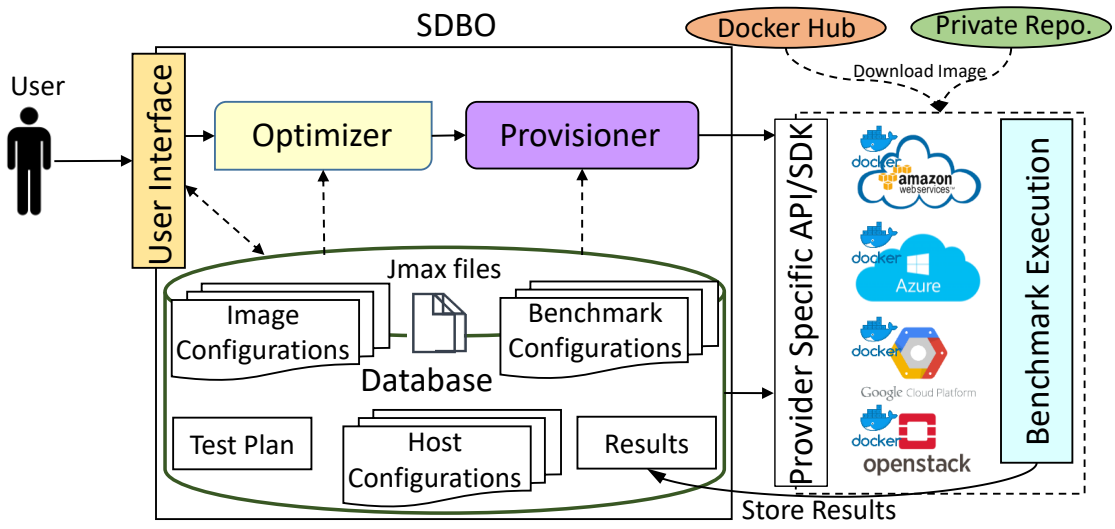


Figure 4.1: System architecture of SDBO

Interface allows the user to choose an existing benchmark application or customize a new application. Moreover, users can easily select the available hosts from different cloud providers, define the benchmarking time for each selected host, and specify the total budget for running the experiments. Next, this configuration information is stored in a relational Database.

The *Optimizer* is designed to create an optimized host list based on the information provided by the user. It retrieves the necessary information (host configurations, benchmark duration and budget) from the Database and applies a heuristic algorithm to generate an optimized host list for running the benchmarking experiments. More details about the *Optimizer* are given in § 4.2.2. The generated host list is automatically stored in the Database. Next, users can choose the *flexible execution* option for benchmark execution. If the user chooses to execute the benchmark experiments, the *Provisioner* will be triggered to provision the resources, deploy the benchmark applications and execute the applications based on the user entered information and optimized host list. The benchmark is executed for the specified interval of time and the completion is notified to the *Provisioner*. The results are stored in the Database in real-time for further evaluation and analysis. Finally, the user is notified after completion of the benchmarking experiment and following that cloud resources are released.

4.2.2 SDBO *design*

Optimizer and its Formal Model. SDBO benchmarks web-application in a multi-cloud environment. Let N represent the number of cloud providers $C_i|i \in \{1, N\}$ where each provider C_i has T type of hosts $v_{i,t}|t \in \{1, T\}$. In our model, we assume a one-to-one mapping between host and container. Consider $\mathcal{C}(v_{i,t})$ to be the unit cost of using $v_{i,t}$, $\tau_{i,t}$ is the time units for which $v_{i,t}$ is chosen to run and $\mathcal{B}$ is the user budget for the benchmark, finding an optimal set of hosts for the benchmark is modelled as a Binary Integer Linear Programming problem (BILP). The defined objective function is given in equation 4.1 subject to constraints as given in equation 4.1a – 4.1c.

$$\textbf{maximize:} \sum_{\mathbf{i=1}}^{\mathbf{N}} \sum_{\mathbf{t=1}}^{\mathbf{T}} \mathbf{x_{i,t}} + \lambda \sum_{\mathbf{i=1}}^{\mathbf{N}} (\sum_{\mathbf{t=1}}^{\mathbf{T}} \mathbf{x_{i,t}} - \mathbf{T}) \tag{4.1}$$

$$\sum_{i=1}^{N} \sum_{t=1}^{T} (\mathcal{C}(v_{i,t}) \times \tau_{i,t}) \leq \mathcal{B} \tag{4.1a}$$

$$\forall i \; \forall t \; \tau_{i,t} \geq 0 \tag{4.1b}$$

$$\forall i \; \sum_{t=1}^{T} x_{i,t} \geq 1, \; \forall t \; \sum_{i=1}^{N} x_{i,t} \geq 1 \tag{4.1c}$$

Where, $x_{i,t}|x_{i,t} \in \{0, 1\}$ is a binary variable which represents whether $v_{i,t}$ is selected or not. The first factor of the optimization problem is to comprehend the maximum selection of hosts and the second considers a penalizing factor to boost the spanning of the maximum number of cloud providers. λ is a tunable parameter which is incorporated to maintain a balance.

Constraint 4.1a states that the total cost of benchmarking different containers running inside the host must be less than the defined budget. Also, the cost is calculated only if $x_{i,t}$ is 1 with a positive execution time for host $v_{i,t}$ (constraint 4.1b). Finally, constraint 4.1c enforces the selection of at least one cloud provider and at least one host configuration.

We developed and implemented a heuristic algorithm for the *Optimizer* to solve the problem formalized above. The algorithm generates an optimized list of hosts while satisfying all the defined constraints. The details about how to create an optimized list of hosts are discussed in Algorithm 2. It first calculates the total cost, $\mathcal{C}^T(v_{i1,t1})$ for each selected host, $v_{i1,t1}$ (line 4). It then performs a local sorting (using merge sort)

Algorithm 1: *Optimizer*

Input: $V1$ - list of hosts $v_{i1,t1}$ selected by the user, $\tau_{i1,t1}$ - time for executing the benchmark on host $v_{i1,t1}$, $\mathcal{C}(v_{i1,t1})$ - unit cost of using host $v_{i1,t1}$, $\mathcal{B}$ - budget

Output: $V2$ - optimized list of hosts

1 $\forall i1\ fine_{i1} = 0$, $V2 = []$, $final\_cost = 0$
2 **for** *each selected provider* $i1$ **do**
3 **for** *each selected host type* $t1$ **do**
4 $\mathcal{C}^T(v_{i1,t1}) = \mathcal{C}(v_{i1,t1}) \times \tau_{i1,t1}$
5 **end**
6 Sort the host $v_{i1,t1}$ in ascending order of total cost $\mathcal{C}^T(v_{i1,t1})$ using Merge sort and store in a list, $List_{i1}$
7 **end**
8 **while** $(final\_cost \leq \mathcal{B})$ **do**
9 Search the first element of all list and find the host $v_{i1',t1'}$ with smallest cost
10 **if** $(fine_{i1'} > 100\ \ \&\ \ \forall i1\ (!empty(List_{i1})))$ **then**
11 Skip $List_{i1}$ from current calculation
12 continue
13 **else if** $(fine_{i1'} \leq 100\ \ \&\ \ \forall i1\ (!empty(List_{i1})))$ **then**
14 Add $v_{i1',t1'}$ to $V2$
15 Delete $v_{i1',t1'}$ from the list $List_{i1'}$
16 $final\_cost = final\_cost + \mathcal{C}^T(v_{i1',t1'})$
17 $fine_{i1'} = fine_{i1'} \times 10$
18 **for** $(\forall\, i1\ <>\ i1')$ **do**
19 **if** $(fine_{i1} \geq 10)$ **then**
20 $fine_{i1} = fine_{i1}/10$
21 **end**
22 **end**
23 **else**
24 Add $v_{i1',t1'}$ to $V2$
25 Delete $v_{i1',t1'}$ from the list $List_{i1'}$
26 $final\_cost = final\_cost + \mathcal{C}^T(v_{i1',t1'})$
27 **end**
28 **end**

for each selected cloud provider, $i1$ according to the increasing host cost and stores it in a temporary list, $List_{i1}$ (line 6). Following that it selects a host with minimum cost globally and adds to the final host list, $V2$ (line 14, line 24) until the final_cost is less than budget, $\mathcal{B}$ (line 8). To maintain fairness and diversity among different cloud providers, there is a provision to add a penalty if the cloud has been selected (line 17). A host is selected only if the penalty imposed on that cloud is less than a defined value (100 for our case) or if there are no other providers left for selection (line 10).

Optimizer complexity analysis. The problem of orchestrating the optimized benchmark for a set of cloud service providers is a complex problem and involves many steps as shown in the above sub-section. The complexity of most of the steps is constant except the creation of an optimized list. The complexity of this step depends on the number of cloud service providers and the number of hosts each cloud service providers have as discussed in Algorithm 2. The time complexity is given as:

$$O((N \times max(T_i)) + max(T_i \times logT_i) + (N \times (N-1) \times max(t_i)))$$

where, $(N \times max(T_i))$ is the complexity to compute the total cost for each selected host, $max(T_i \times logT_i)$ is time to sort the host for cloud provider with maximum number of hosts and $(N \times (N-1) \times max(t_i)$ is the complexity for searching the suitable host based on the cost and imposed fine.

After reducing, the overall complexity is $O(max(T_i \times logT_i) + (N \times (N-1) \times max(t_i)))$.

Provisioner. Once the *Optimizer* generates a benchmark plan, the users can decide whether they want to submit the plan for execution via the user friendly web interface. If the user agrees to perform the experiment, the functions implemented in *Provisioner* will be triggered. First, the *Provisioner* will check the connection and the requirement of the resources on different clouds. Next, it uses a background process application, Hangfire to create and launch the hosts on the selected cloud providers.

Flexible execution. SDBO offers two types of execution strategy (a) *solitary execution* and (b) *manifold execution*. *Solitary execution* is the basic strategy where users can set a particular time interval for evaluating the benchmark on the desired host configuration. The performance evaluation in this case is limited as it executes only for the particular time interval. We know that the host's QoS performance is highly dependent on the system parameters, e.g. current workload, network state, etc. which may vary with time [58]. This variation is especially significant for the web-applications due to the continuous execution and the resource preemption in the virtualized environment.

To capture this variation, we propose *manifold execution* strategy that executes the benchmark application in the same host but in multiple time intervals. The user is

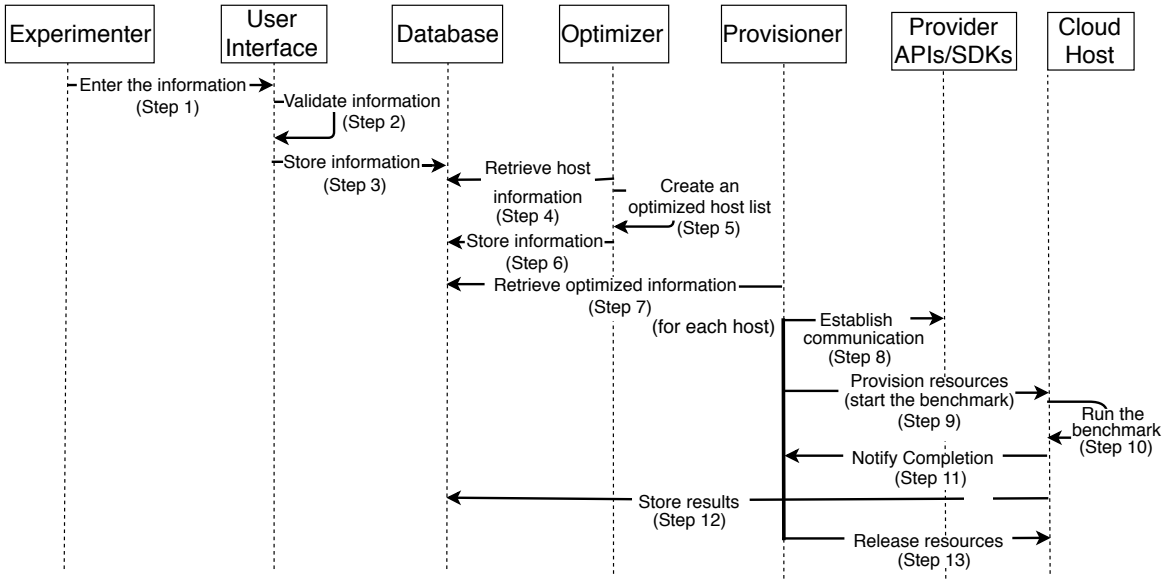


Figure 4.2: SDBO execution workflow

asked to define the number of iterations along with other parameters for the optimizer. The optimizer then generates an optimized list of hosts which is associated with the execution timestamps. As a result, the *Provisioner* can schedule the deployment and execution based on the host configurations and its associated execution timestamps.

4.3 Execution workflow

SDBO is implemented using *Microsoft ASP.NET Core 2.1 C#* and Javascript. The current version of code along with samples and installation details are available as an open-source project on github [24].

The following describes the main steps of the execution workflow of SDBO, which is also shown in Figure 4.2.

Define the benchmark experiment. To start the benchmark, the experimenter needs to enter different information as depicted by Step 1 in Figure 4.2. The details of the experiment definition are given in Figure 4.3. To define a new experiment, first, we need a benchmark application that can be selected from the available ones or created from the given template. Each application consists of an application image and a load generator image. The images are referred from Docker hub or another repository. Thus, users can refer their own images which are available on any pub-

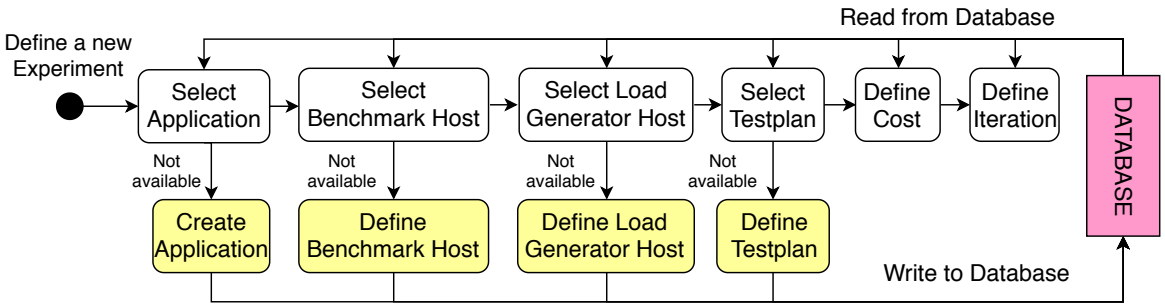


Figure 4.3: Benchmark experiment definition

lic repository. The second component is the host configuration which is required to execute the benchmark. The *User Interface* allows us to choose a name, description, host type (application/load generator) and cloud-specific (e.g. credentials, template) information. To guarantee secure communication, TLS and HTTP authentication is also provided.

The third component is the test plan that includes all the parameters for the load generator to create a continuous load for the benchmark application. SDBO allows us to define the workload using a test plan where we can provide the required file (e.g. JMX, AJAX) for continuous workload generation. New test plans can be easily added by filling the template provided by the framework. Moreover, we also need to specify the maximum budget, execution time for benchmark and flexible execution interval.

Validate and store the entered information. The user input may not be valid all the time. To avoid any error caused by this, SDBO provides a validation mechanism that uses a jQuery validation plugin for the user's input validation (Step 2 in Figure 4.2). The user is notified to enter new information if the data is invalid. After the data validation, all the inputs are stored in the Database (Step 3).

Create an optimized list. The *Optimizer* retrieves the input host information from the Database (Step 4) to generate an optimized host list for the execution (Step 5). At the same time, the optimized host list is stored in the Database (Step 6).

Benchmark execution. To start the benchmark execution, first *Provisioner* retrieves the optimized host list from the Database (Step 7). Next, a background process is started which handles a separate thread for each selected experiment. For each thread, the communication with the cloud host is established using the provider-

specific APIs/SDKs (Step 8) and then resources are provisioned for benchmarking (Step 9). After completing the execution, *Provisioner* is notified (Step 11) and the result is stored in the Database (Step 12) for visualization and further analysis. The experiment result provides different performance metrics including benchmark statistics (e.g. started at, benchmark length), benchmark container metrics (e.g. CPU percentage mean, CPU percentage range, memory percentage mean, network input total, network output total), benchmark web-server metrics (e.g. throughput, average response time, number of requests) and Apdex rating (Apdex count, Apdex rating, Apdex satisfied count,). Finally, the resources are released (Step 13). The whole process is automatically repeated depending on the number of iterations specified for the benchmark execution.

4.4 Metrics profiling

SDBO can support benchmarking for different types of web-applications including e-commerce, social media and banking system. It does not only capture the basic web-application features, e.g. response time, throughput illustrated in §4.4.1, but also supports more complex and advanced metrics (see §4.4.2).

4.4.1 Basic metrics

Response Time (ΔT). Response time is the total time taken by the web-application to process a request and generate its response. It is a basic metric to evaluate the performance of any web-application. Normally, response time depends on many factors varying from the host infrastructure, scheduling policy and the current load on the system to the host capability and network capacity to handle a user's request. Average response time $\mu(T)$ and standard deviation of the response time $\sigma(T)$ is used frequently to measure the performance of the web-applications. Lower response time represents better performance.

Throughput (TP). Throughput represents the host performance in terms of the number of requests that can be handled per unit time. Consider that there are total

Table 4.1: Apdex acceptable zones

Level	Satisfied	Tolerated	Frustated
Multiplier	<= T	>T or <= 4T	>4T

N number of sample requests which are successfully executed in Δt time interval where $\Delta t = (Start\ time - Finish\ time)$, throughput is calculated as $TP = N/\Delta t$.

CPU Usage (CPU). It gives the percentage of CPU used by the container while executing the process. We obtain this information from docker stats APIs [13] embedded with our orchestrator.

Memory Usage ($Memory$). Docker stats APIs also allow us to obtain the percentage of memory used by the monitored container.

Network Throughput (Net). This metric indicates how much data can be transferred from a client to the target container in a unit time interval and is represented in Mega bits per seconds (Mbps).

Block I/O (I/O). Block input/output refers to the amount of data written to or read from the block storage devices in a unit time interval and is also represented in Mbps. We collect Net and I/O also from Docker stats APIs.

4.4.2 Advanced metrics

Based on the collected basic metrics which are stored in our database, users can perform more complex queries to profile the complex systems.

Apdex Score. Apdex (Application Performance Index) [4] is considered as an open standard developed to standardize the methods for benchmarking, tracking and reporting the application performance. It utilizes the Response Time (ΔT) to check the user satisfaction level for an application's performance. Based on a defined threshold for the response time **T**, Apdex defines three acceptable zones namely Satisfied, Tolerated or Frustrated (Table 4.1) for the application performance. The threshold can be set based on the application requirements e.g. threshold, **T** is set to 0.5 sec in [51].

An Apdex score is calculated using the number of requests satisfied and tolerated out of the total requests received. The contribution of satisfied and tolerated requests for

the user satisfaction level is 100% and 50% respectively. Let NR, SR and TR be the total number, satisfied number and tolerated number of requests respectively, an Apdex score is calculated as given in equation 6.11. The value of an Apdex score lies between 0 and 1 with higher values representing better satisfaction levels.

$$Apdex\ Score = (SR + TR/2)/NR \tag{4.2}$$

Host Stability. The stability of host machine is the metric to measure the consistency of the system performance. It is defined as the inverse of variability experienced by different basic metrics. Given the average μ_i and standard deviation σ_i for ith basic system metric ($i \in M$) executed for time T, variability is calculated as given in equation 4.3.

$$Variability = 1/T \sum_{t=0}^{T} \sum_{i=0}^{M} (\sigma_{i,t}/\mu_{i,t}) \tag{4.3}$$

Thereby, host stability is calculated as $Host\ Stability = 1/Variability$. Hosts with smaller stability values show that the performance is inconsistent and is not suggested for execution.

Host Suitability. The host suitability metric represents the worthiness of a host in terms of performance and cost. It is computed using equation 4.4.

$$Host\ Suitability = TP/Cost \tag{4.4}$$

where, TP is the throughput, $Cost$ is the per unit execution cost for that particular host. The higher the value of host suitability the better is the host.

4.5 Evaluation

To illustrate the effectiveness of SDBO, we performed a case study using a simple web-application benchmark. The details are presented in the section below.

Table 4.2: Experiment host configuration

CSP	Host Type	CPU cores	Memory	Disk	Price/hr($)
AWS	t2.nano	1	0.5	EBS	0.0066
	t2.micro	1	1	EBS	0.0132
	t2.small	1	2	EBS	0.026
	t2.medium	2	4	EBS	0.052
	t2.large	2	8	EBS	0.1056
	m4.large	2	8	EBS	0.116
	t2.xlarge	4	16	EBS	0.2112
	c4.xlarge	4	7.5	EBS	0.237
	m4.2xlarge	8	32	EBS	0.464
	c4.2xlarge	8	15	EBS	0.476
Azure	Standard_B1s	1	1	2	0.0118
	Standard_B1ms	1	2	2	0.0236
	Standard_B2s	2	4	4	0.0472
	Standard_F2	2	4	8	0.119
	Standard_B2ms	2	8	4	0.0944
	Standard_D2_v3	2	8	4	0.116
	Standard_B4ms	4	16	8	0.189
	Standard_A4_v2	4	8	8	0.222
	Standard_B8ms	8	32	16	0.378
	Standard_D8_v3	8	32	16	0.464

4.5.1 Experiment setup

SDBO is tested both in simulation and on a real testbed. The simulation is to test the scalability of our proposed optimization algorithm, and the real testbed is to evaluate the system performance. The experiment setup is detailed as follows.

Scalability evaluation. Our algorithm is tested on a Lenovo PC with Intel(R) Core(TM) i5-6200U CPU @2.3GHz - 2.4GHz with 16 GB memory and 512 GB SSD. We collected 20 host configurations from AWS and Azure as the input dataset shown in Table 4.2.

Benchmark application and its deployment. SDBO is published on Google Cloud App Engine (*B2 instance class*) London (europe-west2). Therefore, the users can access the system from any place and run their benchmarking applications via the user interface. PostgreSQL Databaseis associated with the SDBO, and stores different configuration of hosts and benchmark images for running the experiments. The Database is also deployed on a Google cloud *n1-standard-2* instance with 2 vCPUs,

7.50 GB memory, 128 GB disk. All these components are running independently following the microservice architecture.

We utilized a popular benchmark application, TPC-W[27] with SimplCommerce that emulate the activities of a sample e-commerce web-application. This application is containerized and used to benchmark various type of hosts on AWS and Azure as shown in Table 4.2. The load on the web-application is created by Apache JMeter[17] according to the test plans defined by the user. To emulate real traffic, JMeter is not configured on the same cloud where the benchmark applications are running. The containerized load generator is deployed on Digital Ocean cloud and the host is a *Standard droplet* machine with 6 vCPUs, 16 GB memory and 320 GB SSD disk.

4.5.2 Cost optimization

In this section, we evaluate the performance of our optimizer which aims to maximize the number of hosts within the constraint of users' budgets and pre-defined benchmarking time.

To highlight the advantages of the optimizer, we considered 20 host configurations from AWS and Azure (see Table 4.2). Moreover, we assume that the user would like to run their benchmarking experiment for 3.5 hours, with four different budgets $ 0.5, $ 1.0, $ 1.5 and $ 2.0. We compared the performance of our optimized selection method (*Opt*) with the random selection method (*Rand*).

Figure 4.4 demonstrates that the optimized option selects the higher number of host in all the cases, compared to the random selection method. The reason is because the optimizer always selects the host with a lower price first and then it moves to a higher cost host. This is based on the logic that a user wants to deploy their web-application on the cheapest hosts that can meet their QoS requirements. Our algorithm design fits to this logic very much. However, the random selection method selects any host which may not be cost optimized. In addition to this, our method can provide a more stable number of hosts as shown in Figure 4.4, where the *Opt* has a much smaller variance than *Rand*.

We also evaluate the scalability of our algorithm by simulating a scenario with a

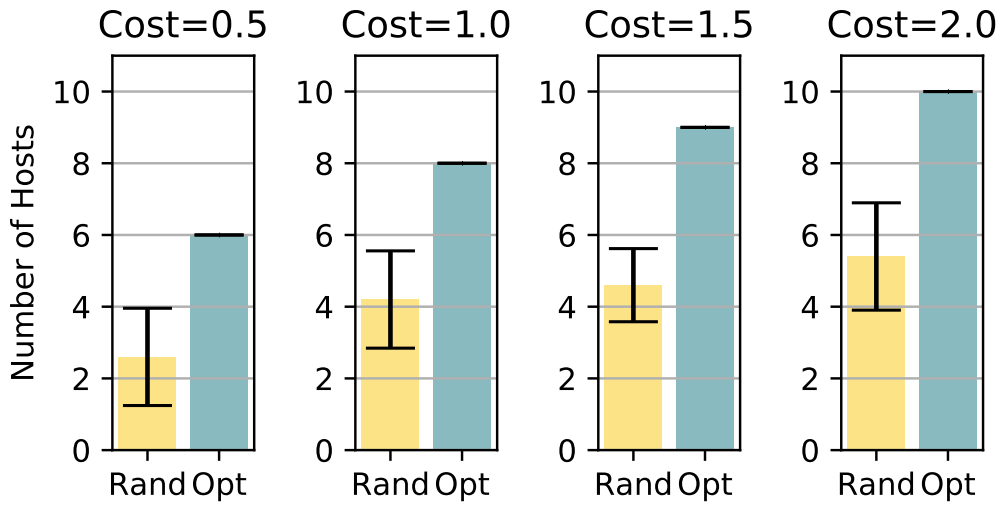


Figure 4.4: Comparing the optimized result with random selected result

varying number of cloud providers with each provider having 50 different host configurations available. Figure 4.5 shows the execution time of different cases with an increasing number of cloud providers varying from 1 to 30. The result shows that the execution time only increases linearly as the number of cloud providers increases. Moreover, the maximal execution time is 5.7 milliseconds for 30 cloud providers, which is comparatively very small as compared to the deployment time.

4.5.3 Basic metrics profiling

In this subsection, we present the benchmark results of an optimized test case. We select a subset of the hosts from Table 4.2 as the input to our optimizer that then generates 6 hosts (highlighted with the gray color in Table 4.2) for benchmarking experiments.

To obtain the throughput of each deployed benchmark application, we emulated the *bursty request*, i.e. we send the maximal number of requests to the web-applications simultaneously without causing any response error. In other words, the web-applications are fully saturated. Table 4.3 shows the maximal number of requests for each selected host. Figure 4.6 illustrates the value of basic metrics (as specified in §4.4.1) of the selected hosts, collected from the experiments.

CPU Usage. Figure 4.6a shows the CPU usage of each selected host. The result

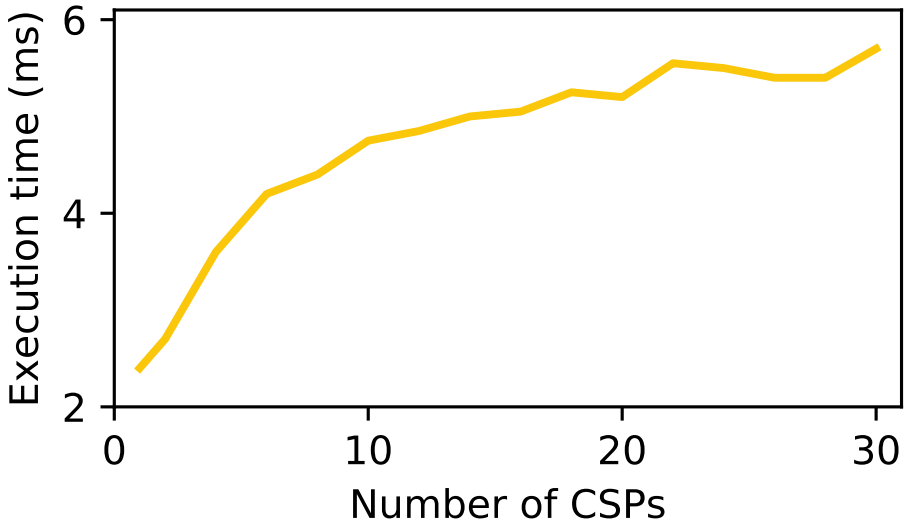


Figure 4.5: Schematic diagram showing the execution time complexity of the Optimizer

Table 4.3: Number of requests to saturate the host

CSP	Seq	Host	No. of requests (to saturate)
AWS	A	t2.small	300
	B	t2.medium	600
	C	t2.xlarge	1500
Azure	D	Standard_B1ms	300
	E	Standard_B2s	600
	F	Standard_B4ms	1500

clearly shows that CPU usage decreases as we increase the size of the host. Also, the more powerful hosts have less variation for CPU usage. For example, the variance of the CPU usage for the big size hosts C and F is only 7% and 5% and that for small host A and D reaches 24% and 12% respectively. Except for small-sized hosts, the performance of AWS to Azure is almost comparable. For the small size host, there is a huge performance difference (52.5% degradation) as Azure has less CPU usage compared to AWS for processing the same number of requests.

Memory Usage. The memory usage (Figure 4.6b) also shows a similar trend except for the variation which is much less (highest is 0.76 for host E) as compared to CPU usage. The highest memory usage is noticed for host D followed by host A with 16.1% and 15.3% respectively. Note that the memory usage is varying only in the initial phase, after that the usage is almost constant. It is caused by the property of the

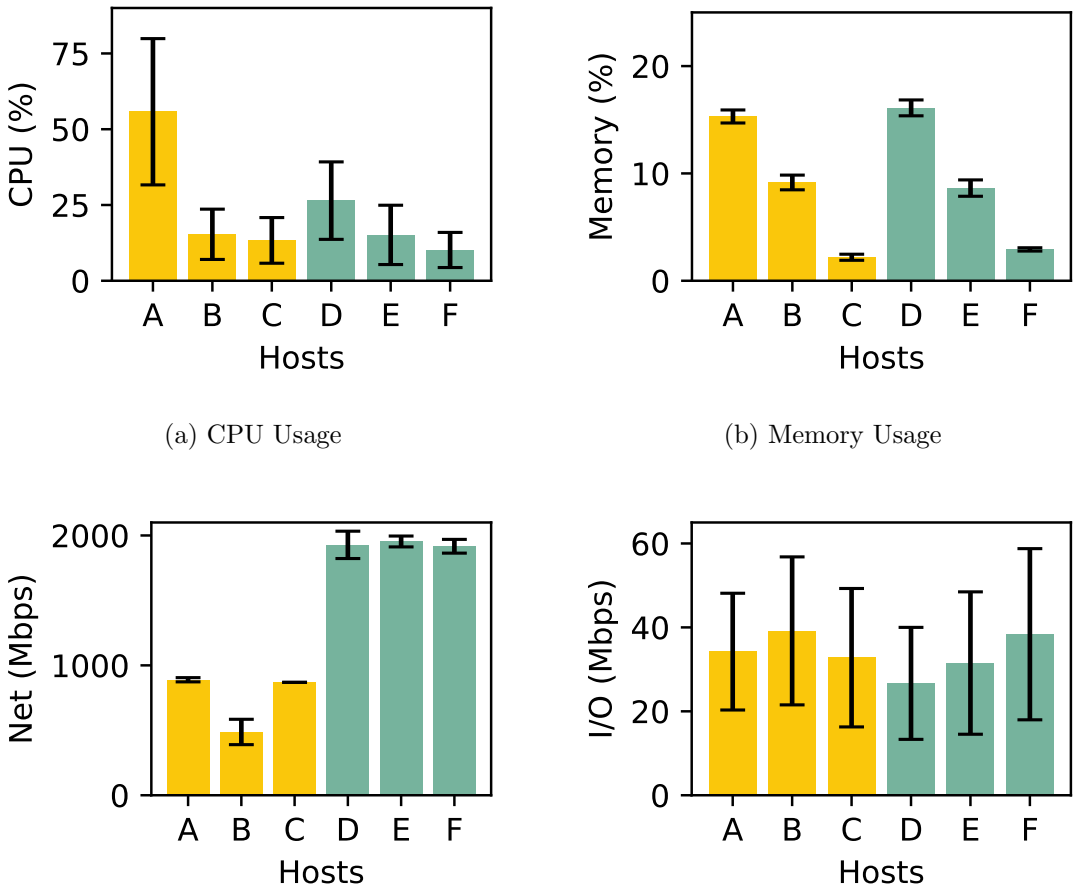
Docker container, the memory once allocated is not released back until the container is terminated or restarted.

Network Throughput. The result in Figure 4.6c shows that the hosts from Azure have about twice the network throughput, compared to the hosts from AWS. The hosts from the same cloud provider have the same network throughput except for host B from AWS where the throughput is much less (487.2 Mbps) than others A (889.2 Mbps) and C (869.2 Mbps).

I/O Throughput. As compared to the above three basic metrics, block I/O shows different trends (see Figure 4.6d). The I/O throughput of AWS hosts is very random which is because of the selected hosts that offer EBS support. Thus, the throughput is also affected by the time elapsed between the I/O requests and EBS server [16]. Moreover, our benchmark application is block I/O intensive, so the collected statistics are not the maximal I/O throughput of each host. This is demonstrated very well in Azure hosts (see D, E, F in Figure 4.6d) that the I/O throughput increases with the increase in the number of requests.

System Throughput. Figure 4.7a illustrates the throughput of different hosts. It is clearly depicted from the figure that the throughput increases linearly with the capacity of the host and AWS hosts have comparatively better throughput than Azure for similar sized machine except for host B. The reason for lower throughput for host B is the degraded network throughput as depicted in Figure 4.6c. Therefore, the network becomes the bottleneck in this case. The highest throughput achieved by AWS host is 8.93 requests per second (host C).

Response Time. Figure 4.7b shows the results of the response time. The results show that the response time is significantly affected by the network throughput and the number of requests. Figure 4.6c show that host B has the worst network throughput which causes the significantly higher response time as shown in Figure 4.7b, i.e. 252.2 seconds. If the network throughput is constant, the response time increases with the increase in the number of requests. Note that we try to saturate the web-application until it reaches the maximal number of requests that it can handle without causing errors. Thus, a large number of requests are queuing and waiting for being processed,



(a) CPU Usage

(b) Memory Usage

(c) Network

(d) Block I/O

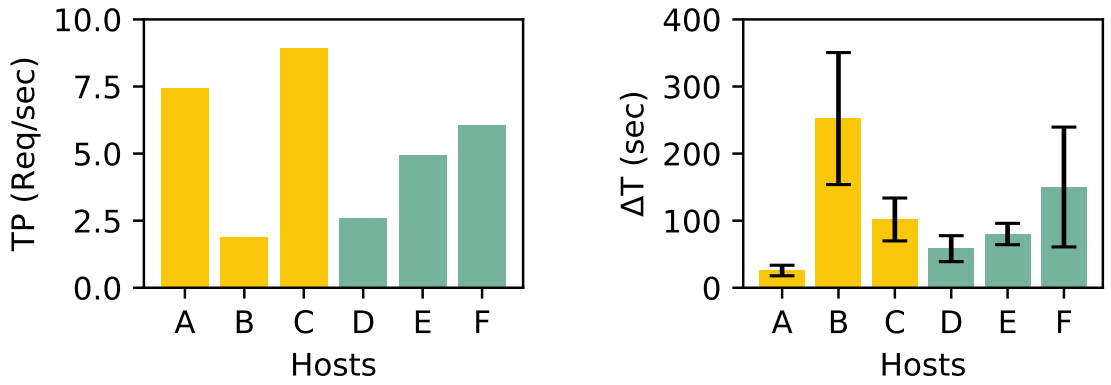
Figure 4.6: Basic container system metrics while specifying the workload to 300 requests per second with ramp up period as 0 seconds. CPU and memory usage are given in percentage while network and block I/O throughput are in Megabits per second (Mbps). Black bar on top represents the standard deviation.

and this is the main reason that causes a high response time for many requests.

4.5.4 Advanced metrics profiling

In this subsection, we compute different advanced metrics based on the collected basic metrics that can help in selecting the cloud provider and the hosts for the actual deployment.

Apdex Score. We calculate the Apdex score for the same case as discussed in §4.5.3. Since we are considering the case of a saturated system where the response time is high, we set the threshold for the response time to 50 second. The higher the Apdex score,



(a) Throughput

(b) Response Time

Figure 4.7: System throughput and response time. Both these metrics are calculated by stabilizing the server with enough requests. Black bars in (b) show the standard deviation.

Table 4.4: Advanced metrics profile

CSP	Hosts	Apdex Score	Host Stability	Host Suitability
AWS	A	1	1.115	286.399
	B	0.15	0.790	36.313
	C	0.5	0.834	42.311
Azure	D	0.8	0.921	109.979
	E	0.5	0.772	104.470
	F	0.4	0.846	32.042

the better users' satisfaction. Table 4.4 shows the Apdex score for all selected hosts. The result clearly shows that smaller hosts have better Apdex scores as compared to larger hosts. We have explained why the smaller hosts have lower response time (see §4.5.3 **Response Time**). The lowest score of 0.15 is noticed for host B due to its bad network throughput (see Figure 4.6c).

Host Stability. A host with a higher stability value is considered best as it signifies less performance variation with the elapsed time. The result in Table 4.4 shows that the stability of small and large host instances are higher. The highest value is for host A with a stability index of 1.115 followed by host D with the index of 0.921. The worst stability index is for host E with a value of only 0.772.

Host Suitability. Host suitability is computed as discussed in equation 6.12. A

host with a higher suitability value is considered to be better as it provides better throughput to cost ratio. The suitability index for the selected hosts show a downward trend with increasing size. For both AWS and Azure, smaller machines have better suitability values as shown in Table 4.4.

4.5.5 Flexible execution

Continuous execution of a benchmark for a longer duration is the best way to capture the performance variation. However, the benchmarking cost in this case is very high. To capture the performance variation of the changing environment in a defined budget, SDBO offers the *flexible execution* module.

We do not have access to the cloud hypervisor, therefore, we are not able to emulate the resource changing or preemption of the hosts. As an alternative, we emulate the performance variation of our web-application by changing the number of requests in a period of time. If our tool can observe the performance variation with the changing number of requests, it can also capture the variations that may be caused by other reasons.

To this end, we define 9 test plans, each plan is defined with a timestamp and the number of requests need to be sent as shown in Figure 4.8 depicted by *Cont.* For example, the first plan is to send 15 requests starting at 00:00 minute timestamp. Following that, the second plan sends 50 requests starting at 30:00 minute timestamp. We keep the web-application (benchmark application) running for 360 minutes to cover all timestamps from the test plans. For the *Opt* case, we randomly selected 5 test plans and sort them based on the timestamp as shown in Figure 4.8 and Table 4.5. The web-application (benchmark application) is executed for 10 minutes, if and only if the timestamp is reached. The above described two experiments were executed simultaneously with the same host configuration (AWS $t2.medium$).

Table 4.5 shows response time and throughput collected from both scenarios. The result clearly shows that SDBO can capture the same performance with a maximal variation of 15% in Case I for response time and 5.9% in Case III for throughput. The cost for the optimized method is much less than the continuous way of deployment as the total time of deployment for *Opt* is only 50 minutes as compared to 360 for *Cont.*

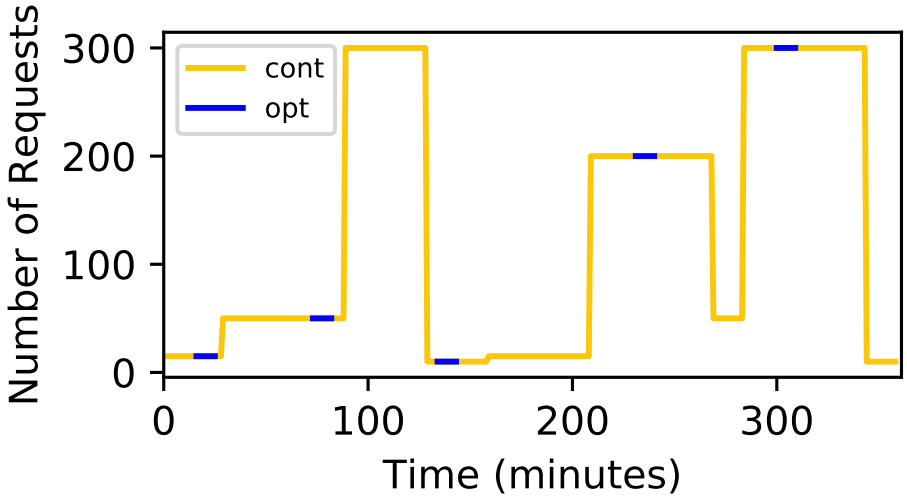


Figure 4.8: Workload pattern for continuous and optimized execution

Table 4.5: Comparison of Optimized; *Opt* and Continuous; *Cont* method for Response Time and Throughput. Values in [] represent standard deviation.

Case	No. of Req	Response Time (sec)		Throughput (Req/sec)	
		Cont.	Opt.	Cont.	Opt.
Case I	10	0.684 [±0.29]	0.789 [±0.36]	6.71	6.54
Case II	15	2.114 [±0.10]	2.122 [±0.15]	6.23	6.17
Case III	50	4.105 [±0.13]	4.441 [±0.18]	10.49	9.87
Case IV	200	21.381 [±1.34]	22.482 [±2.04]	6.90	6.59
Case V	300	23.463 [±6.13]	24.649 [±4.18]	7.56	7.25

4.6 Related work

The web-application benchmarks need to be deployed on various host configurations in the multi-cloud environment. Orchestrating the systematic deployment consists of the following steps [149]: (i) defining the benchmark with their attributes and relationships, (ii) defining the host machine configuration (e.g. CPU cores, location), (iii) instantiating the cloud host complying the application requirements, (iv) monitoring the resources to ensure the QoS and SLA parameters, and (v) controlling the overall processes. Performing all these steps manually is tedious, error-prone and requires a lot of time and diverse knowledge of architecture and accessing mechanism of all these environments. There are different frameworks available that automate/semi-automate the orchestration steps. [90, 130] evaluated the performance of containers

for scientific applications where a few of them [153, 159] evaluate for big data applications. However, most of these works are intended for a single cloud environment, without considering the complexity of interacting with various APIs/SDKs provided by different cloud providers.

There are few existing frameworks that handle the orchestration of benchmarks in a multi-cloud environment. CloudBench [134] and Smart CloudBench [50] automates the benchmark execution in multi-cloud environment. However, it is not easy to define the benchmarks using these frameworks. Also, they are not specific to the containerized environment. Additionally, Varghese et al. proposed a framework called DocLite [142] to evaluate the performance of VMs using containerized microbenchmarks. Microbenchmarks are executed on different VMs and the ranking is evaluated by using the set of weights provided by the user for different system parameters. This framework is specific for scientific application and may not be applicable for web-application. Our proposed SDBO orchestrates the benchmark for web-application while allowing users to define and deploy the benchmark in a very interactive and user-friendly way.

Additionally, there are some commercial tools, e.g. CloudHarmony [9], Cloud Spectator [8] available that perform the benchmark for users but are not specific for a particular application. Also, they do not provide all the required metrics specific to that particular application for making proper decisions before final resource selection and provisioning. The limitations of the existing work are briefly summarized as follows.

Constraints. The cloud providers offer shared computing resources to their customers, which makes the cloud environments dynamic and the SLA very hard to guarantee [129]. Moreover, the web-application is very sensitive to the dynamically changing environment that directly affects user's satisfaction. Capturing or monitoring the changing behavior of the cloud environments requires the users to run their benchmark applications over a considerable time, which is very costly. Existing benchmark frameworks are not able to solve the trade-off between the limited budgets and the long-time benchmarking experiments.

Additionally, the variety of cloud providers offer a massive configuration choice of hosts. For instance, Amazon EC2 provides 43 types of hosts for their customers

excluding self customized hosts. It is not possible to run the benchmark applications overall available resources. The state-of-the-art systems do not consider this case that provides an optimized recommendation to help users in selecting the hosts from the massive number of available hosts spanned across multiple cloud providers.

To overcome the limitations discussed above, our proposed SDBO framework contributes towards three main features. First, it automatically orchestrates *any* containerized benchmark application across multi-cloud environments based on the users' specifications. This will remove the huge burden from the user side, i.e. the users do not need to manually deploy the benchmark applications and define the benchmark plans over various cloud datacenters. Second, it captures the dynamicity and the uncertainty of the cloud environment while ensuring the cost does not exceed the users' budget. This dynamicity and uncertainty may affect the performance of web-application, therefore reducing users' satisfaction [71]. Finally, we should be able to find an optimal set of hosts for benchmarking in a fixed budget that covers the diversity, in terms of host configurations, both within a cloud provider and across the cloud providers. This feature improves the chance for a user to find the best mix of hosts for deploying their web-applications. In other words, with our design, the users have more chance to find the cheaper hosts which meet the performance requirements of their web-applications.

4.7 Discussion

The experiment results show that the *Optimizer* of SDBO selects a higher number of hots for benchmarking the containerized microservices as compared to the *Random* selection approach. The execution time of *Optimizer* is also small as compared to the deployment time. The test case experiments on real-cloud validate the orchestration process. Results also show that SDBO can capture the performance variation of web-application benchmark execution for a longer duration in a limited budget. The *flexible execution* model can achieve the same response time and throughput as *continuous execution* with a maximum 15 % and 5.9 % respectively.

Limitations. Although, SDBO can orchestrate the benchmarking of any type of web-application, it is not able to orchestrate geo-distributed web-application with numerous components located in different geographic locations. Also, the current chapter performed only a small test experiment with e-commerce benchmark application. This can be extended for a large number of host configurations and other benchmark applications.

4.8 Conclusion

To facilitate web-application benchmarking in multiple cloud with cost efficiency and flexibility, we proposed SDBO which is the first cost-efficient web-application benchmarking orchestrator. In this chapter, we have presented the architecture and implementation of SDBO along with some case study that shows the effectiveness of SDBO compared to the existing frameworks. SDBO provides the smart user interface that expedites the handling of the benchmark even for a non-expert user. Also, the cost optimization offered by the orchestrator helps the user to select a variety of hosts while *flexible execution* captures the long time performance variation in a limited budget. Although this chapter discusses SDBO from the perspective of web-applications, it can be easily used for HPC applications.

5

A user-centric cost-efficient geo-distributed web-applications deployment via automatic benchmarking

Contents

Summary

In the previous chapter, we have proposed *SDBO* orchestrator for benchmarking simple web-application in a multi-cloud environment. However, SDBO is not able to handle the deployment of a complex geo-distributed web-application (GWA) which is executed across geographically separated cloud-enabled data centers to provision internationalized user requirements (e.g. currency, taxation). In this chapter, we propose GeoDeploy; an orchestrator suitable for deployment of GWA using benchmarking. We analyze the performance of GeoDeploy using both *numerical analysis* and *real-cloud experiment* and demonstrate that our approach outperforms other baseline methods.

5.1 Introduction

Modern web-applications (WA) like *Google*, *Amazon* and *Alibaba* pursue a business model where there is a desire to offer their services to users distributed across the globe reachable by the Internet. This global service delivery has *two* key constraints (or requirements): 1) End-user of these WA spread across all over the world, and hence ensuring quality of service (QoS) (e.g. low latency) for majority users is not trivial; 2) Privacy and data sovereignty laws in some countries restrict the raw data transmission across some regions [146, 157]. GDPR (General Data Protection Regulation of the EU) restricts a transfer of the personal data between an EEA (European Economic Area) country or non-EEA country [33].

To overcome these constraints, these services are usually geographically distributed in a multi-cloud environment in a federated manner [91, 118, 148, 151]. An application supported in this way is commonly referred to as a Geo-Distributed Web-Applications (GWA).

Their deployment must be achieved across varied datacenter hardware/software supporting services. An added issue in their deployment are the numerous laws and regulations often declared at the commercial region boundaries or country borders. Therefore, determining the optimal approach to deploy all the components that make

up a GWA appropriate (both legally while considering QoS issues) in a geographic sense is a non-trivial research problem.

Content-distribution networks (CDN) [34, 44, 73] share the similar purpose, serving users worldwide. CDN aims to deliver a large volumes of data such as video to users, but this chapter focuses on ensuring the QoS (e.g. low latency) of a GWA. The CDN has to handle the long-running requests and require high-bandwidths to meet QoS requirements. Moreover, CDN does not have the restrictions of privacy and data sovereignty laws. The GWA is designed to quickly and simultaneously respond many users' requests. As a result, the rich solutions developed for CDN are not applicable for GWA.

An approach is to source a set of candidate deployment solutions and evaluate these solutions over real environment. Various studies from academia [93, 95, 141] and industry [9, 10] evaluate the performance of cloud hosting configurations. These works focus on computation intensive applications which tend not to be user facing in the context of time sensitivity and interactivity, and reflect the evaluation (or benchmarking) for elasticity and scalable thresholds associated to Big Data style applications. [25, 116, 135, 138] are developed for WA benchmarking, but can only be deployed on a cloud datacenter manually. A number of works [78, 128] also propose to automatically orchestrate the WA on a cloud datacenter, benchmarking the performance of the deployed WA. These systems do not consider a WA that has multiple components (or replicas) and the components have some dependencies. Moreover, they fails to find an optimized deployment solution for a given WA or GWA. Our proposed method can be further adapted to these systems, adding an important feature to evaluate the performance of a GWA.

All existing studies able to benchmark the deployment of WA, fails to apply for GWA that have following challenges. 1) *satisfying users distributed geographically:* a suitable deployment solution of GWA needs to meet the requirements of users spread all over the world. The requirements mainly consist of fast response time (e.g. in milliseconds), privacy protection and so on. 2) *diversity of the cloud resources:* the cloud providers (CPs) offer massive configurations for their hosts (e.g. virtual machines, containers), including CPU, memory, locations and so on. This diversity outlines a

combinatorial challenge for optimal deployment of a GWA. 3) *limited budget and time for benchmarking:* searching and benchmarking the available solutions for deploying a GWA are vast and cannot be evaluated within limited budget (or time).

To the best of our knowledge, this is the first work that considers optimizing the deployment of GWA in multi-cloud environments through automatic benchmarking. In this chapter, we propose, develop and validate SDBO that automatically 1) generate a set of candidate deployment solutions based on the requirements of the GWA, 2) run benchmarks of these deployment solutions over real cloud environments and 3) recommend the most suitable deployment solution based on the GWA requirements. Although there are various GWA QoS requirements to satisfy, this work considers only response time metrics to select a deployment solution.

SDBO incorporates a novel variant of Particle Swarm Optimization (PSO); Adaptive PSO (APSO) which generates a set of optimal candidate deployment solutions considering geo-location of the cloud hosts and users of the GWA, budget limitation, configurations of cloud hosts such as CPU and memory and costs. The choice of APSO is based on the large solution search space with high dimensionality which converges without being trapped in the local minima.

We summarise the main contribution of our work as follows:

- We propose SDBO, a novel user-centric cost-efficient GWA deployment orchestrator that automatically deploys and benchmarks GWA's in a multi-cloud environment.
- A novel algorithms Adaptive Particle Swarm Optimization (APSO) algorithm that ensures optimal selection of candidate deployment solution.
- A comprehensive experimental evaluation using real-world multi-cloud environment to validate the performance of the proposed APSO algorithm in comparison to the baseline methods.

Outline. In § 5.2, we outline the background and highlight the research problems addressed by this chapter. Then, we illustrate the system design of GeoDeploy in

§ 5.3 followed by system design §5.4. Next, we present our proposed Adaptive PSO algorithm (see § 5.5) and the deployment optimization (see § 5.6). In § 5.7, we evaluate the experimental results. We give a discussion highlighting our work and limitations in §5.9. Before drawing a conclusion in § 5.10, the related work is presented in §5.8.

5.2 Background and motivation

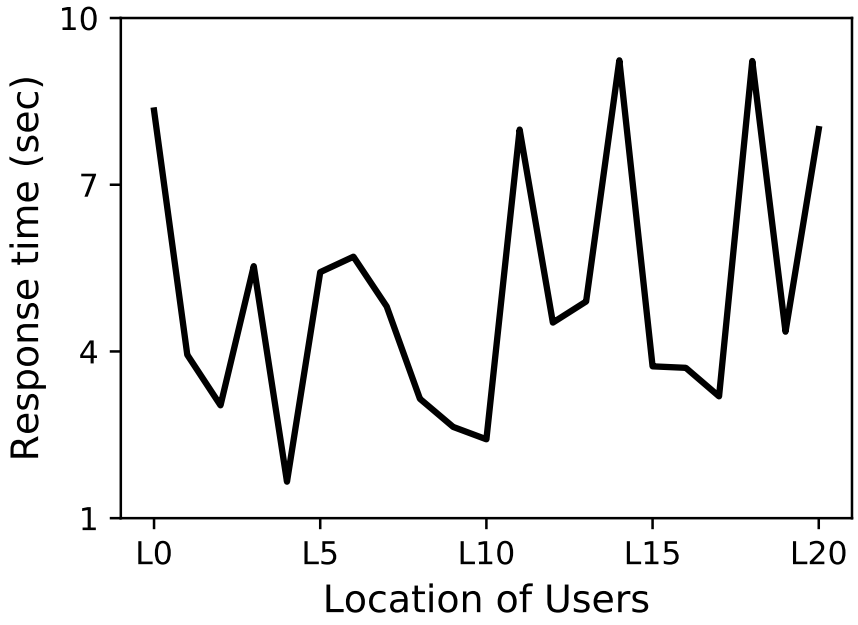
In this section, we first discuss the architectural details of GWA. We then analyze the challenges of deploying applications in a geo-distributed environment.

5.2.1 Geo-distributed web-application

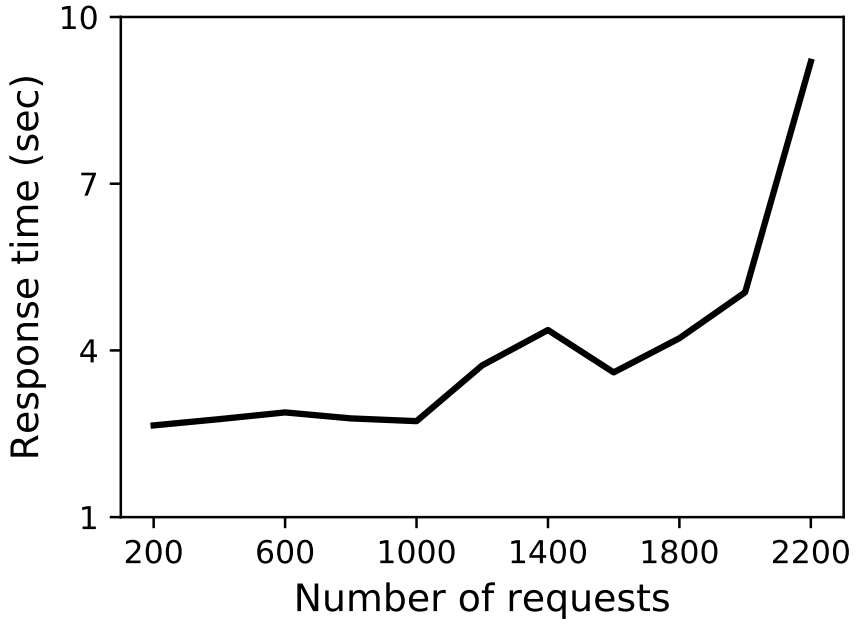
A GWA follows a sophisticated architecture with a set of user-interfaces deployed close to the users in order to provide low latency. Each user-interface may have a local database to cache the frequently visited contents. Moreover, these user-interfaces connect to one or more central databases, because moving a large volume of data is not easy and the sensitive data must be stored at a specific location/region due to privacy and ethical issues [145].

Variety and uncertainty of deployment environment. The cloud computing provides an opportunity for WA owners to realize the global deployment. Since there are more than 267 CPs available in the marketplace, offering numerous host configurations at each datacenter geo-location (e.g. AWS offers 275 instances in Europe (London) datacenter). Moreover, these hosts are inter-connected using WAN. However, the available bandwidth between a pair of DCs can be variable. The authors in [143] reported that the ratio of the highest pair to the lowest pair bandwidth is greater than 20 in both Amazon EC2 and Microsoft Azure.

Location matters. The response time is dramatically affected by the location between the user and the WA host. Figure 5.1a shows a huge fluctuation of the response time when a WA is deployed on the AWS London datacenter and numerous users are accessing from different geo-locations. It shows that the highest latency is almost 6 times to the lowest one. Additionally, Figure 5.1b shows that with the fixed host



(a) The response time for users sending requests from different geolocations. L0 represents a specific geolocation of a User.



(b) The response time for a users with varying number of requests

Figure 5.1: The response time is affected by both location between host and user as well as the capacity of the host.

capacity the latency increase dramatically when the host is saturated, i.e. when more than 2000 requests are sent simultaneously.

5.2.2 Deployment challenges

Importance of deployment location. Assume that a company wants to open its global business, starting to build a GWA. The central database is deployed at its headquarters inside the UK along with a user-interface. Due to the limited budget, the company can only set-up another user-interface that is deployed in China. The users from China can experience low latency of visiting this GWA. However, if many requests need to be redirected to the central database, the users from China may experience huge latency because of the long-distance communication and limited bandwidth. However, no literature has considered to tackle this issue.

Importance of choosing host types. User's experience (e.g. response time) is significantly dependent on the capacity of the hosts executing GWA. The cxloud providers (CPs) offer tons of host configurations; the GWA owner may not have a clear idea of choosing the suitable ones, without executing them. If the hosts could not provision the sufficient computing resources, it will also cause high response time or even outage problems. In contrast, hiring a powerful host may cause computing resources under provisioning, which can bring dramatic resource wastage. As a result, the hosts must be benchmarked before deploying the real GWA.

Benchmarking geo-distributed web-application. Benchmarking GWA is challenging from both *system* and *algorithm* perspective. The *system* challenge is that the ideal benchmark orchestrator is able to interact with various CPs that offer different ways to access their computing resources and automatically run the benchmark application on these resources while providing the required evaluation results. However, most of the research is focused on single cloud scenarios [49, 128]. [78, 79] tackled the problem of deploying containerized web-application on multiple clouds belonged to different CPs, but these tools fail to orchestrate a GWA benchmarking.

Regarding the *algorithm* challenge, the desired orchestrator has to select the host from massive available candidates within a limited budget. This challenge is amplified when we aim to maximize the user's experience at a global level. In other words, the orchestrator has not only to consider *which* types of hosts should be benchmarked, but also *where* to run the selected hosts. As a result, the proposed algorithm needs to

have the following desirable properties.

Close to users. The generated benchmarking solutions need to deploy the GWA close to its users, which is one of the important tricks for reducing the response time significantly.

Increase the diversify hosts. If we can benchmark more diverse deployment solutions, including host configuration and location, we will have more chances to find a suitable solution.

Consider the benchmarking time and budget. To benchmark a GWA, the time and budget must be under consideration. Both directly influence how many *selected* deployment solutions can be obtained.

5.3 System overview

5.3.1 Web-application deployment model

A WA can be represented as a directed acyclic graph (DAG) $G_{web} = (V_{web}, E_{web})$, where V_{web} represents the web-application components (microservices) and E_{web} represents the dependency between two related components including data flow and transactions. The underlying computing resources offered by the variety of cloud providers is assumed as a complete graph $G_{res} = (V_{res}, E_{res})$ where the nodes V_{res} represents the set of available hosts provided by various cloud providers and edges E_{res} represents the network connections among the hosts. Each pair of distinct nodes (v^i_{res}, $v^j_{res} \in V_{res}$) in G_{res} is connected by a pair of edges ($e^{i,j}_{res}$ and $e^{j,i}_{res}$). A host v^i_{res} is a 4 tuple system $\langle c, h, loc, p\rangle$ where, $c \in C$ is a cloud provider, $h \in \mathcal{H}$ represents the host type, $loc \in \mathcal{L}$ represents the datacenter geo-location and p represents the pricing of using the host. Each $h \in \mathcal{H}$ is represented by 4 attributes {name, cpu, memory and storage} while p includes the unit execution cost of v^i_{res} and unit data downloading and uploading cost from and to v^j_{res}.

To model the deployment of a web-application G_{web} to the cloud providers G_{res}, we use an *injective* mapping σ that creates a bipartite graph $G = (V_{web} \cup V_{res}, E)$ where, $e \in E$ between v^i_{web} and v^i_{res} indicates the component $v^i_{web} \in V_{web}$ is running on

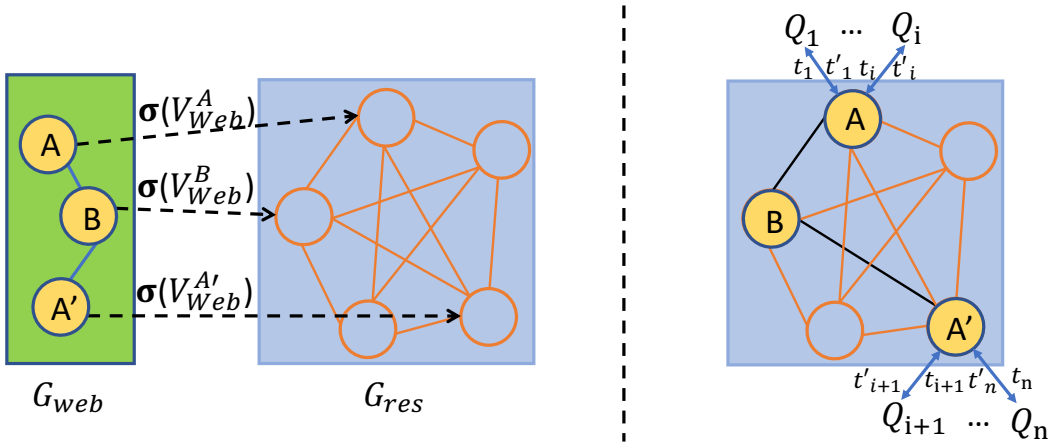


Figure 5.2: An example of the bipartite graph G

host $v^i_{res} \in V_{res}$. Let $\theta_t \in \Theta$ be a deployment solution obtained by applying an *injective* mapping that maps each component $v^i_{web} \in V_{web}$ to a unique host $v^i_{res} \in V_{res}$ ($v^i_{web} \rightarrow v^i_{res}$). Since $|V_{res}| \gg |V_{web}|$, the mapping should be strictly injective (not surjective). Figure 5.2 shows an example of deploying G_{web} to G_{res}.

As a result, we formulate our design goals as a *two-phase* optimization problem as follows.

5.3.2 Problem formalization

Based on the desirable properties of the *selected* deployment solution, in this chapter, we aim to develop an algorithm that maximizes the type of hosts and the hosts' locations within the limited budget thereby increasing the chance of obtaining a deployment option which can maximize user's satisfaction. As a result, we formulate our research problem as a *two-phase* optimization problem as follows.

First phase We aim to find a set of $\Theta = \{\theta_1, \theta_2, ...\theta_T\}$ which can maximizes the types of hosts h and the hosts' locations loc within the defined budget $\mathcal{B}$. For $\theta_t \in \Theta$, the location of each underlying host must be *unique* and the type of host deploying different components is *similar* in the configuration. In this paper, we consider two types of cost: execution cost and communication cost both given in terms of time. Equation 5.1a indicates the execution cost for θ_t, where $\mathcal{P}(v^i_{res})$ is the cost for running v^i_{res} in a unit time and $\mathcal{O}(v^i_{res})$ is the total execution time. The communication cost is illustrated in equation 5.1b, $\mathcal{D}_{e^{i,j}_{res}}$ represents the size of data transferred from v^i_{res} to

v^j_{res} and $\mathcal{N}(E^{v^i_{res},v^j_{res}})$ is the cost of transferring one unit of data from v^i_{res} to v^j_{res}. Note E_{res} is a DAG, and $\mathcal{N}(E^{v^i_{res},v^j_{res}})$ may not equal to $\mathcal{N}(E^{v^j_{res},v^i_{res}})$, i.e. $\mathcal{N}(E^{v^i_{res},v^j_{res}}) \neq \mathcal{N}(E^{v^j_{res},v^i_{res}})$. Total cost $\mathcal{C}^{\theta_t}$ for θ_t is shown in 5.1c.

$$\mathcal{C}_{exe} = \sum_{v^i_{res} \in \theta_t} \mathcal{O}(v^i_{res}) \times \mathcal{P}(v^i_{res}) \tag{5.1a}$$

$$\mathcal{C}_{com} = \sum_{v^i_{res},v^j_{res} \in \theta_t} \mathcal{D}_{e^{i,j}_{res}} \times \mathcal{N}(E^{v^i_{res},v^j_{res}}) \tag{5.1b}$$

$$\mathcal{C}^{\theta_t} = \mathcal{C}_{exe} + \mathcal{C}_{com} \tag{5.1c}$$

We thus formulate our goal as an integer linear programming problem given in equation 5.2 with a set of constraints as shown in equation 5.2a-5.2c.

$$\textbf{maximize} \quad |\Theta| \tag{5.2}$$

subject to:

$$\theta_i \neq \theta_j \rightarrow h_i \neq h_j \; \forall \theta_i, \theta_j \in \Theta, h_i, h_j \in \mathcal{H} \tag{5.2a}$$

$$\forall \theta_i \in \Theta, \; v^1_{res}(loc) \neq v^2_{res}(loc), \; v^1_{res}, v^2_{res} \in \theta_i \tag{5.2b}$$

$$\sum_{\theta_i \in \Theta} (\mathcal{C}^{\theta_i}) \leq \mathcal{B} \tag{5.2c}$$

Our main aim is to maximize the number of $|\Theta|$. It allows the developer to evaluate more options in the aim of finding the most suitable option θ^{best}. Cons 5.2a states that the selected option must be unique in terms of its type of host, and each θ_i only considers one type of host. Moreover, for each deployment solution θ_i, the location of each host must be different as shown in Cons 5.2b. Finally, Cons 5.2c represents the total incurred cost must be less than or equal to the total budget $\mathcal{B}$.

Second phase. In the **first phase**, we generate a set of deployment solutions Θ. These solutions will be deployed and executed on the real cloud environment, and the execution results are collected. The solution with "minimal" response time is selected and resulted for the deployment of the desired GWA.

Table 5.1: A summary of symbols used in the chapter

Symbols	**Description**
$G_{web} = (V_{web}, E_{web})$	Graph of GWA
$G_{res} = (V_{res}, E_{res})$	Fully connected graph of cloud provider hosts
$v^i_{res} \in V_{res}$	Host i of G_{res}
$c \in \mathcal{C}$	Cloud Provider for host v^i_{res}
$h \in \mathcal{H}$	Host type for host v^i_{res}
$loc \in \mathcal{L}$	Datacenter geo-location for host v^i_{res}
$\theta_t \in \Theta$	Solution t in set of solution Θ
$\mathcal{P}(v^i_{res})$	Unit execution cost host v^i_{res}
$\mathcal{O}(v^i_{res})$	Total execution time for v^i_{res}
$\mathcal{N}(E^{v^i_{res}, v^j_{res}}_{res})$	Unit data transferr cost from v^i_{res} to v^j_{res}
$\mathcal{D}_{e^{i,j}_{res}}$	Size of data transferred from v^i_{res} to v^j_{res}
$\mathcal{C}^{\theta_t}_{exe}, \mathcal{C}^{\theta_t}_{com}$	Execution cost and transfer cost for a solution θ_t
$\mathcal{C}^{\theta_t}$	Total cost of a solution θ_t
$\mathcal{B}$	User's budget
s	Population of particle
P_i, P^{local}_i, Vel_i	Position, local best position and velocity of a particle
P^{global}_{best}	Global best position
ζ	Optimization objective (maximize or minimize)
$\overline{Pos}$	Output solution set
$\eta = \lvert\overline{Pos}\rvert$	Number of output solution
ω	Inertia weight
$C1, C2$	Self recognition and scoial constant factor
$Clus = \{Clus_1, ..., Clus_K\}$	K disjoint clusters
u_j	Data point reresenting cluster center for $Clus_j$
a_i	Data point representing host v^i_{res}

5.4 System design

GeoDeploy is a user-centric and cost-effective middleware that aims to offer a full stack benchmarking solutions for GWA. It is a comprehensive integration tool that optimizes and automates the GWA deployment on multiple clouds. Correspondingly, there are two main subsystems in GeoDeploy: *benchmark planner* and *benchmark orchestrator*, as depicted in Figure 5.3.

Benchmark planner. When a GWA and benchmark budget are submitted to GeoDeploy, the *benchmark planner* takes these information along with the collected host information to generate a set of deployment solutions for benchmarking using our proposed algorithms (detailed in §5.6) or other baseline algorithms such as Random

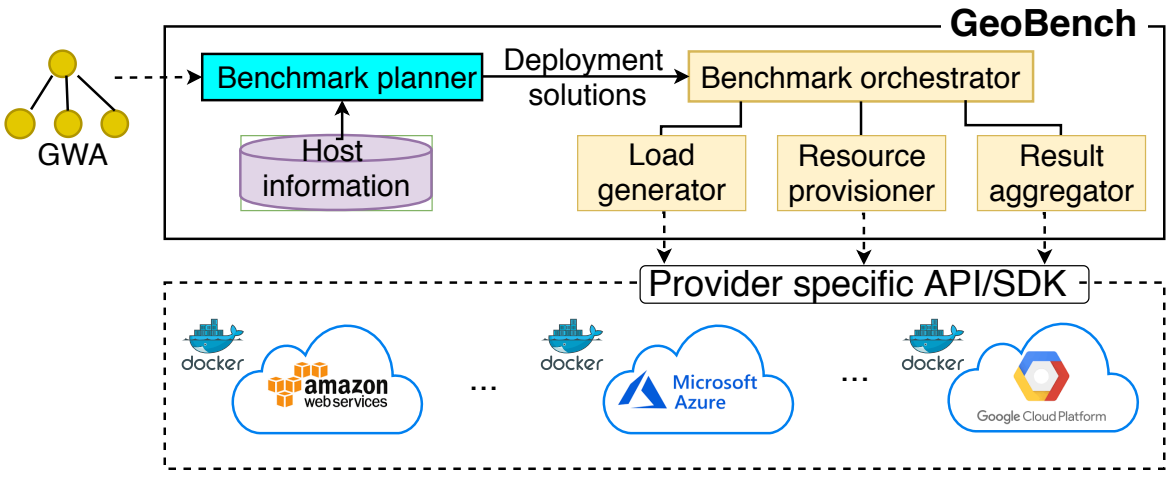


Figure 5.3: System architecture of GeoDeploy

selection and Greedy approach.

Benchmark orchestrator. It first receives the set of deployment solutions generated by the *benchmark planner* and performs a sanity check. It then executes those solutions in the real cloud environment. First, the *Resource provisioner* automatically provision the cloud resources and deploy the given GWA based planned deployment solutions. It hides the complexity of cloud provider deployment in terms of accessing provider-specific APIs, installing and accessing containers, establishing communications, etc. Further, to evaluate the performance of the deployed GWA, the *Load generator* is developed to emulate the users visiting the GWA globally. To this end, the *Load generator* can automatically create a pool of clients across the world according to the registered data centers in GeoDeploy. Each client is specified with a *test plan* that contains the request sending rate, request types and the number of total requests. After the benchmarking experiments, the *Result aggregator* collects the results including users' response time and system statistics, and then recommends a suitable deployment solution for the given GWA.

5.5 Adaptive PSO algorithm

In this section, we present the proposed *APSO* (Adaptive Particle Swarm Optimization) algorithm which is an extension of Particle Swarm Optimization (*PSO*) [84]. Our proposed *APSO* is able to find a sub-optimal solution that minimizes or maximizes

the deployment cost for a given GWA. Algorithm 2 illustrates the key steps of *APSO*.

Algorithm 2: *Adaptive PSO*

Input: s - population size, $G_{res} = (V_{res}, E_{res})$ - connection graph of cloud providers host, η - number of required solution, ζ - optimization constraint

Output: $\overline{P}$ - list of η resulting solutions

for $\forall i \in \{0, s\}$ **do**
  // Initialize vectors P, Vel and P^{local}
  $P_i \leftarrow$ `rand` $(0, |V_{res}|)$
  // Validate the solution P_i
  $P_i \leftarrow$ `VPG` (P_i)
  $Vel_i \leftarrow$ `rand` $(-|V_{res}|, |V_{res}|)$
  $P_i^{local} \leftarrow P_i$
  // Compute the fitness function $\mathcal{F}_i$
  $\mathcal{F}_i \leftarrow$ `fitness` (P_i, G_{res})
end
//Initialize the output $\overline{P}$
$P^{srt} \leftarrow$ `sort` $(P, \mathcal{F}, \zeta)$
$\overline{P} \leftarrow P^{srt}[0, \eta]$
// Initialize the global best solution P_{best}^{global}
$P_{best}^{global} \leftarrow P^{srt}[0]$
//Repeat till the termination condition not reached
while !$terminate$ **do**
  //Update the solutions P
  for $\forall i \in \{0, s\}$ **do**
    $P_i^{new}, Vel_i^{new} \leftarrow$ `update` $(P_i, Vel_i, P_i^{local}, P_{best}^{global})$
    // Validate the updated solution P_i^{new}
    $P_i^{new} \leftarrow$ `VPG` (P_i^{new})
    // Compute the fitness function $\mathcal{F}_i^{new}$
    $\mathcal{F}_i^{new} \leftarrow$ `fitness` (P_i, G_{res})
    // update the local best solution P_i^{local}
    if $\mathcal{F}_i^{new} \succ \mathcal{F}_i^{old}$ **then**
      $P_i^{local} = P_i^{new}$
      if $\mathcal{F}_i^{new}$ *is better than* $\mathcal{F}_{best}^{global}$ **then**
        //update the global best solution
        $P_{best}^{global} = P_i^{new}$
  end
  //Update the output $\overline{P}$
  $\overline{P} \leftarrow$ `Replace`$(P^{new}, \overline{P})$
end

First, we initialize three vectors $P = \{P_1, P_2, ..., P_s\}$, $P^{local} = \{P_1^{local}, P_2^{local}, ..., P_s^{local}\}$ and $Vel = \{Vel_1, Vel_2, ..., Vel_s\}$. P records the current deployment solutions; each P_i represents a solution θ_i. The best deployment solution obtained for each P_i during

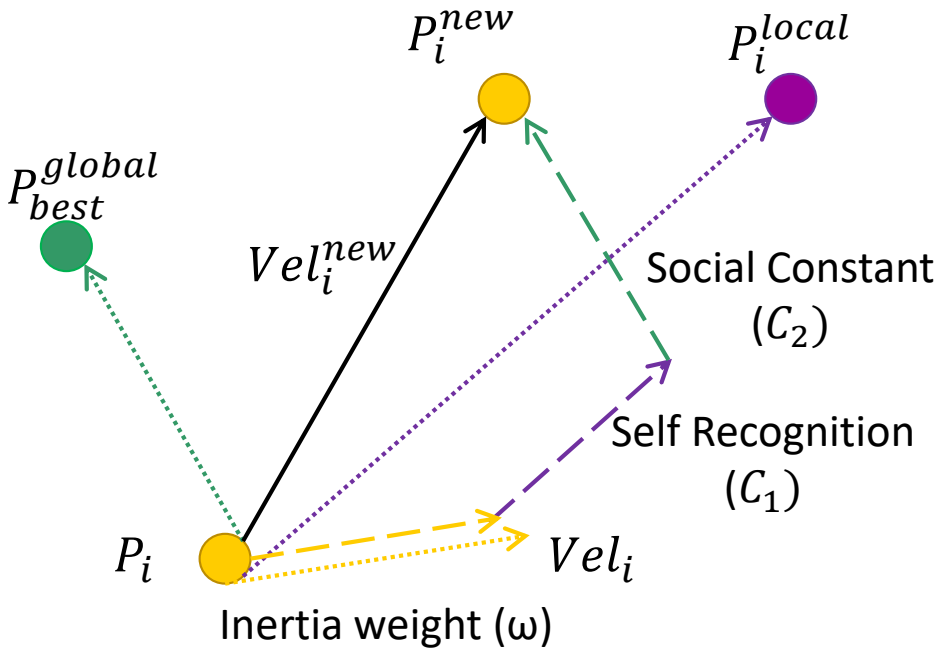


Figure 5.4: Movement of a solution P_i in $APSO$

the run-time of the algorithm is stored in P^{local}. In Vel, each Vel_i represents the variable to update the value of P_i. We initialize P_i by randomly mapping a host from V_{res} for each component of the GWA. The generated P_i is checked for the validity of the deployment solution as specified by the constraints in §5.3.2 and concealed only if found valid (Algorithm 2 Line 3-5). Otherwise, it will be discarded and a new valid one is generated. To allow a uniform update of current deployment solution P_i to a new one P_i^{new}, Vel_i is initialized to a set of randomly generated integer values between $-|V_{res}|$ and $|V_{res}|$ (Algorithm 2 Line 6). P^{local} equals to P at this stage.

Next,the fitness value $\mathcal{F}_i$ of each P_i is computed by using equation 5.1c (Algorithm 2 Line 9). Here, the fitness value $\mathcal{F}_i$ represents the total cost $\mathcal{C}_{\theta_i}$ for deploying P_i. Based on the obtained fitness values, we can select η solutions $\overline{P}$ as shown in Algorithm 2 Line 12-13. ζ represents the objective of the algorithm, i.e. find a list of deployment solutions that maximizes or minimizes the fitness value and η is the desired number of output solutions. At the same time, we choose the P_i with best fitness value as the global best solution P_{best}^{global} (Algorithm 2 Line 15).

To find better deployment solution i.e. to update P_i to P_i^{new}, we used *three* variables, Vel_i, P_i^{local} and P_{best}^{global}. Figure 5.4 shows how to update P_i using these *three* variables and equation 5.3 and equation 5.4 are used to compute the new deployment solution.

First, we compute a Vel_i^{new} which is the key variable to update P_i, which is affected by the *three* variables. Where ω defines how much Vel_i can contribute to the updating process. $C1$ is a factor which affects similarity between P_i^{new} and P_i^{local}; and $C2$ affects the similarity between P_i^{new} and P_{best}^{global}. We set the value of $C1$ and $C2$ as 2 suggested by [45]. The randomly generated $F1$ and $F2$, following uniform distribution between 0 and 1, reduce the bias introduced by P_i^{local} and P_{best}^{global}. To restrict the search space within the available host $|V_{res}|$ for computing both Vel_i^{new} and P_i^{new}, we apply a modular operation.

$$\begin{aligned} Vel_i^{new} = (\omega \times Vel_i + C1 \times F1 \times (P_i^{local} - P_i) \\ + C2 \times F2 \times (P_{best}^{global} - P_i))\%|V_{res}| \end{aligned} \tag{5.3}$$

$$P_i^{new} = (P_i + Vel_i^{new})\%|V_{res}| \tag{5.4}$$

The new deployment solutions are checked and the fitness values the valid solutions will be calculated. If the new fitness value $\mathcal{F}_i^{new}$ is better than the fitness value of P_i^{local} and P_{best}^{global}, it should be updated (Algorithm 2 Line 27-30). Finally, we update the solutions $\overline{P}$ by replacing the exiting solutions with that in P^{new} which have better fitness values (Algorithm 2 Line 32). The described steps (Algorithm 2 Line 16-30) are repeated until it meet the termination condition.

Termination. In this chapter, the *terminate* is set that either repeat 100 iterations or there is no updating of the P_{best}^{global} in 30 iterations.

5.6 Optimize the deployment

This section presents the detailed description of our proposed approach which consists of *four* main components, including *clustering*, *budget allocation*, *deployment solution generation* and *benchmarking in real-world environment*.

To obtain the "best" deployment solution, each component is executed sequentially as shown in Algorithm 3 and Figure 5.5. An input set should be provided, in which

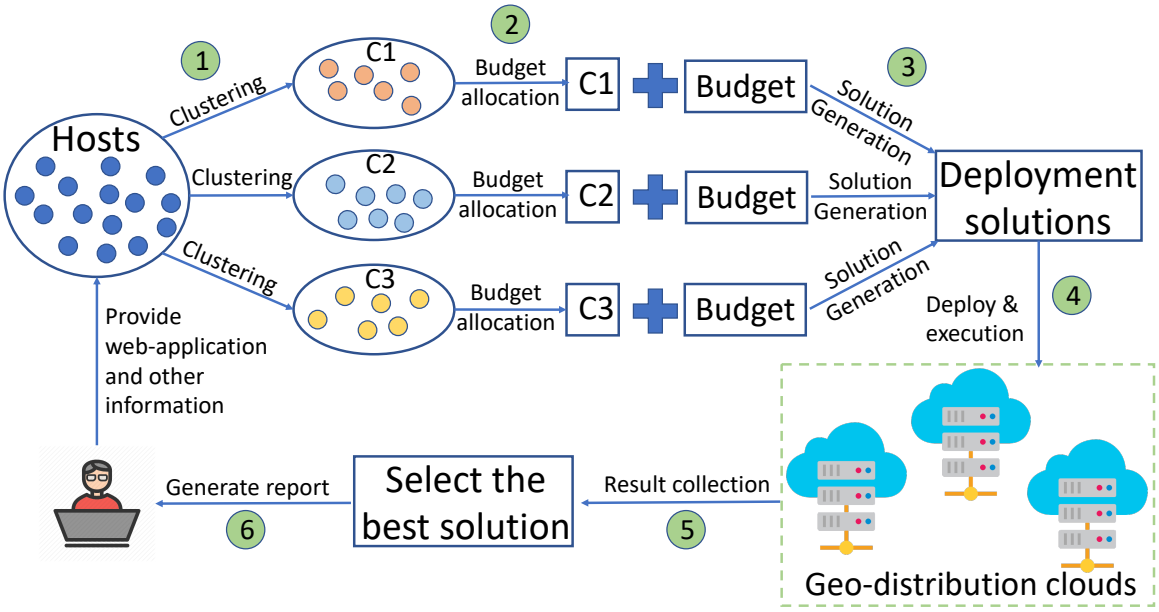


Figure 5.5: The execution workflow of the proposed method

the GWA, benchmarking budget $\mathcal{B}$ and benchmarking time $\mathcal{O}$ are provided by the user, and the host information V_{res} including the CPU, memory, location and price, about provider is pre-collected. In the first step, we partition the host into K clusters (see Algorithm 3 Line 2). Next, the given budget $\mathcal{B}$ is allocated to K clusters in step 2 (see Algorithm 3 Line 4). In step 3, based on the allocated budget, clustered hosts and other provided information, a set of deployment solutions are generated (see Algorithm 3 Line 6). The obtained solutions are automatically deployed and executed on the corresponding hosts (step 4). Finally, the benchmarking results are collected and then a report is generated and sent back to the user (illustrated in step 5, 6 and Algorithm 3 Line 8). The following subsections indicate the technical details of each key component.

5.6.1 Clustering

Our first step is to create a set of clusters to group all the nodes V_{res}, i.e. each node $v_{res}^{i} \in V_{res}$ is mapped to one and only one cluster with all the nodes belonging to the same cluster bearing similar host type characteristics ($h \in \mathcal{H}$). Each host type h is associated with many dimensions of resources such as CPU, memory, storage, bandwidth, etc. In this chapter, our clustering algorithm considers CPU and memory.

Algorithm 3: : GeoDeploy's algorithm overview

Input: $G_{web} = (V_{web}, E_{web})$ - dependency graph of the web-application, $G_{res} = (V_{res}, E_{res})$ - connection graph of cloud providers host, $\mathcal{O}$ - benchmarking time, $\mathcal{B}$ - User's Budget

Output: θ - obtained solution

1 // Partition the V_{res} hosts' configuration into K clusters
2 $K, Clus \leftarrow Clustering(V_{res})$
3 // Allocate $\mathcal{B}$ to K clusters
4 `allocateBudget`($Clus$, $\mathcal{B}$);
5 // Generate solutions for each cluster
6 $\Theta \leftarrow \sum_{i \in K}$ `genSolution` ($Clus_i$, $\mathcal{B}_i$)
7 // Benchmark the selected solutions and choose the best solution
8 $\theta \leftarrow$ `Benchmark` (Θ)

We employed $K - means$ algorithm to partition the data points into K disjoint clusters $Clus = \{Clus_1, ...Clus_K\}$ [82]. Initially, each cluster $Clus_j$ is assigned with an arbitrary cluster center u_j. A data point a_i included CPU and memory information is allocated to a cluster $Clus_j$ based on the closeness to the cluster center u_j using equation 5.5. $||\cdot||$ represents the function (Euclidean function) to measure the distance between the data point a_i and the cluster center u_j. With each iteration, u_j is also updated according to equation 5.6. Here $c_j \in [0, 1]$ in an index variable which is only equal to 1 if a_i is allocated to cluster $Clus_j$. These steps are repeated until the cluster center u_j remains unchanged.

$$arg \quad \min_j ||a_i - u_j||^2 \tag{5.5}$$

$$u_j = \frac{\sum_j (c_j \times a_i)}{\sum_j c_j} \qquad c_j = 1, 0 \tag{5.6}$$

It is to be noted that the parameter K needs to be chosen and given as an input to the K-means algorithm. Determining the number of cluster K is essential to achieve optimal partitioning. In this work, we employed *Elbow Method* to find the optimal value of K [108]. It starts with different values of $k \in K$ and computes a total *Intra Cluster Variation* ICV_k for each cluster size k as given in equation 5.7.

$$ICV_k = \sum_{j=1}^{k} \sum_{a_i \in Clus_j} ||a_i - u_j||^2 \tag{5.7}$$

The optimal cluster number K appears when the difference between ICV_k and ICV_{k+1} dropped significantly. It is worth mentioning that the clustering stage is done only once on a host data set. The stage is repeated only when the dataset is changed.

5.6.2 Budget allocation

As we discussed above, deployment solutions are generated for each cluster, so it is necessary to fairly distribute the given budget $\mathcal{B}$ to the clusters $Clus$. Since, the unit execution cost $\mathcal{P}(v^i_{res})$ of host v^i_{res} varies from one cluster to other, it is not suitable to divide the budget $\mathcal{B}$ equally among the clusters. The average cost for benchmarking a cluster is a good reference for budget distribution to ensure fairness. However, evaluating the average cost for a cluster requires computing the cost of each possible solution which is not feasible. Alternatively, we use the mean of solutions which have the maximum and minimum cost as the reference to distribute the budget $\mathcal{B}$. To this end, we interact our Algorithm 4 with the *APSO* (discussed in §5.5). For each cluster $k \in K$, we first obtain a solution θ_k with maximum cost $\mathcal{C}^{\theta_k}_{max}$ and minimum cost $\mathcal{C}^{\theta_k}_{min}$ by setting the optimization constraint ζ as *maximization* and *minimization* respectively (see Algorithm 3 Line 3-7). Next, the average cost for each cluster k, $\mathcal{C}^{\theta_k}_{av}$ is computed by normalizing the maximum and minimum cost as shown in Line 9. Finally, the user's budget $\mathcal{B}$ is distributed to each k cluster using the equation as shown in Line 12 and output $\mathcal{B}_k, k \in \{1, K\}$ is returned.

5.6.3 Deployment solution generation

In this stage, the cluster budget $\mathcal{B}_k$ is distributed to find a set of solutions for each cluster $Clus_k$ as given in Algorithm 5. First, we generate η set of solution $\overline{P}_k$ for each cluster k using *APSO* algorithm (Line 3). Next, each solution $\overline{P}_k[i_k]$ is added individually to a final list Θ^{fin}_k till no solution can be added in the remaining budget $\mathcal{B}^{left}_k$ (Line 6). To maximize the utilization of the budget, we combine all the remaining cluster budget $\mathcal{B}^{left}_k$ (Line 12). Later, the budget $\mathcal{B}^{left}$ is allocated to each cluster in

Algorithm 4: : Budget allocation

Input: $Clus = Clus_1, ..., Clus_K$ - K clusters of cloud providers host, $\mathcal{B}$ - User's Budget

Output: $\mathcal{B}_k, \forall k \in \{1, K\}$ - clustered budget

1 **for** $\forall k \in K$ **do**
2 // Get a solution θ_k with maximum total cost $\mathcal{C}_{max}^{\theta_k}$
3 $\Theta^{max} \leftarrow APSO(Clus_k, \zeta = maximization)$
4 $\theta_k^{max} = \Theta^{max}[0]$
5 // Get a solution θ_k with minimum total cost $\mathcal{C}_{min}^{\theta_k}$
6 $\Theta^{\min} \leftarrow APSO(Clus_k, \zeta = minimization)$
7 $\theta_k^{min} = \Theta^{min}[0]$
8 // Compute the average cost of $\mathcal{C}_{av}^{\theta_k}$
9 $\mathcal{C}_{av}^{\theta_k} = \frac{\mathcal{C}_{min}^{\theta_k} + \mathcal{C}_{min}^{\theta_k}}{2}$
10 **end**
11 // Distribute the budget $\mathcal{B}$
12 $\mathcal{B}_k = \frac{\mathcal{C}_{av}^{\theta_k}}{\sum_{k=1}^{K} \mathcal{C}_{av}^{\theta_k}} \times \mathcal{B} \qquad \forall k \in K$

a descend order based on its reference (refer to Algorithm 3 Line 9) to compute more solutions (Line 14-18). Finally, the clustered solutions Θ_k^{fin} from each k cluster are merged together to get the final list of solution Θ' for the real deployment.

5.6.4 Benchmarking in real-world environment

The set of solutions obtained in the previous step Θ' is finally deployed using the *benchmark orchestrator* discussed in §5.4. More details about the deployment and the evaluations are given in §5.7. Finally, the evaluation results are collected and analyzed to find θ^{best}.

5.7 Evaluation

In this section, we evaluate our GeoDeploy for both algorithm and system performance. We first performed numerical analysis on the data collected from different cloud providers which results in a small set of deployment solutions which are then deployed on the real-cloud environment. We compared our proposed algorithm with some baseline methods in terms of diversity and scalability. We further show that *selected* solutions obtained by our algorithm have a better user experience compared to other methods by executing in the real multiple clouds environment.

Algorithm 5: *Solution generation*

Input: $Clus_k, k \in \{1, K\}$ - K clusters of cloud providers host, $\mathcal{B}_k, \forall k \in \{1, K\}$ - cluster budget

Output: Θ' - optimized list of hosts

// Get a set of solution Θ_k for each cluster k
for $\forall k \in K$ **do**
  $\overline{P}_k = \texttt{APSO}(Clus_k, \zeta = minimization)$
  $\mathcal{B}_k^{temp} = 0$, $\Theta_k = [], i_k = 0$
  while $(\mathcal{B}_k^{temp} \leq \mathcal{B}_k)$ **do**
    $\Theta_k^{fin} \leftarrow \overline{P}_k[i_k]$
    $\mathcal{B}_k^{temp} \leftarrow \mathcal{B}_k^{temp} + \mathcal{C}^{\overline{P}_k[i_k]}$
    $i_k \leftarrow i_k + 1$
  end
end
// Compute the left budget $\mathcal{B}^{left}$
$\mathcal{B}^{left} \leftarrow \sum_{k=1}^{K}(\mathcal{B}_k - \mathcal{B}_k^{temp})$
// Utilize $\mathcal{B}^{left}$ to add more solutions
for $k \in \{K, 1\}$, **do**
  if $\mathcal{B}^{left} \leq \mathcal{C}^{\overline{P}_k[i_k]}$ **then**
    $\Theta_k^{fin} \leftarrow \overline{P}_k[i_k]$
    $\mathcal{B}^{left} \leftarrow \mathcal{B}^{left} - \mathcal{C}^{\overline{P}_k[i_k]}$
    $i_k \leftarrow i_k + 1$
  end
end
// Merge the solutions Θ_k^{fin} to get the final list Θ'
$\Theta' \leftarrow \sum_{k=1}^{\mathcal{K}} \Theta_k^{fin}$

5.7.1 Experiment setup

Environment for algorithm evaluation. We evaluated GeoDeploy and it comparison algorithms on a PC with Intel(R) Core(TM) i5-6200U CPU @2.3GHz - 2.4GHz with 16 GB memory and 512 GB SSD.

Dataset and evaluation methodology. We considered three main cloud providers i.e. AWS[1], Microsoft Azure[2] and Google Cloud[3] and for each of them we selected four datacenters allocated in *UK South (London)*, *US West (Oregon)*, *South America (Sao Paulo)* and *Asia Pacific (Singapore)*. As a result, the input of our algorithm includes 776 host configurations with their execution and communication cost, which

[1]https://aws.amazon.com/ec2/pricing/on-demand/
[2]https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/linux/
[3]https://cloud.google.com/compute/vm-instance-pricing

is available on GitHub[4].

In the algorithm evaluation, we compared our proposed algorithm with four baseline algorithms namely, *Random Selection* - randomly select the solution till the budget is available, *Greedy with Computation Cost (Greedy)* - select the solutions starting from cheapest computation cost one till the budget is available and *Clustered Greedy with Computation Cost (Clus Greedy)* - first create K clusters and for each cluster follow *Greedy with Computation Cost (Greedy)* strategy, and evaluated them in three metrics: (i) the number of obtained solutions; (ii) the diversity of the hosts in the obtained solutions; and (iii) the utilization of the budget. To accurately determine the effectiveness, we repeat each experiment for 20 times independently and calculate the average.

Benchmark application and its execution environment. Once the *selected* deployment solutions are obtained by our algorithm and baseline algorithms, our GeoDeploy can automatically deploy the obtained solutions, based on the provided host configurations, cloud providers and locations. Then, we emulated a pool of 50 users that are deployed in different geo-locations and a set of them are selected to continuously send the requests to the deployed WA. The average response time is the main metric for evaluating each solution.

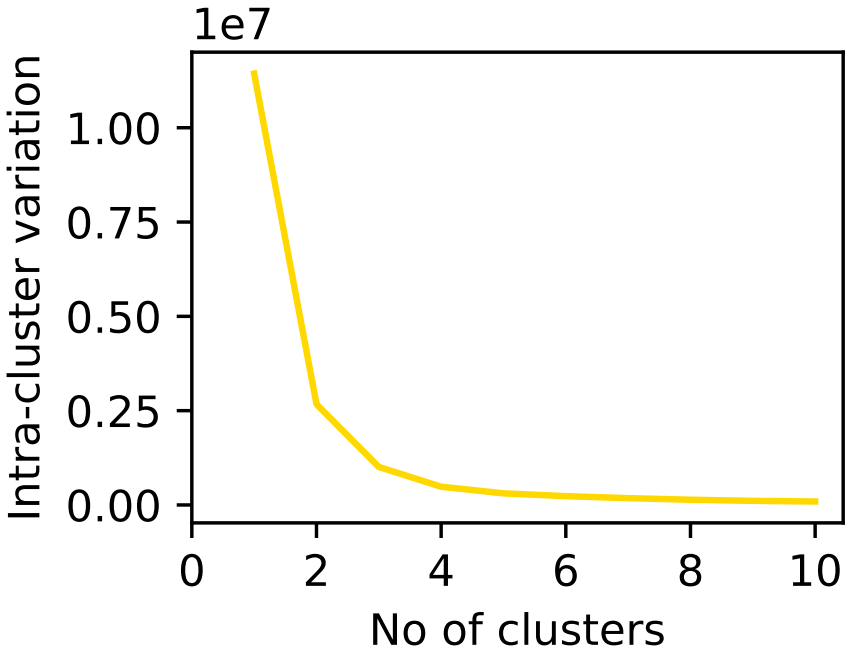
5.7.2 Algorithm evaluation

5.7.2.1 Clustering

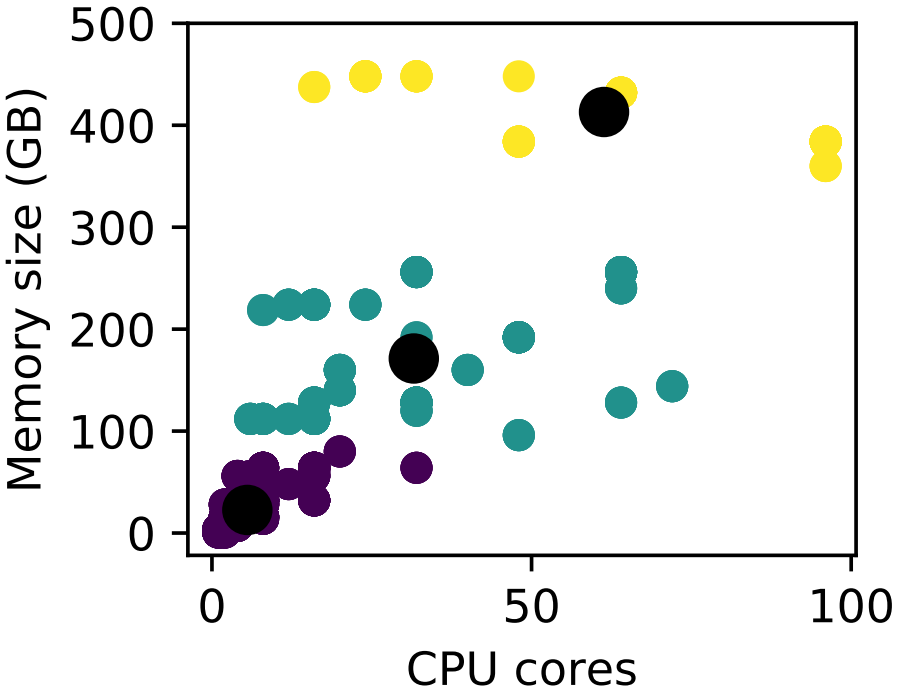
As we discussed in §5.6.1, we partition the host configurations into various clusters and then perform our algorithm through the clustered hosts to increase the diversity. This directly brings a question: *what is the best cluster number?* Using *Elbow method*, we vary the cluster size from 1 to 11 and find the best cluster size. Figure 5.6a shows that at the cluster size 3, a clear bent is visible which indicates it as the best cluster. Next, we clustered our data with $K = 3$. Figure 5.6b depicts the generated cluster with its centroid represented using a black circle.

We set the same inputs for all the evaluated algorithms. The *budget* is varied from

[4]https://github.com/DNJha/Middleware2020



(a) Elbow Method



(b) K-means Clustering Result

Figure 5.6: Clustering the given data to find the best cluster size (a) and clustering result (b).

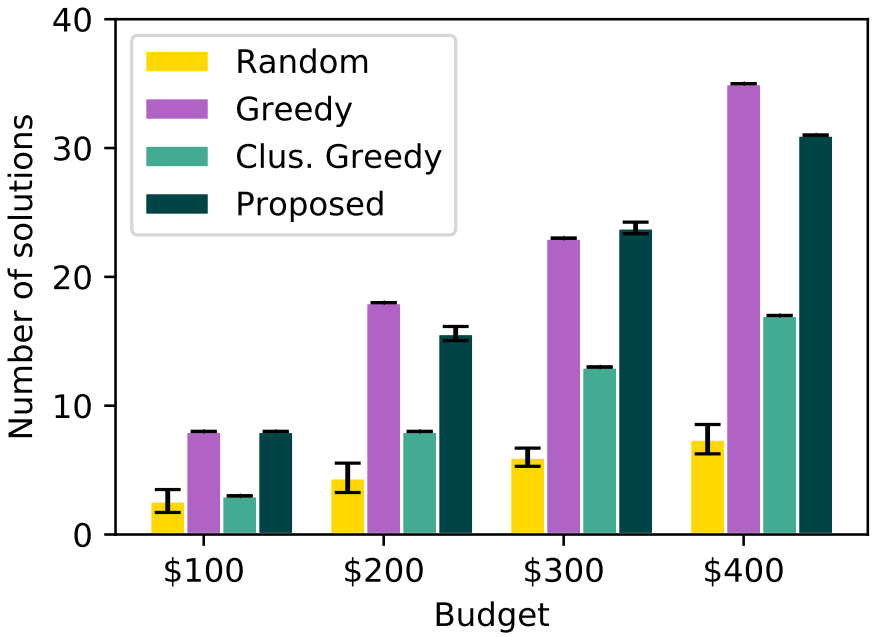
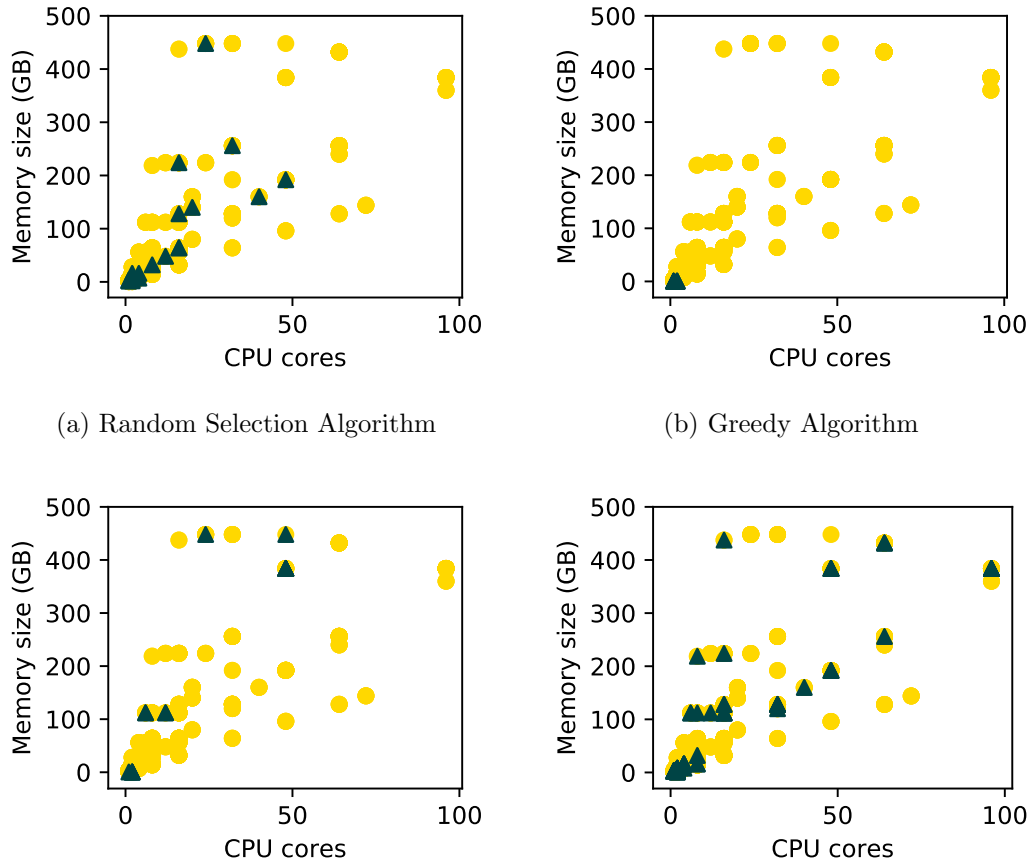


Figure 5.7: Number of solutions generated. Black bar on top represents the standard deviation.

$100 to $400, and we set the *size of the data* transferred among the components of a GWA via randomly selecting values between 4.3 and 344 GB/hour. This is based on the analysis of the real GWA workload executed on the experiment system in a controlled environment where each user request consumes approximate 25 - 500 KB data per seconds to repose. Based on the analysis, we assume that about 50 - 200 users attempt to access a component of GWA at any given hour.

5.7.2.2 Number of solutions vs. budget

In this subsection, we consider a WA consists of three components, i.e. one central database and two web-servers. Figure 5.7 shows that our algorithm obtained more solutions compared to *Random* and *Clus. greedy* for all the cases. On average our algorithm generates 72.5 % more solutions than *Random* and 40.5 % more solutions than *Clus. greedy*. Moreover, the proposed algorithm is comparable with the *Greedy approach* with an average of 6 % less number of obtained solutions. Since *Greedy* concentrates on host configurations with low computation cost, the expected number of solutions are high, however, GWAs always have a high amount of data transfer



(a) Random Selection Algorithm

(b) Greedy Algorithm

(c) Clustered Greedy Algorithm

(d) Proposed Algorithm

Figure 5.8: The host diversity comparison between GeoDeploy and the baseline methods. Each host configuration is represented as a circle in 2D space of CPU and memory values. The triangle shows the host configurations selected by the respective methods.

which is not covered by this approach leading to a comparable number of solutions.

Figure 5.9 shows our algorithm outperforms in utilizing budget compared to others, which is more than 17 times better than *Clust Greedy*. The wasted budget for *Random* is too variable, indicated by its high error rate (standard deviation) in Figure 5.9, and on average the wastage is almost 14× as compared to our proposed approach. Again, the wastage with *Clus. Greedy* is very high with an average of 16× higher as compared to the proposed method.

To show the fair comparison result, we have also performed *paired t-test* where we compared the proposed approach with baseline methods. The t-test results are based on equation 5.8.

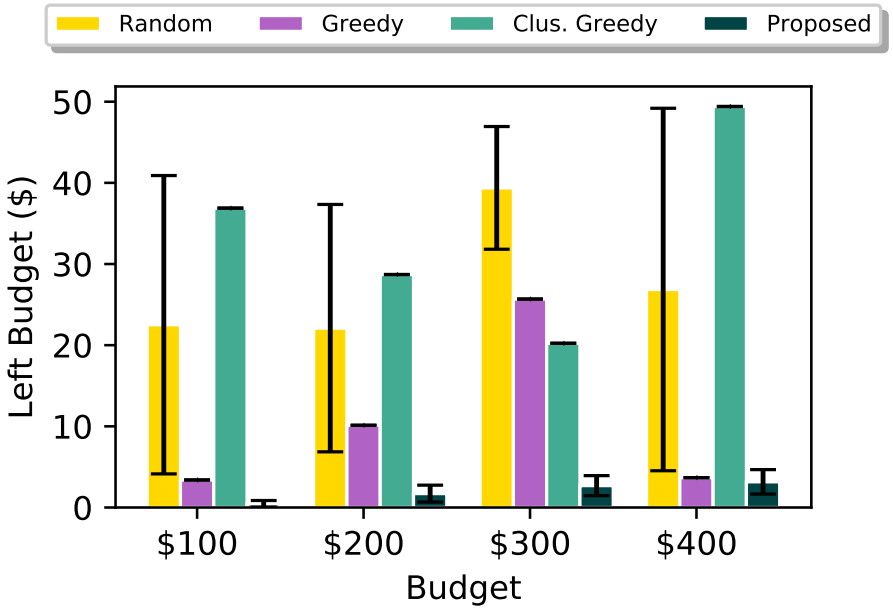


Figure 5.9: Average total unutilized Budget for different methods with varying Budget values. Black bar on top represents the standard deviation.

$$t = \frac{\mu_1 - \mu_2}{\Sigma_p \times \frac{2}{\sqrt{n}}} \tag{5.8}$$

where, μ_1 is the mean value obtained from the proposed method, μ_2 is the mean value obtained from other baseline methods, n is the number of samples and $\sigma_p = \sqrt{\frac{\sigma_1^2+\sigma_2^2}{2}}$ is the composite variance. Table 5.2 gives the t-test results for the number of solutions obtained. The result clearly shows that except for the *Greedy* case, proposed approach is far batter than the other approach. As explained in §5.7.2.2, *Greedy* approach finds the cheapest solution therefore it shows better performance as compared to the proposed approach.

Table 5.3 gives the t-test results for the value of un-utilized budget. In this case, higher negative value shows that our approach utilizes budget in a better manner. The obtained result shows that the mean μ and variance σ^2 of *Random* approach is comparable to the proposed approach. The worst value is obtained for *Clus. Greedy* approach.

Table 5.2: Paired t-test result for comparing the number of solutions obtained. The paired comparison is performed with respect to the proposed approach. Higher the value larger the difference (negative value for Greedy shows the better result as compared to the proposed approach.

Budget	**Random**	**Greedy**	**Clus. greedy**
$100	167.96	0.0	388000.0
$200	246.68	-6208.51	19660.29
$300	592.72	3104.18	41906.55
$400	574.47	-310400.0	1086400.0

Table 5.3: Paired t-test result for comparing the un-utilized budget. The paired comparison is performed with respect to the proposed approach. Higher the negative value larger the difference.

Budget	**Random**	**Greedy**	**Clus. greedy**
$100	-41.16	-13478.33	-167051.63
$200	-44.99	-6029.97	-19336.02
$300	-64.35	-11772.28	-8982.27
$400	-29.70	-177.44	-15849.90

5.7.2.3 Effectiveness of diversity

The key to find the most suitable hosts for a WA is to increase the diversity of the hosts in the generated solutions. Figure 5.8 shows the diversity of results computed by different algorithms with the budget $400, where the yellow dots are the available hosts and the triangles represent the hosts selected by different algorithms.

The results (Figure 5.8d) clearly show that our proposed approach is scattered better than others. *Random Selection approach* also provides good scatter but the total number of solutions is 68% less the proposed algorithm. *Greedy* approach has the worst diversity and the total number of solutions is 85% less than the proposed approach. *Clus. greedy* has better diversity, compared to *Greedy* approach. However, there are no solutions selected for smaller size host configurations as shown in Figure 5.8c. The reason behind this is that the total budget is distributed according to the average computation cost only. The communication is established only if the solution is valid and selected which can not be predicted beforehand. Since GWA have high data communication cost, none of the solutions can be selected in the allocated budget. This results in a selection of average 75 % fewer host configurations as compared to the proposed approach.

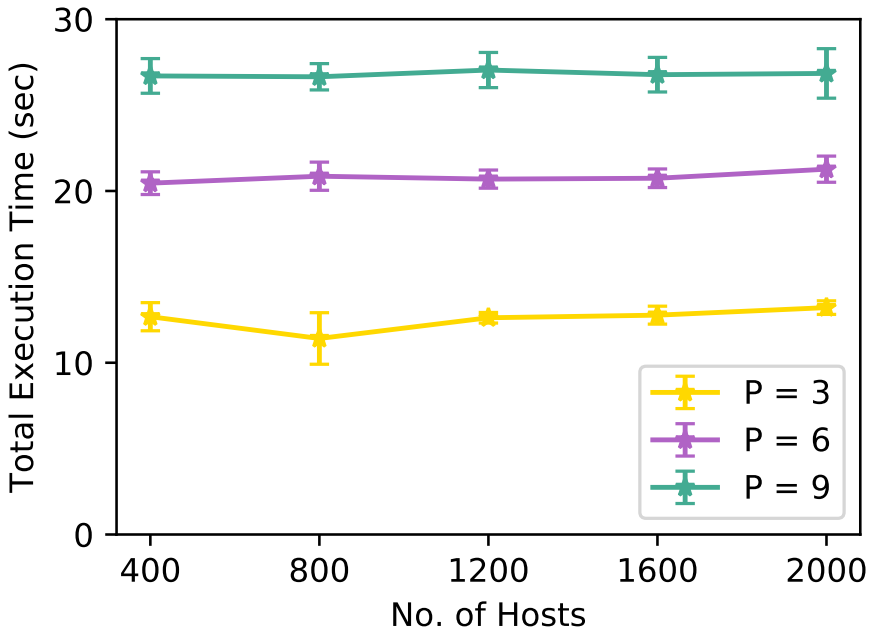


Figure 5.10: Total execution time obtained by varying the number of hosts and replicas for GWA

5.7.3 Scalability test

It is necessary for our proposed approach to scale with the increasing number of hosts and replicas of GWA components. We evaluated the scalability of our proposed approach for three GWA cases a) Problem Size $P = 3$, b) Problem Size $P = 6$ and c) Problem Size $P = 9$. The number of cloud providers is set to 5. We increase the number of hosts varying from 200 to 2000. To maintain consistency, we increased the Budget for each case such that $\frac{Budget}{ProblemSize} = 50$ in every case. We also set the number of datacenter geo-locations to be 12 so that a valid solution is always generated.

Figure 5.10 depicts the result obtained. The figure clearly shows that the total execution time does not significantly increase with increasing the number of host configurations for any case. The maximum increase is visualized for P = 3 with a value of 6 % as compared to the average value. The figure also shows that the execution time increases with the increase in the problem size. With the increased problem size, the algorithm complexity increase as first it needs to search more elements, second for each new element the validity is tested and finally, the fitness function is computed. However, the total execution time does not increase exponentially with the increasing

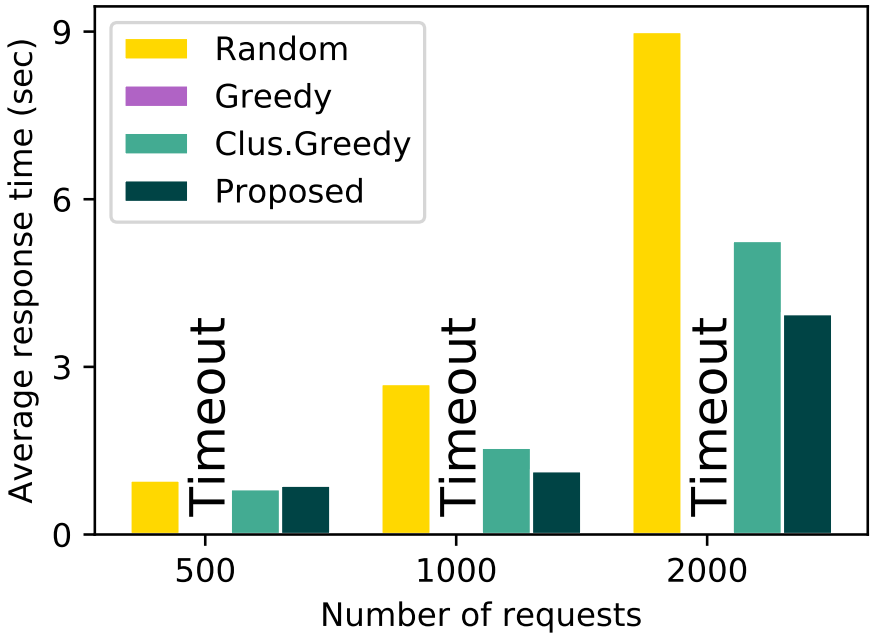


Figure 5.11: Average response time obtained by executing the methods for text data only. *Timeout* represents the requests are not responded.

problem size.

5.7.4 GWA execution in real cloud environments

To evaluate the advances of our proposed algorithm, we deploy all the solutions generated from different algorithms in §5.7.2.2 with budget $100 on the real cloud environment using the *benchmark orchestrator* as discussed in §5.4. In each experiment, we evaluate the solutions by using both *text data* and *image data* and computed its response time.

Figure 5.11 shows the average response time obtained by executing all the solutions from different algorithms. The result clearly shows that the proposed algorithm outperforms others with an average of 3.3× and 1.3× less response time compared to *Random* and *Clus. greedy* respectively. One point to notice here is that *Greedy* approach results to *timeout* in all the cases as the host size is very small and is not able to handle even 500 requests.

Similar trend is observed for image data as shown in Figure 5.12. In general, the proposed method achieve less response time, i.e. 2.1× and 1.3× less, compared to

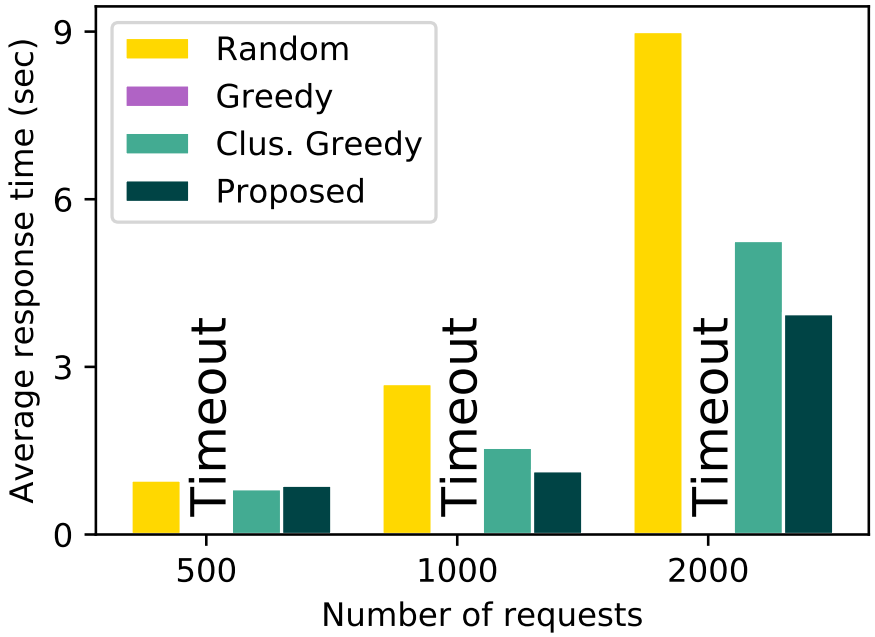


Figure 5.12: Average response time obtained by executing the algorithms for image data. *Timeout* represents the requests are not responded.

Random and *Clus. Greedy* respectively.

5.8 Related work

Web-Application benchmark. To evaluate the performance of WA, many benchmark tools have been developed and used by academic and industry [25, 49, 116, 135, 138, 158]. TPC-W [116], TPC-E [138] and SPECWeb2009 [25] are the are some traditional benchmarks for assessing the performance of WA. However, these benchmarks are not suited for Web 2.0 due to the lack of interactive content and support of mobile users. CloudStone [135] is an open-source toolkit supporting social Web 2.0 application but the implementation of the client-side is very simple, which can not emulate the real-world workloads. To provides a realistic client-base realization, WPBench [158] and BenchLab [49] leverage the web browsers to emulate the users visiting the web sites. However, benchmarking a WA requires to run the benchmark tools on different computing resources while assessing the performance of these resources based on the collected results. This process is complex, error-prone and should be automated. GeoDeploy is designed to orchestrate the benchmark tool, running on multiple cloud

data centers automatically and cost efficiently.

Benchmark orchestrator. Various orchestartion frameworks are available for orchestrating benchmark applications. [51, 54, 57, 78, 109, 128]. SmartDBO [78], Cloud WorkBench [57, 128] and Smart CloudBench [51] allow orchestrating WA benchmark for different cloud providers. Cloud WorkBench [128] allows a reusable benchmark definition leveraging *Chef* and provisioning the benchmark on heterogeneous cloud. However, the benchmarking definition is complex. Smart CloudBench [51] not only automates the benchmark provisioning but also provides a comparison between various cloud providers. However, both these works ignore the benchmarking cost and time which is necessary to consider while benchmarking in cloud environment. SmartDBO [78] orchestrates the benchmarking process with a simple user interface allowing users to interact. It also considers the benchmarking cost and time while selecting the provisioning process. However, none of them can support the benchmarking of GWA.

Geo-distributed orchestration system. Orchestrating the application in geo-distributed manner have been well studied [74, 96, 107, 150–152]. [150, 152] focuses on optimizing the deployment of scientific workflow across federated clouds. These applications have well-defined computation logic, which is easy to be optimized and deployed. Orchestrating or scheduling big data systems in a geo-distributed environment is a challenge. [72, 74, 87, 96, 107, 117, 143] develop various orchestration and optimization algorithms to efficiently run big data systems across multiple cloud data centers while considering the limitation of bandwidths. These systems are not sensitive to the response time for users, compared to WA, and therefore do not consider the latency caused users' requests are submitted from a different location. Our GeoDeploy is the first system that tackles the challenge of benchmarking GWA, which aims to maximize users' satisfaction (e.g. low response time). The users may send their requests from all over the world.

5.9 Discussion

Experiment results highlight the performance of GeoDeploy. Numerical analysis

shows that GeoDeploy can generate up to 72.5 % more number of solutions as compared to the baseline approaches in a given budget. The results also show that the solutions generated are more diverse as compared to any other approach. In terms of scalability, APSO-based optimization approach is scalable with an increasing number of cloud host, however, a slight increase is noticed when the size of GWA components increases. Finally, the real-cloud experiment shows GeoDeploy can lead to a solution with up to 2.1× better response time as compared to the other baseline approaches.

Limitations. While the experiment results encourage the benchmarking of GWA, there are few other aspects to be considered. Sine the cloud environment is very dynamic, to analyze the exact performance variation benchmark execution needs to be performed for a longer duration. Executing benchmarks for a longer duration is very costly and applicable. [79] used a flexible execution that schedules the benchmark for a longer duration while actually executing for a very short time period thus, covering the long-term variation in a defined budget. Developing similar techniques to adapt GeoDeploy's execution plan under the defined budget is part of our future work plan.

5.10 Conclusion

In this chapter, we consider the problem of finding a suitable deployment option for GWA in a multi-cloud environment using benchmarking. We argue that, in order to find the best deployment solution, it is necessary to *increase the diversity of hosts* while maintaining the *dependency of GWA components* for benchmarking. We design GeoDeploy, a novel GWA benchmarking orchestrator that incorporates a variety of novel heuristics for the above-mentioned issues. We implement GeoDeploy with AWS, Azure and Google cloud platform but can be easily extended for any other CPs. Experiment results confirm that GeoDeploy outperforms the baseline methods in simulation as well as real-cloud experiments thus provide better user response time.

6

Deployment of streaming application microservices in cloud-edge environment

Contents

Summary

This chapter presents a heuristic model for the deployment of streaming application microservices with multiple conflicting criteria in cloud-edge environment. It does so by leveraging a well known multi-criteria decision-making method Analytic Hierarchical Processes (AHP) and a basic deployment framework PATH2iot. The applicability of the deployment model is validated using a real-world digital healthcare analytics use case. The results show that our model is able to find the optimal deployment solution with varying user preferences.

6.1 Introduction

Advances in IoT technology are transforming the society and the economy through their widespread impact, e.g. smart homes, smart healthcare and smart agriculture. This will continue to grow: research by Cisco predicts that 50 billion smart devices will be connected to IoT by 2020 [52]. To extract and process the massive streaming data collected from these data sources, several stream processing engines are developed that run in cloud providing common programming frameworks e.g. Apache Storm [3], Amazon Kinesis [2].

However, this cloud-based approach is not suitable for many critical IoT applications for three main reasons. Firstly, some applications require close coupling between the IoT data generators and actions taken based on the analysis of the data [155]. For these applications, the centralized cloud-based analytics approach might introduce unacceptable message transfer delays, and there may be a major problem due to network failure. Secondly, sending all the raw data from sensors to the cloud for analysis is not possible in cases where it may require higher bandwidth than is available or affordable [132]. Thirdly, sending all data to the cloud may flatten the battery of devices such as healthcare wearables too quickly [110].

To address these challenges, an alternative approach is to run part of the application on, or close to, the sensors that generate the data. This approach has been made possible by the introduction of what has become known as edge devices. The development of

smarter IoT and edge devices, with some local storage and processing (e.g. healthcare wearables) opens up a tremendous opportunity for local analytics. Smartphones and field gateways can perform some basic analytic operations on the data, as well as acting as a network bridge between IoT devices and the cloud [42, 139]. As given in [110], executing a subset of application microservice on/near to IoT device increases the IoT device battery hours up to 4× as compared to the entire execution on cloud.

In the literature, an IoT application can be represented using a set of queries (Q_is) which can be modelled as a directed acyclic graph (DAG) with data transformation operations (O_is) as its nodes, and dataflow dependencies (or control flow dependencies for computational synchronization, if/as needed) between data transformation operations O_i representing microservices as its vertices (see Figure 6.1) [114]. Unfortunately, distributing applications across such a wide range of infrastructure (clouds, edge devices, sensors) (see Figure 6.1) is an extremely challenging task for a systems programmer, especially for critical IoT applications such as in healthcare and city management. Key challenges include:

- *heterogeneity of IoT infrastructure* - applications need to be deployed across a wide variety of heterogeneous platforms with differing programming interfaces and capabilities (e.g. sensors may have very limited computational capabilities). Data must be communicated between them over a variety of networks, each with its own idiosyncrasies and limitations.

- *meeting multiple, non-functional requirements* - optimizing several requirements is challenging as the requirements may be conflicting to each other e.g. performance, energy, cost, dependability.

These challenges leads to the following research questions:

RQ 1. How to model the problem of mapping complex IoT application across distributed IoT infrastructure with heterogeneous hardware and software configurations?

RQ 2. How to compute an optimal set of hardware and software configurations for each micro-operations/microservices of an IoT application considering conflicting non-functional requirements?

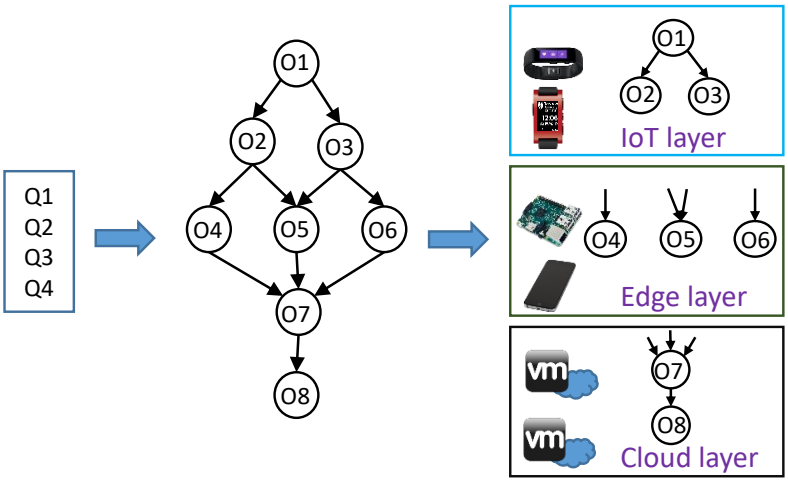


Figure 6.1: Distributed deployment of IoT application

Early efforts centered on the deployment of IoT applications across cloud and edge datacenters are mostly theoretical. Moreover, these solutions have not considered automatic computation partitioning and deployment. Nor have they considered the optimization of multiple conflicting non-functional requirements during the deployment process. Michalak et al. developed *PATH2iot* framework [110] that decomposes a complex IoT application into self-contained micro-operations/microservices. Based on the deployment criteria, *PATH2iot* automatically distributes the set of micro-operations across IoT infrastructure platforms, while respecting their run-time data and control flow dependencies. However, this work does not consider the multiple non-functional requirements for making deployment decisions.

6.1.1 Contributions

To address the limitations of existing works, this chapter makes the following new contributions:

1. We provide a formal model for the deployment of a general IoT application across a distributed IoT infrastructure (e.g. sensors, edge devices, cloud).

2. We prove that the partitioning and deployment of an IoT application across a distributed IoT infrastructure is a strong NP-hard problem.

3. We introduce a new heuristic model *ABMO (AHP Based Multi-objective Optimization)* based on Analytic Hierarchical Processes (AHP) [124] for finding the optimal deployment plan taking into account multiple, potentially-conflicting non-functional requirements. This includes user preferences for making decisions based on a set of non-functional requirements. *ABMO* works on the top of *PATH2iot* framework.

4. A comprehensive experimental evaluation is carried out using a real-world, digital health-care scenario for verifying the performance of the proposed decision-making technique.

Outline. The rest of this chapter is organized as follows. A formal model for the proposed framework is presented in §6.2. It also discuss the complexity analysis of the deployment problem. §6.4 describes the system model of the proposed framework. §6.5 evaluates the proposed framework on the real-world healthcare IoT application. §6.6 discuss the relevant related work. Before discussing conclusion in §6.8, §6.7 highlight the results and present the limitations of this work.

6.2 Formal model

In this section, we give some basic concepts and formally define our problem.

6.2.1 Basic concepts

Definition 1. An IoT application A is a triple $\langle DS, Q, \Gamma \rangle$ where

1). DS represents a continuous stream of data generated by the IoT device.

2). $Q = \{q_1, q_2, ..., q_k\}$ is the set of k queries defined by the user as a description of the computation. The set Q is logically decomposed using a stream optimization function $\mathcal{P}()$ into a set of computational micro-operations O, which need to be deployed on the processing elements, as given in equation (6.1).

$$O = \{o_1, o_2, ..., o_l\} = \mathcal{P}\{q_1, q_2, ..., q_k\} \tag{6.1}$$

The dependency among various micro-operations is represented by a topologically ordered DAG. Each micro-operation o_l has specific hardware and software requirements R_H and R_S respectively. Some constraints $Cons$ are also associated with the requirement specification.

3). Γ represents the identity property of the application and is represented as $\langle id, r_H, r_S, cons \rangle$ where id is the identifier of the application, r_H and r_S are the set of hardware and software requirements and $cons$ is the set of constraints defined for the hardware/software requirement of the application.

Definition 2. An IoT infrastructure I is a quadruple $\langle D, E, C, \lambda \rangle$ where,

1). D is the set of IoT devices d, each represented by a set of 4-tuple $\langle id, Type, S_H, S_S \rangle$ where id is the identifier, $Type$ represents the type of device and S_H and S_S respectively represents the hardware and software support provided by the device d.

2). E is the edge datacenter consisting set of edge devices e, each denoted by $\langle id, S_H, S_S \rangle$ where id is the identifier, S_H and S_S respectively represents the hardware and software support provided by the edge device e.

3). C is the set of cloud datacenter components (VMs or containers) c, each denoted by $\langle id, S_H, S_S \rangle$ where id is the identifier, and S_H and S_S respectively represents the hardware and software capabilities provided by c.

4). $\lambda \in \{\{D \times E \times C\} \cup \{D \times E\} \cup \{D \times C\} \cup \{E \times C\}\}$ is a set of all the available connections from IoT device to edge to cloud.

Definition 3. The set R of non-functional requirements is a sequence of r elements $R = \{R_1, R_2, ..., R_r\}$ where each element R_i can have either numeric or boolean values. Unspecified values of element R_i are denoted by $\emptyset$.

Numerous non-functional requirements may be associated with the application. Our approach is general, but those considered in this chapter are discussed in §6.3. Figure 6.2 shows the hierarchical representation for all the non-functional requirements considered in this chapter.

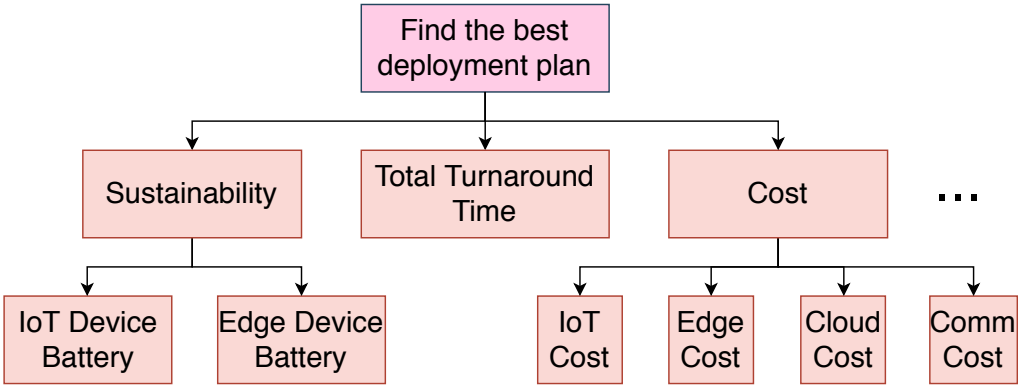


Figure 6.2: Non-functional requirements

6.3 Non-functional requirements

The details of different non-functional requirements considered in this chapter are as follows.

Sustainability. This considers the energy impact in terms of the battery life of all the IoT and edge devices. Battery life is very important to consider as it can be a limiting factor: for example, healthcare wearables that don't last a full day on a single charge of the battery are not going to be used in practice. Battery power is represented in terms of Energy Impact (EI) which is measured in Milli Joule (mJ). As an example, the energy impact for a BLE connected IoT device EI_D and edge device EI_E is calculated as given in equation (6.2) and equation (6.3).

$$EI_D = OS_{idle} + \sum_{i}^{n} c\_cost_i + \frac{msg\_count_D \times n\_cost_D + BLE_{active} \times BLE_{dur}}{cycle\_length} \tag{6.2}$$

$$EI_E = OS_{idle} + \sum_{j}^{n1} c\_cost_j + \frac{msg\_count_E \times n\_cost_E + RF_{active}}{time\_slice} \tag{6.3}$$

Where, OS_{idle} is the power consumption by the IoT device/edge device caused by the operating system, c_cost and n_cost are the overall power consumption of the IoT device (D) and edge device (E) for each computation and transmission respectively, msg_count specifies the number of messages transmitted from IoT device to edge device or edge device to cloud device, BLE_{active} is the power consumption for the Bluetooth

low energy active state that is activated just after the message transfer, BLE_{dur} is the period of BLE_{active} state, *cycle_length* is the time duration after which the whole cycle repeats and *time_slice* is the particular period of time considered for edge device.

The energy impact value represents the average power consumed by the IoT device and the edge device. The battery life is inversely proportional to the energy impact. The battery life in hours (bat_D) for IoT device D is computed as given in equation (6.4).

$$bat_D \propto (EI_D)^{-1} \Rightarrow bat_D = \beta_D \times (EI_D)^{-1} \tag{6.4}$$

Where, β_D is a constant called $maxBat$ that depends on IoT device D and is given by equation (6.5) where $bat(C)$ is the battery capacity and $bat(V)$ is the battery voltage.

$$maxBat = bat(C) \times bat(V) \times 3.6 \tag{6.5}$$

Similarly, for an edge device, the battery life in hours bat_E is calculated in terms of energy impact EI_E and a device-specific constant β_E as given in equation (6.6).

$$bat_E = \beta_E \times (EI_E)^{-1} \tag{6.6}$$

Total Turnaround Time. The raw data generated by the accelerometer is partially processed by the IoT device and is then transferred to edge devices for further processing. The final processing takes place in a cloud datacenter which also saves the result for further processing (e.g. for cross-population analytics that can generate better predictive models). The total turnaround time T^3 is given by summing the time taken by IoT device T_D, edge device T_E and cloud datacenter T_C from data generation to data computation and is given by equation (6.7).

$$T^3 = T_D + T_E + T_C \tag{6.7}$$

Each layer performs some operation and then sends the data to the upper layer. The time taken by IoT device T_D is given in equation (6.8). The total time is equal to the cycle length as it includes both computation time and time to send data to the edge device.

$$T_D = cycle\_length \tag{6.8}$$

The time taken by edge and cloud datacenter are given in equation (6.9a) and (6.9b) respectively.

$$T_E = T_E(comp) + T_{E \to C}(trans) \tag{6.9a}$$
$$T_C = T_C(comp) \tag{6.9b}$$

Where, $T_{E \to C}(trans)$ is the transmission time from edge to cloud and $T_x(comp)$ is the computation time taken by either edge device or cloud datacenter. For the sake of simplicity, we are not considering any waiting time or queueing time at any part of the IoT infrastructure.

Cost. Performing the operations on either IoT device, edge or cloud datacenter incurs some cost in terms of electricity charge, set up cost, cloud VM cost or storage cost. There is an additional cost associated with the data transfer. The total cost ($Cost_{Total}$) is given by the sum of the cost incurred by an IoT device ($Cost_D$), edge device ($Cost_E$), cloud datacenter ($Cost_C$) and communication cost ($Cost_{comm}$). The cost incurred by an IoT device is determined in terms of electricity cost in charging the device plus a fixed set up cost. The electricity cost depends on the power consumed (Energy Impact) by the device and the per unit electricity rate ρ_{elec}. The value of $Cost_D$ is given in equation (6.10a). Similarly, the cost for an edge device depends on the setup cost and electricity cost as given in equation (6.10b). The cost for cloud datacenter can be given in terms of launching cost and the processing and storage cost of a VM as given in equation (6.10c).

$$Cost_D = (EI_D \times \rho_{elec}) + Cost_D(set_{up}) \tag{6.10a}$$
$$Cost_E = (EI_E \times \rho_{elec}) + Cost_E(set_{up}) \tag{6.10b}$$
$$Cost_C = Cost_C(VM_{proc}) \tag{6.10c}$$

Data is transferred from the IoT device to the edge, and from the edge device to the cloud. For an IoT device, the data transfer costs drain the battery (see sustainability above) but the data transfer from edge device to cloud datacenter costs not only in terms of the edge device's battery charge cost but also the network charge e.g. for transferring data over a 3G or 4G network. The communication cost is calculated by

multiplying the quantity of data transferred (msg_count_E) with the data rate charge (ρ_{data}) as given in equation (6.11).

$$Cost_{comm} = (msg\_count_e + msg\_header) \times \rho_{data} \tag{6.11}$$

6.3.1 Problem definition

Definition 4:. Let $A = \langle DS, Q, \Gamma \rangle$ be an IoT application and $I = \langle D, E, C, \lambda \rangle$ be the IoT infrastructure. A possible deployment plan Δ_i is a mapping from operation $O : \mathcal{P}(Q)$ to $\lambda \in \{\{D \times E \times C\} \cup \{D \times E\} \cup \{D \times C\} \cup \{E \times C\}\}$ ($\Delta_i = O \rightarrow \lambda$) if and only if:

1. all $o_j \in O$ must be mapped to some IoT infrastructure $I_k \in I$.
2. for each $o_j \in O$, $I_k \in I$, if $(o_j \rightarrow I_k) \in \Delta$ then $R_H(o_j) \preceq S_H(I_k)$, $R_S(o_j) \preceq S_S(I_k)$ and $satisfied(Cons(o_j))$
3. $\sum_{j=1}^{max} R_H(o_j) \leq S_H(I_k)$ and $\sum_{j=1}^{max} R_S(o_j) \leq S_S(I_k)$.

The definition given above considers all the constraints to meet the optional deployment requirements. Condition (1) guarantees that all the operations must be deployed on some IoT infrastructure. Condition (2) allocates the operations o_j only to infrastructure I_k, which satisfies their hardware requirements $R_H(o_j)$ and software requirements $S_H(o_j)$, along with any other deployment constraints $Cons$ defined for the operation o_j. Condition (3) limits the number of operations to be deployed on an infrastructure component so that the hardware and software requirements are enough to satisfy the demands of selected operations $o_j | j \in \{i, max\}$.

Definition 5: Given the available possible plans Δ, find the best possible plan $\Delta_{best} \in \Delta$ that optimizes all non-functional requirements such that for any other plan $\Delta_i \in \Delta$, $\Delta_i \leq \Delta_{best} | \forall R_l \in R$ and $\Delta_i < \Delta_{best} | \exists R_l \in R$.

6.3.2 Complexity analysis

Given an IoT application A and IoT infrastructure I, finding the optimal deployment plan Δ_{best} from the set of possible plans Δ that optimizes all R non-functional re-

Table 6.1: A summary of symbols and abbreviations used in the chapter

Symbol	Explanation
A	IoT application
DS	Data Stream
Q	Set of queries q_k
Γ	Identity property of the application
O	Set of operations o_l
$\mathcal{P}()$	Stream optimization function
R_H	Hardware requirements of micro-operation o_i
R_S	Software requirements of micro-operation o_i
$Cons$	Constraints for micro-operation o_l
id	Identifier of the component A
r_H	Hardware requirements of application A
r_S	Software requirements of application A
$cons$	Constraints for application A
I	IoT infrastructure
D	Set of IoT devices d
E	Set of edge devices e
C	Set of cloud datacenter components c
S_H	Hardware support
S_S	Software support
λ	Connection between different IoT components
R	Set of non-functional requirements R_i
ϕ	Unspecified values for R_i
Δ	Set of mappings Δ_i for Operation O to λ
$\preceq$	Satisfied by
P_L	Set of logical plans
P_{OD}	Set of all possible deployment plans
P_{PD}	Set of all physical deployment plans
M	Comparison Matrix
CR	Consistency Ratio
CI	Consistency Index
RI	Random Index
eig	Maximum eigen Value of comparison matrix M
W_j	Weight for non-functional requirement j
$\mathcal{N}$	Number of plans to shortlist
EI	Energy Impact
T^3	Total Turnaround Time

quirements is strong NP-hard and can be proved by reduction from the Bin-Packing problem.

Bin-Packing is known to be a strong NP-hard problem, which is non-solvable in any polynomial time [63]. It is defined as: given a set of o objects with sizes $s_1, s_2, ..., s_o$

and a set of bins $B_1, B_2, B_3, ...$ of same capacities C, find the smallest integer $k \in N$ such that all the o objects got mapped to some bins B_i following the condition that for any $i = \{1, 2, ..., k\}$, $\sum_{i \in B_k} s_i \leq C$.

Proposition 1: Finding an optimal mapping for the deployment of streaming application in cloud-edge environment is NP-hard.

Proof: Considering the formal definition of the Bin-packing problem as given above, it is possible to transform the Bin-Packing problem into the simplest deployment problem in polynomial time. The transformation is as follows.

Change all the bins B_i to IoT infrastructure deployment nodes I (IoT device, edge device or cloud datacenter components) with equal hardware capacities and no software support. Change all the objects o to operations O with s_o hardware requirements, no software requirements, no constraints and no dependency between operations.

This maps the Bin-packing problem into the simplest case of our deployment problem. This transformation can be easily achieved in polynomial time. Thus, proves that the simplest case of our problem is at least as hard as the Bin-packing problem which is already strong NP-hard, making the generic streaming application deployment problem $\in$ strong NP-hard.

Inherently, as given in proposition 1, finding a solution of the Bin-packing problem in polynomial time leads to finding a solution of the generic streaming application deployment problem in polynomial time. No such algorithm exists for any NP-hard problem, therefore we need a heuristic algorithm to find a solution.

6.4 System model

Our proposed framework is built on top of the *PATH2iot* framework. Figure 6.3 summarizes the internal processing of our proposed framework. The framework is divided into three components. The first is the *User Input* that accepts a set of EPL queries which defines – in high-level terms - the computation, the state of the deployment infrastructure, and the non-functional requirements to be placed on the system. The second is the *PATHfinder* implementation, where all the deployment decisions are performed automatically, while the third is the *PATHdeployer* that performs the physical

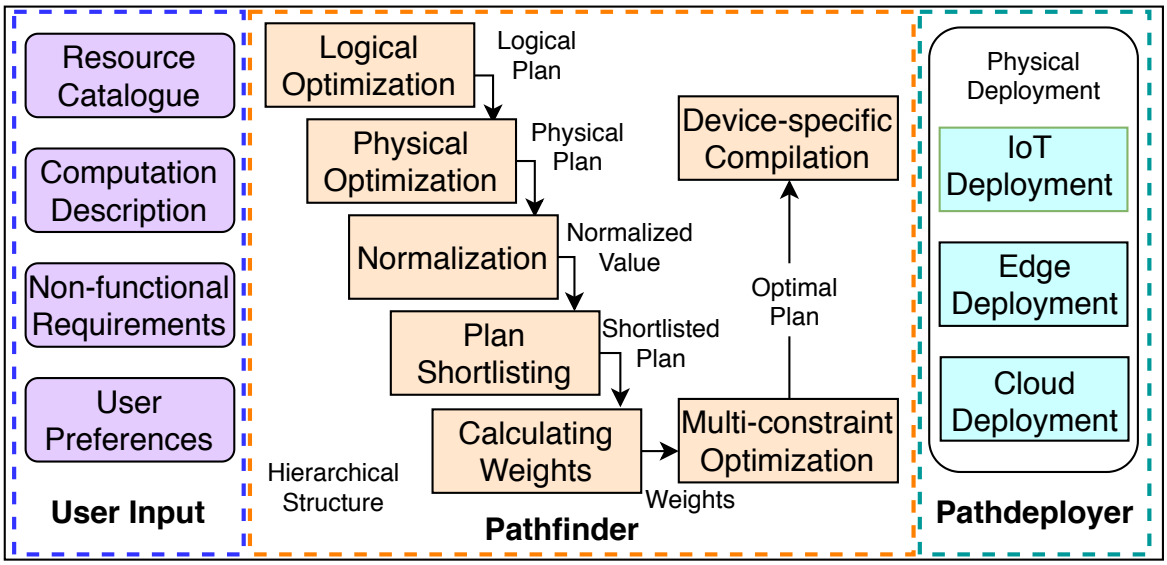


Figure 6.3: Holistic representation of the deployment plan

infrastructure deployment. *PATHfinder* is again divided into three stages, namely, *Initial Optimization*, *ABMO* and *Device Specific Compilation*. A detailed discussion of each component is given below.

6.4.1 User Input

The whole system is driven by the following inputs:

Resource Catalogue. It holds a description of all the relevant features of the IoT infrastructure platforms over which the computations can be distributed. This includes the infrastructure capabilities in terms of hardware support S_H and software support S_S. It represents the state of the infrastructure, i.e. a description of the available cloud resources, IoT and Edge devices with their current state (e.g. active/disabled, battery level, battery capacity, network bandwidth). It also holds the constraints that need to be recognized when making deployment decisions. This includes a definition of User Defined Functions (UDF) that are supported by the system with their placement constraints, along with the Energy Impact coefficients for the supported operators. The optimizer accesses this information in the form of a JSON file.

Computation Description. To allow automatic partitioning over a set of platforms, it requires the computation to be defined in a high-level declarative language which can be analyzed, distributed and optimized. In this chapter, we adopt the approach

of [110] and use a Complex Event Processing (CEP) based relational model in which an Event Processing Language (EPL) is used to define the computation.

Non-functional Requirements. This defines all the requirements that are required to be optimized. The set of non-functional requirements considered in this paper are discussed in detail in §6.3. The non-functional requirements are represented by a top-down hierarchical structure of level L where lower-level elements are grouped under some higher-level elements based on a common property, e.g. lower-level attributes IoT cost, Edge cost and Cloud cost are grouped as they all calculate the cost.

User Preferences. This is one of the key user inputs as it defines the relative importance of the non-functional requirements. A user submits pairwise comparison values for all the non-functional requirements in the form of a CSV file.

6.4.2 PATHfinder

PATHfinder is an internal module of the proposed framework and is divided into three stages as explained below.

6.4.2.1 Initial Optimization

This stage is divided into two consecutive phases *Logical Optimization* and *Physical optimization*. The details are given below.

1) Logical optimization. The high-level user description for the computation is decomposed into a DAG, and the operators of the DAG are topologically sorted. Therefore, each operation is executed in a valid sequence. Various stream optimization techniques can be used to optimize the DAG [**?**]. A list of topologically sorted logical plans P_L is created that acts as input for the next step.

2) Physical optimization. This step generates a set of physical deployment plans based on the logical plans P_L generated by the previous step. For each logical plan, it first creates all possible deployment plans P_{OD} based on the topological ordering of the operations.

Algorithm 6 is used to shortlist the plans that satisfy the constraints and non-functional requirements and generates a list of physical deployment plans P_{PD}.

Algorithm 6: Physical Optimization

Input: P_{OD} – list of optional deployment plans, R_H – hardware requirements, S_H – software requirements, $Cons$ – defined constraints, I – IoT infrastructure
Output: P_{PD} – list of physical deployment plans

```
for all p ∈ P_OD do
    if (R_H(p) ⪯ R_H(I) && S_H(p) ⪯ S_H(I) && Satisfied(Cons)) then
        ADD P_OD to P_PD
    end
end
```

Table 6.2: Satty scale for assigning the priority value

Value	Priority Scale	Value	Priority Scale
1	Equal	7	Demonstrated
3	Moderate	9	Extreme
5	Strong	2, 4, 6, 8	Intermediate

6.4.2.2 AHP Based Multi-objective Optimization (ABMO)

This stage uses the Analytic Hierarchical Process (AHP) [124] for calculating the rank of each physical deployment plan. The whole process is divided into four steps as given below.

1) Calculating Weights. This step uses AHP to first find the weights for each non-functional requirements, based on user preferences. The preferences are measured by using Satty scale [**?**] as given in Table 6.2. A reciprocal comparison matrix M is constructed from the user-preferences following the rules as given in equation (6.12).

$$M_{ij} = \begin{cases} 1, & \text{when } i = j \\ X, & \text{when } i > j \} \\ 1/M_{ij}, & \text{when } i < j \end{cases} \tag{6.12}$$

Before calculating the weights of the non-functional requirements, it is necessary to check whether the provided user-preference values are consistent or not. The consistency of comparison matrix is verified by checking the consistency ratio (CR) value. CR is calculated using the maximum eigen value (eig), size of the comparison matrix ($M.size$) and the Random Index (RI) values. The default RI values are given in the Table 6.3. Equation (6.13) is used to calculate the CR.

$$CR = CI/RI \tag{6.13}$$

Table 6.3: Random Index (RI) value

Size	1	2	3	4	5	6	7	8	9	10
RI	0	0	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49

Algorithm 7: Calculating Weights

Input: $Pref_{jk}$ – preference of jth non-functional requirement over kth non-functional requirement, R_r – list of r non-functional requirements, M – Reciprocal comparison matrix

Output: W – non-functional requirements weight

construct M using $Pref_{jk}$ following equation (6.12)
if (!$Consistent(M)$) **then**
    Notify user to enter new values
    return -1
else
    W = average (principle eigen vector (M))
end
`Consistent(`M`)`
$eig = max(eigenvalue(M))$
$CI = (eig - M.size)/(M.size - 1)$
$CR = CI/RI$ |RI is the Random Index as given in Table 6.3
if ($CR < 0.1$) **then**
    Matrix M is consistent
    return TRUE
else
    Matrix M is inconsistent
    return FALSE
end

where, $CI = (eig - M.size)/(M.size - 1)$. If the comparison matrix M is found to be consistent, AHP is used to calculate the final weights W_j for non-functional requirement j. Otherwise, the user is instructed to re-enter the preference values. AHP uses principal eigenvector to calculate the priority of each non-functional requirement. The final weights are computed by averaging the priority values. The pseudo-code for the whole process is explained in Algorithm 7.

2) Normalization. Directly comparing different non-functional requirement values is not possible as each requirement can have a different data-type and range. Also, the optimal value depends on the type of requirements: in some cases, higher is better e.g. Battery Power, while in others lower is better e.g. Cost. It is necessary to normalize all these values to one type and range.

The normalization is performed according to the data type of the non-functional requirements e.g. boolean, numerical, etc. and the function whether to maximize or

Algorithm 8: Normalization

Input: P_{PD} – list of physical deployment plans, R_r – list of r non-functional requirements, $type_{R_j}$ – data type of the non-functional requirement R_j, $Option_{R_j}$ – option for the non-functional requirement R_j to be maximized or minimized, $R_j(P_i)$ – value of jth non-functional requirement for ith plan

Output: $norm(R_j(P_i))$ – normalized value of jth non-functional requirement for ith plan

initialize $norm(R_j(P_i))$ to 0 for all $R_j(P_i)$
for *each* $R_j \in R_r$ **do**
  if *(*$type_{R_j} == Numeric$*)* **then**
    if ($Option_{R_j} == maximize$) **then**
      $Sum(R_j) = \sum_i R_j(P_i)$
      $norm(R_j(P_i)) = R_j(P_i)/Sum(R_j)$
    else if ($Option_{R_j} == minimize$) **then**
      $Sum1(R_j) = \sum_i R_j(P_i)$
      $norm1(R_j(P_i)) = Sum1(R_j)/R_j(P_i)$
      $Sum(R_j) = \sum_i norm1(R_j(P_i))$
      $norm(R_j(P_i)) = norm1(R_j(P_i))/Sum(R_j)$
    else
      option is not valid
    end
  end
  if *(*$type_{R_j} == Boolean$*)* **then**
    if ($Option_{R_j} == maximize$) **then**
      $Sum(R_j) = \sum_i R_j(P_i)$
      $norm(R_j(P_i)) = R_j(P_i)/Sum(R_j)$
    else if ($Option_{R_j} == minimize$) **then**
      swap 0 with 1
      do same as maximize
    else
      option is not valid
    end
  end
  if *(*$type_{R_j} == Unordered\_set$*)* **then**
    find the maximal set R_{max}
    $norm1(R_j(P_i)) = size(R_j(P_i) \cap R_{max})/size(R_{max}))$
    repeat the same process as in $Numeric$ with $maximize$ using the value $norm1(R_j(P_i))$ in place of $(R_j(P_i))$
  end
  if *(*$type_{R_j} == Numeric\_Range$*)* **then**
    find the optimal Range R_{opt}
    $norm1(R_j(P_i)) = length(R_j(P_i) \cap R_{opt})/length(R_{opt}))$
    repeat the same process as in $Numeric$ with $maximize$ using the value $norm1(R_j(P_i))$ in place of $(R_j(P_i))$
  end
end

minimize. The steps of normalization process are summarized in Algorithm 8.

3) Plan Shortlisting. The complexity of the *Pathfinder* depends on the number of physical deployment plans. To manage the complexity of the *Pathfinder*, we shortlist $\mathcal{N}$ plans from the full list of physical deployment plans. If the total number of the

Algorithm 9: Plan Shortlisting

Input: P_{PD} – list of physical deployment plans, R_r – list of r non-functional requirements, $norm(R_j(P_i))$ – normalized value of jth non-functional requirement for ith plan, $\mathcal{N}$ – maximum number of plans to be generated, $grade(P_i)$ – grade of the plan (P_i)

Output: P_{PPD} – list of shortlisted physical deployment plans

```
initialize grade(P_i) to 0 for all P_i ∈ P_PD
for each P_i ∈ P_PD do
    for each R_j ∈ R_r do
        grade(P_i) = grade(P_i) + norm(R_j(P_i))
    end
end
sort grade(P_i) in descending order
select top N plans and store the list in P_PPD
```

plans is less than $\mathcal{N}$ we can skip this step.

Algorithm 9 explains the steps involved in the pruning process. It first ranks all the plans and finally selects the top $\mathcal{N}$ plans for further evaluation.

4) Multi-constrained Optimization. This step combines the final weights of the non-functional requirement with the corresponding values of the plans to find the final rank of the plans. This step aims to find the optimal plan over the selected plans. The rank of each plan P_i is computed using equation (6.14), where W_j is the weight for non-functional requirement j, and $norm(R_j(P_i))$ is the normalized value for the plan P_i. The plan with the highest rank is selected for the physical execution.

$$Rank(P_i) = \sum_{j=0}^{r} (W_j) \times norm(R_j(P_i)) \tag{6.14}$$

6.4.2.3 Device-specific Compilation

Once the execution plan is derived, the framework converts the selected plan into a deployment configuration which is finally transmitted to *PATHdeployer* for deployment over the available IoT infrastructure.

6.4.3 PATHdeployer

Following PATH2iot [110], there are two stages to deploy the optimized plan: Cloud Deployment and Edge and IoT Deployment. The two stages are detailed below.

Cloud Deployment. is the first phase, where the framework verifies through the ZooKeeper that the destination *D2ESPer* instances are available in the infrastructure and then transmits the deployment configuration to them. The configuration information is then parsed and dynamically compiled EPL statements are executed within the Esper CEP engine, which is wrapped inside the *D2ESPer* tool.

Edge and IoT Deployment. occurs once the cloud deployment has been completed. The configuration information is passed through a REST API endpoint. Pre-installed agents on IoT and Edge devices pull regularly configuration from the endpoint and once it has been received start the processing accordingly.

6.4.4 Time complexity

This section computes the time complexity of our proposed framework. Let o be the number of operations and $\overline{I}$ is the number of IoT infrastructure components. Consider the non-functional requirements R is represented in a hierarchical structure with L number of levels The complexity of each phase is given below.

Path Finder. This phase is divided into three stages, and the complexity of each stage is given below.

a). *Initial Optimization:* There are two steps for this stage, the complexity of *Logical optimization* depends only on the operator reordering operation. For one *Logical optimization*, the maximum complexity of operator-reordering is O(o). Varying the window size between a given range, $R1$ and $R2$ with defined step size *Step* creates $\{(R1 - R2)/Step\}$ plans. Since, $R1$, $R2$ and *Step* is constant, the complexity for creating a set of logical plans, P_L is O(o). For *Physical optimization*, finding all the possible deployment plans for one logical plan takes O($o \times \overline{I}$) searches. Thus, the total complexity for finding all the optional deployment plan P_{OD} for the given set of logical plan P_L is O($P_L \times o \times \overline{I}$). For finding the physical deployment plan P_{PD}, we have to check all the available possible deployment plan making the complexity as O(P_{OD}) = $P_L \times o \times \overline{I}$.

After reduction, the overall complexity for *Initial Optimization* step is O($P_L \times o \times \overline{I}$).

b). *ABMO:* This stage consists of four steps, *Calculating Weights*, *Normalization*, *Plan Shortlisting* and *Multi-constrained Optimization*. *Calculating weights* compares the user preferences for each attribute by using matrix manipulation and computes eigen vector as the priority values. Given n elements, the complexity to calculate the normalized eigen vector is $O(n^3)$. Considering N_{lev} number of attributes at level lev and $r_{sub,lev}$ number of sub-attributes at level lev of subth attribute at level $(lev-1)$, the complexity to calculate the normalize eigen vector is given as $O(\sum_{lev=1}^{L}\sum_{sub=1}^{N_{lev-1}}(r_{sub,lev})^3)$. The complexity value seems large but is comparatively very small when compared with the comparison complexity without using AHP. The total complexity for comparing all the low level elements without using AHP is $(\sum_{sub=1}^{N_0}(r_{sub,1}))^3$ which is $>>$ $(\sum_{lev=1}^{L}\sum_{sub=1}^{N_{lev-1}}(r_{sub,lev})^3)$ when there are a large number of elements which can be represented in a hierarchical structure. AHP reduces the complexity by breaking the problem in a hierarchical structure resulting in more, smaller comparisons.

For the *Normalization* step, the time taken to normalize any non-functional requirement value is $O(P_{PD})$ irrespective of the data type. The complexity to normalize all R non-functional requirements is $O(R \times P_{PD})$. Therefore, the final complexity for Normalization step is $O(R \times P_{PD})$. *Plan shortlisting* step finds the grade for each plan which is $O(R \times P_{PD})$ complex. The time taken to sort the grade values is $O((P_{PD})^2)$ and finally selecting the top $\mathcal{N}$ plan is $O(1)$. Therefore, the final complexity for Plan Shortlisting is given as $O((P_{PD})^2)$. For final step, *Multi-constrained optimization* step, the complexity to calculate the rank for plan is $O(R)$. For all $\mathcal{N}$ plan, the complexity is $O(R \times \mathcal{N}) = O(R)$ as $\mathcal{N}$ is a constant. Finally, the time taken to find the best value from sorted plan is $O(\mathcal{N}) = O(1)$.

Thus the total time complexity for second stage is reduced to $O(\sum_{lev=1}^{L}\sum_{sub=1}^{N_{lev-1}}(r_{sub,lev})^3 + R \times P_{PD} + (P_{PD})^2)$.

c). *Device-specific Compilation* The complexity of this step is constant as there is only one execution plan for the deployment.

Path Deployer. The complexity of this step is also constant as the deployment is always performed for one selected plan.

Thus, the overall time complexity of our proposed framework is summarized as $O((P_L \times o \times \overline{I}) + \sum_{lev=1}^{L} \sum_{sub=1}^{N_{lev-1}} (r_{sub,lev})^3 + R \times P_{PD} + (P_{PD})^2)$.

6.5 Experimental evaluation

In this section, we evaluate the performance of our proposed framework on a real testbed.

6.5.1 Experimental setup

For experimentation, we choose a healthcare based application that captures the physical activity and blood glucose level of a type II diabetes patients [122]. To analyze the activity, we used the Pebble Steel smartwatch that contains an embedded accelerometer. The smartwatch connects to an LG G4 smartphone via a Bluetooth Low Energy (BLE) network and the phone is then connected to the cloud via a 4G mobile network. The data analysis uses a *Step Count* algorithm [160] that accepts the raw accelerometer data and generates activity information. Our proposed framework automatically decides where to place the components of the analysis while optimizing different criteria (maximizing the battery life on the smartwatch and mobile phone, minimizing the turnaround time, etc.). The starting point is a description of the required computation as a set of EPL rules:

1. **INSERT INTO** *AccelEvent*
   SELECT getAccelData(25, 60)
   FROM *AccelEventSource*

2. **INSERT INTO** *EdEvent*
   SELECT Math.pow($x \times x + y \times y + z \times z$, 0.5) **AS** ed, ts
   FROM *AccelEvent* **WHERE** vibe = 0

3. **INSERT INTO** StepEvent
   SELECT * **FROM** *EdEvent*
   MATCH_RECOGNIZE (**MEASURES** A **AS** ed1, B **AS** ed2 **PATTERN** (A B) **DEFINE** A **AS** (A.ed > THR), B **AS** (B.ed ≤ THR))

4. **INSERT INTO** *StepCount*
   SELECT count(*) **AS** steps
   FROM *StepEvent*.win:flexi_time_batch(30, 120, 15, sec)

5. **SELECT** persistResult(steps, 'step_sum', 'time_series') **FROM** *StepCount*

Using PATH2iot [110], the proposed framework can interpret this into a graph of basic operations, and then considers all the possible solutions that map these operations onto the available IoT infrastructure (Pebble watch, mobile phone and cloud). For each option, a cost is derived, and the one that has the maximal value of the non-functional requirements is selected.

In order to calculate the energy consumption of the Pebble Steel watch and the Mobile Phone, we must know the energy consumption of performing each operation (i.e. per sample received from the accelerometer). A series of experiments were conducted in which the watch and the mobile phone was connected to a Monsoon Power Monitor to measure the energy expenditure with each executed operation. Based on Central Limit Theorem (CLT), we assume that the entire data set follows Gaussian distribution. Therefore, we compute the 95% confidence interval ($Conf$) using equation (6.15), where sd and $\bar{X}$ are the standard deviation and mean of the sample and n is the size of the sample. The results are shown in Table 6.4 and Table 6.5. Notably, the Energy Impact shown in the two tables represents the mean of the sample.

$$Conf = \bar{X} \pm 1.96 \times \frac{sd}{\sqrt{n}} \tag{6.15}$$

To calculate the battery life resulting from each option, we need to know the overall capacity of the battery. The battery capacity and battery voltage of Pebble watch and LG G4 are 130 mAh, 3.7 V and 3000 mAh, 3.85 V respectively. The $maxBatteryLife$ of Pebble watch and Mobile Phone β_D and β_E is calculated as $\beta_D = 130mAh \times 3.7V \times 3.6 = 1731.6J$ and $\beta_E = 3000mAh \times 3.85V \times 3.6 = 41580J$.

To calculate the total transmission time from Mobile phone to cloud, we considered the Wi-Fi data rate from our phone (30 Mbps). Also, to calculate the Total Cost, we considered the standard electricity cost (ignoring the average fixed cost) as £0.155 /KWh [5] and standard data-rate cost as £0.01 /MB [22]. We have neglected the VM cost for our experiment as the VM cost is almost the same for all the cases.

6.5.2 Experimental results and analysis

We have compared three scenarios, where different non-functional requirements of the IoT infrastructure have been monitored. The detail of these scenarios are as follows:

Table 6.4: Power Consumption Coefficients for the Pebble Steel Watch.

Operation	Energy Impact (mJ)	95% Conf Int
OS_{idle}	1.78	± 0.0370
25 Hz sampling	0.06	± 0.0153
SELECT	0.09	± 0.0416
ED	0.34	± 0.0665
POW	0.03	± 0.1039
WIN	0.06	± 0.0605
net_cost	5.06	± 0.2747
BLE_{active}	12.12	

Table 6.5: Power Consumption Coefficients for the LG G4 mobile phone.

Operation	Energy Impact (mJ)	95% Conf Int
OS_{idle}	56.28	± 4.520
n_cost	161.62	± 6.813
RF_active	2497.10	± 226.288

1) Baseline. The generated raw accelerometer data (under 25 Hz) is streamed from the Pebble Steel watch to the cloud as quickly as possible. Given the software restrictions, this is done in a batch of 10 accelerometer samples, therefore with a frequency of 2.5 messages per second. This scenario gives the best Turnaround Time but consumes maximum energy.

2) Optimized by energy. The main focus, in this case, is to optimize the energy consumption of IoT devices to increase the battery running hours. This is outlined in [110], where, the optimizer selects the deployment plan where some of the operators are placed on the wearable watch, reducing the amount of data required to be transmitted and pushing windows closer to the data source in this case directly on the wearable device, with a fixed window size of 120 seconds, greatly reducing the energy required to keep the Bluetooth connection opened. The result shows that we achieved a significant improvement in the energy consumption of the wearable device of 453% as compared to the Baseline approach. However, the Turnaround Time in this scenario is higher as compared to the Baseline approach.

3) Optimized by multiple conflicting criteria. Depending on the type of application and user requirements, the module automatically selects the best deployment plan. It can be easily converted to the previous scenarios i.e. Baseline or Optimized by

energy by setting up the user preference highest for Turnaround Time or Sustainability respectively. To illustrate the effectiveness of our proposed plan, we considered three cases with different user-preferences, as given below:

Case 1: In this case, the battery constraints are more important, so making the user-preferences inclined towards sustainability. The comparison matrix, C_1 for level 1 is given below. For other levels, we considered an equal priority for all the attributes.

$$C_1 = \begin{array}{c|ccc} & Sus. & T^3 & Cost \\ Sus. & 1 & 7 & 9 \\ T^3 & 1/7 & 1 & 2 \\ Cost & 1/9 & 1/2 & 1 \end{array}$$

Case 2: In this case, total turnaround time has a higher preference as compared to other non-functional requirements. The comparison matrix, C_1 for this case is given below:

$$C_1 = \begin{array}{c|ccc} & Sus. & T^3 & Cost \\ Sus. & 1 & 1/7 & 2 \\ T^3 & 7 & 1 & 9 \\ Cost & 1/2 & 1/9 & 1 \end{array}$$

Case 3: In this case cost has given higher preference as compared to other non-functional requirements. The comparison matrix, C_1 for this case is shown below:

$$C_1 = \begin{array}{c|ccc} & Sus. & T^3 & Cost \\ Sus. & 1 & 2 & 1/7 \\ T^3 & 1/2 & 1 & 1/9 \\ Cost & 7 & 9 & 1 \end{array}$$

After completing the *Initial Optimization* stage, a total of 108 optimal plans got selected which acts as input for the $ABMO$. After execution of $ABMO$, the final result gives $Plan107$, $Plan9$ and $Plan1$ for the physical deployment in **Case 1**, **Case 2** and

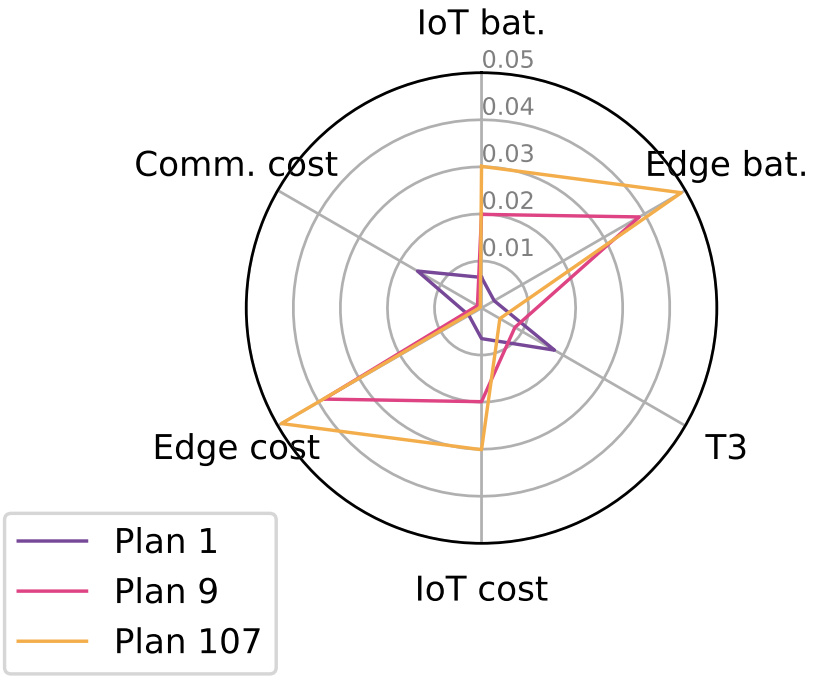


Figure 6.4: Normalized non-functional requirement values for selected plans

Case 3 respectively. Figure 6.4 presents a clear comparison of the normalized values for all three selected plans. The figure clearly shows that $Plan1$ has the best normalized value for *communication cost* and $T3$ with the value of (0.0157 and 0.0179) as compared to $Plan9$ and $Plan107$ with values of (0.0010 and 0.0082) and (0.0003 and 0.0045) respectively. However for remaining non-functional requirements, $Plan9$ and $Plan107$ gives better results.

Based on the user preference values, the final rank is calculated as discussed in §6.4.2.2. Depending on the users' preferences, our proposed approach always select the optimal solution. Figure 6.5 shows the actual rank value for the best three plans. For **Case 1**, $Plan107$ has the highest rank value of 0.0345 followed by $Plan9$ with the rank value of 0.0261. For **Case 2**, the highest rank (0.0187) is shown for $Plan9$ followed by $Plan107$ $(0,0158)$. Finally, for **Case 3**, $Plan1$ gives the best result (rank value of 0.0128) followed by $Plan107$ (rank value of 0.0124).

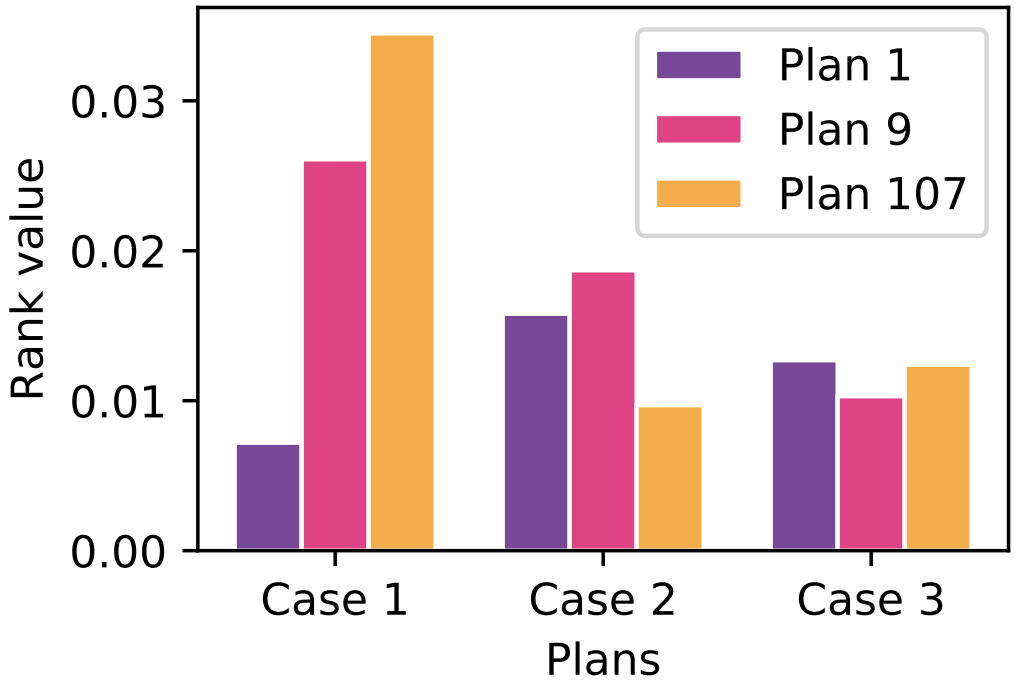


Figure 6.5: Final rank in different cases

6.6 Related work

Compared to the web-application, the current streaming application needed to be distributed and deployed across the edge and cloud environment. As discussed in §1.1 edge devices bring multiple challenges. The deployment problem of stream processing in the cloud environment has been extensively studied in the literature. Frameworks such as Stream [35], Flextream [70], Naiad [113], Cayuga [46] from academia and Apache Storm [3], Amazon Kinesis [2], Google MillWheel [31], Time-Stream [119] from industry are common examples of stream processing frameworks. However, these approaches are limited only to the cloud environment.

Very few models are available in the literature that considers the deployment of streaming application across edge and cloud environments. Work in [69] presents a Mobile Fog framework for large-scale distributed streaming applications in the edge-cloud environment. The model is very simple and is evaluated for a traffic data stream using Omnet++ [140] simulator. [125] proposes a theoretical model for the deployment of IoT application in fog computing infrastructure. It analyses the latency and energy

performance in fog environment and compares with that in the cloud environment.

[127] proposes a programming infrastructure, Foglets for distributing the deployment across edge and cloud environment. Foglets are used for managing the distributed deployment of the application with latency and sensor mobility parameters taken into account. The evaluation is performed by simulating a vehicular network workload on Docker-based foglets.

In [144], an infrastructure model, LEONORE, is presented that provisions applications on resource-constrained edge devices. It uses both pull (allowing independent schedule provisioning) and push (allowing immediate schedule initiation) based mechanism for flexibility and control over the deployed application. Using the distributed provisioning mechanism, it improves the scalability and reduces the network delay. However, the model does not consider the application's QoS requirements in the deployment process.

Kea [56] is another system for offloading the sensor data computational for processing on edge or cloud. It is based on the application metric profiling performed beforehand and the user's weight for all the metrics. It also takes into consideration the hardware capabilities, communication energy and latency costs. Similar work is done in [81] which considers the problem of dynamic computation offloading in wearable healthcare devices. An improvement of 21.1% in battery life is achieved by partitioning the computation between the wearable and a mobile device However, the approach is derived and validated using a simulation environment. Also, the main focus of both these works is to make a decision about offloading the data processing to cloud or not.

A general model to support QoS-aware deployment is presented in [47], where a multi-component IoT application is deployed across fog infrastructure. The fog infrastructure here considers both edge and cloud environment. A simple Java-based prototype, FogTorch is presented to illustrate the proposed model. Although the deployment of an application across the IoT infrastructure is considered, it does not address how to optimize the deployment solution. Also, this is an abstract model that does not consider how to generate the distributed IoT application and how to perform the physical deployment.

In addition to these frameworks, various simulation environments are proposed for

modelling the application deployment in edge-cloud environment e.g. iFogSim [67], EdgeCloudSim [137] and IotSim-Edge [76]. However, these environments are very generic and are not able to give the infrastructure specific performance evaluation.

Early efforts centered on the deployment of IoT applications across cloud and edge datacenters are mostly theoretical. Moreover, these solutions have not considered automatic computation partitioning and deployment. Nor have they considered the optimization of multiple conflicting non-functional requirements during the deployment process. In this chapter, we implemented and evaluated an optimized framework to find a suitable deployment solution for the distributed stream processing application. The framework incorporates the user preferences along with a high-level computation description to generate the deployment solution which optimizes the conflicting non-functional requirements.

6.7 Discussion

The experiment results highlight the applicability of the proposed framework for a healthcare-based step-count application. The results show that our proposed approach can find the most suitable solution for any user preferences. Although the proposed approach uses the hierarchical-based management method for making the deployment decision, it can also be done using fully distributed-based management approach. However, the hierarchical approach has smaller time complexity as it categorizes the non-functional attributes into smaller groups which are then compared within the group to compute the respective weights. On the contrary, the distributed approach compares all the attributes exhaustively which makes the computation larger and time-consuming.

Limitations. Although, the proposed work obtains the optimal deployment solution, the whole process is static i.e. the deployment solution is found before the actual deployment. This may not be suitable for cases where the user preference changes at run-time, for example, if the battery of IoT device is found to be less than a minimum value, IoT device will stop transferring data to the cloud. Such a decision can only

be possible if the deployment decision can be made dynamically. This requires the deployment plans to adjusted automatically with the changing environment.

6.8 Conclusion

The proposed framework provides a novel framework to facilitate the partitioning and deployment of streaming application across the distributed IoT infrastructure. The case study shows that the proposed framework always chose the optimal deployment plan based on user preferences. The approach is very general; while we have described a healthcare use case in this chapter, it can be directly applied to any other domain in which non-functional requirements are vital. The system also works well when a model derived by machine learning is used to classify, or predict behavior: in this case, the model is simply treated as an operation in the computational graph, and the proposed framework is then used to decide where to place it in order to meet the non-functional requirements. Finally, whilst the focus of this chapter is on stream processing IoT application, the approach can be easily applied to batch query processing workload. To this end, a batch query can be modelled as an operation, within our IoT graph, deployed at either edge or cloud layer. Since batch processing is normally executed over massive historical data, batch query operation is most likely to be mapped to the cloud layer.

7

Conclusion

Contents

Summary

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

7.1 Thesis summary

The evolution of microservice architecture that modularizes the application into smaller independent components (known as microservices) gives the flexibility for developers to engineer an application with independent, self-contained and portable run-time components. Complemented with containers, microservices can be deployed across any cloud and edge environment. However, deployment of microservices is challenging as it requires efficient and scalable techniques that undermine the heterogeneity of underlying infrastructure and provides a naïve platform. Based on the application requirements, the techniques also need to handle the functional and non-functional requirements in an optimal manner.

In this thesis, we explored numerous challenges for the deployment of microservices in a cloud and edge environment, and proposed solutions that ease the deployment process. In particular, this thesis contributes as

Chapter 2 first presents an overview of microservices and the available deployment environments. It then evaluates the current work on the performance characterization of microservices which is one of the key features for making any deployment decision. Based on the requirements of the application microservices, we discuss the relevant related work and highlights the research gap. To bridge this gap, in this thesis we proposed a set of algorithms and frameworks facilitating the deployment of microservices.

Chapter 3 discusses the performance evaluation of containerized microservices for HPC applications in the interfering environment with cases of inter-container and intra-container interference. It uses CEEM methodology [101] to evaluate the

performance of all these microservices. The results present comprehensive details about the performance variation of containerized microservices. The results show that:

- Executing multiple microservices inside a container is also a feasible deployment option as the result shows that the performance is better than the baseline performance for some cases.
- The interference caused by microservices with similar resource requirements is always higher as compared to the microservices with different resource requirements. Also, the effect of interference is higher for intra-container scenarios than inter-container scenarios. The performance in intra-container scenarios is worst for network-intensive stream operations.
- The result also shows that the performance of containerized microservies are comparable with either cgroups enabled or disabled if the system resources in both cases are exactly the same.

Chapter 4 presents an orchestrator, *SDBO* for the deployment of simple web-application microservices in a multi-cloud environment. The proposed approach offers a user interface for users to define or select from the available benchmarks and cloud host configurations. Users also set the benchmark duration and a maximum budget for the benchmark. The inbuilt *Optimizer* component of the orchestrator uses a heuristic method to find a subset of cloud hosts for executing the benchmarks and then the *Provisioner* executes the benchmark on a real cloud infrastructure. *SDBO* is validated with SimplCommerce web-application executing on AWS and Azure cloud environments. With flexible execution, *SDBO* provides a long time performance within the limited budget. Result shows that:

- *Optimizer* is able to choose a higher number of host configurations as compared to random selection algorithm. It also shows that the time complexity of the optimizer does not increase exponentially with the increasing number of cloud providers.
- Various system parameters result from the *Provisioner* which facilitates the fair comparison of host configurations for the selection of an optimal one.

- For a varying workload pattern, flexible execution matches the performance achieved by continuous execution, thus guarantees the longer duration performance within a limited budget.

Chapter 5 presents a deployment framework for complex web-application microservices in cloud environments using benchmarking. Emphasizing a complex geo-distributed web-application, it presents *GeoBench*, which first receives user requirements from the user (web-application graph, number of components, budget, etc.) and finds a suitable deployment solution. The *Benchmark Planner* component has an attached database storing the cloud host configuration information. It leverages K-means clustering [82] and PSO [84] to generate a set of deployment solutions which increases the number and diversity of the cloud hosts. The *Benchmark Orchestrator* component orchestrates the execution of benchmark in the real cloud environment. Experiment results shows that:

- On average, *GeoBench* is able to find more diverse options in the multi-cloud environment as compared to the state-of-the-art methods (e.g. Random, Greedy). Also, it utilizes the users' budget in an optimized manner.
- *Benchmark Planner* can be easily scaled with the increasing complexity of applications (number of microservice components) and the number of cloud host configurations.
- Cloud execution result confirms that *GeoBench* is able to find better solutions as compared to the state-of-the-art methods for different data-types.

Chapter 6 discusses the deployment of streaming application microservices in IoT environments. Emphasizing the deployment of complex streaming applications, which is an NP-hard problem, it presents *ABMO* which extends the basic *PATH2iot* [110] framework. *ABMO* presents a heuristic model which leverages AHP [124] for selecting a suitable deployment option in the IoT environment. The model receives the computation description, non-functional requirements, resource catalogue and user preferences as input. Using the basic *PATH2iot*, it computes the physical optimized plans which acts as input for the multi-constrained optimization. The weights of non-functional requirements are also computed along

with their normalized value. Finally an optimal plan is selected which is then deployed on the IoT environment. The case study shows that based on user preferences, *ABMO* always selects the optimal deployment plan.

7.2 Future research directions

Microservice architecture can be extended and generalized for many other types of application, where their applicability is yet to be reviewed. Here we provide motivation for a number of areas of future research, which can be inspired by the work done in this PhD thesis.

7.2.1 A generic benchmarking orchestrator

In the current work, we implemented an orchestrator for benchmarking web-applications. The orchestrator can also benchmark HPC applications. However, currently it does not support orchestration of batch processing applications or machine learning applications. Additionally, the orchestrator is focused only on the cloud environment. Future work could address the design of a generic orchestrator which is able to benchmark any type of application executing in the cloud-edge platforms.

7.2.2 Modelling the benchmark metrics to handle the infrastructure uncertainty

In the current work, we analyzed the obtained benchmark metrics to find a suitable solution. Also the flexible execution allows the users to execute the benchmark in any pre-defined timestamp. Future work can leverage the feature of flexible execution and develop an advanced sampling method to collect the system metrics that can be fed to some machine learning methods to have a better observation of the uncertainty in cloud and edge environments.

7.2.3 Run-time migration of microservices

The QoS performance of microservices needs to be guaranteed. Since the cloud-edge system performance changes abruptly, it is necessary to manage the deployment of

microservices at run-time. Future research can combine the current benchmark metrics with the live monitoring data to make a deployment decision.

7.2.4 Simulation models for digital twins

Digital twins [60] represent virtual replicas of a physical device or system which can be used for numerous purposes including system analysis and problem identification. Benchmark results integrated with real-time simulation can be easily employed for performing various analysis in digital twins.

Bibliography

[1] Amazon aws. Available at https://aws.amazon.com/.

[2] Amazon kinesis. Available at https://aws.amazon.com/kinesis/.

[3] Apache storm. Available at http://storm.apache.org/.

[4] Apdex. Available at http://www.apdex.org/index.html.

[5] Average variable unit costs and standing charges for standard electricity in uk. Available at https://www.gov.uk/government/uploads/system/uploads/ attachment_data/file/357808/qep_224.xls.

[6] Bonnie++. Available at https://www.coker.com.au/bonnie++/.

[7] Chef. Available at https://www.chef.io/.

[8] Cloud spectator. Available at https://cloudspectator.com/.

[9] Cloudharmony: transparency for the cloud. Available at https://cloudharmony.com/.

[10] Cloudorado: Cloud computing comparison engine. Available at https://www.cloudorado.com/.

[11] Docker. Available at https://www.docker.com.

[12] Docker hub. Available at https://hub.docker.com/.

[13] Docker stats. Available athttps://docs.docker.com/engine/reference/commandline/stats

[14] Google cloud. Available athttps://cloud.google.com/.

[15] Intel math kernel library benchmarks. Available at https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite/.

[16] I/o characteristics and monitoring. Available at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-io-characteristics.html/.

[17] Jmeter. Available at https://jmeter.apache.org/.

[18] Kolla. Available at https://wiki.openstack.org/wiki/Kolla.

[19] Lxc: Linux containers. Available at https://www.linuxcontainers.org.

[20] Microsoft azure. Available at https://azure.microsoft.com/en-gb/.

[21] The netperf homepage. Available at https://hewlettpackard.github.io/netperf/.

[22] Pay as you go rates on three. Available at http://www.three.co.uk/Store/Pay_As_You_Go_Price_Plans.

[23] Rubis. Available at http://rubis.ow2.org/.

[24] Sdbo. Available at https://github.com/smartdockerbenchmarkingorchestrator/Smart-Docker-Benchmarking-Orchestrator.

[25] Specweb2009. Available at https://www.spec.org/web2009/.

[26] Stream benchmark. Available at http://www.cs.virginia.edu/stream/.

[27] Tpc-w. Available at https://cs.nyu.edu/totok/professional/software/tpcw/tpcw.html.

[28] Wkibook. Available at https://www.wikibooks.org/.

[29] Y-cruncher - a multi-threaded pi-program. Available at http://www.numberworld.org/y-cruncher/.

[30] M Aazam and E.-N Huh. Fog computing micro datacenter based dynamic resource estimation and pricing model for iot. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 687–694. IEEE, 2015.

[31] T Akidau, A Balikov, K Bekiroğlu, S Chernyak, J Haberman, R Lax, S McVeety, D Mills, P Nordstrom, and S Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[32] A Al-Fuqaha, M Guizani, M Mohammadi, M Aledhari, and M Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials*, 17(4):2347–2376, 2015.

[33] J. P Albrecht. How the gdpr will change the world. *Eur. Data Prot. L. Rev.*, 2:287, 2016.

[34] S Androutsellis-Theotokis and D Spinellis. A survey of peer-to-peer content distribution technologies. *ACM computing surveys (CSUR)*, 36(4):335–371, 2004.

[35] A Arasu, B Babcock, S Babu, J Cieslewicz, M Datar, K Ito, R Motwani, U Srivastava, and J Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[36] K Bankole, D Krook, S Murakami, and M Silveyra. A practical approach to dockerizing openstack high availability, 2014.

[37] 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*, 37(5):164–177, 2003.

[38] D Bhamare, M Samaka, A Erbad, R Jain, and L Gupta. Exploring microservices for enhancing internet qos. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3445, 2018.

[39] J Bhimani, Z Yang, M Leeser, and N Mi. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[40] E. W Biederman and L Networx. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112. Citeseer, 2006.

[41] M. S Bonfim, K. L Dias, and S. F Fernandes. Integrated nfv/sdn architectures: A systematic literature review. *ACM Computing Surveys (CSUR)*, 51(6):114, 2019.

[42] F Bonomi, R Milito, J Zhu, and S Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[43] A. H Borhani, P Leitner, B.-S Lee, X Li, and T Hung. Wpress: An application-driven performance benchmark for cloud-based virtual machines. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*, pages 101–109. IEEE, 2014.

[44] S Borst, V Gupta, and A Walid. Distributed caching algorithms for content distribution networks. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[45] D Bratton and J Kennedy. Defining a standard for particle swarm optimization. In *2007 IEEE swarm intelligence symposium*, pages 120–127. IEEE, 2007.

[46] L Brenna, A Demers, J Gehrke, M Hong, J Ossher, B Panda, M Riedewald, M Thatte, and W White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1100–1102. ACM, 2007.

[47] A Brogi and S Forti. Qos-aware deployment of iot applications through the fog. *IEEE Internet of Things Journal*, 4(5):1185–1192, 2017.

[48] V Cardellini, V Grassi, F Lo Presti, and M Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 69–80, 2016.

[49] E Cecchet, V Udayabhanu, T Wood, and P Shenoy. Benchlab: An open testbed for realistic benchmarking of web applications. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

[50] M. B Chhetri, S Chichin, Q. B Vo, and R Kowalczyk. Smart cloudbench–automated performance benchmarking of the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 414–421. IEEE, 2013.

[51] M. B Chhetri, S Chichin, Q. B Vo, and R Kowalczyk. Smart cloudbench—a framework for evaluating cloud infrastructure performance. *Information Systems Frontiers*, 18(3):413–428, 2016.

[52] Cisco. Fog computing and the internet of things: Extend the cloud to where the things are. pages 1–6, 2015.

[53] I Cuadrado-Cordero, A.-C Orgerie, and J.-M Menaud. Comparative experimental analysis of the quality-of-service and energy-efficiency of vms and containers' consolidation for cloud applications. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2017)*, pages 1–6, 2017.

[54] M Cunha, N Mendonça, and A Sampaio. Cloud crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds. *Concurrency and Computation: Practice and Experience*, 29(1):e3825, 2017.

[55] M Curiel and A Pont. Workload generators for web-based systems: Characteristics, current status, and challenges. *IEEE Communications Surveys & Tutorials*, 20(2):1526–1546, 2018.

[56] R. B Das, N. V Bozdog, M. X Makkes, and H Bal. Kea: A computation offloading system for smartphone sensor data. In *Cloud Computing Technology and Science (CloudCom), 2017 IEEE International Conference on*, pages 9–16. IEEE, 2017.

[57] C Davatz, C Inzinger, J Scheuner, and P Leitner. An approach and case study of cloud instance type selection for multi-tier web applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 534–543. IEEE, 2017.

[58] Q Duan. Cloud service performance evaluation: status, challenges, and opportunities – a survey from the system modeling perspective. *Digital Communications and Networks*, 3(2):101 – 111, 2017.

[59] C Ebert, G Gallardo, J Hernantes, and N Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[60] A El Saddik. Digital twins: The convergence of multimedia technologies. *IEEE multimedia*, 25(2):87–92, 2018.

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

[62] W Felter, A Ferreira, R Rajamony, and J Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. IEEE, 2015.

[63] M. R Garey and D. S Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. Freeman, 1979.

[64] M Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26. IEEE, 2007.

[65] M Gonçalves, M Cunha, N. C Mendonça, and A Sampaio. Performance inference: A novel approach for planning the capacity of iaas cloud applications. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 813–820. IEEE, 2015.

[66] 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*, 1(1):45–64, 2012.

[67] H Gupta, A Vahid Dastjerdi, S. K Ghosh, and R Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[68] J Higgins, V Holmes, and C Venters. Orchestrating docker containers in the hpc environment. In *International Conference on High Performance Computing*, pages 506–513. Springer, 2015.

[69] K Hong, D Lillethun, U Ramachandran, B Ottenwälder, and B Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.

[70] A. H Hormati, Y Choi, M Kudlur, R Rabbah, T Mudge, and S Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 214–223. IEEE, 2009.

[71] J. A Hoxmeier and C DiCesare. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, page 347, 2000.

[72] K Hsieh, A Harlap, N Vijaykumar, D Konomis, G. R Ganger, P. B Gibbons, and O Mutlu. Gaia: Geo-distributed machine learning approaching {LAN} speeds. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 629–647, 2017.

[73] M Hu, J Luo, Y Wang, and B Veeravalli. Practical resource provisioning and caching with dynamic resilience for cloud-based content distribution networks. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):2169–2179, 2013.

[74] Y Huang, Y Shi, Z Zhong, Y Feng, J Cheng, J Li, H Fan, C Li, T Guan, and J Zhou. Yugong: geo-distributed data and job placement at scale. *Proceedings of the VLDB Endowment*, 12(12):2155–2169, 2019.

[75] D. M Jacobsen and R. S Canon. Contain this, unleashing docker for hpc, 2015.

[76] D. N Jha, K Alwasel, A Alshoshan, X Huang, R. K Naha, S. K Battula, S Garg, D Puthal, P James, A. Y Zomaya, *et al.* Iotsim-edge: A simulation framework for modeling the behaviour of iot and edge computing environments. *arXiv preprint arXiv:1910.03026*, 2019.

[77] D. N Jha, S Garg, P. P Jayaraman, R Buyya, Z Li, G Morgan, and R Ranjan. A study on the evaluation of hpc microservices in containerized environment. *Concurrency and Computation: Practice and Experience*, page e5323, 2019.

[78] D. N Jha, M Nee, Z Wen, A Zomaya, and R Ranjan. Smartdbo: smart docker benchmarking orchestrator for web-application. In *The World Wide Web Conference*, pages 3555–3559, 2019.

[79] D. N Jha, Z Wen, Y Li, M Nee, M Koutny, and R Ranjan. A cost-efficient multi-cloud orchestrator for benchmarking containerized web-applications. In *International Conference on Web Information Systems Engineering*, pages 407–423. Springer, 2019.

[80] C. T Joseph and K Chandrasekaran. Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. *Software: Practice and Experience*, 49(10):1448–1484, 2019.

[81] H Kalantarian, C Sideris, B Mortazavi, N Alshurafa, and M Sarrafzadeh. Dynamic computation offloading for low-power wearable health monitoring systems. *IEEE Transactions on Biomedical Engineering*, 64(3):621–628, 2017.

[82] T Kanungo, D. M Mount, N. S Netanyahu, C. D Piatko, R Silverman, and A. Y Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*, 24(7):881–892, 2002.

[83] A Karmel, R Chandramouli, and M Iorga. Nist definition of microservices, application containers and system virtual machines. Technical report, National Institute of Standards and Technology, 2016.

[84] J Kennedy and R Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[85] T Killalea. The hidden dividends of microservices. *Communications of the ACM*, 59(8):42–45, 2016.

[86] A Kivity, Y Kamay, D Laor, U Lublin, and A Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, pages 225–230, 2007.

[87] K Kloudas, M Mamede, N Preguiça, and R Rodrigues. Pixida: optimizing data parallel jobs in wide-area data analytics. *Proceedings of the VLDB Endowment*, 9(2):72–83, 2015.

[88] H Knoche. Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 121–124, New York, NY, USA, 2016. ACM.

[89] C. G Kominos, N Seyvet, and K Vandikas. Bare-metal, virtual machines and containers in openstack. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pages 36–43. IEEE, 2017.

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

[91] C Krintz. The appscale cloud platform: Enabling portable, scalable web application deployment. *IEEE Internet Computing*, 17(2):72–75, 2013.

[92] P Leitner and J Cito. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3):15:1–15:23, April 2016.

[93] P Leitner and J Cito. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology (TOIT)*, 16(3):1–23, 2016.

[94] J Lewis and M Fowler. Microservices guide. Available at https://martinfowler.com/microservices/, 2014.

[95] A Li, X Yang, S Kandula, and M Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2010.

[96] H Li, H Xu, and S Nutanong. Bohr: similarity aware geo-distributed data analytics. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 267–279, 2018.

[97] Z Li, M Kihl, Q Lu, and J. A Andersson. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 955–962. IEEE, 2017.

[98] Z Li, K Mitra, M Zhang, R Ranjan, D Georgakopoulos, A. Y Zomaya, L O'Brien, and S Sun. Towards understanding the runtime configuration management of do-it-yourself content delivery network applications over public clouds. *Future Generation Computer Systems*, 37:297–308, 2014.

[99] Z Li, L OBrien, R Cai, and H Zhang. Towards a taxonomy of performance evaluation of commercial cloud services. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 344–351. IEEE, 2012.

[100] Z Li, L OBrien, R Ranjan, and M Zhang. Early observations on performance of google compute engine for scientific computing. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, volume 1, pages 1–8. IEEE, 2013.

[101] Z Li, L O'Brien, and H Zhang. CEEM: A practical methodology for cloud services evaluation. In *2013 IEEE Ninth World Congress on Services (SERVICES)*, pages 44–51. IEEE, 2013.

[102] Z Li, L O'Brien, H Zhang, and R Cai. A factor framework for experimental design for performance evaluation of commercial cloud services. In *2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 169–176. IEEE, 2012.

[103] S. H Liew and Y.-Y Su. Cloudguide: Helping users estimate cloud deployment cost and performance for legacy web applications. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 90–98. IEEE, 2012.

[104] LightStep. Global microservices trends: A survey of development professionals. Available at https://go.lightstep.com/rs/260-KGM-472/images/global-microservices-trends-2018.pdf, 2018.

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

[106] F Liu, J Tong, J Mao, R Bohn, J Messina, L Badger, and D Leaf. Nist cloud computing reference architecture. *NIST special publication*, 500(2011):1–28, 2011.

[107] G Liu, H Shen, and H Wang. Cooperative job scheduling and data allocation for busy data-intensive parallel computing clusters. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–11, 2019.

[108] M. A Masud, J. Z Huang, C Wei, J Wang, I Khan, and M Zhong. I-nice: A new approach for identifying the number of clusters and initial cluster centres. *Information Sciences*, 466:129–151, 2018.

[109] N Michael, N Ramannavar, Y Shen, S Patil, and J.-L Sung. Cloudperf: A performance test framework for distributed and dynamic multi-tenant environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 189–200, 2017.

[110] P Michalák and P Watson. Path2iot: A holistic, distributed stream processing system. In *Cloud Computing Technology and Science (CloudCom), 2017 IEEE International Conference on*, pages 25–32. IEEE, 2017.

[111] R Morabito. Power consumption of virtualization technologies: An empirical investigation. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 522–527. IEEE, 2015.

[112] R Morabito, J Kjällman, and M Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *2015 IEEE International Conference on Cloud Engineering (IC2E)*, pages 386–393. IEEE, 2015.

[113] D. G Murray, F McSherry, R Isaacs, M Isard, P Barham, and M Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[114] M Nardelli, S Nastic, S Dustdar, M Villari, and R Ranjan. Osmotic flow: Osmotic computing+ iot workflow. *IEEE Cloud Computing*, 4(2):68–75, 2017.

[115] P Pietzuch, J Ledlie, J Shneidman, M Roussopoulos, M Welsh, and M Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49. IEEE, 2006.

[116] M Poess and C Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[117] Q Pu, G Ananthanarayanan, P Bodik, S Kandula, A Akella, P Bahl, and I Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.

[118] B Qian, J Su, Z Wen, D. N Jha, Y Li, Y Guan, D Puthal, P James, R Yang, A. Y Zomaya, O Rana, L Wang, M Koutny, and R Ranjan. Orchestrating the development lifecycle of machine learning-based iot applications: A taxonomy and survey. *ACM Comput. Surv.*, 0(ja).

[119] Z Qian, Y He, C Su, Z Wu, H Zhu, T Zhang, L Zhou, Y Yu, and Z Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[120] R Ranjan, B Benatallah, S Dustdar, and M. P Papazoglou. Cloud resource orchestration programming: overview, issues, and directions. *IEEE Internet Computing*, 19(5):46–56, 2015.

[121] Z Ren, W Wang, G Wu, C Gao, W Chen, J Wei, and T Huang. Migrating web applications from monolithic structure to microservices architecture. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, Internetware '18, pages 7:1–7:10, New York, NY, USA, 2018. ACM.

[122] L Roberts, P Michalák, S Heaps, M Trenell, D Wilkinson, and P Watson. Automating the placement of time series models for iot healthcare applications. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 290–291. IEEE, 2018.

[123] C Ruiz, E Jeanvoine, and L Nussbaum. Performance evaluation of containers for hpc. In *European Conference on Parallel Processing*, pages 813–824. Springer, 2015.

[124] T. L Saaty. Axiomatic foundation of the analytic hierarchy process. *Management science*, 32(7):841–855, 1986.

[125] S Sarkar and S Misra. Theoretical modelling of fog computing: a green computing paradigm to support iot applications. *Iet Networks*, 5(2):23–29, 2016.

[126] M Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[127] E Saurez, K Hong, D Lillethun, U Ramachandran, and B Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 258–269. ACM, 2016.

[128] J Scheuner, J Cito, P Leitner, and H Gall. Cloud workbench: Benchmarking iaas providers based on infrastructure-as-code. In *Proceedings of the 24th International Conference on World Wide Web*, pages 239–242, 2015.

[129] D Serrano, S Bouchenak, Y Kouki, F. A de Oliveira Jr, T Ledoux, J Lejeune, J Sopena, L Arantes, and P Sens. Sla guarantees for cloud services. *Future Generation Computer Systems*, 54:233–246, 2016.

[130] P Sharma, L Chaufournier, P Shenoy, and Y. C Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*, pages 1–13, 2016.

[131] W Shi, J Cao, Q Zhang, Y Li, and L Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[132] W Shi and S Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[133] J Siegel and J Perdue. Cloud services measures for global use: the service measurement index (smi). In *2012 Annual SRII Global Conference*, pages 411–415. IEEE, 2012.

[134] M Silva, M. R Hines, D Gallo, Q Liu, K. D Ryu, and D Da Silva. Cloudbench: Experiment automation for cloud environments. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 302–311. IEEE, 2013.

[135] W Sobel, S Subramanyam, A Sucharitakul, J Nguyen, H Wong, A Klepchukov, S Patil, A Fox, and D Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proceedings of CCA*, 2008.

[136] S Soltesz, H Pötzl, M. E Fiuczynski, A Bavier, and L Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *ACM SIGOPS Operating Systems Review*, 41(3):275–287, 2007.

[137] C Sonmez, A Ozgovde, and C Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3493, 2018.

[138] T. P. P. C TPC. Tpc benchmark^{TM} e. 2010.

[139] L. M Vaquero and L Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[140] A Varga and R Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and ..., 2008.

[141] B Varghese, O Akgun, I Miguel, L Thai, and A Barker. Cloud benchmarking for maximising performance of scientific applications. *IEEE Transactions on Cloud Computing*, 7(1):170–182, 2016.

[142] B Varghese, L. T Subba, L Thai, and A Barker. Doclite: A docker-based lightweight cloud benchmarking tool. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 213–222. IEEE, 2016.

[143] R Viswanathan, G Ananthanarayanan, and A Akella. {CLARINET}: Wan-aware optimization for analytics queries. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 435–450, 2016.

[144] M Vögler, J. M Schleicher, C Inzinger, and S Dustdar. A scalable framework for provisioning large-scale iot deployments. *ACM Transactions on Internet Technology (TOIT)*, 16(2):11, 2016.

[145] P Voigt and A Von dem Bussche. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

[146] A Vulimiri, C Curino, P. B Godfrey, T Jungblut, J Padhye, and G Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 323–336, 2015.

[147] 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*, 119:97–109, 2018.

[148] Z Wang, B Li, L Sun, and S Yang. Cloud-based social application deployment using local processing and global distribution. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, page 301–312, New York, NY, USA, 2012. Association for Computing Machinery.

[149] D Weerasiri, M. C Barukh, B Benatallah, Q. Z Sheng, and R Ranjan. A taxonomy and survey of cloud resource orchestration techniques. *ACM Computing Surveys (CSUR)*, 50(2):26, 2017.

[150] Z Wen, J Cała, P Watson, and A Romanovsky. Cost effective, reliable and secure workflow deployment over federated clouds. *IEEE Transactions on Services Computing*, 10(6):929–941, 2016.

[151] Z Wen, T Lin, R Yang, S Ji, R Ranjan, A Romanovsky, C Lin, and J Xu. Ga-par: Dependable microservice orchestration framework for geo-distributed clouds. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):129–143, 2019.

[152] Z Wen, R Qasha, Z Li, R Ranjan, P Watson, and A Romanovsky. Dynamically partitioning workflow over federated clouds for optimising the monetary cost and handling run-time failures. *IEEE Transactions on Cloud Computing*, 2016.

[153] M. G Xavier, M. V Neves, F. D Rossi, T. C Ferreto, T Lange, and C. A De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–240. IEEE, 2013.

[154] Y Xing and Y Zhan. Virtualization and cloud computing. In *Future Wireless Networks and Information Systems*, pages 305–312. Springer, 2012.

[155] M Yannuzzi, R Milito, R Serral-Gracià, D Montero, and M Nemirovsky. Key ingredients in an iot recipe: Fog computing, cloud computing, and more fog computing. In *Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2014 IEEE 19th International Workshop on*, pages 325–329. IEEE, 2014.

[156] K Ye and Y Ji. Performance tuning and modeling for big data applications in docker containers. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6. IEEE, 2017.

[157] Y Yuan, D Ma, Z Wen, Y Ma, G Wang, and L Chen. Efficient graph query processing over geo-distributed datacenters. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 619–628, 2020.

[158] K Zhang, L Wang, X Guo, A Pan, and B. B Zhu. Wpbench: a benchmark for evaluating the client-side performance of web 2.0 applications. In *Proceedings of the 18th international conference on World wide web*, pages 1111–1112. ACM, 2009.

[159] Q Zhang, L Liu, C Pu, Q Dou, L Wu, and W Zhou. A comparative study of containers and virtual machines in big data environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 178–185. IEEE, 2018.

[160] N Zhao. Full-featured pedometer design realized with 3-axis digital accelerometer. *Analog Dialogue*, 44(06):1–5, 2010.