MICROSERVICE ORCHESTRATION STRATEGIES FOR CONTAINERIZED CLOUD ENVIRONMENTS

Thesis

Submitted in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

CHRISTINA TERESE JOSEPH

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

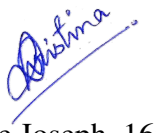
SURATHKAL, MANGALORE - 575 025

May, 2021

DECLARATION

by the Ph.D. Research Scholar

I hereby declare that the Research Thesis entitled **Microservice Orchestration Strategies for Containerized Cloud Environments** which is being submitted to the **National Institute of Technology Karnataka, Surathkal** in partial fulfilment of the requirements for the award of the Degree of **Doctor of Philosophy** in Department of Computer Science and Engineering is a bonafide report of the research work carried out by me. The material contained in this Research Thesis has not been submitted to any University or Institution for the award of any degree.



Christina Terese Joseph, 165092 CS16F02

Department of Computer Science and Engineering

Place: NITK, Surathkal.

Date: May 8, 2021

CERTIFICATE

This is to certify that the Research Thesis entitled **Microservice Orchestration Strategies for Containerized Cloud Environments** submitted by **Christina Terese Joseph** (Register Number: 165092 CS16F02) as the record of the research work carried out by her, is accepted as the Research Thesis submission in partial fulfilment of the requirements for the award of degree of **Doctor of Philosophy**.

Dr. K. CHANDRA SEKARAN Ph.D.
Professor, Dept. of Computer Sc. & Engg.
National Institute of Technology Karnataka
Surathkal, Srinivasnagar - 575 025
Mangalore, Karnataka State, India.

Prof. K. Chandrasekaran

Research Guide

07/06/2021

(Signature with Date and Seal)

17/6/2021

Chairman - DRPC

(Signature with Date and Seal)

ACKNOWLEDGEMENTS

"In order for the light to shine so brightly, the darkness must be present" -

Francis Bacon

First and foremost, I would like to express my gratitude to my supervisor, Prof. K. Chandrasekaran for enabling me to identify the darkness and throwing light on the topics, that came under my preview, to formulate my research.

I would like to thank my Research Progress Assessment Committee members, Dr. Deepu Vijayasenan, Department of Electronics and Communication Engineering, NITK and Dr. Jeny Rajan, Department of Computer Science and Engineering, NITK for their valuable evaluation and feedback throughout the progress of my research. I am thankful to Prof. A. Kandasamy, Department of Mathematical and Computational Sciences, NITK, whose charisma and adroitness in problem solving helped to enlighten my thoughts.

I would also like to extend my sincere thanks to Dr. Alwyn Roshan Pais, Head of the Department of Computer Science and Engineering, all the faculty members, support staff, fellow research scholars and my overseas research peers for the unstinting support rendered over the course of my research. I am grateful to NITK Surathkal for providing all the necessary infrastructure and facilities to pursue my research.

My parents and life partner John Paul Martin were my perennial source of inspiration that helped me all the way in accomplishing this remarkable goal.

Above all, I thank God Almighty for showering me with abundant blessings that constantly strengthened me to pass through all the difficult times.

Christina Terese Joseph

ABSTRACT

The explosion in the popularity of the Internet paralleled with the impetuous evolution of computing and storage technologies has brought about a revolutionary shift in the way computational resources are provisioned. The Cloud computing paradigm facilitates the lease of computational resources as services on a pay-per-use basis. Cloud developers have rapidly embraced the Microservice Architecture to accelerate the development and deployment of Cloud applications. However, the dynamism, agility and distributed characteristics of microservices pose significant challenges in the resource orchestration of microservice-based Cloud environments. Effectively utilizing the distributed resources of the Cloud to obtain performance gains is an issue of paramount importance. Hence, this work focusses on the orchestrational challenges in microservice-based Cloud environments.

In order to achieve the desired level of scalability and elasticity, microservice-based Cloud applications are typically packaged in containers. Therefore, microservice orchestration strategies for Cloud environments must effectively manage container clusters to automate processes such as resource allocation, autoscaling and load balancing. In terms of system performance, a key concern is the initial placement of the microservice applications. Placing microservice applications without considering the interactions among the microservices forming an application results in a performance penalty. Accordingly, an interaction-aware microservice placement strategy, called Interaction-aware Microservice Allocation (IntMA) that preserves the Quality of Service and maintains resource efficiency, is devised in this research. The interaction pattern is modeled using a doubly weighted interaction graph, which is then used to assign the incoming microservice applications to appropriate nodes in the Cloud datacenter. Experiments on the Google Cloud Platform substantiated that our proposed approach attains better objective values than the existing placement policies.

The dynamism of microservice-based Cloud environments renders it essential to revisit the initial placement decisions and perform rescheduling. Rescheduling strategies must strive to resolve degradations in the performance due to fluctuations in the workload. Existing rescheduling strategies, tailored for hypervisor-based virtualization environments, do not consider the features specific to containers. Therefore, this research work also explores the impact of container configuration parameters on microservice application performance. The experiments revealed that larger values for container CPU throttling led to higher response times. In order to circumvent this, a Throttling and Interaction-aware Anticorrelated Rescheduling Framework (TIARM) for microservices, is proposed. Experimental results elucidate the efficacy of the proposed approach in enhancing the performance of containerized Cloud environments.

Keywords: Cloud computing, Container virtualization, Microservice Architecture, Microservice orchestration, Resource management, Microservice allocation, Microservice re-scheduling.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

Abbreviations	Expansion
API	Application Programming Interface
AUFS	Advanced multilayered Unification File System
BPM	Business Process Modelling
CI/CD	Continuous Integration and Continuous Delivery
CNA	Cloud Native Application
DDD	Domain Driven Design
DSL	Domain Specific Language
ECS	Elastic Container Service
FaaS	Function as a Service
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
HCL	HashiCorp Configuration Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
JSON	JavaScript Object Notation
k8s	Kubernetes
MAPE	Monitor-Analysis-Plan-Execute
MSA	Microservice Architecture
NSGA	Nondominated Sorting Genetic Algorithm
OASIS	Organization for the Advancement of Structured Information Standards
PaaS	Platform as a Service
QoS	Quality of Service
REST	REpresentational State Transfer
RPC	Remote Procedure Call
SaaS	Software as a Service
SDN	Software Defined Networks
SOA	Service-Oriented Architecture
SOC	Service-Oriented Computing
TCP	Transmission Control Protocol
UML	Unified Modelling Language
VM	Virtual Machine

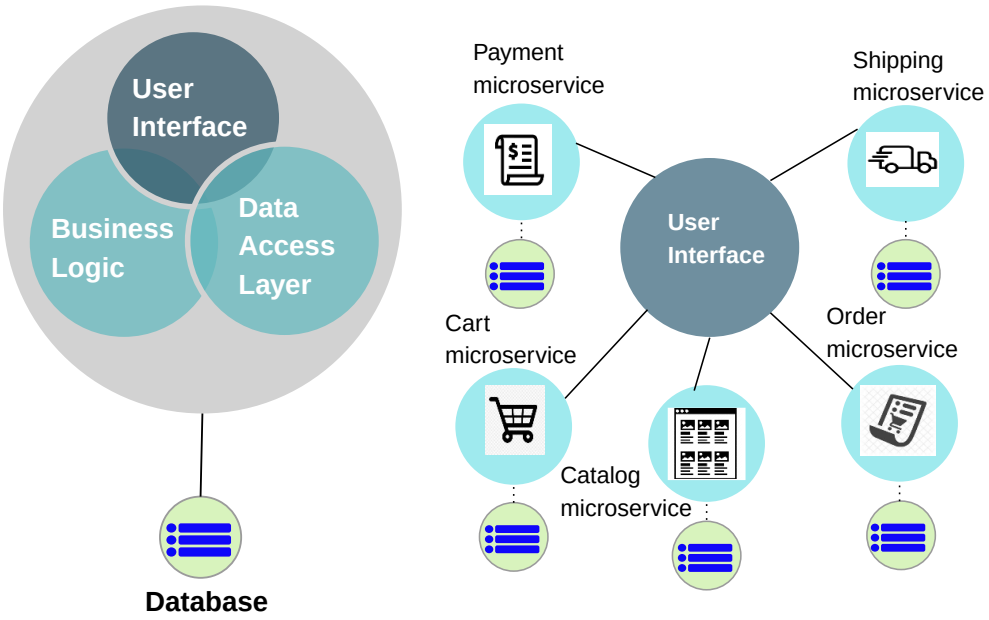
CHAPTER 1

INTRODUCTION

The Cloud Computing paradigm has rapidly proliferated as a platform to host applications across different sectors of the Information and Communication Technology (ICT). As per Gartner forecasts (Katie 2019), the Cloud service industry is estimated to grow at a rate three times greater than the overall Information Technology (IT) industry through 2022. According to reports (Thomas et al. 2019), nearly all Fortune 500 companies rely on the Cloud to host their organizational workloads. Cloud users leverage on-demand access to a shared pool of hardware and software resources through different service models, such as, Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

Over the past few years, there has been a phenomenal change in the approach used to develop and deliver Cloud applications and services. Enterprise solutions backed by Cloud-based systems must heed to rapidly evolving customer needs by developing software that can be scaled and updated at a swift pace. In this regard, the Microservice Architecture (MSA) pattern has been gaining a foothold in the software development industry and in particular, in Cloud application development.

In yesteryears, applications were packaged and deployed as a single component, called monolith. An example of an e-commerce application designed using the monolithic structure has been illustrated in Figure 1.1a. The monolithic design follows a modular structure with the business logic at its core, packaged as a monolith. The monolith contains all the modules of the application, including the user interface, busi-



(a) Monolithic application

(b) Microservice application

Figure 1.1: A monolithic application transformed to a microservice-based application

ness logic and data access layers. Such applications are easier to develop, test and deploy. They also support scaling requirements. However, once deployed, several modifications made to the monolith tampers the intial structure of the monolithic application and results in several drawbacks, such as an exponential increase in the Lines of Code (LoC), increased start-up times and hindrance in upgrading project dependencies. The initially simple monolithic application thus becomes too complex and difficult for the developer to handle. The large monolithic application offers higher resistance to future development due to the high level of interdependence among the different layers. The evolution of the software thus becomes too cumbersome. Another major challenge is the resistance offered by the large monolithic applications to agility and continuous deployment. Other challenges include lower reliability values and less support for scalability (Balalaie et al. 2014).

As a solution to these problems, several organizations such as Amazon and Netflix adopted the Microservices Architecture pattern where a single application is split into constituent independent microservice units. In the MSA, individual microservices are developed to implement each functional unit of the application, as depicted in Figure

1.1b. Separate microservices perform different e-commerce operations such as receiving customer orders, price listing and displaying product availability, adding items to customer carts, managing customer payments and shipping the placed orders. Each microservice accesses its own database and can be deployed independently. Since the application is decomposed to multiple manageable units, the MSA application is less complex and offers the benefits of modularity. Accordingly, the individual microservices can be developed and maintained with lesser efforts. The other tangible benefits of the MSA include improved scalability, agility and fault tolerance. Each microservice unit can be scaled independently, thereby eliminating the need to scale the entire application. As the MSA enables the independent deployment of each unit, changes that are to be made locally to one unit can be tested and deployed without the need for coordinated deployment, thereby supporting agile development. Additionally, since different modules are deployed independently, bugs in individual modules do not hamper the availability of the entire application, which is in contrast with the case of monolithic applications where bugs in any of the modules can bring down the entire application.

Cloud applications pertaining to the MSA are characterised by features such as hyperagility and resilience that empower IT enterprises to reap maximum benefits offered by the Cloud and cater to the evolving client requirements. The Microservice Architecture has also been influencing the Edge Computing, Fog Computing and Internet of Things (IoT) paradigms (Yousefpour et al. 2019). Since the introduction of the architecture in 2014, there has been a tremendous explosion in the number of organizations that have resorted to microservices (Fritzsch et al. 2019; Henry and Ridene 2020). The MSA is thus an emerging distributed architectural style that is predominantly used in the design of distributed systems. The following section presents the evolution of the Microservices Software Architecture.

1.1 EVOLUTION OF MICROSERVICES SOFTWARE ARCHITECTURE (MSA)

The first object-oriented language, SIMULA, was developed in the 1970s (Meyer 1988). In the early 1980s, Bjorn Stroustrup adopted the object-oriented programming features into the C programming language, to develop C++ (Stroustrup 1994). By the 1990s,

most of the computing systems made use of object oriented programming. To provide object-oriented features for multi-user environments, two distributed programming models were introduced (Vinoski 1997): the Distributed Component Object Model (DCOM) and the Common Object Request Broker Architecture (CORBA). The object-oriented models were too complex as every action involved objects. Besides, the learning curve was very steep and security measures were also not stringent. This led to the development of service-based systems, where the entire system is centered around units called services (Tsai 2005).

1.1.1 Service Oriented Computing (SOC)

Service Oriented Computing (SOC) is a paradigm for designing software applications for distributed systems (Papazoglou 2003). SOC relies on services as the fundamental constructs and promotes the rapid development of software applications. These systems comprises of three categories of entities: the provider who is responsible for providing the service, the consumer who makes use of these services, and the most important entity, the registry, where the providers display the functionalities offered by the services hosted by them, which enables the consumers to identify the service suitable to their needs/requirements and use them. The paradigm involves various services, with less coupling among them, which interact with each other to deliver a business process.

1.1.2 Service Oriented Architecture (SOA)

In order to realize the vision of SOC, an architectural pattern was developed in the late 1990s, the Service Oriented Architecture (SOA). The Organization for the Advancement of Structured Information Standards (OASIS) reference framework for SOA defines a service as "*A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface*" (MacKenzie et al. 2006). The SOA is thus an architectural style where business processes are implemented as different components, with less degree of coupling, communicating with each other using a middle layer with messaging capabilities. The Enterprise Service Bus (ESB) is the central controller which controls and coordinates the flow of execution of services. The degree of coupling in the system was slightly lower, when compared to monolithic ap-

plications, but this was not sufficient to completely realize the goal of Service Oriented Computing. In particular, the SOA did not provide much support for agility and for rapid deployment (Zimmermann 2016).

1.1.3 Inception of the Microservices Software Architecture (MSA)

In 2011, software architects proposed the microservices architectural style, though it was only in 2012, that the term 'microservices' was used to refer to the architectural style. The initial discussions on the architectural style were presented by Fred George (George 2013) and James Lewis (Lewis 2012). Parallely, the Netflix team led by Adrian Cockroft, re-designed all their applications to follow the 'fine-grained' SOA principles, which was in alignment with the characteristics of the microservices architectural style. Martin Fowler defines the microservices architectural style as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API" [1]. The microservice components focus on single business capabilities and are independently deployable. Architecting applications as microservices, will result in an increased number of components which lead to an increase in the complexity of management of the system. The MSA is often considered to be strongly related to SOA (Bogner et al. 2018). The Microservices Architectural (MSA) style inherits most of its features from the distributed computing paradigm and ideologies from Business Process Modelling (BPM) (Geisriegler et al. 2017). The underlying concepts of MSA are presented in Section 1.2.

1.2 FUNDAMENTAL CONCEPTS OF MICROSERVICE ARCHITECTURE (MSA)

Microservices architecture is a software architectural pattern where applications are decomposed into autonomous, loosely coupled, independently deployable units, called microservices (Newman 2015). Generally microservices are formed by dividing an application across the functional boundaries, resulting in each microservice dealing with only one function. Each of the microservices is run separately and communicates via

[1] martinfowler.com/microservices.html

lightweight mechanisms such as REpresentational State Transfer (REST) over Application Programming Interface (API). In the following subsections, the internal structure of microservices and the theoretical concepts underlying MSA are presented.

1.2.1 Internal structure of Microservices

The microservice components are usually characterized by the single responsibility principle. Each microservice unit generally exposes their functionalities in the form of interfaces. The microservices can be considered to have a layered structure [2]as shown in Figure 1.2, consisting of the following layers:

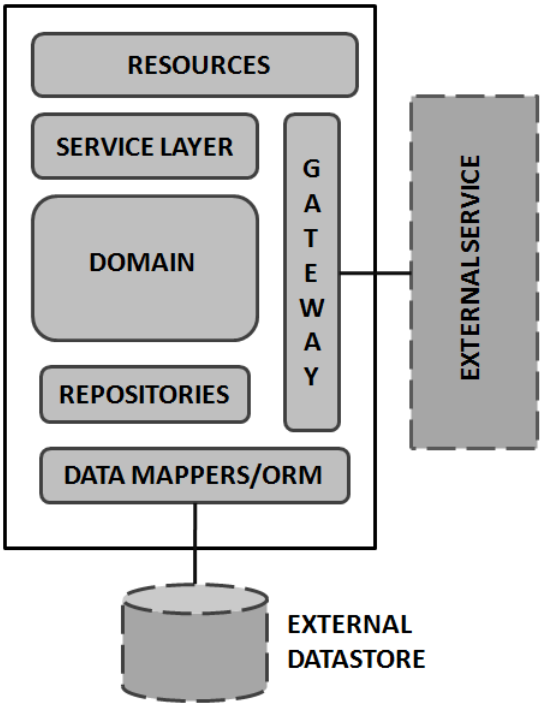


Figure 1.2: Internal structure of a Microservice component

- Resources: This layer maps the requests into messages to the objects in the domain. They carry out sanity checks for each request, transform the request into invocations on the domain layer and provides a response in the specified format.

- Domain model Layer: This layer represents the business domain and contains the business logic. This layer includes:

[2]martinfowler.com/microservices.html

  - Services: The services layer contains implementations of business logic which handles multiple domain objects.
  - Domains: This layer includes entities, value objects and aggregate roots, all of which are used to represent 'things' in the business model.
  - Repositories: This layer acts as a storage for a collection of domain entities.

- Gateway: This layer enables communication with remote services. The layer is responsible for marshalling requests and responses from and to objects of the domain.

- Data Mapper: This layer provides persistence aspects for the domain objects across requests. The layer generally includes Object Relational Mapping (ORM) Models to transform entities or objects into atomic scalar values, which can be directly stored in relational databases.

1.2.2 Domain-Driven Design

Developing software intuitively can lead to software systems that are extremely complex. Software developers tackle this challenge by adopting the principles of Domain Driven Design (DDD). Introduced by Eric Evans (Evans 2004), the DDD aims at developing solutions for a particular 'domain' or problem space. The domain is further decomposed into sub-domains, which are termed 'context maps'. Solutions are designed for each context map, and these solutions are called 'bounded contexts'. The solutions involve two different type of objects 'aggregates' and 'value objects'. Aggregates are analogous to 'objects' in the Object Oriented Design (OOD), and usually represent objects in a hierarchy, where only the object at the root node can be accessed from the external world. On the other hand, value objects represent values like identifiers or measurement values. By default, they are static and does not vary. Systems designed according to the DDD principles deploy individual units corresponding to each bounded context. Each unit incorporates a domain layer that contains the business logic, an infrastructure layer that houses the infrastructure that runs the other layers, and a client layer that acts as the interface to the application. In addition, there are 'domain services' that has higher capabilities and can simultaneously access heterogeneous

domain objects or bounded contexts.

Microservices are designed to handle the responsibility of a single functionality. A better perspective to view microservices would be as components that provide solution for one bounded context. Thus, microservices may be designed adhering to the credo of the DDD, which enables the units to be autonomous and independent. Once the bounded contexts have been identified, the developer may proceed to identify the aggregates and value objects corresponding to the bounded context. This can then be translated to design microservice systems.

1.2.3 Circuit Breaker Pattern

A client request to a microservices system, usually results in a cascade of microservice calls. Once the request has been processed by all the microservices in the workflow, the result will be delivered back to the client. Such a distributed system, is more vulnerable to failures. One of the promised characteristics of microservice architectures is fault tolerance and graceful shut down. To enforce this characteristic, the failure of one microservice instance must not hinder the users from getting responses from the other microservices in the system. One measure to ensure this is to introduce circuit breakers in between the microservice units (Montesi and Weber 2016). Circuit breakers are proxies which function similar to the electric circuit breakers. In order to ensure that the failure of one particular microservice, has reduced impacts on the functioning of the system as a whole, these proxy circuit breakers trip down after a certain number of failed requests to a service, and all further requests made to the service is terminated immediately.

1.2.4 12-factor app

Though a consensus on an exact definition for microservices, is yet to be attained, there are several guidelines on the approaches that may be adopted to decompose a system into microservices. One of the most referred guidelines is the 12-factor app, which was originally introduced by the Heroku in 2011 [3], to guide the development of applications to be run on the Heroku platform. In other words, these provide the guidelines to

[3] 12factor.net/

develop Cloud-native applications [4](Cito et al. 2015). The intent of the 12-factor app is to build scalable and portable applications that are deployable in the Cloud. These targets also align with those of the microservices architectures. Thus, the 12-factor app can be adopted as a patten for developing microservices (Torkura et al. 2017). These steps ensure that the application may be deployed and modified easily by the same developer or other developers, to incorporate new features.

After development, the microservice applications are generally packaged and deployed as containers. Section 1.3 sheds light on the container virtualization technology.

1.3 CONTAINER VIRTUALIZATION AND CONTAINER TECHNOLOGIES

The virtualization technology, which is considered as the backbone enabler of Cloud computing, can be broadly classified into two: hypervisor-based virtualization and container-based virtualization. In hypervisor-based virtualization, a Virtual Machine Manager (VMM) runs atop the host Operating System (OS) and each Virtual Machine (VM) instance includes its own guest OS. Containerization provides lightweight virtualization through creation of self-contained application packages from image files. The guest instances operate at virtualization layers at the OS kernel level.

Since 1970s, containerization has been used to provide isolated environments since the 1970s. However, it was only by the year 2012, that Linux distributions introduced kernel features such as *namespaces*, *chroot* and *control groups*, leading to the emergence of Linux Containers (LXC). Various container technologies such as OpenVZ, and CoreOS have been introduced. In 2013, the open-source Docker container technology was released, which catapulted the applicability of containers in the software industry. Docker image files follow a layered structure known as Advanced multilayered Unification File System (AUFS).

Cloud based applications were usually run on VMs. Cloud applications based on the microservices architecture offers several advantages over the conventional monolithic architectures. However, these advantages may be fully exploited only when it is deployed in environments which provide adequate support for the architecture. When

[4] `cyberlearn.hes-so.ch/mod/resource/view.php?id=766219&redirect=1`

microservices are deployed on VMs, the limitations of VMs, such as virtualization overhead, degraded performance, will hinder the microservices from providing the expected benefits. Based on this observation, several researchers explored the use of lightweight virtualization techniques in deploying microservices (Amaral et al. 2015; Kang et al. 2016b; Peinl et al. 2016; Stubbs et al. 2015).

Container virtualization can be materialized in different forms: application containers and system containers. Application containers provide a more viable option for the deployment of microservices. These containers wrap applications along with their dependencies into a self-contained unit which can be deployed independently. They have *lower startup-times* and offer *near-native performances* in several scenarios. Another feature of containers is that they contain the application and all the libraries required to run the application. This improves the *portability* of the application. These characteristics of containers provide the agile environment required to deploy microservices. These features of containers make it the unanimously accepted technology to enable seamless execution of microservices in the Cloud. The various activities in the lifecycle of containers such as creation, deployment and termination, are managed by container management platforms, as detailed in Section 1.4.

1.4 CONTAINER MANAGEMENT PLATFORMS

The emerging dominance of containers in the Cloud Computing environment, has led to the development of several platforms dedicated for the management of container-based systems. These platforms may be adopted to run microservices in containers. A detailed study on the different platforms has been presented by Peinl and Holzschuher (2015). The major platforms are Apache Mesos Marathon, Amazon Elastic Container Service (ECS), Docker Swarm, Google Kubernetes (earlier Google Borg) and Hashicorp Nomad. A comparison of the major container management platforms is provided in Table 1.1. Container runtimes may be employed to spawn application containers in single host scenario or across multiple hosts.

Container Management Platforms	Placement / Allocation	Scaling rules	Load Balancing	Monitoring	Application Description
Apache Mesos Marathon	Dominant Resource Fair (DRF) Sharing	Threshold-based (CPU or memory utilization)	Marathon LB (HAProxy) or virtual IP based	HTTP endpoints	Application definition (JSON)
Amazon ECS (Elastic Container Service)	Bin packing, spread, affinity, distinct instance	Threshold-based (CPU or memory utilization)	Application Load Balancers, Network Load Balancers, and Classic Load Balancers	Cloudwatch	Task Definition (JSON)
Docker Swarm	Bin packing, spread, random	No in-built support	Layer 4 TCP (internal) DNS and virtual IP based Round Robin	Prometheus	Service definition (declarative)
Google Kubernetes (k8s)	Fit Predicate (labels), node/pod affinity & anti-affinity & taints	Threshold-based (CPU or memory utilization)	Layer 4 TCP Round Robin (kube-proxy) Layer 7 HTTP Round Robin	Heapster, cAdvisor	Service (REST)
Hashicorp Nomad	Bin Packing	Manual CPU/ memory based scaling	HAProxy TCP/HTP Load Balancing	Nomad agent with HTTP endpoint	Jobspec (HCL)

Table 1.1: Comparative Analysis of container management platforms

In single host systems, multiple containers are handled by the Linux kernel by considering them as equivalent to the other processes running in the system. The scheduler used to schedule containers on a single host is the Completely Fair Scheduler (CFS). The CFS maintains a list of incoming requests in the form of a red-black tree. The nodes are structured on the basis of their execution times.

The container management platforms enable spawning containers across multiple hosts connected to form a cluster. The platform takes care of deploying containers, provides load balancing capabilities and scaling functionalities. The platform decides where a container should be deployed in the cluster, initiates the container and provides monitoring capabilities (usually with the help of other monitoring tools). In such multi-host systems, the allocation policies generally followed are as follows:

- **Dominant Resource Fair Sharing (DRF)**: This strategy attempts to maintain fairness across different requests with multi-resource requirements. When job requests arrive, the resource requirements for the various resources are considered. The proportion of each resource requested (with respect to the available resources) is calculated. For each job, the resource with a higher proportion is considered to be the dominant resource. Then, the strategy strives to allocate equal proportion of the dominant resource for all the jobs.

- **Bin Packing:** This strategy attempts to complete all the resources on one node before allocating containers to the next node. It tries to allot as many containers as possible to the current node.

- **Spread:** The strategy places the containers on the different nodes in the cluster by selecting them in a round-robin manner.

- **Random:** A node is selected at random to host the container. The selected node must satisfy the constraints specified.

- **Distinct Instance:** Each incoming container request is allocated to a distinct host.

- **Custom labels/rules:** The allocation is done based on the information provided

by the user. The information may be in the form of tags, affinity values or anti-affinity values.

Microservice applications are generally composed of numerous microservice components hosted across multiple containers. Tackling the operational complexities of such distributed systems presents numerous challenges. Though existing systems have employed these container management platforms as a middleware, they possess several limitations. The current container management systems do not support dynamic management of the container system. In addition, the allocation, load balancing and scaling rules adopted are ingenious. To overcome this, developing efficient microservice orchestration mechanisms is extremely important.

1.5 MICROSERVICE ORCHESTRATION

The resources in a microservice ecosystem must be managed across different phases of the microservices' lifecycle by means of processes and/or services to perform operations like resource selection, monitoring and controlling. All these activities collectively fall under microservice orchestration. For container-based microservices, orchestration involves automation of the process of co-ordinating the work of individual microservice application containers. This involves the automation of tasks such as load balancing, service discovery, provisioning and deployment. The recent practice of incorporating microservices across different layers of the application stack results in having multiple components that may not seamlessly integrate as a single unit. Having individual entities governed by different demands and dependencies raises the need for microservice orchestration. In such scenarios, conventional service orchestration approaches may deem insufficient due to the specific characteristics of microservices such as agility, independence and asynchronous communication. In this thesis, the primal focus is on the microservice orchestration activities of microservice allocation and microservice re-scheduling.

In a nutshell, the contributions of this thesis can be categorised into three. First, solutions to different problems in microservice-based environments are studied. Second, a strategy for performing allocation of microservice components in Cloud datacenters,

is proposed. Third, a framework for performing rescheduling in microservice-based Clouds is proposed.

1.6 MOTIVATION

According to recent studies, the global microservices market is expected to grow at a Compound Annual Growth Rate (CAGR) of 22.4% over the years 2018 to 2023 (Research and Markets 2018). Enterprises worldwide have rapidly embraced the MSA to develop and deploy applications. The switch to MSA enabled organizations to overcome the performance barriers emanated by traditional monolithic applications. An example is the significantly large performance improvements attained by Walmart's migration to microservices. The Information Technology (IT) team of Walmart in U.S. was confronted with downtime issues on Black Fridays for two successive years. This originated from the failure in being able to scale for handling around 6 million page tracking hits per minute. Replatforming to MSA enabled Walmart to resolve all downtime issues and moreover resulted in energy and cost savings (around 40% and 50% respectively) (Vizard 2015).

Cloud applications structured as microservice components facilitate the optimal usage of the Cloud resources (Wu 2017). These microservices running on the Cloud infrastructure exhibit many beneficial features like scalability and resilience. To ensure this, the ecosystem of microservices must be dynamically orchestrated to improve resource utilization, minimize costs and meet the heterogeneous Quality of Service (QoS) requirements. The deployment units have different options for packaging, either as VM or container images. In containerized Cloud environments, the microservices are deployed across clusters of multiple containers that must be coordinated to ensure seamless service delivery to end-users. Microservice orchestration includes activities such as selection, deployment and dynamic configuration management that takes into consideration the heterogeneities of the microservices, container engines and Cloud datacenter resources that support the execution of the microservice applications (Zhong and Buyya 2020). The focus of this research work is on the scheduling and re-scheduling activities in containerized environments.

Scheduling strategies form the crux of microservice orchestration platforms. They play a primal role in harnessing the benefits of the underlying resources. Although a plethora of approaches have been proposed to address the application scheduling problem in Cloud environments, research focussing on the specific characteristics of microservices is essential (Fazio et al. 2016). *Using the traditional scheduling approaches may result in compromised QoS values as, contrary to the standalone monolithic applications, microservice applications are constituted of multiple interacting microservice units* (Esposito et al. 2016). *Accordingly, there is a dire need to consider the communication among the multiple microservices constituting an application, while taking scheduling decisions.* Hence, the scheduling strategy developed in this research attempts to place the frequently interacting microservices together.

Regardless of the optimality of the initial scheduling strategy, the quality of the arrangement degrades with fluctuations in resource utilization and workload rate (Rodriguez and Buyya 2020; Zhong and Buyya 2020). Therefore, *re-arrangement of the microservice containers through re-scheduling activities is vital. The majority of existing research targets the re-scheduling in hypervisor-based virtualized datacenters. However, adopting these approaches in containerized datacenters would fail to achieve optimal performance.* Hence, the re-scheduling solution developed in this research, considers the container configuration parameters in combination with the resource usage statistics to perform re-scheduling in containerized datacenters.

1.7 DISTRIBUTED COMPUTING PARADIGMS EMPLOYING MSA

The emerging MSA architectural pattern has been applied to several computing realms, such as ubiquitous computing (Fano and Gershman 2002). The MSA may be used to realize other recent computing paradigms/ models, such as:

- **Reactive Computing:**

  Reactive programming is programming with asynchronous data streams. A stream is a sequence of ongoing events ordered in time. The emitted events are captured only asynchronously, by defining a function that will execute when a value is emitted and another function when an error is emitted. Responsiveness is one of

the key characteristics of reactive systems (Bonér et al. 2014). Microservices can be effectively used to achieve Reactive Computing functionalities.

- **Osmotic Computing:**

  The Osmotic Computing paradigm was conceptuated by Villari et al. (2016). This paradigm took birth at the intersection of the Cloud and Edge Computing domains. Applications in the Osmotic Computing paradigm are deployed as microservices. One of the central elements of the Osmotic paradigm is the decision maker which maps each microservice to the best location (Nardelli et al. 2017). Villari et al. (2017) also proposed an Osmotic flow, which is a model for modelling and executing the IoT workflow applications in an Osmotic environment. Sharma et al. (2017) propose a fitness-based algorithm which considers the load, energy and processing time requirements and then take a threshold-based decision for the placement of the service request. Medical domains can also harness the potential of the osmotic computing environment, by leveraging them to develop Hospital Information Systems (HIS). Carnevale et al. (2017) describes one such approach , where the application is decomposed into microservices and deployed across the Cloud and edge resources.

- **Serverless Computing and Functional PaaS:**

  Serverless computing allows users to build and run applications and services without thinking about servers. One of the models that enable clients to leverage serverless computing capabilities is through Function-as-a-Service (FaaS) (Van Eyk et al. 2018). FaaS provides a platform allowing the developers to execute code in response to events without the complexity of building and maintaining the infrastructure. The tenets of serverless and FaaS can be materialized by adopting fine-tuned microservices (Van Eyk et al. 2019). Contrary to other computing paradigms, FaaS eliminates the need to keep components up and running at all times by employing time-sharing principles, thereby cutting down operating costs to a great extent.

- **Cloud Computing:**

  Cloud Native applications (CNA) are applications built specifically for running on the Cloud. Though there exists no formal definition for the term, these applications are said to exhibit certain properties called the CNA properties, which include scalability and elasticity (Khan 2017). Other properties include dealing with constant failures, support for Continuous Integration and Continuous Development (CI/CD) and security. In order to develop applications with the CNA properties, it was necessary to decompose applications into functional units. Thus, the CNA are architected as microservices that are generally stateless and communicate using Representational State transfer or Remote Procedure Call (RPC)- based calls. The CNA are mostly run on containers, which provide isolation and enable packaging of all dependencies as a single unit. Decomposing applications into microservices provide agility and ease the maintainability of microservices.

- **Other Distributed Computing Paradigms:** The Industry 4.0 Revolution has fostered the adoption of IoT, Fog computing and other related edge computing paradigms (Yousefpour et al. 2019). The aforementioned paradigms necessitate the development and deployment of applications as interconnected, autonomous modules. These features are perfectly in alignment with the characteristics of MSA-based applications (Pallewatta et al. 2019). In addition, other requirements of IoT applications such as robustness, modularity, scalability and resilience are also met by the MSA (Buzachis et al. 2019; de Santana et al. 2019). Consequently, the MSA has been widely embraced by application developers.

1.8 ORGANIZATION OF THE THESIS

This thesis is organized as illustrated in Figure 1.3. Chapter 1 provides a brief preface to the domain of microservices and containerized Cloud environments. Chapter 2 presents a taxonomy of the different aspects of MSA and a comprehensive study on the related research works. Chapter 3 describes the research problem that forms the focus of this thesis. Chapter 4 presents an interaction-aware allocation algorithm to effectively place

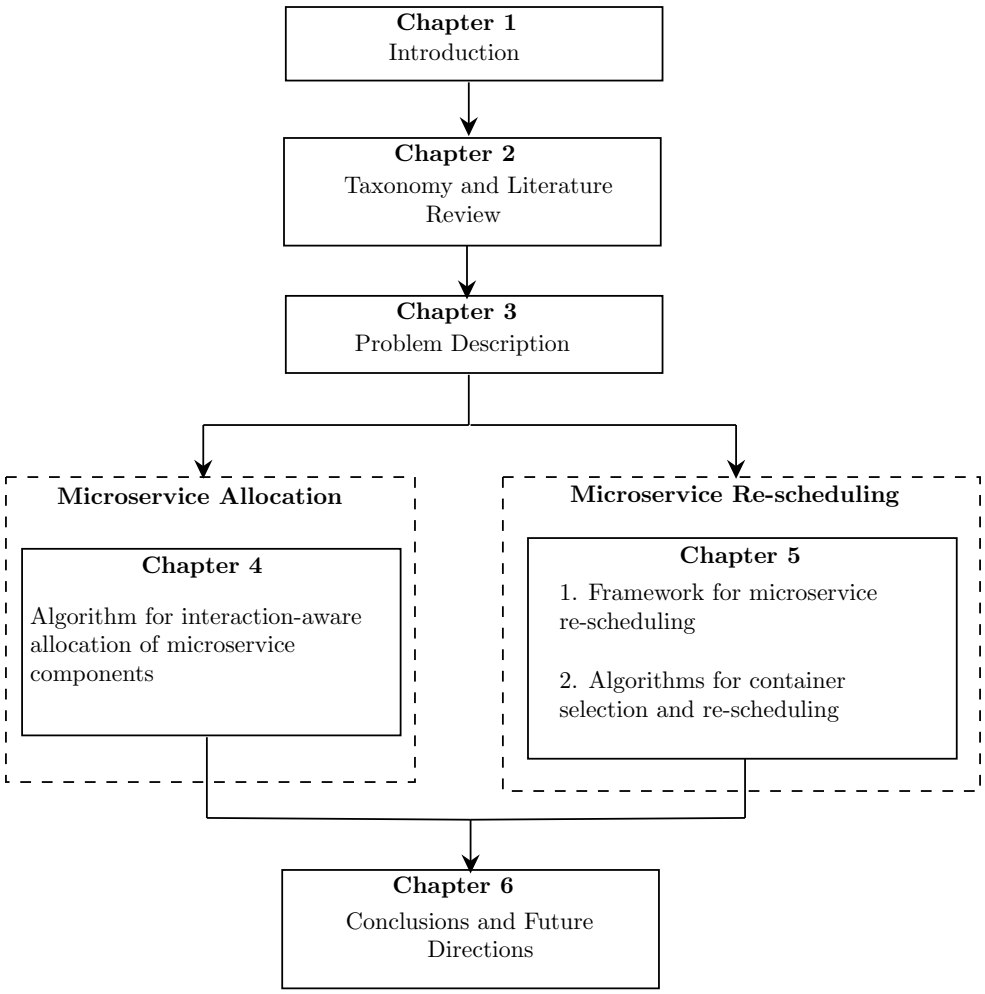


Figure 1.3: Organization of the thesis

microservice components in containers across the multiple hosts in a Cloud datacenter. Chapter 5 describes the proposed framework to perform dynamic re-scheduling in Cloud datacenters. Chapter 6 outlines the findings of the thesis and also provides the future research directions.

CHAPTER 2

LITERATURE REVIEW

Researchers and practitioners have adopted microservices into several application domains such as Internet of Things, Cloud Computing, Service Computing and Healthcare. Apart from the development of microservice-based applications, devising techniques to efficiently coordinate and manage the various microservices is equally essential. Applications developed in alignment with the microservices principles require an underlying platform with management capabilities to coordinate the different microservice units and ensure that the application functionalities are delivered to the user. A multitude of approaches have been proposed for the various tasks in microservices-based systems. However, since the field is relatively young, there is a need to organize the different research works. Accordingly, a multi-level taxonomy to categorize the existing research is provided in Section 2.1 of this chapter. The section also presents a comprehensive review of the research approaches directed towards microservice architectures. Section 2.2 presents the outcome of the study conducted and enlists the research gaps in the domain.

2.1 TAXONOMY BASED ON DIFFERENT ASPECTS OF MICROSERVICE ARCHITECTURES

Since 2014, when the microservice architecture was first introduced, several researchers have attempted to overcome the various challenges in adopting microservice architectures to develop systems and applications. As the trend in research community, there are a lot of studies, including systematic reviews and mapping studies (Dragoni et al.

2017; Fernández Villamor et al. 2010; Katuwal 2016; Kratzke and Quint 2017; Mazzara et al. 2016; Pozdniakova and Mazeika 2017; Salah et al. 2016; Zimmermann 2016) that attempt to review the research works carried out in this domain. These include studies that consider sub-domains of MSA related concerns, such as security (Almeida et al. 2017) and granularity (Hassan et al. 2020) of microservices . A majority of these studies consider the software engineering aspects of microservices. The systematic studies review the works, with the aim of identifying possible research directions and the research trends in the domain. They do not attempt to provide a classification taxonomy for the various aspects of microservice architectures. This work considers all research works and solutions by the industry, in the different sub-domains of microservices and attempts to classify the research studies into different categories, based on the aspect considered.

The proposed taxonomy classifies the existing research broadly into two: research addressing *development phase* and *operational phase* aspects. The classification taxonomy is illustrated in Figure 2.1. The taxonomy considers research efforts that address issues either in the development phase or the operational phase.

The developmental phase concerns include the challenges faced in the development of microservice-based systems (such as deciding the size of microservices, approaches to model the requirements for the microservice-based systems, all of which come under the Software Engineering concerns, already discussed in existing surveys). Research studies that fall under this category have been briefly discussed in Section 2.1.1.

The operational phase concerns include the challenges faced once the microservices has been deployed into the system. Rolling out new updates, constant monitoring, and other issues come under this category. This category of research works has not been considered in the earlier studies. The issues in this category can be further classified into two: the infrastructural management issues and the service management issues. The works in each category have been reviewed and analyzed in Section 2.1.2.

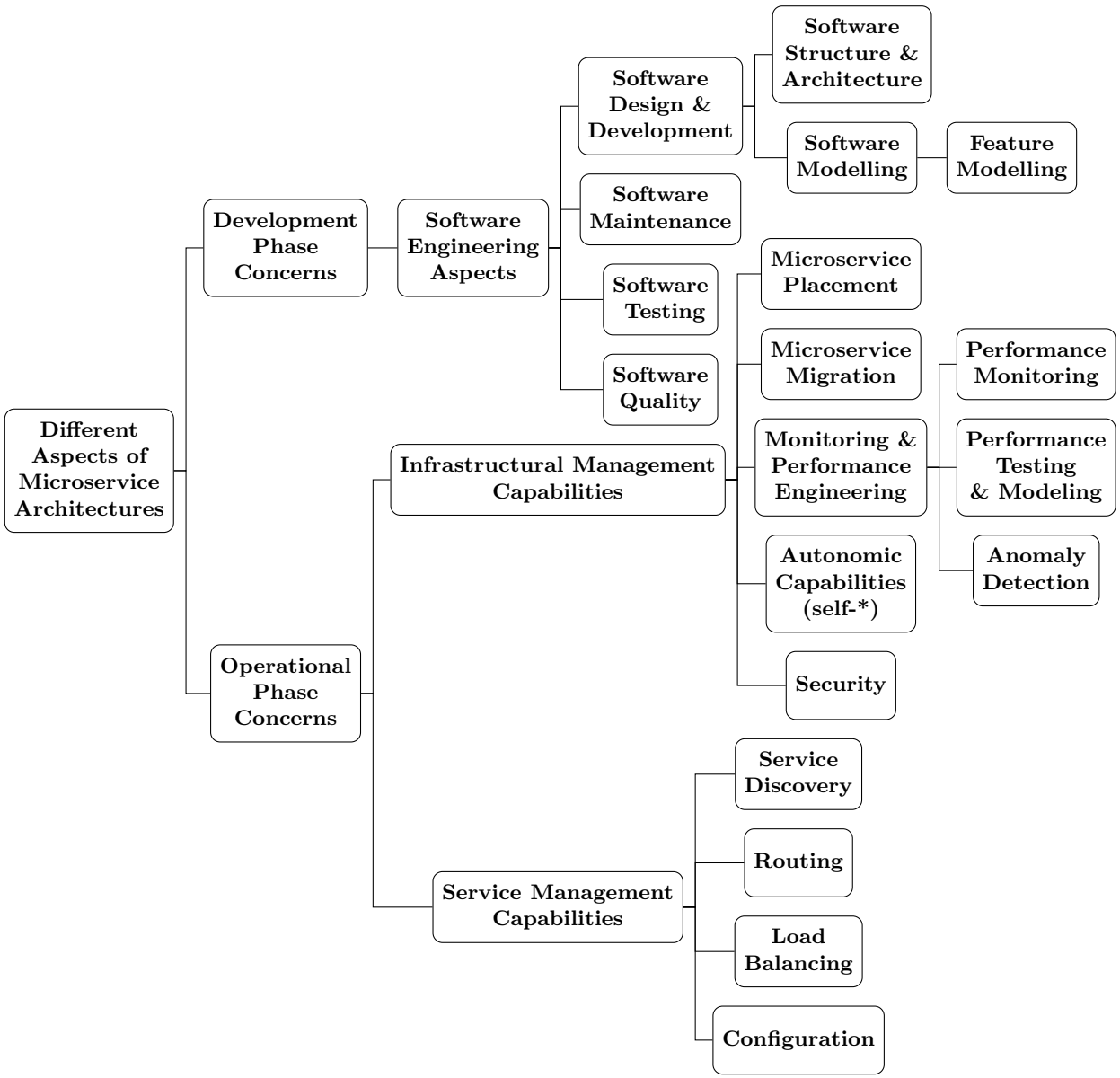


Figure 2.1: Taxonomy for the different research works in the literature under the domain of Microservices Architectures

2.1.1 Developmental Phase Concerns

This category considers challenges or issues faced in the development of microservices based systems. The challenges come under the Software Engineering challenges.

Software Engineering

Microservice architectures are by and large associated with continuous software engineering, where software is updated on a very frequent basis (O'Connor et al. 2017). It is common that Software Engineers use microservices to develop systems that are agile (Rosenberg et al. 2017). The process of engineering microservices poses several challenges. There is a need for research on the design of microservices (Tai 2016), as the architectural pattern brings about several challenges in the field of Software Engineering. There are different Software Engineering sub-disciplines that have been considered by researchers working in the domain of microservices such as software design & development, software maintenance, software testing and software quality.

1. **Software Design & Development:** The design of microservice system encompasses different elements, which include: Design of the service (or microservice), Solution, Process and Tools, Organization and Culture (Nadareishvili et al. 2016). The design element involves the design of the microservice units themselves. The decision challenge is to correctly determine the microservice boundaries and the granularity. It also deals with the designing of the APIs. The solution provides a global view of the microservice system, which follows certain architectural patterns. Selection of the tools and enablers for the development of the microservices system, directly influences the characteristics of the system, such as resistance to change, and therefore, is a vital part of the design phase. Apart from this, the formation of teams developing the microservices, is another design-level concern. Cultural factors also contribute to determining the properties of the developed system. The scope of the MSA architectural style is often restricted only to the back-end or server-side of applications. Harms et al. (2017) demonstrate the viability of the microservices architectural style for providing the front-end services. The different types of front-end architectures are: monolith,

plug-in and the self-contained systems (SCS). Tizzei et al. (2017) conducted a study on how Software Product Lines (SPL) processes can be integrated with the microservices architectural concepts.

- **Software Structure & Architecture:** The design of microservices can adopt different architectural patterns. The architectural pattern language developed by Chris Richardson (2017) is a conglomeration of different patterns with relationships defined between them. Granchelli et al. (2017a) proposed the MicroArt tool that employs model-driven concepts and can be effectively used by the software architects (Di Francesco 2017). Kleehaus et al. (2018) proposed Microlyze, that can enable developers to recover the software architecture of microservices, using data collected in the monitoring process. At the design level, different decisions can lead to different patterns. In order to ensure that the design complies with the patterns, Zdun et al. (2017) proposed few metrics that can be used to assess the compliance with the patterns. Most of the metrics are associated with the number of components shared among the microservices. Ideally, microservice architectures strive to keep 'sharing' at the bare minimum.

- **Software Modelling:** In order to develop microservices-based systems, it is necessary to devise techniques capable of representing user requirements as design specifications. This implies that there is a need for researchers to work on developing methods and languages for modelling microservice-based applications. Rademacher et al. (2017) presented a study on the model-driven development techniques for SOA that may be equally pertinent for microservices architectures as well. Metamodels for MSA must capture additional information, such as the container technology used and communication protocol, among other details. Modelling languages for MSA must take into consideration the features of MSA, such as single capability and polyglot nature, where each team can have their own choice for the modelling language as well. Petrasch (2017) proposed extended Unified Modelling Language (UML) diagrams with the 'microservice' stereo-

type to support MSA. The UML diagrams are then subjected to Enterprise Integration Patterns (EIP) to extract accurate specifications to engineer microservice applications that also reflect the communications carried among the microservice components. Diepenbrock et al. (2017) developed an approach to apprehend the ontological information in domain metamodels, that can be used to model microservices. Microservice ambients proposed by Hassan et al. (2017) can be used to model microservices. Granchelli et al. (2017b) proposed an approach for Model Driven Reverse Engineering. They developed a MicroART prototype that is capable of extracting the architecture of the system, when provided with the source code. Zúñiga-Prieto et al. (2017) proposed an extended architecture model that can integrate several models of microservices and can be used to guide design decisions for the code generation phase. Düllmann and van Hoorn (2017) proposed a metamodel for MSA that can be used to generate synthetic microservices to simulate the changes arising in the deployed microservices, which can then aide the collection of data for monitoring. Model-integrating components along with microservices can be applied to better support continuous software engineering (Derakhshanmanesh and Grieger 2016). The modelling language combining both these concepts can provide as a model language whose applicability spans the entire lifecycle of the software. The Community Application Editor (Lange et al. 2016) tool can be used to model web applications based on microservices.

One of the important instruments used by software engineers for domain modelling is feature modelling, that can be used to represent the variability in the applications.

Feature modelling Klock et al. (2017) proposed the MicADO (Microservice Architecture Deployment Optimizer) tool which accepts the architecture model and workload model to develop a possible feature placement model depicting the features provided by each microservice. Given the set of features to be provided by a microservice system, and the expected work-

load, the tool outputs the features that will be provided by each microservice, by applying the Genetic Algorithm metaheuristic.

2. **Software Maintenance:** This phase starts once the software has been developed. The maintenance activities involve rectifying faults in the working of the software and several other tasks such as modifying the software to adapt to external changes, or providing enhancements to the software to improve the performance. Microservice architectures are expected to reduce the costs incurred for maintenance activities (Esposito et al. 2016). Bogner et al. (2017) observe that most of the metrics, related to measurements of maintainability involving the size and complexity or degrees of coupling and cohesion, are applicable to evaluate the maintainability of microservice-based software as well.

3. **Software Testing:** Along with the benefits that the microservices architectures offers, it also introduces new challenges in the software testing domain. Developing effective testing strategies for the microservices applications is a challenging task (Sundar 2017), owing to the dynamicity and agility involved. The polyglot nature of the microservices architecture, coupled with the continuously changing environment of the microservices architecture deems it challenging to perform testing for the system resilience using conventional methods. However, it is most essential to analyze the fault handling capability of the microservices. Heorhiadi et al. (2016) proposed Gremlin, a technique underpinned by Software Defined Networks (SDN) to enable systematic testing of the microservices- based systems. The faults to be injected are communicated to the control plane in the form of Python recipes, which then takes the appropriate actions on the data plane. The major advantage of the Gremlin approach is that the latency involved in getting the feedback is very low. In order to develop a testing strategy tailored to the requirements of the microservices architectures, Savchenko et al. (2015) analysed the features specific to testing of microservices and then proposed a framework called Mjolnirr, for the validation and testing of microservices. Savchenko and Radchenko (2015) proposed a framework for validation of microservices that spans the entire development cycle of the application, and can provide an abet

for the Mjolnirr system. Testing the functional aspects of the microservice-based systems can be done by adopting a black-box approach, using a learning-based strategy (Meinke and Nycander 2015). The approach involves modelling the systems as Finite State Machines (FSM) and then checking for System Under Test (SUT) errors. In addition, faults may be injected to verify the robustness of the system. Schermann et al. (2016) proposed Bifrost middleware to perform live-testing in microservice-based systems. The proposed middleware, which is based on the formal model specified in a DSL, can be used to automatically perform live testing in different phases across the rollout process.

4. **Software Quality:** The microservices architecture design is perceived by different practitioners in different forms. In view of the fact that there are no formal definitions or guidelines to develop microservices architectures, it deems imperative that the developed software be verified for conformance to design requirements. Possible quantifiers for software quality of microservice architectures include the size of the service, that may be measured by considering the resources accessed and the number of external clients communicating with the service. Communications between the services also needs to be considered to measure the software quality. Asik and Selcuk (2017) proposed a static tool that can be used to effectively analyze the microservice based software and rate the quality of the software. Ulander (2017) observed that structural metrics such as flexibility, reusability and understandability are effective measures of the quality of the developed software. However, these metrics vary according to the characteristics of the system where the software is deployed.

2.1.2 Operational Phase Concerns

Microservice Architectures (MSA) aim at making application development simpler by decomposing the application into different modules and assigning different teams the responsibility of each module. Each team can develop the microservice while choosing from a wide range of technology options that best support their requirements. However, the backstage tasks for supporting such an application is plenty. Removing the com-

plexities in application development is actually virtual and is realized by pulling more responsibilities to the supporting environment. This implies that an ecosystem that supports the microservice-based applications has more responsibilities, such as handling the communication between the entities and ensuring the overall co-ordination and seamless operation of the several distributed entities. This section discusses the various operational functionalities that come under the responsibilities of the supporting infrastructure. The different tasks can be broadly classified into two: infrastructural management capabilities and service management capabilities. Research works falling under the two categories are discussed in Sections 2.1.2.1 and 2.1.2.2 respectively.

2.1.2.1 Infrastructural Management Capabilities

The microservices are generally run on containers that are spawned on physical machines or hosts. Ensuring that the microservices are provided adequate resources without interrupting the microservice activities such as scaling, observing the microservice instances for their runtime behaviour and making corresponding adaptations in the infrastructure, and several other activities come under the management tasks of infrastructure for microservices (Sousa et al. 2016; Wizenty et al. 2017). The infrastructural management capabilities may be viewed as the activities at the boundary between the microservice instances and the underlying infrastructure. The major activities are microservice placement, microservice migration, performance monitoring & engineering, autonomic management and security mechanisms. The research state-of-art for each of the activities are discussed in this section.

1. **Microservice Placement:**

   Container runtimes offer the most suitable deployment strategy for the microservice architectures. With the surge in usage of containers in Cloud data centers, the efficient allocation of containers is a key concern with regard to the resource utilization and the overall performance of the system. There are container orchestration tools to perform the container lifecycle management. However there are unsettled issues in the container resource management. Guerrero et al. (2018a) proposed a NSGA-II based approach for the initial placement of microservices

on containers. The authors considered the objectives of balancing the system workload, system reliability and the network overhead. To quantify the objectives the authors proposed metrics and employed the metaheuristic technique to solve the optimization problem. Selimi et al. (2017) proposed an approach to effectively place services in a network. The state of the network in which the services are to be deployed is also to be taken into account. However, this approach does not consider the specific characteristics of microservices architectures. The MiCADO tool (Kiss et al. 2017) includes an optimised deployment phase, where the application description is taken as input, and an optimal deployment schedule is generated, taking into consideration the QoS parameters and security requirements. However, the authors proposed only a generic framework and implementation specifics are not discussed. Other relevant research works include the works in the literature for container allocation.

2. **Microservice Migration:** Microservices architecture is transpiring as the predominant application architecture for Cloud Computing. In this milieu, the necessity of orchestrators and schedulers cannot be neglected. One of the techniques which the orchestrators may employ to ensure optimality in the environment is microservices migration, which involves discontinuing the execution of a microservice at the current location and re-scheduling the microservice to another node. This process may be initiated for the objectives of better energy consumption or balanced load across all nodes or for maintenance purposes. Rusek et al. (2016) proposed de-centralized approaches based on the bio-inspired swarm algorithms for initiating the migration process. The probability to migrate a microservice is deduced from the pheromone calculated value. As an extension to this research, Karwowski et al. (2017) also proposed a swarm-intelligence based approach for an environment that consists of non-homogeneous hosts, varying in CPU and memory capabilities.

3. **Performance Monitoring & Performance Engineering:** Performance Engineering deals with ensuring that the non-functional requirements are met. Performance Engineering considers the supporting infrastructure and monitors the

non-functional requirements, in order to ensure that the service levels are met. By virtue of the agility and Continuous Integration and Continuous Deployment/Delivery (CI/CD) properties of MSA, there is a need to investigate how performance engineering activities can be carried out for microservices. The conventional methods may prove insufficient. Performance Engineering involves: performance monitoring, performance testing & performance modelling and anomaly detection.

- **Performance Monitoring:** Microservice Architectures deployed in an environment undergo rapid and continuous changes, in the form of new versions replacing older ones or existing versions being upgraded and so on. All these changes can be vital in the context of application performance monitoring. In a microservices-based system, performance monitoring involves the collection of several metrics such as service request response times, resource consumption, interactions among the components, etc. Haselböck and Weinreich (2017) performed a comparison study of the various monitoring tools. These tools can be used to collect metrics at the service level and/or the infrastructure level. Based on the results obtained from their study, they proposed models that can aid developers and operators in decision-making for microservices monitoring. The models can be used to take decisions on which monitoring tool may be employed to collect the metrics that the user is interested in. Mayer and Weinreich (2017) conducted a study to collect the parameters of microservices based systems that needs to be monitored. Apart from the system metrics, certain static and dynamic metrics also need to be monitored, such as API-related metrics and workload of each service. Thalheim et al. (2017) observed that transforming the monitored metrics into useful information is equally important as the monitoring activity itself. A framework, SIEVE, was proposed to perform the metrics reduction using clustering, to reduce the dimensionality and then derive the dependencies among the metrics by performing the Granger causality test.

- **Performance Testing and Modelling:** Performance tests are performed on software systems, to analyze whether the non-functional requirements are satisfied or not. One of the main challenges offered by the microservice systems is testability. The different component microservices will have to be tested individually and each of these microservices may be modified on the fly. To add to the complexities, each of the microservice component may be developed using different languages, which should be taken into consideration for developing effective testing strategies. de Camargo et al. (2016) proposed techniques to carry out the performance tests on microservices in an automated manner. The proposed approach requires each microservice to include a test specification which provides the methods available and sample input data to generate requests to the microservice. Heinrich et al. (2017) evaluated the performance of the nested and master-slave models for microservices running on containers. Khazaei et al. (2016) proposed a performance model to check the efficiency of the provisioning phase for microservices. The authors resorted to metrics like response time, utilization and rejection rate.

- **Anomaly Detection:** In constantly changing environments of the Microservices Architectures, performance monitoring can be a challenging task. The observed metrics will have to be constantly analyzed to detect if there are occurrences of any unwanted events. In order to ensure that the system is functioning as expected, the data must be analyzed and anomaly detection techniques must be applied to detect any deviations from the expected behaviour. One method to detect anomalies is to measure the response times. However, owing to the continuously changing nature of the environment, several false positives may be detected, arising due to the increase in response times due to new versions of the microservices instances being released or due to upgrades performed on the existing microservices. In order to deal with this, microservice architectures may be generated from models to resemble the actual scenario, and these generated microservices

may be used to collect data for monitoring. At a fundamental level, anomaly detection is usually done by collecting historical data, deducing a baseline from the collected data and analysing the future data to identify any deviations from the baseline data. Any deviation is considered to be an anomaly. However, this technique cannot be used to detect anomalies in the microservice architectures. Event information also needs to be taken into account for anomaly detection in MSA-based systems. Düllmann (2017) proposed the Event-Aware Anomaly Revision (EAR), which involves event-triggered increasing and reduction of the anomaly threshold.

4. **Autonomic Capabilities for MSA-based systems:** The dynamicity of microservices introduces several changes in the environment, to which the system has to respond appropriately. The management of microservices can be done in an autonomic manner, where the system is self-managing. Autonomic Systems are capable of adapting to the changes in its environment with minimal intervention of human users. Autonomicity in cyber-physical systems is generally achieved by employing the Monitor-Analysis-Plan-Execute (MAPE) control loop. Toffetti et al. (2015) observed that the management functionalities can be incorporated as part of the microservices itself rather than separating it out to an external entity. Regular monitoring must be done to ensure that the topology is maintained and any failures occurring must be handled to avoid any hindrance in the functioning of the system. Florio (2015) observed that self-adaptive mechanisms are required to handle the fluctuating workload in a system of microservices. Baylov and Dimov (2017) proposed a reference architecture, for developing microservices systems with self-adaptive capabilities. The proposed architecture, inspired by reference models for SOA, augmented each service with an autonomic manager, responsible for performing the actions of the MAPE loop. The self-management capabilities are generally classified into four: Self-Configuration, Self-Healing, Self-Optimization and Self- Protection. In addition to this, systems that support elasticity also exhibit autoscaling mechanisms. The following subsections discuss the works that incorporate Self-Configuration, Self-Optimization and Autoscal-

ing capabilities in microservices-based systems.

- **Self-Configuration:** Self-configuring systems for microservices are capable of deploying the microservice components in an autonomic manner. Gabbrielli et al. (2016) proposed an approach that autonomically derives the optimal placement mapping for microservices. The authors considered a system where the microservices are reconfigurable. The approach relied on a tool, Zephyrus, that is adept in generating the optimal architecture that convey the number of instances and their distribution, when provided with an abstract-level description of the microservices-based application.
- **Self-Optimization:** The resource allocation task in dynamic environments is an NP-hard problem. Several researchers have attempted to develop solutions for the task of optimizing the configuration in microservice systems. HoseinyFarahabady et al. (2017) proposed a solution for the resource allocation problem in microservice environments. A controller controlled the resource allocation process based on the predicted future events, while satisfying QoS constraints. The authors considered a metric termed as the QoS detriment value, which measures the QoS violations.
- **Autoscaling:** One of the quintessential features of microservices is scalability. The microservice instances responsible for one functionality receives service requests from the client. When the number of client requests to the microservice increases, replicas of the microservice may be instantiated to better serve the requests. Similarly, as the number of requests decreases, few replicas of the microservices may be terminated. These scenarios are frequently encountered in an ecosystem of microservices. Thus, it is advocated that these actions may be carried out in an automated manner, by incorporating autonomicity. Kampars and Pinka (2017) observed that effective scaling mechanisms take decisions based on the observed values of application-level metrics. The objective of these autonomic systems is to maintain the QoS level perceived by the users. The authors proposed a Context Driven approach which incorporates machine learning and graph

processing capabilities provided by the Spark framework to take scaling decisions. Kukade and Kale (2015) proposed a threshold-based scaling mechanism for microservices. The services provided and the history of the client requests are constantly monitored to take scaling decisions. López and Spillner (2017) observed that too many replicas may also deteriorate application performance, due to the Amdahl's law effect and several services competing for the limited amount of resources available. The authors adopted a mathematical approach to determine the optimal number of service instances, based on the application characteristics and the number of incoming requests. MiCADO- Microservices based Cloud Application-Level Dynamic Orchestrator proposed by Kiss et al. (2017) provided automated scalability for Cloud Applications deployed as microservices. The MiCADO tool takes in the application topology and the QoS requirements to generate the deployment of services for the application.

5. **Security:** The offerings of microservices architectures do not come free of cost. There are several challenges inherent in the microservices architectures. One specific challenge is considering the security aspect. Several features of microservices introduces few challenges specific to the security aspect of microservice architectures. The polyglot stack functionality of microservices induces the risks of higher number of security culpabilities. The highly dynamic nature of the microservices also raises issues in discoverability. Various microservice instances may be incorporated on-the-fly while existing ones may be discontinued. The short span of the development cycles of microservices also make the testing process cumbersome. Due to extensive communication between the various instances, several points of entry and exit contribute in increasing the attack surface. Besides security concerns after the deployment of microservices, security measures must be adopted while developing the applications as well.

2.1.2.2 Service Management Capabilities

Client or user requests to microservices are directed to the appropriate microservice instance. The interactions between the client and the microservice instances are governed by the service management activities, such as service discovery, routing, load balancing and configuration management. Research efforts towards these service management activities are discussed in this section.

1. **Service Discovery:** Microservice Architectures decompose applications into multiple constituent microservices, which communicate with each other using lightweight mechanisms. At present, microservices do not conform to a standard definition. However, in general, certain features are considered to be possessed by most microservices. The feature with substantially high importance is the discoverability feature, which implies that the different microservices must be able to locate the other microservice instances currently running in the system. Service discovery is the process which enables the clients or the consumers to identify the location of the providers of the services they require and thereby enables communication between these two entities. Microservice architectures require service discovery mechanisms to provide the IP address and port number to correctly locate other microservice instances (Khan 2017). Service discovery solutions may be centralized or de-centralized. MSA call for de-centralized solutions that can be modified dynamically to reflect the correct location information of each service provided by the system, which changes frequently during runtime. Microservices-based systems are highly dynamic- microservices are spawned and de-registered on the fly. During the lifetime of the microservice instance, it may be migrated to other hosts. A service discovery solution for microservices must be capable of handling all these scenarios. The service discovery solution must be scalable (Tellago 2016), as it should be scaled in or out, corresponding to the changes in the microservices deployed. Stubbs et al. (2015) proposed a de-centralised solution for the service discovery problem in microservices, using Serfnode. Serfnode is an open-source, lightweight Docker image that comprises additional capabilities exhibited by the Serf agent. Long et al. (2017) proposed

a service discovery mechanism that adopts the concepts of information-centric networks. Similar to the named data networks, each microservice is assigned a hierarchical name which is updated in the registry using routing protocols that are name based. When a request arrives, the service that offers the longest prefix match, is invoked.

Consul and etcd are cluster-based solutions (Rotter et al. 2017) that uses the Raft consensus algorithm to ensure consistent records. The discoverability of stateful microservices must also be considered. Policies are required to allow the correct instance of a stateful microservice to be invoked. All the session data is maintained in a common persistent storage, which also maintains the details of the entity owning each session. Gadea et al. (2016) proposed a solution which couples service discovery information, changes in microservice descriptions and other updates related to the deployed microservices. All the information is stored in a NoSQL database, to which the clients can subscribe to be notified when any updates are made in the database. Messina et al. (2016) proposed to use separate database servers as a cluster to provide database-as-a-service. Requests from the clients are received by a load balancer, which are then directed to the appropriate database server.

2. **Routing:** Applications designed using microservice architectures encompass several microservice units. Different microservices provide different functionalities. Different instances of microservices providing the same functionalities are also available in the system. Upon receiving a client service request, routing capabilities are required to determine the microservice instance to which the request may be directed. The client requests are directed to the API Gateway, the only component of the microservices systems that is publicly exposed. A routing layer handles the responsibility of forwarding requests from the API Gateway to the appropriate microservice instance (Bakshi 2017; Naily et al. 2017; Versteden and Pauwels 2016). The routing layer is usually provided as part of the infrastructure layer. Researchers have attempted to develop solutions for routing in microservice applications. One unique challenge faced by researchers is that

there may be different versions of the same microservice. For usecases like performance or resilience testing, the request may be targetted at a specific version. In this context, there is a need for routing mechanisms that are version-aware. A group of researchers from IBM developed Amalgam8 [1] that performs routing by considering the version of the microservice to be accessed. The routing solution is inspired by the Software Defined Networks (SDN) concepts (Rajagopalan and Jamjoom 2015; Rajagopalan et al. 2016).

3. **Load Balancing:** Microservice ecosystems contain several microservice units interacting with each other. One of the major attractions of MSA is the ability for different microservice units to be scaled depending on the load. A load balancing component may be introduced to effectively distribute the incoming requests among the different instances of the same microservice. The load balancing task may be performed either at the server-side or at the client-side, giving rise to the two types of load balancing. In the server-side load balancing, all the requests are directed to a proxy address which hosts the load balancer and also maintains a directory of all the microservice instances in the system. The server then applies the load balancing algorithm to determine which instance can handle the current request. The client-side load balancing entrusts more power in the client. Each client first directs the requests to the service discovery agent which returns all the potential microservice instances to handle the request, from the service registry. On receiving the information of all the running instances, the client then applies load balancing algorithms such as the Round-Robin algorithm, to determine where their current request must be directed to. Alternatively, a local cache may be housed at the client side, to avoid the communication with the service registry for each request. The client-side mechanism is considered to be more suitable for the decentralized microservices systems. The Netflix Ribbon is an example of a client-side load balancer [2].

4. **Configuration Management:** Dynamic environments of the microservices re-

[1] `www.amalgam8.io/`
[2] `github.com/Netflix/ribbon`

quire tools for configuration management, which ensure that all the dynamic entities in the system follow the same configuration settings. Configuration management ensures that all the containers running microservices adhere to the requirement specifications. Even with container technologies like Docker and Rkt, where containers are spawned according to the specified configuration, configuration management tools are still required to create the base image and also to manage configuration settings of the containers that are running. Tools such as Chef and Ansible [3] can be used in collaboration with the container technologies, to ensure that all the running containers heed to any modifications to the configuration made at the server node and thus help in the automation of the lifecycle of microservices (Familiar 2015).

2.2 RESEARCH GAPS

The extensive study conducted brought to light that the domain of MSA presents much scope for further research. In this context, the domain offers several research challenges that demand solutions. Few of these challenges that directed the research presented in this thesis are as follows:

- In the case of developmental concerns of MSA, a considerably large body of literature exists. However, only a limited number of studies focus on the challenges posed in the operational phase of microservice systems. The operational phase presents numerous challenges such as energy profiling, workload modelling, resource demand profiling and estimation.

[3] https://www.chef.io/ansible/

Research Work	Management Task	Managed Unit	Solution approach	Parameters considered
Guerrero et al. (2018a)	Container Allocation	Microservices in containers	NSGA-II	Latency, network usage and resource usage
Filip et al. (2018)	Container Scheduling	Microservices in containers	Round Robin, First Come First Serve (FCFS), Heuristic based	Bandwidth, latency between physical nodes
Buyya et al. (2018)	Container orchestration	Containers	Architectural Framework	Pricing model of resources and QoS
Wan et al. (2018)	Application Deployment	Microservices in containers	Execution Container Placement and Task Assignment algorithm	Deployment, execution and communication cost
Tao et al. (2017)	Container deployment	Containers	Fuzzy Inference System	Resource usage

Table 2.1: Summary of research works on initial placement

Research Work	Strategy	Optimization Objectives	Managed Unit	Solution Approach	Evaluation Platform
(Wang et al. 2019b)	VM Re-scheduling	Improved Resource utilization	VM	Host switching behaviour in symbiotic environments	Simulation experiments
(Sharma et al. 2019)	VM Re-scheduling	Reduced Failure occurrence rate	VM	Failure prediction based on Exponential Smoothing	Simulation experiments
(Mahdhi and Mezni 2018)	VM Re-scheduling	Energy efficiency	VM	Kernel Density Estimation (KDE) technique	Simulation experiments
(Witanto et al. 2018)	VM Re-scheduling	Cloud provider goals-specific	VM	Deep Neural Network to select among multiple algorithms	Simulation experiments
(Wang et al. 2019a)	VM Re-scheduling	Energy efficiency	VM	Temperature model based	Simulation experiments

Table 2.2: Summary of research works on re-scheduling

- A significant concern in the operational phase is the initial placement or allocation of the microservice workload. Allocation mechanisms in the container orchestration engines are based only on the resource capacity and do not consider the nature of the workload. There is a need for allocation strategies that are further optimized in terms of the various QoS parameters in containerized systems.

- Microservice applications are composed of multiple microservice entities that frequently interact with each other. The pattern of microservice interaction is equally important as the interacting entities. There is a lack of research on allocation approaches that determine the placement strategy of microservice components based on the frequency of interactions and the characteristics of the interacting entities (as inferred from Table 2.1).

- A multitude of research has been directed towards re-scheduling in hypervisor-based virtualized datacenters. On the contrary, there is a dearth of research that targets the re-scheduling in containerized datacenters (as inferred from Table 2.2), which is another crucial issue in the operational phase.

- Unlike virtual machine re-scheduling approaches, the approaches for re-scheduling in containerized datacenters must consider the container configuration parameters. There is a need to investigate the impact of re-scheduling activities on the performance of microservice workloads.

2.3 SUMMARY

This chapter mainly focusses on the research works dealing with the operational phase concerns of microservices. The study presented in this chapter briefly discusses research works on developmental phase concerns of microservices, without delving into much details. The study also categorises works on the operational phase of microservices. However, the design concerns of microservice architectures are not considered in this study. The domain of microservice architectures holds much scope for future research especially in the resource management and orchestration of microservice-based systems. Considering the features particularly exhibited by microservices, there is a

need for research on tasks such as workload modelling, resource profiling, energy-efficiency and securing the microservice based systems.

The study presented in this chapter furnishes a body of knowledge to the domain of Microservice Architectures by i) discussing a taxonomy to classify the various works in the literature, ii) classifying and reviewing the existing research contributions on the various aspects of Microservice Architectures and iii) pointing out the open challenges in the domain.

CHAPTER 3

PROBLEM DESCRIPTION

Adopting microservices in Cloud environments results in an ecosystem of microservices interacting with each other and their environment. Microservices make application delivery more simple and efficient. The microservice components that are part of an application follow a simplified structure. However, this reduction in complexity of software design and architecture is at the cost of thrusting more responsibilities on to the underlying infrastructure layer (Abeysinghe 2016). Managing the operational complexity of such a distributed system is a challenge, which must be addressed through orchestration mechanisms.

3.1 SCOPE AND FOCUS OF THE THESIS

In this thesis, the resource orchestration challenges for Cloud environments have been investigated. The objective of the thesis is to enhance the performance of Cloud environments by addressing the resource management concerns for an ecosystem of containerized microservices. The focus is on Cloud environments where containers are used to deploy microservice workload applications as illustrated in Table 3.1.

Facet	**Thesis Scope**
Virtualization level	Operating system level virtualization - Containers
System resources	Processing and memory resources
Workload application	Microservice workload applications
Orchestration techniques	Microservice allocation and microservice re-scheduling

Table 3.1: Scope and focus of this thesis

Cloud applications deployed as microservices must meet specific performance requirements. In order to facilitate effective utilization of the Cloud resources by the microservice applications, the Cloud platform must be augmented with microservice orchestration capabilities. A platform deploying microservices must possess different orchestrational capabilities to support the deployed microservices. These capabilities include all the activities to support the microservice applications during the operational phase. The activities in such a platform can be broadly classified into two: the *service management capabilities*, which involve the operations interfacing the client and the microservices, and the *infrastructural management capabilities*, which involve the operations interfacing the microservices and the underlying hardware. A framework that deals with the operational phase challenges of MSA has been diagrammatically represented in Figure 3.1.

From the customers' perspective, the flow of activities in the framework starts with users submitting requests to microservices. Users can take the form of any digital computing device such as mobile phones, laptops, Personal Computers, etc. The requests are decussated at the microservices platform. On the arrival of user requests, the service management operations are carried out to perform tasks such as identifying the microservice that can handle the request (service discovery and load balancing) and directing the request to the appropriate microservice instance (routing). A registry of services contains the information of all the different microservice modules in the platform. Service discovery is performed using the information contained in the service registry. For uniformly distributing load across the running microservice instances, load balancing mechanisms are applied. In Figure 3.1, the first component (first white box in Figure 3.1) enlists the service management capabilities.

The user requests are serviced by the different microservice module types 'Microservice A' to 'Microservice n'. Each microservice module type has different instances μ_{xy}, where x represents the microservice module type of which it is an instance (ranging from A to n) and y represents the index of the specific instance. The different microservice instances can communicate with each other using messaging interfaces (generally REST-based). On receiving the requests, the microservices process the re-

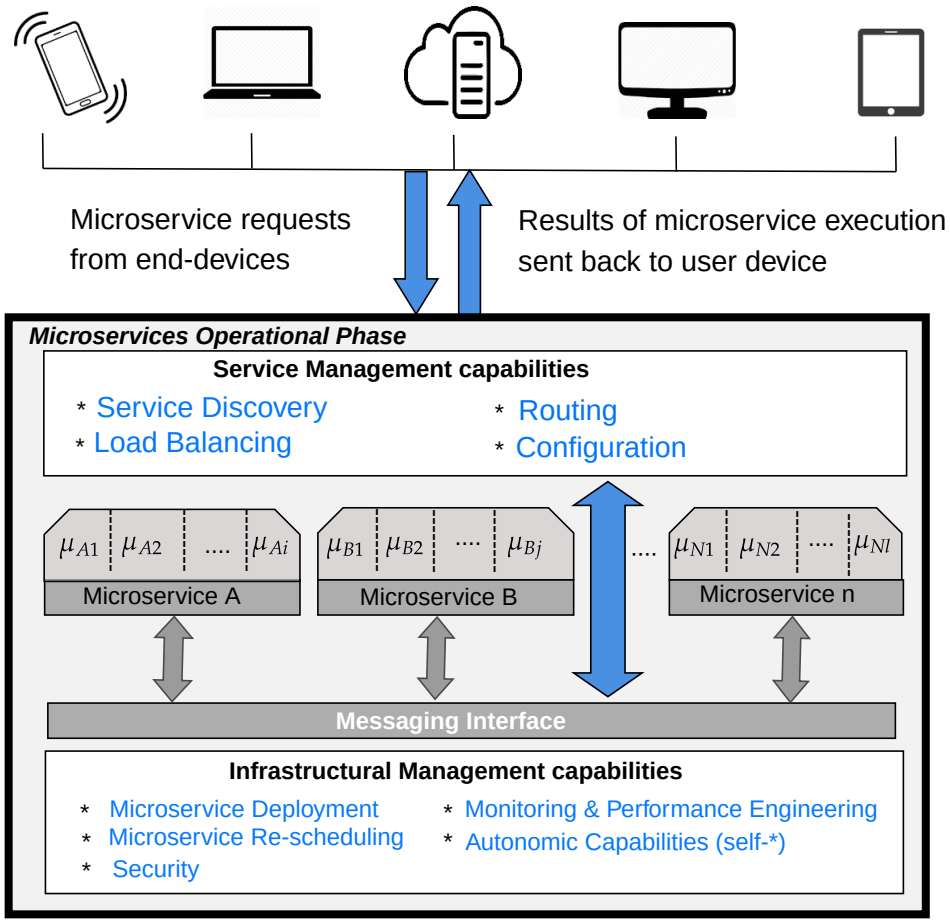


Figure 3.1: A framework for the deployment and execution of microservices

quests. During the processing, other microservices may be invoked in a cascaded manner. The results of processing are directed back to the end users leveraging the service management capabilities.

Infrastructural management capabilities ensure that the required microservices are up and available to receive and process the user requests. The infrastructural management operations involve actions performed over the entire lifecycle of the microservice, starting from deployment. Identifying the appropriate node to host the microservice (allocation), observing the runtime metrics of the microservice (monitoring) and securing the different processes during the microservice execution are all examples of operations carried out as part of the infrastructural management. The underlying hardware

is thus managed by leveraging the infrastructural management capabilities (depicted by the second white box in Figure 3.1).

3.2 RESEARCH PROBLEM AND OBJECTIVES

The aim of the research work presented in this thesis is to address the challenges pertaining to infrastructural management capabilities for microservice deployment platforms. In particular, this work explores the research problems of microservice deployment and microservice re-scheduling in containerized Cloud environments.

In order to address these issues, the research problem investigated in this thesis is as follows:

To design and develop algorithms, for the efficient allocation and re-scheduling of microservice components, which meet the microservice performance objectives in containerized Cloud environments.

The first goal of this work is to devise a strategy to schedule microservice containers in the Cloud datacenters. This process consists of assigning the microservice workload containers to suitable hosts or nodes in the datacenter. The mapping is to be done in such a manner that the various Quality of Service (QoS) parameters are met. In general, for service-based distributed systems, these QoS parameters are defined as performance metrics encompassing both functional requirements such as response time and non-functional requirements like energy efficiency and security. The microservice scheduling problem is a variant of the bin packing problem, which is NP-hard (Beloglazov and Buyya 2012) and therefore an optimal solution cannot be obtained in polynomial time. In this thesis, the terms scheduling, deploying, placing and allocating have been used interchangeably and refers to the activity of mapping each microservice container onto the most aptly suited resource.

The second goal of this work is to develop an approach for microservice re-scheduling in Cloud datacenters. The highly dynamic nature of the workloads arriving at the Cloud datacenters leads to unprecedented variations in the resources offered by the Cloud providers. As a consequence, Cloud providers must leverage techniques to effectively reallocate or reschedule microservice containers in order to adapt to the fluctuations

in the resource usage. This is essential to eliminate unwarranted degradations in the performance of the hosted microservice applications. The process of microservice re-scheduling can be considered as a multi-stage problem that involves the activities of identifying the conditions under which the re-scheduling process is to be triggered, identifying the candidate nodes or hosts to be subjected to re-scheduling and determining the apt alterations in the assignment of the microservice containers to the datacenter resources.

To address the aforementioned goals, the research objectives identified are as follows:

1. To propose a model for the deployment of microservice components considering QoS parameters.

2. To design and develop a resource-aware allocation algorithm, to place the microservice components, maintaining the interdependencies among the components.

3. To design a dynamic microservice re-scheduling algorithm which meets the microservice performance objectives.

4. To implement and evaluate the effectiveness of the microservice re-scheduling algorithm.

3.3 RESEARCH CHALLENGES

Resource orchestration in distributed systems involves numerous inherent challenges such as resource and workload heterogeneities and support for QoS constraints (Hilman et al. 2020; Krauter et al. 2002). Strategies devised for resource orchestration must carefully consider these particularities. The orchestration strategies must have a general approach to handle the heterogeneity of resources in the Cloud environment (Bittencourt et al. 2018; Gonzalez et al. 2017). Furthermore, a key characteristic of distributed systems is resource co-allocation. Accordingly, resource attributes must be considered as a defining characteristic of the Cloud datacenter nodes while determining

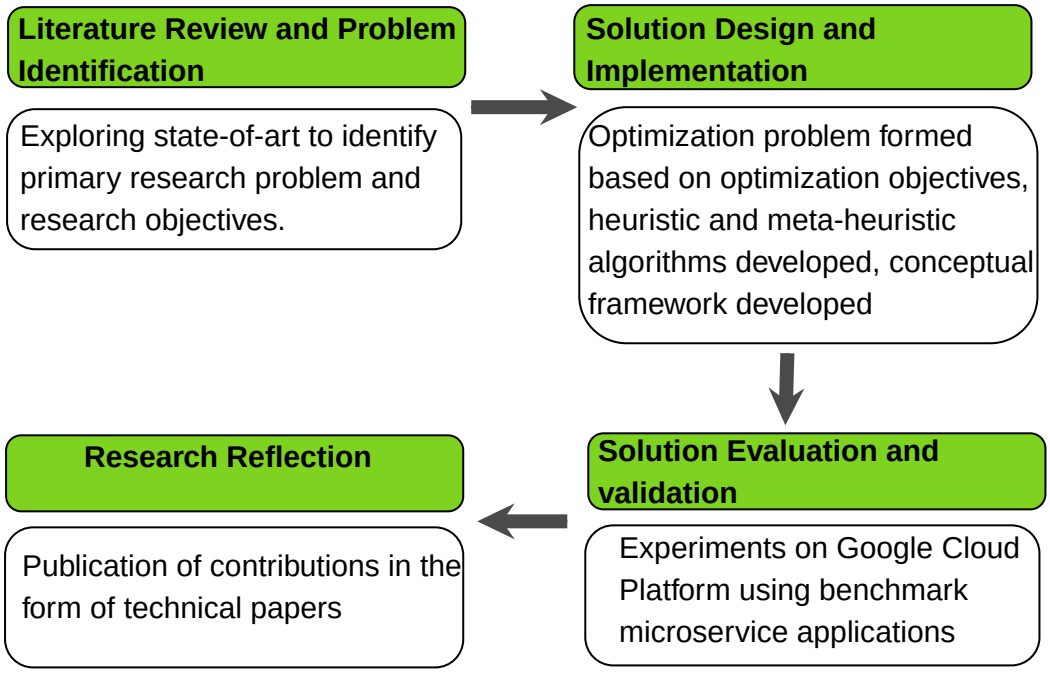


Figure 3.2: Overview of the research methodology followed in this thesis

the strategy. A wide spectrum of applications that vary in resource and performance requirements can be deployed in the Cloud distributed environment. The suitability of the orchestration strategies for the heterogeneous workload must be thoroughly investigated by considering diverse microservice applications in the validation process. In order to handle the performance variability in Cloud environments, re-allocation strategies that consider the QoS attributed must be incorporated.

3.4 RESEARCH METHODOLOGY

Research in the field of information systems is determined by two paradigms, namely, behavioral science and design-science (Hevner et al. 2004).

Considering the synthetic nature of the software engineering domain, the research presented in this thesis is in alignment with the field of study in the design-science paradigm, as shown in Figure 3.2. The artifacts resulting from this work are portrayed as a set of models, frameworks and algorithms in the application domain of Cloud computing. The research activities were conducted in different phases:

1. **Literature Review and Problem Identification:** In this phase, an exhaustive

study of the research in the domain of Microservice Architecture, was conducted. Based on this study, the research gaps in the field of study were identified. Following this step, the primary research problem and the research objectives were formulated.

2. **Solution Design and Implementation:** In this phase, potential solutions were devised and developed to accomplish the research objectives identified in the previous step. The outcomes of this step include an optimization model, algorithms and a framework.

3. **Solution Evaluation and validation:** In this phase, the different solution contributions were evaluated and validated. The microservice allocation and re-scheduling approaches were evaluated by conducting experiments in real Cloud environments using microservice benchmark applications. Extensive experiments were conducted on the Google Cloud Platform using reference microservice workload applications with different characteristics. This phase aimed at establishing the performance enhancements attained by the proposed contributions.

4. **Research Reflection:** Research reflection includes activities to illustrate the impact of the research outcomes. In this regard, the outcomes of this research have been communicated to the scientific community via the channels of technical conference proceedings and journal publications.

3.5 RESEARCH CONTRIBUTIONS

The contributions of this thesis can be broadly classified into three, namely systematization and analysis of prior research, proposition of an efficient dynamic microservice allocation strategy and presentation of a dynamic microservice re-scheduling technique. An outline of the contributions is provided in Figure 3.3. The core contributions are as follows:

1. A comprehensive review of the literature on Microservice Architectures.

    - A multi-level taxonomy to classify existing research works on Microservice Architectures, based on the aspect considered.

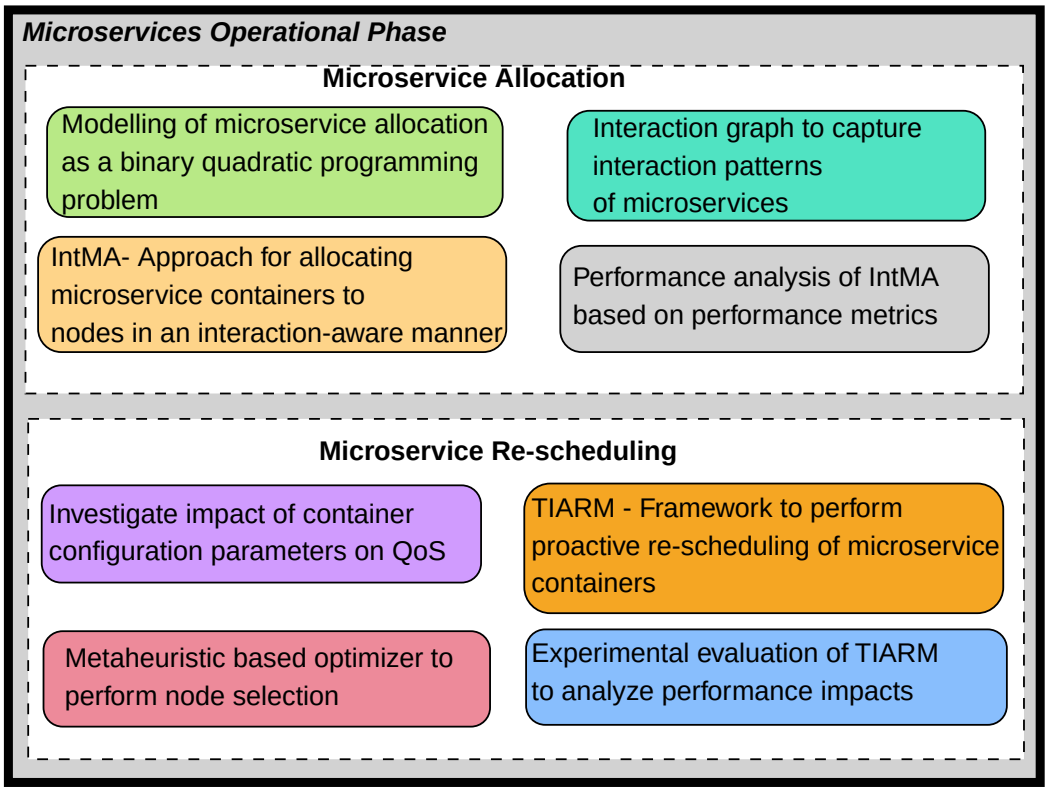


Figure 3.3: Outline of the contributions of this thesis

- A survey and analysis of the various works that discuss different aspects of Microservice Architectures.

2. A dynamic interaction-aware allocation strategy for containerized microservices.

   - Formulation of the microservice allocation problem as a $(0/1)$-Quadratic programming Problem (QPP), which includes the constraints on computation and memory resource requirements.
   - A doubly weighted complete graph, called the *Interaction Graph* to capture the interaction patterns between microservices, which includes the frequency of interactions and the characteristics of the interacting components. A framework for recording the information required to generate the Interaction Graph.
   - A novel and efficient approach, **IntMA**-Interaction-aware Microservice Allocation, is proposed to allocate the microservices to different nodes in the

Cloud, in an interaction-aware manner.

- An analysis of the performance of the proposed approach by collecting the microservice-specific metrics such as response time, throughput and a comparison with a well-known heuristic based algorithm, IntRR-Interaction-aware Round Robin, and the Kubernetes default scheduling policies.

3. An approach for rescheduling in container-based Clouds running microservice applications.

    - An examination of the impact of the container configuration parameters on the observed QoS values to establish the relationship between the different variables.
    - A framework, **TIARM**, that enables Cloud providers to control and coordinate adaptations in the runtime deployment of microservice containers. The framework proactively identifies scenarios in which re-scheduling is to be done and also determines the targets for the re-scheduling activities.
    - Design and implementation of a novel metaheuristic-based optimizer to find the best set of nodes suited for a set of migrating containers, taking into account the migration overheads incurred. The optimizer also ensures that the CPU resource demands complement each other, to reduce the resource contention.

Research Contribution	Specifics of Research Contribution	Research Objective (RO) addressed	Thesis Chapter presenting Contribution	Chapter derived from Publication
Comprehensive review on MSA	Multilevel Taxonomy		Chapter 2	Joseph, C. T. & Chandrasekaran, K. (2019). Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. Wiley Software Practice and Experience, 49(10), 1448-1484.
	Survey and analysis of existing works			
Dynamic interaction-aware allocation strategy	Formulation of allocation problem as QPP	RO 1	Chapter 4	Joseph, C. T. & Chandrasekaran, K. (2020). IntMA: Dynamic Interaction-Aware Resource Allocation for Containerized Microservices. Elsevier Journal of Systems Architecture.
	Modeling interaction patterns as a doubly weighted complete graph	RO 1		
	IntMA allocation approach	RO 2		
	Performance analysis of IntMA	RO 2		
Rescheduling Approach	Examination of impact of the container configuration parameters on the observed QoS values	RO 3	Chapter 5	Joseph, C. T. & Chandrasekaran, K. Nature-Inspired Resource Management and Dynamic Rescheduling of Microservices in Cloud Datacenters, *Wiley Concurrency and computation: practice and experience*(“Under Review”)
	TIARM framework to perform rescheduling	RO 4		
	Metaheuristic-based optimizer to find the best destination nodes suited for a set of migrating containers	RO 4		
	Performance analysis of TIARM	RO 4		

Table 3.2: Details of Contributions of this Thesis

- Experimental demonstration of the effectiveness of the proposed framework through extensive experiments using well-known microservice benchmark applications in real-time Cloud environments and reveal the superior performance of the proposed re-scheduling approach in comparison with other baseline strategies.

The work presented in this thesis addresses the objectives listed in Section 3.2. The contributions drawn from the thesis corresponding to the different objectives are provided in Table 3.2. Due to the logical dependencies between the objectives, the research work directed towards the first and second objectives are jointly presented in Chapter 4. Similarly, the research work directed towards the third and fourth objectives are jointly presented in Chapter 5.

CHAPTER 4

IntMA: DYNAMIC INTERACTION-AWARE RESOURCE ALLOCATION FOR CONTAINERIZED MICROSERVICES IN CLOUD ENVIRONMENTS

The allocation strategy adopted for deploying/placing the microservice application containers to the different nodes in a Cloud data center determines the efficiency of the microservice-based systems. An inefficient allocation strategy may lead to increased response times and low throughput which affects the user experience. Resource management techniques that consider the resource requirements and other key elements such as communication among the microservice components, are essential to ensure seamless running of microservice applications. Determining the right amount of resources to be allocated to an incoming microservice application is an intricate task which involves many challenges (Fazio et al. 2016).

In view of the recent adoption of container virtualization to run applications in the Cloud (Goldschmidt et al. 2018), researchers have addressed resource management for containers. Fazio et al. (2016) discussed the various challenges in adopting microservices in the Cloud. Though the application characteristics suit the Cloud environment, there are several operational challenges that needs to be tackled. Guerrero et al. (2018a) proposed a Genetic Algorithm based solution for allocation of containers running microservices in the Cloud environments. The objectives of cluster reliability and load balancing were considered in the deployment strategy based on the Ant Colony metaheuristic proposed by Lin et al. (2019). Netaji and Bhole (2019) developed a hybrid

container allocation approach that combines the Whale and Lion metaheuristic algorithms. Another work by Guerrero et al. (2018b) proposes the use of Non Dominated Sorting Genetic Algorithm (NSGA)-II for the container allocation problem in multi-cloud environments. Filip et al. (2018) considered microservice placement across heterogeneous environments. Wan et al. (2018) considered the minimization of the total cost involved with application deployment in container based systems. Zheng et al. (2019) devised an autoscaling approach that differentiates workloads based on their characteristics. Adhikari and Srirama (2019) adopted the accelerated Particle Swarm Optimization technique to place containers in IoT environments. Pallewatta et al. (2019) considered microservice placement in Fog environments. A recent study presented by Rodriguez and Buyya (2019) highlighted the challenges involved in the resource management for container-based systems, which form the crux in container orchestration systems. Buyya et al. (2018) proposed a framework for the cost-efficient orchestration of containers in the Cloud environment. They also emphasised the need for more research works to be carried out in the container-based Cloud environments, specifically on the initial placement of containers and to optimize the placement at run time through migration, rescheduling or autoscaling of the clusters. Tao et al. (2017) proposed an algorithm for the container resource allocation based on Fuzzy Inference System (FIS). Kaewkasi and Chuenmuneewong (2017) developed an Ant Colony Optimization (ACO) based scheduling algorithm for the allocation of containers. Kang et al. (2016a) proposed an energy efficient brokering system for the containers. The aforementioned research works do not consider the interaction patterns among the microservices to be allocated. The frequency of interactions and the interacting entities are significant factors in determining the values of microservice-specific response times.

Few researchers considered interactions among microservices. Wen et al. (2019) developed GA-Par that adopts Genetic Algorithm to perform microservice composition that fulfills the security requirements of the user and deploy the resulting microservice workflow while ensuring system dependabillity. Štefanič et al. (2019) considered time critical microservice applications in the Software Workbench for Interactive, Time Critical and Highly self-adaptive Cloud applications (SWITCH) workbench that captures

the application logic details using Topology Orchestration Specification for Cloud Applications (TOSCA). However, GA-Par and SWITCH workbench considers mainly the logical invocations and does not consider the service runtime chain. Thus, there is a need for placement schemes that consider the runtime dependencies among microservice components.

The primal focus of this chapter is to address container orchestration for microservice allocation in Cloud with an objective of minimizing the communication among the physical nodes deploying the application. It is assumed that each microservice component is hosted in an independent container which is to be mapped to a suitable physical node. Several mission-critical systems designed using microservice architectures have stringent constraints on response time. So, the objective of minimizing the response time is considered, by ensuring that interacting microservice components are deployed on the same node.

This chapter presents two heuristic based interaction-aware microservice allocation algorithms, Interaction-aware Microservice Allocation (IntMA) and Interaction-aware Round Robin (IntRR). The remainder of this chapter is organized as follows: Section 4.1 presents a use case that motivates the proposed approach, Section 4.2 provides the formal description of the system, Section 4.3 details the proposed methodology and Section 4.4 discusses the experimental configurations that were adopted to validate the proposed approach. Sections 4.5 and 4.6 provide a detailed analysis of the results.

4.1 MOTIVATION

On the basis of the review of the existing literature, it is observed that the research works on deployment of microservices give equal weight to any interaction between microservice components. The pattern of microservice interaction is equally important as the interacting entities. There is a lack of research that considers the frequency of interactions and the characteristics of the interacting entities.

The intuition behind the work discussed in this chapter is that the communication latency between the microservice components forming the application is a key factor which affects the response time experienced by the user (Zhang et al. 2018). This

implies that in order to reduce the response time, it is crucial to reduce the microservice communication latency. This latency depends on various parameters such as the interconnecting interface, the network characteristics and most importantly the network distance between the communicating entities. In this work, the focus is on reducing the network distance between the entities that frequently interact with each other. The term 'interaction' is used to denote the invocation of one microservice component by another. To portray the need for such a system, a use case scenario is described.

Consider a microservice application with 3 components. The i^{th} component is represented as M_i. Let M_1 be dependent with M_2 and M_3. M_2 is dependent with M_1 and M_3, M_3 is dependent with M_1 and M_2. The existing approaches for microservice application deployment consider all the dependencies to be of equal weights. Suppose that M_1 exhibits the highest dependency with M_3. Employing the existing approaches would place M_1 and M_2 on the same node before considering M_3. However, the number of times that M_3 invokes M_1 is much larger when compared to the number of times that M_2 and M_1 invokes each other. In such cases, more time is spent in the communication between the modules M_1 and M_3. If these modules are on different hosts, the communication latency between them increases which leads to an increase in the overall response time.

Therefore, it is of utmost importance that the frequency of the interactions and the characteristics of the interacting microservices, be considered while performing the application deployment. Interactions that are more frequent are considered to be more critical in deriving the application deployment strategy.

4.2 FORMAL DESCRIPTION OF THE SYSTEM MODEL

In this section, a formal model for the aspects of container-based Cloud data centers used in the proposed microservice allocation approach is presented.

Consider a Cloud datacenter with N physical hosts/nodes, $H = \{h_1, h_2, ..., h_N\}$. Let h_z denote an individual host/node. Each host/node is characterised by the tuple $< cpu, mem >$ denoting the number of processing cores and memory units on the physical host/node respectively. Every host/node is capable of running application con-

tainers (hosted on bare-metal or on virtual machines).

Applications comprise of different microservice components (referred to as microservices or modules in the subsequent sections). The microservice components are run on containers hosted on the physical machines/nodes. Microservice components belonging to the same application may interact among themselves for servicing the requests submitted by users. The microservices belonging to different applications thus form an ecosystem of interacting microservices. Let d_{ij} denote the number of interactions between the microservice components v_i and v_j, respectively. Each microservice component is characterised by a 2-tuple, $< cpu\_req, mem\_req >$ corresponding to the number of processing cores and the amount of memory requested by the microservice component. Every host in the datacenter will have some resources currently in use by other processes. For the allocation process, only the remaining resource or residual resource capacity must be considered. Let F denote the set of feasible nodes with sufficient residual resource capacities. Each feasible node is denoted as f_k. The residual CPU and memory capacities (denoted as $h_z^{cpu\_res}$ and $h_z^{mem\_res}$, respectively) on each node can be computed from the current CPU and memory utilization on the node ($h_z^{cpu\_util}$ and $h_z^{mem\_util}$), using Equation 4.1.

$$h_z^{u\_res} = h_z^u(1 - h_z^{u\_util}); u \in \{cpu, mem\} \tag{4.1}$$

Each microservice is allocated to the available host which satisfies the resource requirements of the microservice. The model can be extended to include other resources such as storage, bandwidth, etc. Table 4.1 provides the different elements in the system with their representations. The microservice allocation policy determines the host on which the microservice must be deployed. The allocation problem is first modelled as a quadratic programming problem and then an interaction-aware allocation policy, IntMA, to determine the mapping of microservice requests to the feasible nodes, is proposed in the subsequent sections.

Notation	Element	Description
H	Physical Host/ Node	Set of all physical hosts/nodes in the system
N		# physical hosts/nodes in the system, i.e. $\|H\|$
h_z		z^{th} physical host/nodes, $\forall_{z=1}^{N} h_z \in H$
h_z^{cpu}		# of processing cores on z^{th} physical host h_z (expressed in millicpu, mCPU)
h_z^{mem}		Amount of memory on z^{th} physical host/node h_z (expressed in bytes)
$h_z^{cpu\_util}$		CPU utilization of z^{th} physical host/node h_z, in current time interval (expressed in %)
$h_z^{mem\_util}$		Utilization of memory z^{th} physical host/node, h_z in current time interval (expressed in %)
$h_z^{cpu\_res}$		Residual processing capacity on z^{th} physical host/node h_z
$h_z^{mem\_res}$		Residual amount of memory on z^{th} physical host/node h_z
F		Feasible nodes; Set of all physical nodes with $h_z^{u\_res} > min(v_i^{u\_req})$ where $u \in \{cpu, mem\}$
Q		# feasible nodes in the system, i.e. $\|F\|$
f_k		k^{th} feasible node, $\forall_{k=1}^{Q} f_k \in F$
V	Microservice Component	Set of microservice components in the application
M		# microservice components in the application, i.e. $\|V\|$
v_i		i^{th} microservice component, $\forall_{i=1}^{M} v_i \in V$
$v_i^{cpu\_req}$		# of processing cores requested by i^{th} microservice component v_i (expressed in millicpu, mCPU)
$v_i^{mem\_req}$		Amount of memory requested by i^{th} microservice component v_i (expressed in bytes
D	Interaction	Set of interactions among the microservice components, derived from edge weights in interaction graph
d_{ij}		Interaction between microservice components v_i and v_j, where $d_{ij} \in D$ and $v_i \in V, v_j \in V$

Table 4.1: Notations used in the system model

4.3 PROPOSED METHODOLOGY

Application developers design applications composed of different microservices. The different microservices and their instances are submitted to the Cloud datacenter. Cloud providers that offer support for microservices-based applications extend different functionalities that can be broadly classified into two:

- Service Management Capabilities: These include activities at the interface between users and the microservices. These include routing, service discovery, etc.
- Infrastructure Management Capabilities: These include activities that manage the mapping between microservices and the underlying infrastructure.

In an ecosystem of microservices, the first step is that the application developer submits the different microservices that form an application. Once the microservices arrive, the Cloud provider considers the requirements of each microservice component to derive an optimal placement or deployment strategy for the application. This task falls under the infrastructure management capability. The various microservices are assigned to suitable physical nodes in the datacenter, that can satisfy the resource requirements and other objectives of the microservices. A group of related microservices running on the physical node form a pod, which is the basic unit of management. Once the deployment operation is completed, the microservice containers are started. After the microservice containers are in the 'running' state, user requests are accepted. The user requests for microservices are received by the Cloud platform. The service management capabilities identify the microservice instance and the node hosting the microservice instance, to which the requests are forwarded. After the request has been processed, the service management capabilities ensure that the results are delivered back to the user who initiated the request.

4.3.1 Proposed Framework

The microservices lifecycle can be divided into two phases: the developmental phase and the operational phase. The operational phase involves the service management capabilities and the infrastructure management capabilities. This work deals with mi-

croservice deployment, which falls under the infrastructure management capabilities.

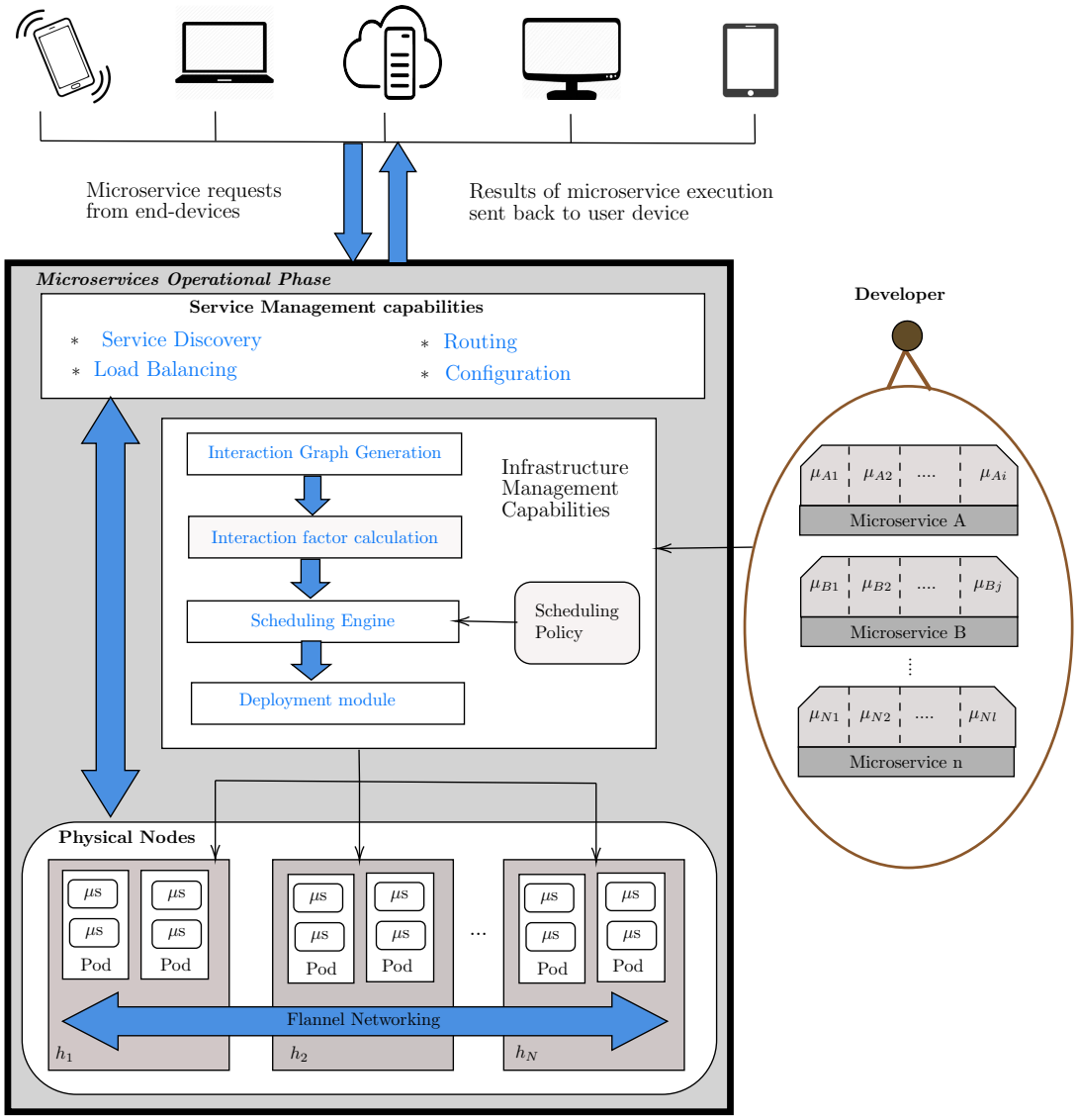


Figure 4.1: Proposed framework for microservice allocation

Figure 4.1 illustrates the proposed framework for microservice deployment. The developer submits different types of microservices, $Microservice\ A$, $Microservice\ B$, $\cdots$, $Microservice\ N$. Each microservice has several instances. For example, the $Microservice\ A$ has different instances $\mu_{A1}, \mu_{A2}, \cdots, \mu_{Ai}$. Each of these instances are considered as different modules while scheduling by the Cloud provider. The microservices scheduling is performed in various steps:

1. Interaction Graph Generation: The first step is to capture the interactions among

the various microservice components. This can be done by subjecting the microservices based application to a load test. The developer provides the application along with a load test script which can then be used to capture the interactions. This information is represented as an interaction graph.

2. Interaction factor calculation: The next step is to calculate the interaction factor which symbolizes the number of interactions among each microservice component. This is represented by the edge weights in the interaction graph. Based on the different equations defined in the subsequent sections, the interaction factor is computed for each pair of microservices.

3. Scheduling Engine: Once the interaction factors are obtained, the next step is to apply different scheduling policies to derive near-optimal scheduling strategies. The Scheduling Engine may employ a solver to obtain the solution for the Quadratic Programming problem representing the allocation problem. IntMA and IntRR are the other scheduling policies discussed in this chapter, which can also be plugged in to the scheduling engine.

4. Deployment module: Once the scheduling strategy is obtained, the deployment module takes care of initiating the assigned microservice components on the corresponding physical nodes. Information about the microservice components is also entered in a registry service, such as *etcd*, to ensure that the microservices are discoverable.

The steps are discussed in detail in the following subsections.

4.3.2 Interaction graph generation

The different components that form part of an application may interact with each other while processing the requests received from the users. In this context, the direction of communication between the microservice components, is immaterial, as communications in any direction will result in interaction between the nodes on which they are deployed. Thus, the interactions among the microservice components can be captured in the form of an undirected graph.

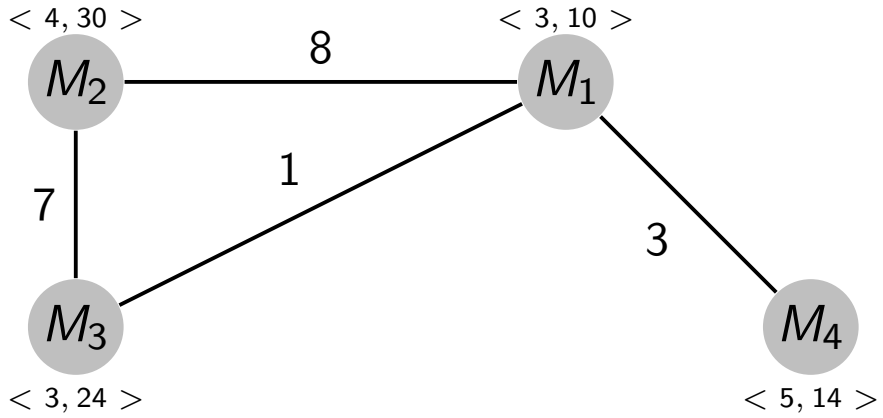


Figure 4.2: Interaction graph for a toy microservice application with 4 microservice components

The interaction graph G is an undirected complete doubly-weighted graph denoted by $< V, T, E, D >$ where the microservice components form the vertices, V and the interactions among the components form the edges, E. T represents the weights associated with the vertices in V. The weights in T are 2-tuples $< v^{cpu\_req}, v^{mem\_req} >$, representing the processing and memory requirements of the associated microservice at vertex v. All the vertices of the graph G are connected with edges. D denotes the set of edge weights. Each edge is associated with a weight $d_{ij} \in D$ denoting the number of interactions among the microservice components (which is the sum of interactions in both directions) at the associated vertices. The weight of each edge can be computed as defined in Equation 4.2.

$$d_{ij} = \begin{cases} t, & \text{where } t \text{ denotes the number of interactions, if microservice } i \text{ interacts} \\ & \text{with microservice } j \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

The interaction graph for a toy microservice application with 4 modules is illustrated in Figure 4.2.

4.3.3 Interaction factor

The aim is to allocate the microservice components to the physical nodes in such a manner that the microservices with frequent interactions are placed on the same node or next to each other as much as possible. This ensures that communication across different physical nodes is minimal thereby reducing the latencies involved in request processing. To quantify the interactions across different nodes, we define a parameter, $interaction\_factor$ in Equation 4.3.

$$interaction\_factor, c_{v_i h_z v_j h_w} = \begin{cases} \frac{1}{2} d_{v_i v_j}, & \text{if } h_z \neq h_w \\ 0, & \text{if } h_z = h_w \end{cases} \tag{4.3}$$

where v_i, v_j are microservice components and h_z, h_w are the physical nodes to which the microservices v_i and v_j are respectively allocated. For each pair of interacting hosts h_z and h_w, the cost of interaction is equally divided among them, by assigning the interaction factor value for each of the hosts as half of the edge weight. Thus, interaction factor value is dependent on the number of interactions, d_{ij} and also on the physical node h_z and h_w hosting these microservices. It must be noted that the interaction factor will be zero for all cases where h_z and h_w are the same, regardless of the edge weight value.

4.3.4 Optimization Model

The system comprising of hosts in the datacenter and microservice components to be allocated to these hosts can be formulated as a $(0/1)$ Quadratic Programming Problem (QPP) with linear constraints, which is a special case of the class of Mixed Integer Non-Linear Programming (MINLP) optimization problems (Caprara 2008). The objective of the problem is to minimize the total cost of interaction, subject to the resource capacity constraints. The values of the decision variables of the problem denote the node to which the module is allocated. The binary decision variable denoted as $x_{v_i h_z}$ is defined in Equation 4.4.

$$x_{v_i h_z} = \begin{cases} 1, & \text{if } v_i \text{ is allocated on node } h_z \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

In other words, if the value of $x_{32} = 1$, it implies that in the optimal allocation strategy, the 3^{rd} microservice component is placed on the 2^{nd} host.

$$Minimize \sum_{\substack{z=1,w=1 \\ z \neq w}}^{N} \sum_{\substack{i=1,j=1 \\ i \neq j}}^{M} c_{v_i h_z v_j h_w} * x_{v_i h_z} * x_{v_j h_w}$$

subject to,

$$\forall h_z \in H, \sum_{i=1}^{M} v_i^{cpu\_req} * x_{v_i h_z} \leq h_z^{cpu\_res} \tag{4.5}$$

$$\forall h_z \in H, \sum_{i=1}^{M} v_i^{mem\_req} * x_{v_i h_z} \leq h_z^{mem\_res} \tag{4.6}$$

$$\forall v_i \in V, \sum_{z=1}^{N} x_{v_i h_z} = 1 \tag{4.7}$$

The optimization problem aims to reduce the inter-node communications and thereby reduce the latency experienced by the users. Equations 4.5 and 4.6 provide the constraints on the processing and memory requirements. The sum of the resources requested by all the microservice components placed on the host must not exceed the total residual resource capacity. Equation 4.7 ensures that each microservice component is placed on one and only one host in the system.

The number of terms in the optimization objective, can be expressed in terms of the number of hosts, N and the number of microservice components, M as provided in Equation 4.8.

$$M * (M-1) * N * (N-1) \tag{4.8}$$

4.3.5 Model Example

A toy microservice application with 4 components is considered. The communications between the different microservice components has been recorded in Figure 4.2. In this section, a quadratic programming problem is formulated to represent the allocation of

the application components across a system with 2 nodes $H = \{h_1, h_2\}$. The resource capacities of the nodes are given as: $< 8, 50 >, < 10, 50 >$. The total number of terms in the objective function, according to Equation 4.8 is $4 * (4 - 1) * 2 * (2 - 1) = 24$ terms.

From the interaction graph, the values of d_{ij} are obtained as:
$d_{12} = 8, d_{14} = 3, d_{13} = 1, d_{23} = 7, d_{24} = 0, d_{34} = 0$

The objective function after eliminating the terms with coefficient 0, is given in Equation 4.9.

$$\begin{aligned} Minimize\ & 4x_{11}x_{22} + 0.5x_{11}x_{32} + 1.5x_{11}x_{42} + \\ & 4x_{21}x_{12} + 3.5x_{21}x_{32} + 0.5x_{31}x_{12} + 3.5x_{31}x_{22} + \\ & 1.5x_{41}x_{12} + 4x_{12}x_{21} + 0.5x_{12}x_{31} + 1.5x_{12}x_{41} + \\ & 4x_{22}x_{11} + 3.5x_{22}x_{31} + 0.5x_{32}x_{11} + \\ & 3.5x_{32}x_{21} + 1.5x_{42}x_{11} \end{aligned} \tag{4.9}$$

subject to

$$\begin{aligned} 3x_{11} + 4x_{21} + 3x_{31} + 5x_{41} &\leq 8 \\ 3x_{12} + 4x_{22} + 3x_{32} + 5x_{42} &\leq 10 \\ 10x_{11} + 30x_{21} + 24x_{31} + 14x_{41} &\leq 50 \\ 10x_{12} + 30x_{22} + 24x_{32} + 14x_{42} &\leq 50 \\ x_{11} + x_{12} &= 1 \\ x_{21} + x_{22} &= 1 \\ x_{31} + x_{32} &= 1 \\ x_{41} + x_{42} &= 1 \end{aligned}$$

The QPP may be solved by any MINLP solver. On solving the problem with AP-Monitor optimization suite (Hedengren et al. 2014), the solution obtained is furnished in Table 4.2. For small problem sizes, classical optimization solutions, provide accurate results. With increase in the problem size, the number of decision variables grow exponentially, making it infeasible to correctly derive and solve the QPP.

Parameter	Mathematical optimization
Objective value	11
x_{11}	0
x_{12}	1
x_{21}	0
x_{22}	1
x_{31}	1
x_{32}	0
x_{41}	1
x_{42}	0

Table 4.2: Solutions obtained from mathematical optimization problem

4.3.6 Proposed Algorithms

In large problem spaces, heuristic algorithms are considered to be more computationally feasible, than classical optimization techniques. This section discusses two heuristic algorithms that can aid in obtaining near optimal solutions for the interaction-aware allocation of microservice components.

First, a baseline approach that was developed to benchmark the performance of the proposed approach is described. The aim of this work is to take into account the interactions among the microservice components while deriving the placement strategy. This attribute will ensure that the inter-node communication is reduced. Algorithm 4.1 describes the Round Robin heuristic (Interference-aware Round Robin, **IntRR**) that considers the microservice interactions for deploying the modules. Both the lists of available nodes and the value of interaction factors are arranged in non-increasing order. Then, each pair of microservices is selected based on the interaction factor value and placed together on the next node traversed in a circular order. The process is repeated till a suitable node is identified for every microservice component.

Initially, all the nodes with residual CPU and memory resources are filtered (in F), which are then sorted based on the availability of the dominant resource on the nodes. The values of the interaction factor (in D) are also sorted in non-increasing order. The variable 'k' is used to keep track of the index of the current node under consideration in the set F. The pair of microservices corresponding to the edge with highest interaction factor is picked. Lines 7 - 9 check whether both the microservices connected by the

edge can be placed in the same node. The corresponding edge interaction factor is removed from D and k is updated to point to the next node in F. In cases where both microservice components cannot be placed together on the same node, but either one of the microservice components can be placed, the individual microservice component is placed on the current node. The values of k and D are also updated accordingly (Lines 11-13). In all other cases, where none of the microservice components can be accommodated, the current node is skipped and the value of k is updated to point to the next node in circular order (Line 15).

Algorithm 4.1: IntRR-Interaction-aware Round Robin algorithm

Input : Interaction Graph, $G =< V, E >$ with edge weights D and vertex weights T
Output: Mapping from V to F

1 Initialize F with all nodes having $h_z^{u\_res} > 0; u \in \{cpu, mem\}$
2 Sort F in non-increasing order of $f_k^{mem\_res}$ `/* for memory intensive applications */`
3 Initialize k as 1
4 Sort weights in D in non-increasing order
5 **while** *all microservices have not been placed* **do**
6 Select $\alpha = max(D)$; Let $\alpha = d_{ij}$ be the weight of the edge connecting the microservices at vertices v_i and v_j
7 **if** *both microservices v_i and v_j are not placed and both can be accommodated on current node* **then**
8 Place both microservices on current node
9 Update D as $D - \alpha$; Update k as $(k + 1)\%Q$
10 **end**
11 **else if** *either one of the microservices not placed and can be accommodated on current node* **then**
12 Place the microservice on current node
13 Update D as $D - \alpha$; Update k as $(k + 1)\%Q$
14 **else**
15 Update k as $(k + 1)\%Q$
16 **end**
17 **end**

The core idea of this algorithm is to ensure that microservice components with more number of interactions among themselves are placed on the same node. However, the algorithm mainly considers interactions among pairs of microservices. In scenarios where several microservices interact frequently with each other, this algorithm may

not provide the best possible solution. To overcome this, another heuristic algorithm is proposed, **IntMA**, a heuristic algorithm inspired by the Prim's Minimum Spanning Tree (MST) algorithm, to perform allocation of microservice components in an interaction-aware manner.

In the initial step of the IntMA algorithm described in Algorithm 4.2, the nodes feasible for scheduling any of the microservice components (satisfies resource requirements), are collected in the set F. The elements in this set are then sorted in non-increasing order of remaining processing capacity (for CPU-intensive applications) or remaining memory bytes (for memory intensive applications). The edge weights are considered in non-increasing order. An array, $ALLOCATED$ is initialized with zeroes. This array keeps track of the node allocated for each microservice component. For the first edge, both the connected microservice components can be placed on the same node, if the first node in F can accommodate the processing and memory requirements of both the microservices combined.

Lines 17-25 iterate through the microservice components that interact with the microservices that have been already allocated on the current node. The algorithm attempts to place as many dependent microservices together on the same node, restricted by the processing and memory resources available on the node. On each node, once all the dependent microservices are exhausted, any remnant microservices with no residual dependencies (ie. does not interact with any unallocated microservices implying that all interacting microservices, if any, have already been placed on a different node) are considered for possible placement on the current node. If the processing and memory requirements can be met by the current node, then, the microservices will be placed on the node. This step is included to ensure maximum utilization of the nodes deploying the application. Then, the next physical node is identified as target for the incoming microservice requests. Lines 41-54 consider this scenario. In this step, microservice components which interact with any other microservice component that has been already placed are considered. While considering the first microservice to be placed on the current node, a re-check is performed to ensure that no re-allocations are required. Once the decision to allocate is made, the residual capacities on the current node are

Algorithm 4.2: IntMA- Interaction-aware Microservice Allocation algorithm

Input : Interaction Graph, $G =< V, E >$ with edge weights D and vertex weights $T = \{< V^{cpu\_req}, V^{mem\_req} >\}$, Set of feasible nodes, F
Output: $ALLOCATED$ Array

1 Initialize F with all nodes having $h_z^{u\_res} > 0; u \in \{cpu, mem\}$
2 Create an array, $ALLOCATED$ with size as $|V|$ and initialize with zeroes
3 Sort F in non-increasing order of $f_k^{mem\_res}$ `/* for memory intensive applications */`
4 Initialize k as 1
5 Select $\alpha = max(D)$; Let $\alpha = d_{ij}$ be the weight of the edge connecting the vertices v_i and v_j
6 **if** *(*$v_i^{cpu\_req} + v_j^{cpu\_req} < f_k^{res\_cpu}$*) and (*$v_i^{mem\_req} + v_j^{mem\_req} < f_k^{res\_mem}$*)* **then**
7 Place (v_i, f_k)
8 Place (v_j, f_k)
9 Assign $ALLOCATED[v_i] = k$
10 Assign $ALLOCATED[v_j] = k$
11 **else**
12 Place (v_i, f_k)
13 Assign $ALLOCATED[v_i] = k$
14 **end**
15 $S :=$ Set of all edges incident on v_i where $ALLOCATED[v_i] = k$
16 $d_{ij} := max(S)$ where $ALLOCATED[v_i] = k$ and $ALLOCATED[v_j] = 0$
17 **while** $d_{ij} \neq NULL$ **do**
18 **if** *(*$v_j^{cpu\_req} < f_k^{res\_cpu}$ *and* $v_j^{mem\_req} < f_k^{res\_mem}$*)* **then**
19 Place (v_j, f_k)
20 Assign $ALLOCATED[v_j] = k$
21 **goto Line** *15*
22 **else**
23 Find next maximum weighted edge from S, d_{ij}
24 **end**
25 **end**
`/* allocate pending microservice components */`
26 **if** $\exists ALLOCATED[v_j] = 0$ **then**
27 $v_j :=$ Microservice component to be placed
`/* check for residual dependencies with unallocated microservice components */`
28 $int\_factor(v_j, v_{unallocated}) := \sum(d_{jl})\ \forall$ microservice components l where $ALLOCATED[v_l] = 0$
29 **if** *(*$int\_factor(v_j, v_{unallocated}) = 0$*) and (*$v_j^{cpu\_req} < f_k^{res\_cpu}$ *and* $v_j^{mem\_req} < f_k^{res\_mem}$*)* **then**
30 Place (v_j, f_k)
31 Assign $ALLOCATED[v_j] = k$
32 **end**
33 **end**
34 Increment k by 1
35 **if** $k > Q$ *or* $\nexists ALLOCATED[v_j] = 0$ **then**
36 **break**
37 **end**
38 **else**
39 $S :=$ Set of all edges incident on v_i where $ALLOCATED[v_i] \neq 0$
40 $d_{ij} := max(S)$ where $ALLOCATED[v_i] \neq 0$ and $ALLOCATED[v_j] = 0$
41 **if** $ALLOCATED[v_i] = (k-1)$ **then**
42 $int\_factor(v_i, v_{allocated}) := \sum(d_{il})\ \forall$ microservice components l where $ALLOCATED[v_l] = (k-1)$
43 **if** $d_{ij} > int\_factor(v_i, v_{allocated})$ `/* greater dependency with current microservice component, so attempt to re-allocate */`
44 **then**
45 **if** *(*$v_i^{cpu\_req} + v_j^{cpu\_req} < f_k^{res\_cpu}$*) and (*$v_i^{mem\_req} + v_j^{mem\_req} < f_k^{res\_mem}$*)* `/* can be accommodated on current node */`
46 **then**
47 Assign $ALLOCATED[v_i] = 0$
48 Place (v_i, f_k)
49 Assign $ALLOCATED[v_i] = k$
50 **end**
51 Place (v_j, f_k)
52 Assign $ALLOCATED[v_j] = k$
53 **goto Line** *15*
54 **end**
55 **end**
56 **end**

Algorithm 4.3: *Place-* Subroutine invoked by **IntRR** and **IntMA** update residual capacities on allocated nodes

Input: Microservice component to be placed, m, Node, n
Subroutine Place:
$n^{res\_cpu} = n^{res\_cpu} - m^{req\_cpu}$
$n^{res\_mem} = n^{res\_mem} - m^{req\_mem}$
if $n^{res\_cpu} < min(V^{cpu\_req})$ *or* $n^{res\_mem} < min(V^{mem\_req})$ **then**
  Increment k by 1
end

updated as shown in Algorithm 4.3. In this Algorithm, the residual resource capacities are updated to reflect the current allocations by deducting the requested resources from the available resources on that node. If the updated residual resource capacities are not sufficient to satisfy the resource requirements of the smallest microservice component, then, the algorithm proceeds to the next node in the sorted list. Algorithm 4.2 terminates either when there are no more microservice components to be placed or when there are no more feasible nodes to accommodate the microservices. On completion of the algorithm, the allocation strategy can be obtained from the $ALLOCATED$ array. The index of the array gives the microservice component identifier and the value at that index gives the identifier of the physical node to which the component has been allocated. If the value is 0, no feasible nodes have been assigned to the component and the microservice continues to wait in the scheduling queue till nodes become available.

4.4 EXPERIMENTAL EVALUATION

This section describes the experiments conducted for the evaluation of the proposed approach against the baseline approach and the default scheduling policy employed in the GKE. The experimental settings, performance metrics and the results are discussed in this section.

4.4.1 Evaluation Environment

In order to evaluate the performance of the proposed solution in real Cloud environments, microservices-based applications were run on the Google Cloud Platform (GCP) (Google 2020). Google Kubernetes (k8s) (Kubernetes 2020a) was employed as the container management platform to coordinate the application containers running across

different nodes. The experimental testbed consists of one k8s cluster with a single master node. The cluster is enabled with autoscaling features, allowing the cluster to comprise of worker nodes ranging from 3 to 10, based on the incoming requests for application pod deployment. Each worker node has 2 virtual CPUs (vCPUs) and 7.5 GB of memory. Every node in the Cloud environment runs the *kubelet* service enabling the containers hosted on it to be managed by a node designated as the master node. Applications are submitted to the master node. The scheduler component running on the master node deploys the application in containers across all the nodes. Docker containers (Docker 2020) were considered to encapsulate the application modules. The containers are configured with resource requirements and limits corresponding to the specifications in the application deployment YAML (YAML Ain't Markup Language) files.

There exists numerous solutions to support logging and monitoring of microservice-based systems, by collecting logs and other metrics. In our experiments, we used the Jaeger, Prometheus and Grafana open-source monitoring tools. Jaeger is a microservices tracing tool (JaegerTracing 2020) which was used to extract the runtime metrics of the microservices containers. Prometheus monitoring tool was employed to collect other time series metrics (Prometheus 2020) during the microservices' lifecycle. Grafana (Grafana 2020) works on top of Prometheus to provide data visualisation capabilities.

The IntRR algorithm and IntMA algorithm were provided as the scheduling policy for the scheduler component. The performance was evaluated and compared with the baseline container deployment algorithm (represented as 'default' policy in the remainder of this chapter) provided by the k8s container platform manager. In the k8s system, 'pod' is the basic unit of deployment. A pod may comprise of one or multiple containers. The default scheduler maintains a queue of pods to be scheduled (Netto et al. 2017). For each pod to be scheduled, the scheduler follows a two-step strategy. In the first step, the nodes are filtered according to resource requirements and other predicates specified by the user. In the next step, the filtered nodes are sorted according to priority functions (affinity-based, balanced resource usage, etc.) . Finally, the pod is created on

Name	Microservice types	No. of instances	Microservice Components
SockShop	9	13	Orders, Payment, User, Catalogue, Carts, Front-end, Shipping, Carts-db, Orders-db
BookInfo	4	6	Details, ProductPage, Ratings, Reviews
HipsterShop	10	12	adservice, cartservice, checkoutservice, currencyservice, emailservice, frontend, paymentservice, productcatalogservice, recommendationservice, shipping service

Table 4.3: Characteristics of reference microservice applications

the node with the highest score.

The results were also compared with a graph-partition based service placement policy proposed by Lera et al. (2018). According to this policy, applications are partitioned into sets of services by finding the transitive closure of the nodes in the graph representing the application. The policy then uses the first-fit decreasing algorithm to map the service sets to the nodes. This policy is represented as the Transitive Closure (TC) policy in the remainder of this chapter.

4.4.2 Workload Microservice Applications

To analyze the performance of the proposed approaches, different microservice applications were used : the Weaveworks Sock Shop application (Weaveworks 2017), Istio Book Info (Istio 2020) application and the Hipster Shop application (Github 2020). These applications were chosen as they represent the real-life microservices based applications. The different characteristics of these applications are provided in Table 4.3.

4.4.2.1 Sock Shop Application

Sock Shop is a reference application designed by Weaveworks to be used for testing the performance of solutions for microservice systems. It includes the user interface of an e-shopping website for socks. The application uses Go, Node.js and Spring Boot technologies. The application consists of different microservices such as front-end, order, payment, catalogue, carts and shipping. The designers of the application included sev-

eral microservices into the application to ensure that the application closely resembles real applications.

4.4.2.2 Istio BookInfo App

The BookInfo is a simple application with fewer number of microservice components. The application includes components to display the information of a book and display the review and ratings corresponding to each book. Envoy sidecars provided with each service are responsible for telemetry collection while enabling communication among the various components.

4.4.2.3 Hipster Shop

The Hipster Shop Cloud native microservice application consists of 10 tiers. The different microservices constituting the application are written in different languages. The application is an e-commerce application enabling users to view items, drop them into carts and buy the items.

4.4.3 Performance metrics

In order to analyze the performance parameters, the following metrics were considered.

- **Response Time:** Application response time is considered to be of primal importance in service based systems. The user QoS also features application response time as a key performance indicator. Response time is defined as the time taken to generate a response to a user request. It is measured from the time the user request is received upto the point in time when the user receives the corresponding response.

  In the context of microservices, application response time can be decomposed into microservice response times. Further, the processing of each microservice request might include a sequence of invocations to other microservice components. In such cases, the microservice response time includes the sum of responses times of all the invoked microservices. When compared to monolithic applications, microservice-based applications may sometimes experience higher response times. However, this increase can be tackled by efficient orchestration

of the microservices belonging to an application.

- **Throughput:** Throughput is the rate at which requests are processed by a service. This can be computed as the number of requests serviced in unit time interval. The rate of requests flowing in the system is impacted by several factors such as time taken for computation, communication latencies and resource capacities of the nodes on which the requests are processed.

  For microservices, in addition to the aforementioned, the time spent in communication between the different microservices also affects the observed throughput. Due to these reasons, throughput can be considered to be one of the vital metrics to evaluate the 'fairness' of an allocation / placement strategy.

- **Interaction factor:** The interaction factor metric is defined as the overhead incurred on the system due to communications between microservices deployed on different nodes. In this work, the communication between two microservice components is referred to as 'interaction'. Two types of interaction factor values are considered:

  - Inter-node interaction factor: The inter-node interaction factor value for a pair of nodes is computed as the sum of interactions between the different microservice components deployed on each node.

  - Overall interaction factor: This metric is calculated as the total of the inter-node interaction factor values across different pairs of nodes on which the application microservices are deployed.

  An increase in the interaction factor value indicates increased communication across the physical nodes deploying the application which incurs an overhead on the system. It leads to higher communication latencies and also significantly increases the response times experienced by the users. Thus lower interaction factor values are desirable.

4.5 RESULTS

The evaluative analysis of the proposed algorithm was done mainly using three applications, Sock Shop application, Istio BookInfo application and Hipster Shop application. Several runs were conducted for each of the experiments and the presented results contain the averaged values across the runs.

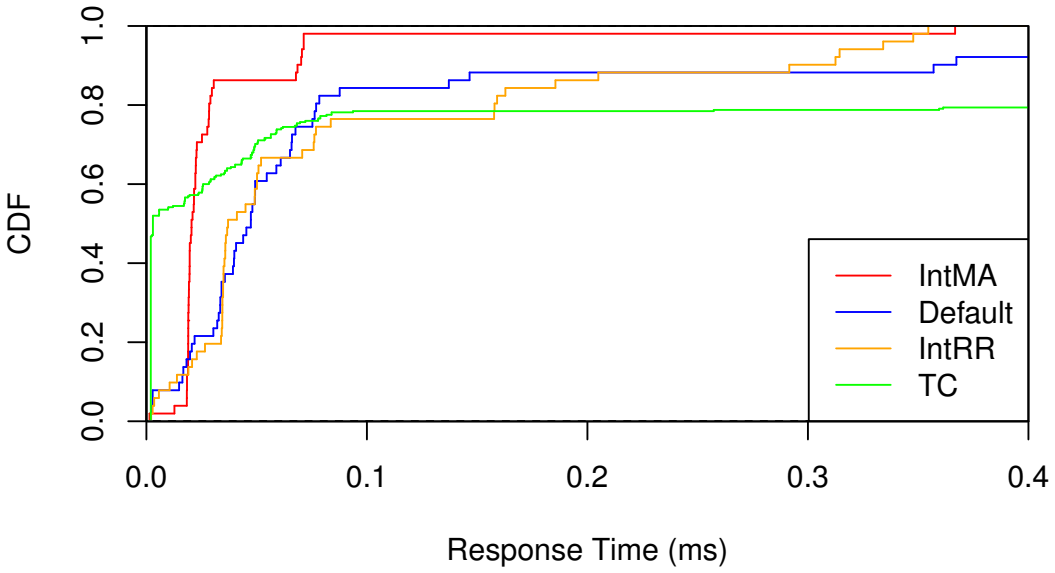
4.5.1 Evaluation of the metrics

The microservices are assigned to different nodes according to the IntMA, IntRR scheduling algorithms and the default k8s scheduling policy. All the performance metrics are evaluated and compared with respect to the values obtained by deploying the application using the k8s scheduler. The values of the performance metrics, response time and throughput obtained when the microservice applications are deployed using the TC policy, are also presented.

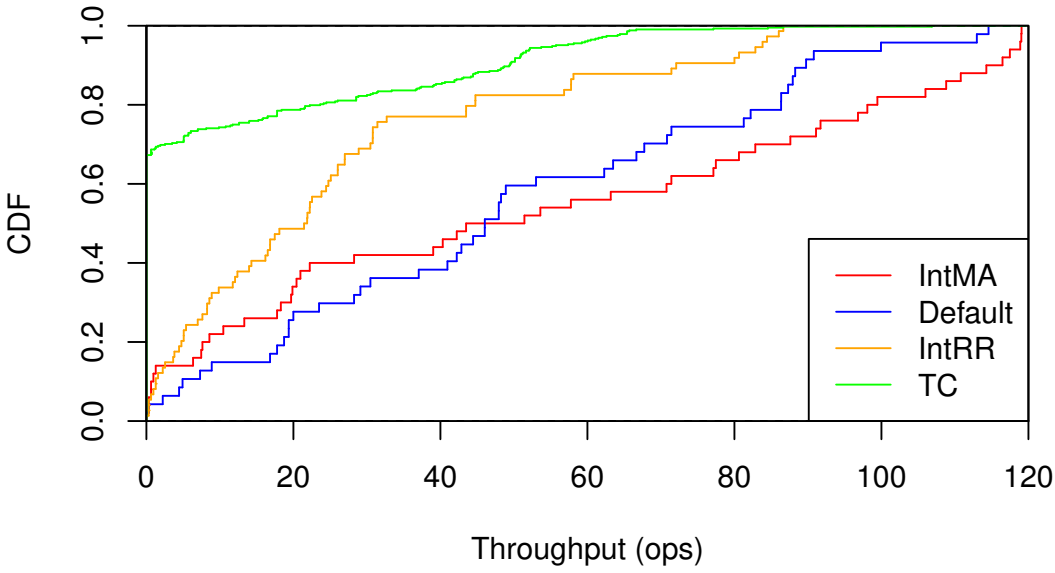
Figure 4.3a shows the Cumulative Distribution Function (CDF) of response time values for different component microservices of the Sock Shop application, deployed using the IntMA, TC, default and IntRR scheduling policies. It is observed that on adopting IntMA, 90% of the requests are serviced within time less than 0.1 ms. It is observed that the response times are lower for the IntMA policy. The lower response times are mainly due to the fact that the microservice units which frequently interact are placed on the same nodes. In the Cloud datacenter, the communication latency among containers hosted on the same node is negligible when compared to the communication across containers on different nodes.

The default scheduler policy, schedules the microservice components without regard for the interaction pattern, thus leading to higher waiting times. From the average response time values (averaged across the entire time interval of 55 minutes) in Table 4.4, it is seen that the response time is lowered by 62% when the microservice application is scheduled using the IntMA approach and there is a 78% increase when the application is scheduled using the IntRR policy. The response time increases by 88% when the application is scheduled according to the TC policy.

Figure 4.3b shows the CDF of throughput for the SockShop application scheduled



(a) CDF of Response Time for SockShop application



(b) CDF of Throughput for Sock Shop application

Figure 4.3: Performance metric values for Sock Shop application

	IntMA	IntRR	TC	Default
Sock Shop application	0.0335	0.1551	0.1640	0.0871
Istio BookInfo application	0.0138	0.0161	0.0147	0.0166
Hipster Shop application	0.1188	0.1883	0.1752	0.1563

Table 4.4: The average response times (in ms) for the different applications

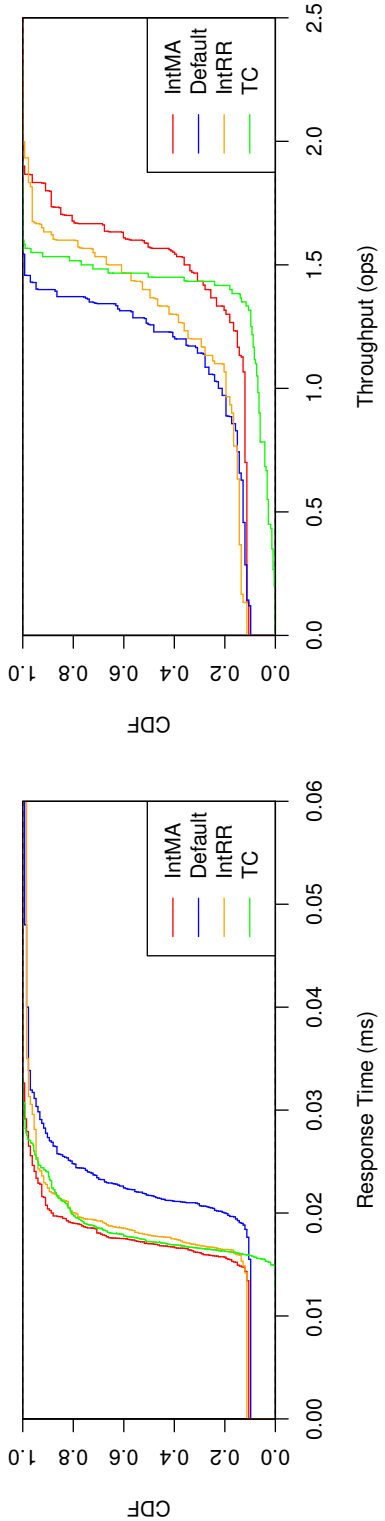
	IntMA	IntRR	TC	Default
Sock Shop application	8.3843	7.9347	7.8423	7.9613
Istio BookInfo application	1.2553	1.1480	1.1473	1.0908
Hipster Shop application	1.2389	1.0415	1.0173	1.0286

Table 4.5: The average throughput (in ops) for the different applications

by the IntMA, default and IntRR approaches respectively. The throughput is measured in terms of operations per second (ops). When the application is deployed using the IntMA policy, throughput is higher than 50 ops in 50% of the cases. For the SockShop application, IntMA provides a 5% increase in the throughput, while the IntRR provides a 0.3% decrease and TC provides a decrease of 1.5%, as can be observed from the average throughput values (averaged across the entire time interval of 55 minutes) in Table 4.5.

Figure 4.4a gives the CDF of response time values for the BookInfo application corresponding to the IntMA, default, TC and IntRR scheduling policies, respectively. It is observed that the IntMA policy reduces the average response time by 16%, while the TC and IntRR witnesses 11% and 3% decrease in the response time respectively. The IntRR policy considers the interactions while alloting the components to the nodes in a circular manner. This may lead to neglection of some interactions. However, the IntMA ensures that the maximum possible number of interacting components are placed together.

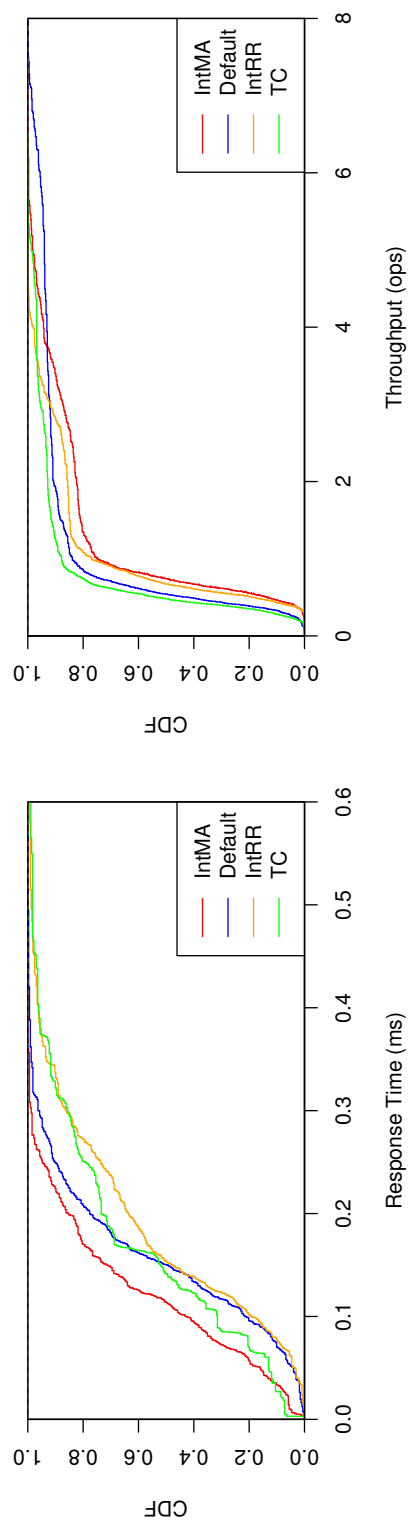
Figure 4.4b shows the CDF of throughput for the BookInfo application scheduled by the IntMA, default, TC and IntRR approaches respectively. For the BookInfo application, IntMA scheduling policy provides a 9% increase in the average throughput, while TC and IntRR leads to a 5.1% and 5.2% increase in the throughput, respectively. The proposed IntMA places the frequently communicating entities close to each other, thereby reducing the communication latencies. This positively impacts the number of



(a) CDF of Response Time for BookInfo application

(b) CDF of Throughput for BookInfo application

Figure 4.4: Performance metric values for Istio BookInfo application



(a) CDF of Response Time for Hipster Shop application

(b) CDF of Throughput for Hipster Shop application

Figure 4.5: Performance metric values for Hipster Shop application

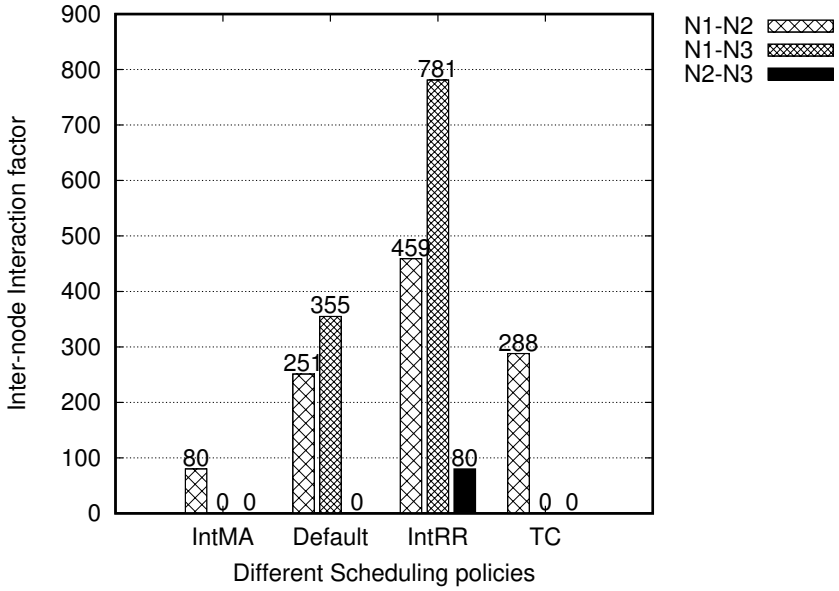


Figure 4.6: Interaction factor value for Sock Shop application across nodes

successful responses returned per unit time.

The CDF of response time values obtained by the Hipster Shop application when scheduled using the IntMA, default, TC and IntRR scheduling policies are plotted in Figure 4.5a. Response time values are highest when the application is deployed according to the IntRR scheduling policy. The least values for response times are obtained when the application is deployed according to the IntMA scheduling policy. The values recorded in Table 4.4 indicate that the IntMA scheduling policy lowers the response time by approximately 36%.

Figure 4.5b illustrates the CDF of throughput values for the Hipster Shop application. It may be noted that instantaneous throughput values in few instants is higher when using the default scheduler. However, the averaged values, which better captures the trend of the throughput values, indicates that the IntMA scheduling policy improves the throughput by 18.9%.

Figures 4.6, 4.7 and 4.8 show the inter-node interaction factors of the different nodes ($N1$, $N2$ and $N3$) hosting the applications - Sock Shop application, Istio BookInfo application and Hipster Shop application. For all applications, the IntMA scheduling policy ensures that there is no interaction between the second and third nodes. For

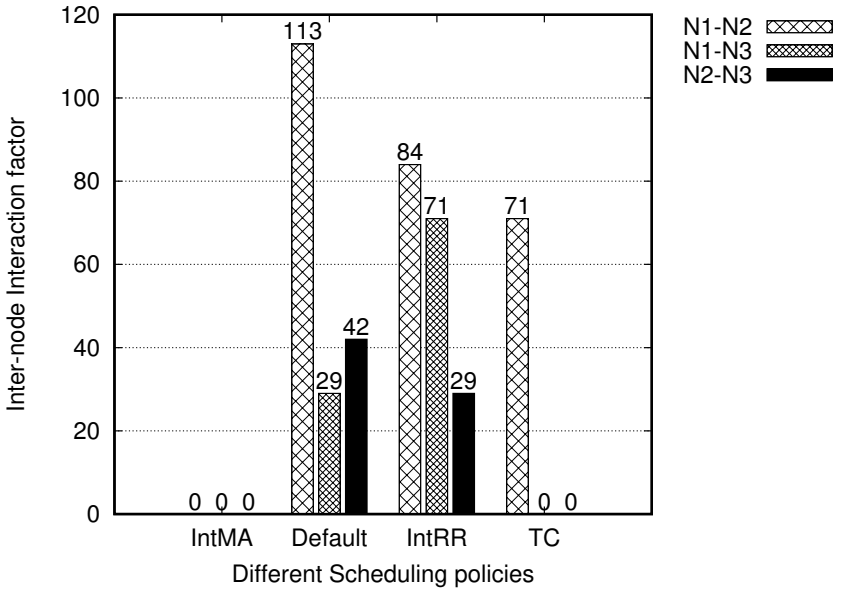


Figure 4.7: Interaction factor for Istio Book Info application across nodes

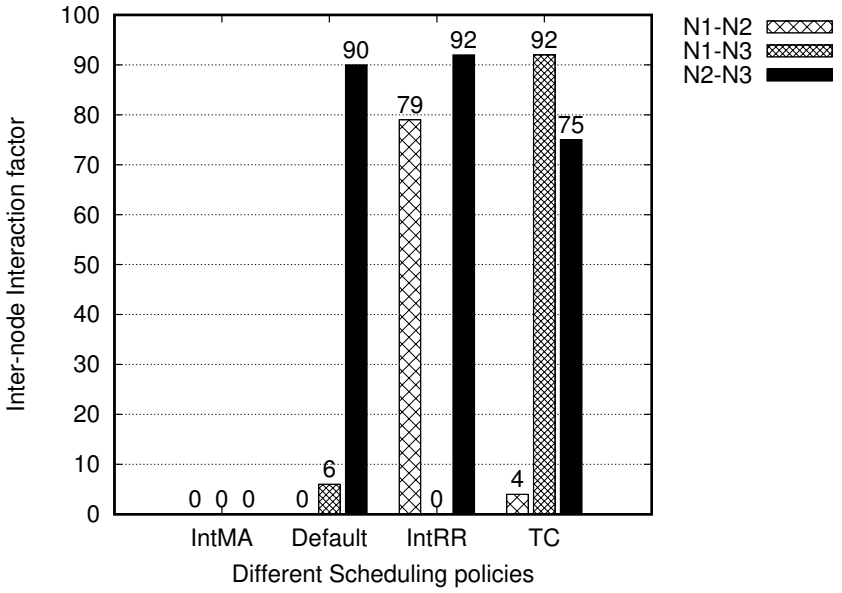


Figure 4.8: Interaction factor for Hipster Shop application across nodes

the Sock Shop application, the first and second nodes interact with each other with an interaction factor of 80. It is also perceived that none of the nodes interact between themselves when Istio BookInfo application and Hipster Shop application are scheduled using the IntMA policy. For the smaller application Istio BookInfo, IntRR has lower interaction factor values than the default policy, whereas for the larger applications, Sock Shop application and Hipster Shop application, the default scheduler provides lower values for the interaction factor. Since the TC scheduling policy places services only based on the transitive closure and does not consider the interaction factor value, TC has higher interaction factor values for all applications.

The higher the interaction factor value, higher the frequency of interactions between the physical nodes. This leaves provisions for increase in waiting times for responses from the component microservices resulting in higher response time values. Thus, lower interaction factor values are desirable. In this context, the proposed algorithm IntMA outperforms the baseline IntRR and TC algorithms and the default scheduling policy. The IntMA algorithm attempts to place all entities that interact with each other, on the same node. If any of the nodes have sufficient capacity, IntMA places all components of the application on that node itself. When all components are placed on the same node, there will not be any interactions pertaining to the application, between different nodes, thereby reducing the internode interaction factor value to 0.

Figure 4.9 shows the overall interaction factor values for the different applications deployed using different scheduling policies. The proposed policy works as expected, with the interaction factor value as zero for Istio BookInfo application and Hipster Shop application. The SockShop application, deployed using the IntMA scheduling policy spans multiple nodes and the resource requirements cannot be satisfied by any individual node alone, results in an overall interaction factor of 80, which is negligible when compared to the factor values obtained while using the default and IntRR scheduling policies. It must be observed that the IntRR algorithm works similar to the default algorithm for applications with fewer number of microservice components. However, the IntRR algorithm fares significantly lower than the other scheduling policies discussed, when the number of microservice components are higher.

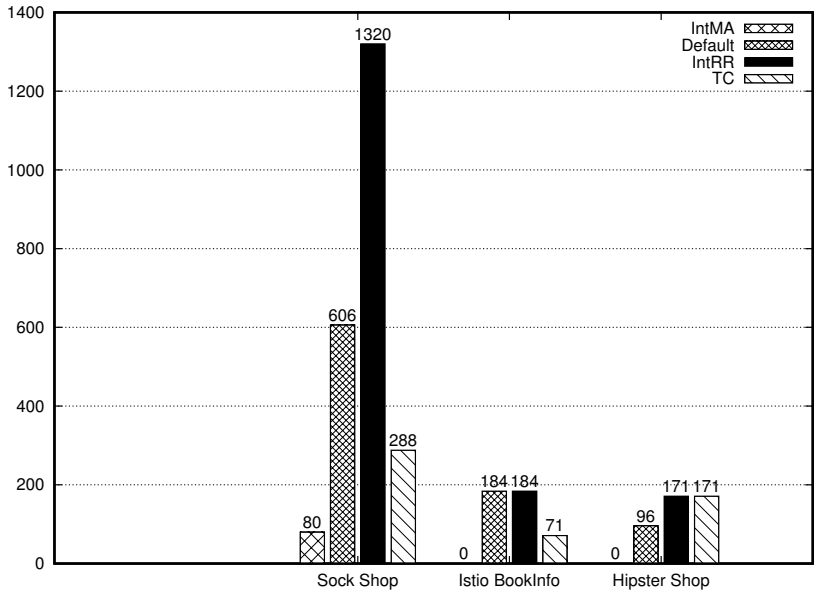


Figure 4.9: Overall Interaction factor value comparison for different schedulers

	IntMA	IntRR	TC	Default
Avg. Response time (ms)	0.0335	0.1551	0.1640	0.0871
Avg. Throughput (ops)	8.3843	7.9347	7.8423	7.9613

Table 4.6: Comparison of different approaches for Sock Shop application

4.6 DISCUSSION

The consolidated results across the different applications for the different scheduling policies are provided in Tables 4.6, 4.7 and 4.8. In all the considered test cases, the IntMA vanquishes the other described scheduling policies. The baseline approach, IntRR performs well in scenarios where the number of microservice components is less. However, it is not preferable to schedule applications with more number of microservice components using IntRR.

The applications were subjected to a load test in order to evaluate how the applications deployed handles user load. For this, a large number of requests were submitted

	IntMA	IntRR	TC	Default
Avg. Response time (ms)	0.01384	0.0161	0.0147	0.0166
Avg. Throughput (ops)	1.2553	1.1480	1.1473	1.0908

Table 4.7: Comparison of different approaches for Istio BookInfo application

	IntMA	IntRR	TC	Default
Avg. Response time (ms)	0.1188	0.1883	0.1752	0.1563
Avg. Throughput (ops)	1.2389	1.0415	1.0173	1.0286

Table 4.8: Comparison of different approaches for Hipster Shop application

to the applications using the open-source Locust tool (Heyman et al. 2020). A detailed analysis of the resource consumption by the workload microservice applications deployed using the different scheduling policies discussed in this chapter is provided in Appendix A. To assess the performance of the scheduler, the scheduling duration values were collected for the different experimental settings.

4.6.1 Scheduling Duration

In our experiments, we defined scheduling duration as the time spent waiting to be scheduled by the submitted pods/microservices. According to the pod's lifecycle in the k8s platform (Kubernetes 2020b), when an application is deployed on the GKE cluster, the associated pods are created. They continue in a 'Pending' state till they are scheduled. This time involves the time spent in waiting for the scheduler and the time required for pulling of the container images. In the experiments conducted, all the container images are pre-pulled from the registry. Thus, the scheduling duration includes the time elapsed since the pod creation to the time when the pod starts running on the scheduled node.

Figure 4.10 presents the average scheduling duration for the different workload applications considered. The results are averaged across the scheduling durations of all pods constituting the application. The scheduling duration details for the individual pods of each workload application is presented in Table 4.9. At the pod level, the IntMA scheduling policy reduces the scheduling duration for some pods, while increasing the scheduling duration for the remaining pods.

However, while considering the average scheduling duration across pods for each application, it is observed that the IntMA scheduling policy increases the scheduling duration by $2 - 5$ milliseconds, while the IntRR scheduling policy performs slightly better where the increase is between $1 - 4$ milliseconds, when compared to the default

Application Details		IntMA	Default	IntRR
Weaveworks SockShop	carts	16 ms	2 ms	16 ms
	carts-db	20 ms	26 ms	17 ms
	catalogue	15 ms	6 ms	20 ms
	catalogue-db	16 ms	2 ms	19 ms
	front-end	17 ms	11 ms	17 ms
	orders	33 ms	25 ms	10 ms
	orders-db	5 ms	5 ms	6 ms
	payment	4 ms	4 ms	7 ms
	queue-master	15 ms	10 ms	14 ms
	rabbitmq	17 ms	3 ms	15 ms
	shipping	12 ms	3 ms	12 ms
	user	8 ms	7 ms	7 ms
	user-db	9 ms	7 ms	10 ms
Istio BookInfo	details	9 ms	2 ms	9 ms
	ratings	19 ms	18 ms	9 ms
	reviews-v1	4 ms	2 ms	4 ms
	reviews-v2	6 ms	2 ms	7 ms
	reviews-v3	4 ms	1 ms	7 ms
	productpage-v1	4 ms	2 ms	4 ms
HipsterShop	adservice	60 ms	117 ms	5 ms
	cartservice	61 ms	59 ms	44 ms
	checkoutservice	41 ms	28 ms	17 ms
	currencyservice	23 ms	32 ms	14 ms
	emailservice	63 ms	23 ms	66 ms
	frontend	38 ms	30 ms	52 ms
	loadgenerator	76 ms	85 ms	94 ms
	paymentservice	46 ms	26 ms	46 ms
	productcatalogservice	18 ms	20 ms	38 ms
	recommendationservice	28 ms	20 ms	42 ms
	Redis-cart	23 ms	20 ms	37 ms
	shippingservice	32 ms	14 ms	39 ms

Table 4.9: Scheduling Duration values for each microservice in the workload applications using IntMA, default and IntRR scheduling policies

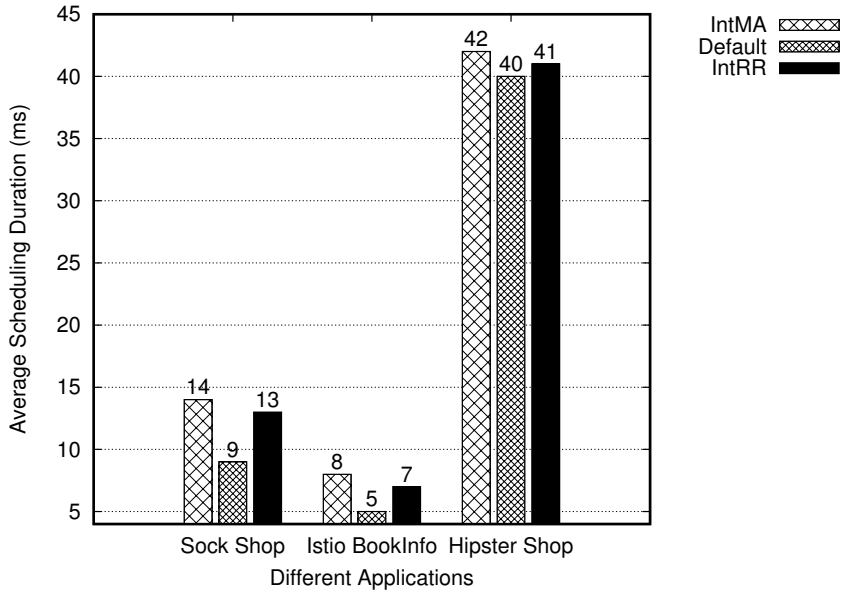


Figure 4.10: Average Scheduling Duration comparison across default, IntMA and IntRR schedulers for different applications

scheduler. However, this increase is negligible when compared to the performance improvement obtained using the IntMA approach in the throughput and response time values.

4.6.2 QPP and Heuristic approach

Table 4.10 provides the solutions obtained for the toy problem presented in Section 4.3.4 using classical optimization, and the IntMA, IntRR algorithms discussed in this chapter. For small solution spaces, all the three methods provide similar results, indicating that the proposed heuristic approach, IntMA closely approximates the QPP. However, as the number of nodes and the number of microservice components increases, the number of terms in the objective function undergoes an exponential increase, thereby introducing several challenges in solving the problem using classical optimization techniques.

4.6.3 Statistical Analysis

In order to evaluate the validity and effectiveness of the proposed approach, statistical analysis was conducted. For hypothesis testing, t-test was performed on the performance metric values for the proposed IntMA and the IntRR allocation policy. The results of the t-test are provided in Table 4.11. The commonly used significance level

Parameter	Mathematical optimization	IntMA	IntRR
Objective value	11	11	11
x_{11}	0	1	1
x_{12}	1	0	0
x_{21}	0	1	1
x_{22}	1	0	0
x_{31}	1	0	0
x_{32}	0	1	1
x_{41}	1	0	0
x_{42}	0	1	1

Table 4.10: Comparison of solutions obtained from mathematical optimization problem, the IntMA algorithm and the IntRR algorithm

Algorithm	***p*-value**
IntMA-Default	0.011
IntMA-IntRR	0.023

Table 4.11: *p*-values obtained from *t*-test

of $\alpha = 0.05$ was used in the conducted tests. For all cases, the p-value obtained was lower than the significance level, thus providing evidence of the effectiveness of the proposed allocation policy in improving the performance metrics.

4.6.4 Threats to Validity

The following threats to validity have been identified.

- Scheduling Duration metric: In the experiments conducted, it was assumed that all images have already been pre-pulled on the nodes on which it runs. The results obtained depend on various conditions such as network throughput, container image size, etc.). The collected results may vary according to the different conditions. Though the actual values may vary, the proportion of the values will be maintained.

- Workload Applications: The experiments considered three open-source web-based microservice applications. In order to evaluate the feasibility of the proposed approach to the generic environment, several other types of applications will have to be considered. The performance must also be evaluated when there are several instances of different applications with confilicting resource requirements.

- Interaction Graph Generation: A pre-requisite for the proposed IntMA is the

availability of a load test script that can be used to generate the interaction graph corresponding to the application. The load test must be structured in such a way that all possible interactions among all the components of the application are simulated in the load test.

4.7 SUMMARY

The transition to microservices brings a wide range of infrastructural orchestration challenges. Microservice application deployment in containerized datacenters must be optimized to enhance the overall system performance. In this chapter, the deployment of microservice application modules on the Cloud infrastructure is first modelled as a Binary Quadratic Programming Problem. In order to reduce the adverse impact of communication latencies on the response time, a novel, robust heuristic approach **IntMA** is proposed for deploying the microservices in an interaction-aware manner with the aid of the interaction information obtained from the Interaction Graph. The interaction pattern between the microservice components is modelled as an undirected doubly weighted complete Interaction Graph. The proposed allocation policies are implemented in k8s. The effectiveness of the proposed approach is evaluated on the Google Cloud Platform, using different microservice reference applications. Experimental results indicate that the proposed approach improves the response time and throughput of the microservice-based systems.

CHAPTER 5

NATURE-INSPIRED RESOURCE MANAGEMENT AND DYNAMIC RESCHEDULING OF MICROSERVICES IN CLOUD DATACENTERS

Devising performance-optimized solutions that guarantee QoS is of paramount importance. Container orchestration platforms incorporate activities to integrate and manage the high footprint microservice applications running on containers, providing support for tasks such as initial deployment, state management and scaling while ensuring fault tolerance (Khan 2017). However, these platforms seldom consider the fluctuations involved in Cloud environments. To alleviate this gap, degradations in the initial deployment configurations may be handled by re-scheduling and consolidating the microservice application containers to new nodes (Rodriguez and Buyya 2019), enabling the nodes with lesser workloads to be switched off for energy-savings.

It is imperative that Cloud providers tackle the performance-energy tradeoff in Cloud resource management. Two significant mechanisms that facilitate this are virtual machine (VM) Consolidation (VMC) and Dynamic Voltage Frequency Scaling (DVFS). VM consolidation techniques reschedule the tasks across fewer number of nodes, enabling the unused nodes to be switched off. VMC techniques can be broadly classified into static and dynamic VMC algorithms (Bermejo et al. 2019). VMC algorithms are generally considered to comprise of three core phases (Khan et al. 2018): Source Node Selection, VM Selection and Destination Node Selection. Most of the existing research on VMC focus on virtual-machine based environments.

Cloud datacenters are now shifting to microservice-based applications running in containers. There is a need to consider the rescheduling of microservice containers to ensure that the performance objectives are met in containerized Cloud data centers (Fazio et al. 2016; Rodriguez and Buyya 2019). Microservice re-scheduling is a NP-hard problem and there exists much scope for further optimization.

Xu et al. (2018) proposed iBrownout for energy-aware scheduling microservices using the concept of *brownout*. The research was further extended in BrownoutCon (Xu and Buyya 2019) to support resource management in containerized Clouds. However, the system does not include any migration of microservice containers and rather switches off the optional components of microservice applications.

Piraghaj et al. (2015) conceptualized a framework for container consolidation that dealt with consolidating containers on virtual machines. Only simulation experiments were carried out. On the contrary, we have considered real-world microservice applications in our system. Rattihalli (2018) proposed a container migration system that also identifies the right-size of each job by offline profiling. The authors have not considered the QoS attained by the system. Rodriguez and Buyya (2018) presented a k8s-based framework for autoscaling and reactive re-scheduling in containerized Cloud environments. However, a better alternative is to use a proactive strategy. Thus, there is a need for systems that proactively trigger re-scheduling activities which includes node and container selection, based on the values monitored at regular intervals, rather than waiting till incoming pods get stuck in the 'pending' state.

These issues served as the motivation to design and develop a novel system to dynamically perform the re-scheduling of microservice containers based on periodically monitored resource utilization values. In this work, the main aim was to achieve better average response time. It was observed that a key factor affecting the response time is the container CPU throttling values. Communication overhead between the different microservice components of an application equally affected the response time. In the context of microservice containers, there are additional configuration factors such as CPU/memory limits, to be considered.

5.1 CPU REQUESTS, LIMITS AND THROTTLING

The k8s container management platform deploys Docker application containers by means of *.yaml* deployment files. These files generally contain resource requirements for the containers. CPU and memory resource configurations can be specified in two types: Resource requests and Resource limits. Requests specify the amount of resources required by the container to ensure proper execution and is considered by the k8s scheduler to schedule the containers. The pods with single or multiple containers are scheduled on to nodes with resources that can satisfy the resource requests. However, the containers are permitted to use resources upto the specified limits.

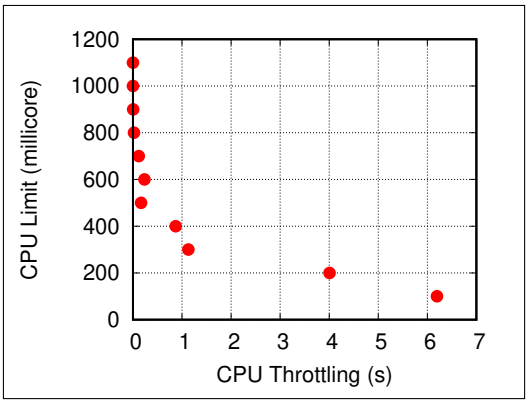


Figure 5.1: Relation between CPU throttling and container CPU Limit - Figure shows the level of container CPU throttling corresponding to the variation in container CPU limits. Containers with lower CPU limits are subjected to more CPU throttling.

The CPU requests and CPU limits can be considered to be soft limits and hard limits respectively. The CPU control groups (*cgroups*) value is set by the CPU requests. Based on the CPU requests and limits of all containers in a pod, pods are categorized into different QoS classes: Guaranteed, Burstable and Best-Effort. As CPU is a compressible resource, containers with CPU resource usage greater than the CPU limits, are throttled as a precaution against the noisy neighbour effect. The importance of container CPU throttling in determining response time, which is one of the key Application Performance Indicators, was analysed by varying the container CPU limits and measuring the response time. Figure 5.1 depicts how the amount of throttling that a container is subjected to, is affected by the CPU resource limits. When the CPU limit is sufficiently high, the containers experience negligible throttling and vice-versa. To capture

the relationship of the variation in response time, different workloads were considered. The CPU limits were varied and service response time values were measured. The Pearson correlation co-efficient between the CPU throttling and response time values was calculated to be 0.611, indicating a moderate to strong positive correlation which is not linear. Further, a value of 1 for both the non-parametric rank-based statistics Spearman rank coefficient and Kendall's correlation coefficient indicate that the response time is monotonically related to the CPU throttling level. In order to evaluate the statistical significance of the observations, t-test analysis was performed. At a significance level of $\alpha = 0.05$, the P-value is 2.315, which falls outside the critical region. Thus, the alternate hypothesis stating that response time and throttling level are correlated, is accepted.

Rescheduling is often done to optimize the overall system performance. Though initial scheduling is done in an optimized manner, the dynamic behaviour of the system can tend to decrease the performance of the current configuration, calling for rescheduling decisions. One of the key parameters indicating the microservice application performance is the response time. High level of throttling generally implies that the current host node does not have sufficient CPU resource to accommodate the container's CPU resource demands. Thus, triggering rescheduling operations to reduce the CPU throttling level can result in an improvement in the performance of microservice-based systems.

5.2 TIARM- THROTTLING AND INTERACTION-AWARE ANTI-CORRELATED RESCHEDULING FOR MICROSERVICES

Traditional re-scheduling approaches guided by the resource usage profiling have been found to be effective in hypervisor virtualization based data centers. However, in the context of microservice containers, there are additional configuration factors such as CPU/memory limits, to be considered. To alleviate this gap, container orchestration platforms must be augmented with re-scheduling strategies that perform relocating of the microservice containers to ensure optimized performance. These strategies must also take into consideration the container configuration parameters. To address the aforementioned challenges, we propose Throttling and Interaction-aware Anti-

correlated Rescheduling for Microservices (TIARM), a system for performing re-scheduling activities in container-based clusters running microservice applications. In the TIARM framework, continuous monitoring is done to perform rescheduling activities which will ensure optimized placement of microservice containers, thereby improving the QoS attained by the system. The proposed TIARM framework thus deals with dynamic rescheduling of containerized microservices using resource monitoring data. Section 5.2.1 presents the context and the overall architecture of the TIARM framework and Section 5.2.2 details the functions of the proposed TIARM framework.

5.2.1 System Architecture

A containerized Cloud datacenter running the k8s Engine consists of multiple heterogeneous hosts with varying resource capacities. Each host can accommodate VMs referred to as 'nodes' in the remainder of this chapter. The VMs are characterized by the number of virtual CPU cores ($vCPU$) and amount of memory resources (measured in terms of GBs). VMs with the same type of resources are grouped into node pools. All the VMs are interconnected to form a cluster.

Several microservice applications are submitted for deployment. In this work, the applications are initially deployed by the k8s scheduler, which first filters and then ranks the available nodes according to different priority functions. Each microservice application consists of several interconnected containers. The containers are segregated into multiple 'pods', which is the basic deployment unit in k8s clusters. The pods are characterized by different requirements and may consist of one/more containers. In order to filter out nodes that do not meet the requirements criteria, different pre-defined predicate rules are applied. In the next step, the node with the highest score is designated to run the pod. Though the scheduler does a fairly good job at initially deploying the submitted applications, dynamic changes in the workload tend to result in sub-optimal configurations. It is essential that rescheduling policies be applied periodically to ensure optimized results by adapting the configurations in response to the fluctuations in the workload.

Figure 5.2 depicts the system architecture of the proposed system. Users submit

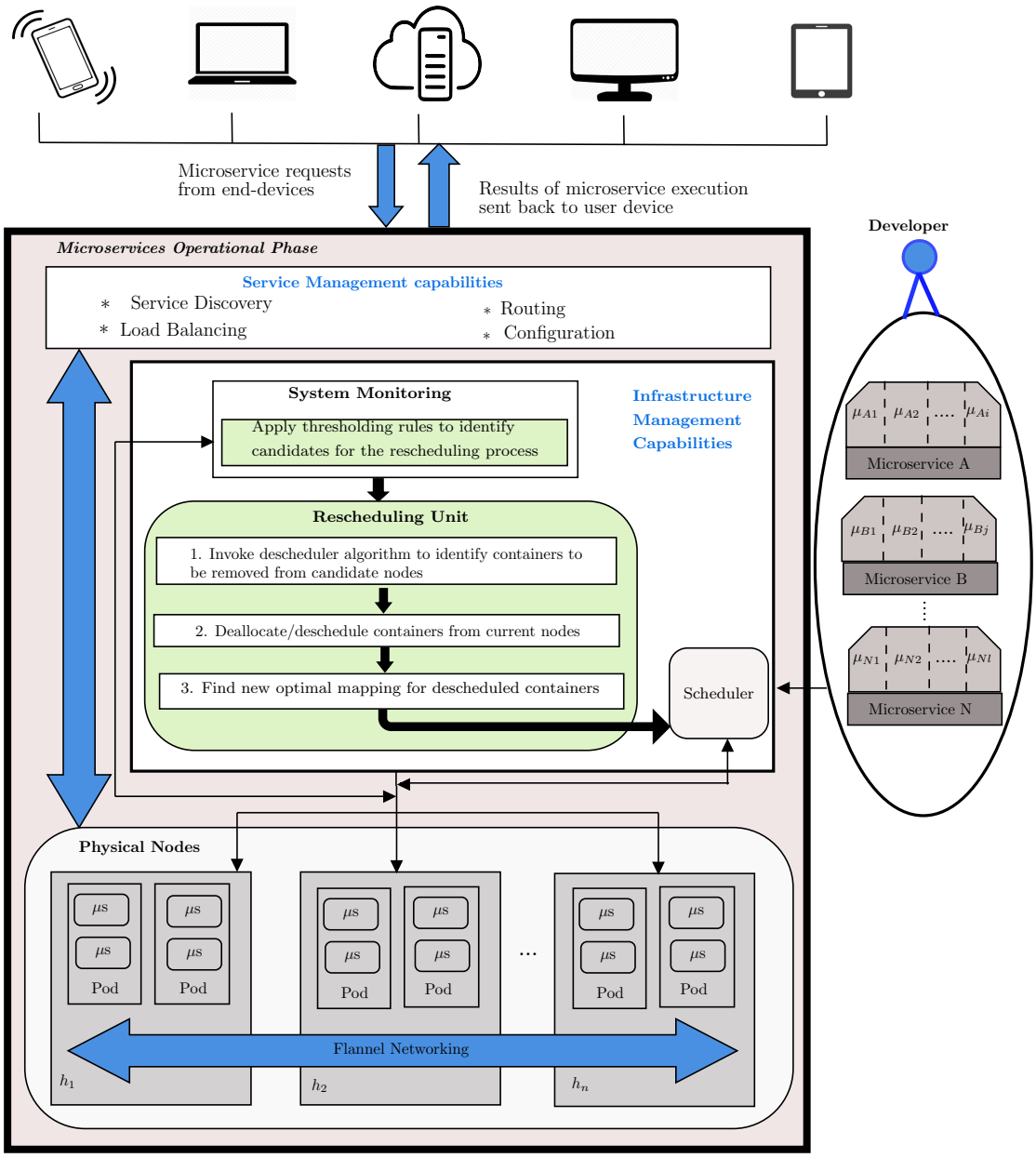


Figure 5.2: System Architecture

requests from end-devices. Requests to deploy microservice applications consisting of different microservices (*Microservice A to Microservice N*) are submitted by application developers. Each microservice can have mutiple instances (Eg: $\mu_{A1}, \mu_{A2}, etc.$). According to the taxonomy presented in Chapter 2, the activities in a microservice-based platform can be broadly categorized into developmental phase concerns and operational phase concerns. Developmental phase concerns are outside the scope of this research. Operational phase concerns can further be sub-categorised into service man-

agement capabilities and infrastructure management capabilities. The infrastructure management capabilities act as the intermediary between the microservice requests and the underlying resources. This includes tasks directly involving operations on the infrastructure layer, such as scheduling, scaling and rescheduling. This chapter focusses on the rescheduling of microservices which is represented by the boxes shaded in green in Figure 5.2.

Once the scheduler has deployed the application on the various nodes, the system monitor monitors the overall activities and invokes the rescheduler unit to trigger the rescheduling process. The activities involved in the proposed framework are as follows:

1. The System Monitoring agent monitors the resource utilization and periodically evaluates the CPU utilization to identify the $Candidate\_Nodes$ that are either over-utilized ($CPU\ Utilization > Over - Load\,Threshold\ (OLT)$) or under-utilized ($CPU\ Utilization < Under - Load\,Threshold\ (ULT)$).

2. The rescheduling unit is invoked with the identified overloaded and underloaded nodes along with their resource usage information. The rescheduling unit sends commands to the descheduler to perform container selection and descheduling operations.

3. The containers/pods to be de-allocated from each candidate node are identified using a weighted linear combination of the CPU throttling level and interaction factor, as discussed in Section 5.2.2.2. The selected containers are de-allocated from the current nodes.

4. Resizing operations are performed for containers with high CPU throttling level as discussed in Section 5.2.2.3.1.

5. The rescheduler module builds a globally optimal rescheduling plan using an extended metaheuristic algorithm based on Multi-Verse Optimization (MVO), as described in Section 5.2.2.3.2.

6. The rescheduling unit uses the binding process to notify the API server to actually perform the transfer and re-allocation of the containers.

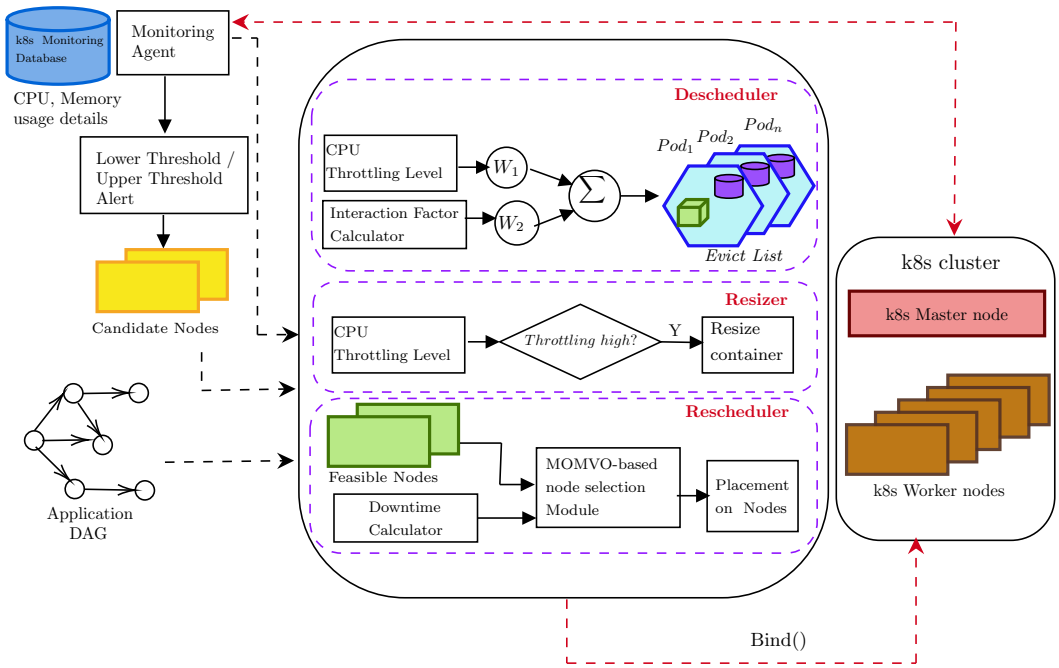


Figure 5.3: Internal Details of the Rescheduling Unit in the proposed system
Microservices rescheduling is handled by this unit which invokes the descheduler, resizer and rescheduler. The unit takes the monitored data, selected nodes and Application DAG as input. The unit invokes Bind() to place the migrating containers on the new destination nodes in the cluster.

5.2.2 Functional Details of the TIARM Framework

Microservices rescheduling consists of the descheduling and rescheduling phases. In the proposed system, the overall rescheduling activities are handled by the rescheduling unit, which invokes the descheduler algorithm (as shown in Figure 5.3) to perform container selection and descheduling. The control is then returned back to the rescheduling unit which performs resizing operations and identifies new nodes to receive the migrating containers and places the containers on the identified nodes. The Rescheduling Unit is invoked by the System Monitoring Agent as discussed in Section 5.2.2.1. Section 5.2.2.2 endetails the descheduler algorithm and Section 5.2.2.3 discusses the other operations of the rescheduling unit. This work assumes the Application Directed Acyclic Graph (DAG) to be available as input. Besides, all microservice application containers must specify CPU limits in their deployment files, which is the preferred way of deploying containers on k8s platforms.

5.2.2.1 System Monitoring Agent

The System Monitoring Agent augments the existing monitoring tools for k8s clusters, as described in Algorithm 5.1. In the Monitoring Phase, the CPU utilizations of the nodes are compared against static upper and lower thresholds to identify overloaded ($Node.OL$ is set to $TRUE$) and underloaded nodes ($Node.UL$ is set to $TRUE$), respectively. The thresholds were defined based on a set of preliminary experiments. The identified nodes are appended to a list $Candidate\_Nodes$, which is forwarded to the Rescheduling Unit.

Algorithm 5.1: Monitoring Phase

```
   Output: List of Candidate Nodes
1  Candidate_Nodes ← []
2  Monitor CPU and memory resource usage of Nodes
3  Monitor CPU and memory resource usage of Pods
4  Monitor CPU throttling level for containers in Pods
5  UT ← Upper Threshold for resource usage of over utilized nodes
6  LT ← Lower Threshold for resource usage of under utilized nodes
7  if Node Resource usage ≥ UT then
8  |  Node.OL = TRUE
9  |  Append Node to Candidate_Nodes
10 end
11 else if Node Resource usage ≤ LT then
12 |  Node.UL = TRUE
13 |  Append Node to Candidate_Nodes
14 end
15 Forward Candidate_Nodes
16 Forward Node resource usage vectors
17 Forward Pod resource usage vectors
18 Forward History of CPU throttling values for containers
```

5.2.2.2 Descheduling Phase

The Descheduling phase selects the containers/pods to be migrated from the overloaded/underloaded nodes. Microservices can be broadly classified into two: *stateful* and *stateless* microservices. In k8s, stateful microservices are deployed as StatefulSets and stateless microservices are generally deployed as Deployments or ReplicaSets. The migration of stateless microservices can be done trivially by spawning a new instance and rerouting all traffic to the new instance. Descheduling and rescheduling activities are

performed only for the migration of stateful microservices. Microservice applications contain multiple microservice units interacting with each other. Each microservice unit is deployed on a container. The different containers forming part of an application tend to interact with each other. The container selection policy used in the proposed system prefers containers with least interactions with other containers on the current node. The intuition is that highly interacting containers spend more time in communication when placed across different nodes, thereby leading to a degradation in the observed response times.

Interaction among application entities are captured in the form of Application DAG where vertices (V) represent the microservice units and edges (E) represent an interaction among the corresponding microservice units. As shown in Figure 5.3, an interaction factor calculator parses the DAG to compute the interaction factor for each microservice running on the candidate nodes. The interaction factor between two microservice containers x and y is computed as provided in Equation 5.1.

$$I(x,y) = \begin{cases} 1, & \text{if } \{x,y\} \in E \textit{ and } x \textit{ , } y \textit{ are running on the same node} \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

The relative amount of CPU throttled time(T) for each container is also considered by the selection policy. Higher values of throttling are indicators of the containers being deprived of the required CPU resources and lead to a negative impact on the response time as inferred from Section 5.1. Such containers are selected for migration to new destination nodes with more CPU resources. The CPU throttled time is calculated from the *cgroups* values forwarded by the monitoring agent as provided in Equation 5.2.

$$T(x) = \frac{\textit{number of throttled periods}}{\textit{number of periods}} \tag{5.2}$$

The Interaction factor values and CPU Throttling values independently characterize the priority of containers to be selected. The proposed Throttling and Interaction factor-aware container selection algorithm (Algorithm 5.2) thus uses a combined metric, Objective Value ($O.V$) which is a weighted sum of both the aforementioned values and is defined for microservice x as :

Algorithm 5.2: Descheduling Phase

Output: List of evicted Pods
Input : $Candidate\_Nodes$, Node resource usage vectors, Pod resource usage vectors, History of CPU throttling values for containers, Application DAG

$Evict\_List = []$
forall $node \in Candidate\_Nodes$ **do**
  $Sorted\_Pods = []$
  forall *pod running on* $node$ **do**
    $[w_1, w_2] \leftarrow$ *Weight vector*
    $T[pod] \leftarrow$ *current CPU throttling value*
    $I[pod] \leftarrow$ *Calculate interaction factor from DAG*
    $O.V[pod] \leftarrow (w_1 * T[pod]) + (w_2 * I[pod])$
    Append pods to $Sorted\_Pods$
  end
  Sort all pods in $Sorted\_Pods$ based on their corresponding $O.V$ values in non-increasing order
  forall *pod* $\in Sorted\_Pods$ **do**
    Add pod to $Evict\_List$
    Update resource usage vector of node after deducting resource usage vector of pod
    if $node.OL == TRUE$ *and updated Node resource usage* $< UT$ *or* $node.UL == TRUE$ *and updated Node resource usage* $== 0$ **then**
      break
    end
  end
end
Deschedule all pods in $Evict\_List$

$$O.V(x) = w_1 * T(x) - w_2 * I(x) \tag{5.3}$$

where $I(x) = \sum I(x, y)\ \forall y \neq x$ *on same node as x*

The values of w_1 and w_2 are experimentally determined. All containers running on the nodes in $Candidate\_Nodes$ are sorted based on the $O.V$ value (in non-increasing order). The containers are iteratively selected and added to the $Evict\_List$, till enough resources have been freed on the nodes. Further, the containers in the $Evict\_List$ are descheduled.

5.2.2.3 Rescheduling Phase

Once the containers have been descheduled, the next step is to identify suitable destinations and transfer the containers to the destination nodes.

In this phase, first, the feasible nodes with residual resources suitable for running the migrating containers are identified (refer Algorithm 5.3), following which the migrating containers are sent to the resizer module. Then, the lists of feasible nodes and migrated containers are forwarded to the Multi-Objective Multi-Verse Optimization (MOMVO)-based Node Selection Module to generate an optimal mapping.

Algorithm 5.3: Rescheduling Phase

Input : $Evict\_List$, $Candidate\_Nodes$, Node resource usage vectors, Pod resource usage vectors, Node details, Pod details

```
Feasible_Nodes = []
min_resource_request =Minimum of resources requested by pods in
  Evict_List
forall node ∈ Nodes − Candidate_Nodes do
    if node.resource_capacity − node.resource_allocated >
     min_resource_request then
        Add node to Feasible_Nodes
    end
    Allocation_vector = Find_suitable_node(Evict_List,Feasible_Nodes);
    Bind(Allocation_vector);
end
```

5.2.2.3.1 Resizer During the deployment of microservice application containers, CPU resource limits are specified. When the CPU usage exceeds the specified limits, containers are throttled. Large throttling values imply that the CPU limits specified are too low. To avoid the adverse effects of throttling on response time, the relocation of containers on to nodes with more CPU resources, must be accompanied by a corresponding increase in the fractional CPU share. The Rescheduling Unit includes a resizing sub-module (second block in Rescheduling unit in Figure 5.3) to perform this action and raises the configured limits for migrating containers with throttling values greater than a threshold, before proceeding to the Node Selection Module. Based on profiling experiments, the threshold is set to 70%.

5.2.2.3.2 MOMVO-based Node Selection Module The Node Selection Module maps each migrating container $\{ms_1, ms_2, ..., ms_n\} \in Evict\_List$ to potential nodes $\{fn_1, fn_2, ..., fn_m\} \in Feasible\_Nodes$. Each ms_i can be characterized by a resource request vector of dim dimension, corresponding to different resources such as CPU, memory, storage, bandwidth, etc. Each potential node is characterized by a resource capacity vector of dimension dim. In this study, we have considered the CPU and memory resources ($dim = 2$). This can be extended to consider any number of resources. The output of this module is a mapping, which when applied, would result in an improvement in the QoS metrics such as response time. Let z_{ij} denote the decision variable, defined as in Equation 5.4.

$$z_{ij} = \begin{cases} 1, & \text{if } ms_i \text{ is allocated on node } fn_j \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

A significant factor in this context is the time required for migrating the container from the current node to destination node. This determines the amount of time that the service will not be available to the users and is referred to as $downtime$. This value is dependent on the transfer and start-up of the containers.

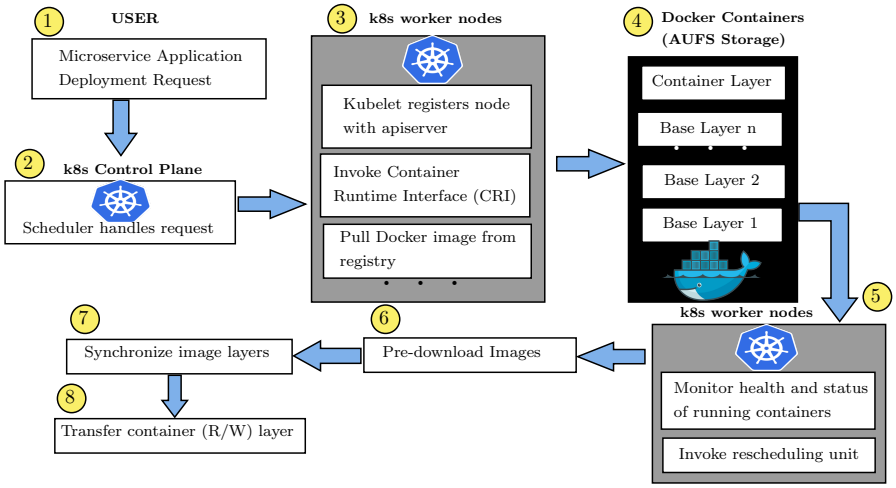


Figure 5.4: Sequence of activities in the proposed rescheduling system
Steps 1-4 handle application deployment requests and initiates the different containers; Steps 5-8 are responsible for migrating the containers across the worker nodes

The overall workflow of the proposed system is depicted in Figure 5.4. Applications are deployed by the k8s control plane on the worker nodes, each running *kubelet* instances which communicates with the *apiserver* and initiates the pulling of container

images. Docker container images consist of different layers each corresponding to different steps in the Dockerfile. The running container adds an additional Read/Write layer (aka container layer) which is the only layer that can be modified at runtime. There are several researches on how live migration of containers can be performed using techniques such as Checkpoint and Restore in Userspace (CRIU) (Karhula et al. 2019; Puliafito et al. 2019). In this work, we consider the approach proposed by Ma et al. (2018, 2017), where the base layers are pre-fetched directly at the destination node (Step 6 in Figure 5.4), then *diff* operation is performed (Step 7) and finally the container layer is relayed across the network (Step 8). This greatly reduces the time required for migration. Though the migration technique has been proposed mainly for edge environments, it is trivial to adopt the same in Cloud environments.

Thus, the $downtime$ is computed using Equations 5.5, 5.6 and 5.7.

$$\begin{aligned} \textit{Downtime} = \textit{Image pulling time} \; + \; \textit{R/W layer transfer time} + \\ \textit{container startup time} \end{aligned} \tag{5.5}$$

where,

$$\textit{Image pulling time} \; = \frac{\textit{Size of container image (in bytes)}}{\textit{Network bandwidth of Node (bytes/sec)}} \tag{5.6}$$

$$\textit{R/W layer transfer time} \; = \frac{\textit{Size of R/W layer (in bytes)}}{\textit{Network bandwidth of Node (bytes/sec)}} \tag{5.7}$$

The size of the R/W layer is measured using the 'docker ps' command and the *container startup time* is calculated from the logs of the running container (at the previous node).

The performance of workloads often depend on other workloads running on the same node. Workloads with positively correlated resource utilization running on the same node offer more risks of over-utilization (Wang et al. 2019b). Coupling microservice containers with complementary resource demands can improve the resource utilization of the node and thus improve QoS values (Shaw et al. 2018). So, anti-correlation values of the resource vectors are also used to guide the node selection process. Anti-correlation values are calculated as the additive inverse of the correlation values calculated using the Pearson Correlation Coefficient.

$$AntiCorr_{xy} = -1 * \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}} \tag{5.8}$$

where x_i and y_i are the resource usage status of the container and node at each sample point within the monitoring interval. The value of $AntiCorr_{xy}$ falls in the range $[+1, -1]$, where values greater than zero indicates that the resource vectors are complementary.

Thus, the objectives of the Node Selection module are as follows:

Objective 1: Minimize the downtime experienced, where downtime is calculated as given in Equation 5.5.

Objective 2: Maximize the anti-correlation between the microservice container and node resource vectors, where anti-correlation is defined in Equation 5.8.

In order to ensure that the solutions generated represent feasible solutions, Constraints 5.9, 5.10 and 5.11 are applied.

$$\forall fn_j \in Feasible\_Nodes, \sum_{i=1}^{n} z_{ij} * ms_i^{cpu} \leq fn_j^{cpu} \tag{5.9}$$

$$\sum_{i=1}^{n} z_{ij} * ms_i^{mem} \leq fn_j^{mem} \tag{5.10}$$

$$\forall ms_i \in Evict\_List, \sum_{j=1}^{m} z_{ij} = 1 \tag{5.11}$$

The proposed system employs an extended Multi-Objective Multi-Verse Optimizer (MOMVO) for the Node Selection Process. The Multi-Verse Optimization (MVO) algorithm inspired by the concepts of cosmology was proposed by Mirjalili et al. (2016). Since the problem involves multiple objectives, a multi-objective variant of MVO (Mirjalili et al. 2017) based on the concept of Pareto dominance is employed to generate the Pareto optimal solutions. The pseudocode is presented in Section 5.2.2.3.2.1. The main advantage of MVO-based algorithms over GA-based algorithms is that any solution can advance the creation of new solutions and elitism is also preserved.

An overview of the MOMVO is provided in Figure 5.5. Different universes are randomly generated, where each universe consists of as many objects as the number of

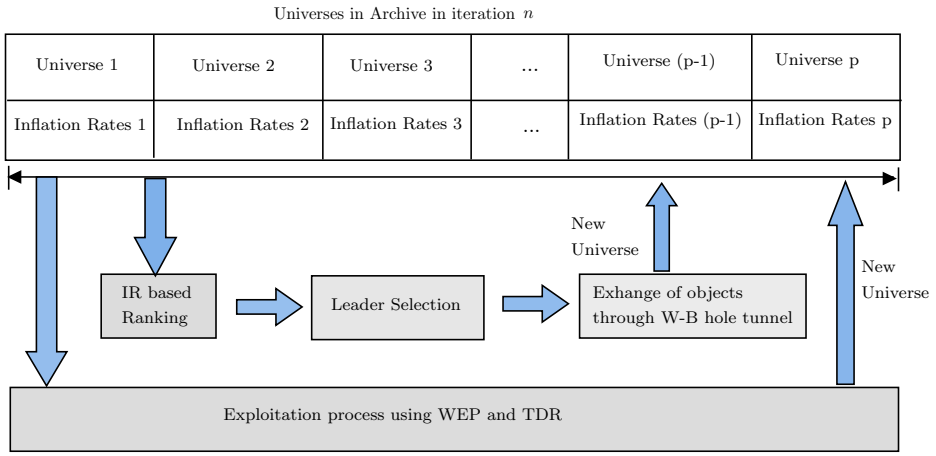


Figure 5.5: Overview of the exploration-exploitation steps in MOMVO

migrating containers. The value of each object gives the potential node that can run the migrating container. Each universe is assigned two values for the inflation rate - one corresponding to the summation of downtime values for each pod and one corresponding to the sum of anti-correlation values for each pod. In order to adapt the stochastic metaheuristic for the discrete solution space, the Smallest Position Value (SPV) rule (Tasgetiren et al. 2004) was applied.

5.2.2.3.2.1 Description of MOMVO-based algorithm for Node Selection: MOMVO is a stochastic algorithm based on the concept of Pareto dominance. In the Node Selection phase, multiple conflicting objectives are considered. In such multi-objective problems, rather than obtaining 'best solutions', solutions that provide the best-tradeoff between the multiple objectives is considered. For a minimization problem, a solution $\vec{x} = x_1, x_2, ..., x_k$ is superior to a solution $\vec{y} = y_1, y_2, ..., y_k$, if the following condition holds:

$$\forall i \in 1, 2, .., n : f_i(\vec{x}) \leq f_i(\vec{y}) \textit{ and } \exists i \in 1, 2, .., n : f_i(\vec{x}) < f_i(\vec{y}) \tag{5.12}$$

where n is the number of objectives for the problem.

The best solutions obtained by using the Pareto dominance comparator are called Pareto optimal solutions and they constitute the Pareto optimal front.

Algorithm 5.4: MOMVO-based Node Selection Algorithm

Input : $Feasible\_Nodes$, $Evict\_List$
Output: Mapping of each evicted pod to a feasible node, $Allocation\_vector$

function $Find\_suitable\_node$**:**
  Create initial random universes
  Sort universes based on Pareto dominance
  Initialize WEP and TDR
  while *end criterion not satisfied* **do**
    for *each universe* i **do**
      $inflation\_rate\_1[i] = downtime(i)$
      $inflation\_rate\_2[i] = AntiCorr(i)$
    end
    for *each universe* i **do**
      Update WEP and TDR
      $Black\_hole\_index = i$
      for *each object* j *in universe* **do**
        Generate random number $r1$
        if $r1 <$ *Normalized Inflation rates of universei)* **then**
          $White\_hole\_index =$ *Index of leader selected from archive*
          Replace object j with object from $Leader$ universe
        end
        Generate random number $r2$
        if $r2 < WEP)$ **then**
          Generate random numbers $r3$, $r4$
          if $r3 < 0.5$ **then**
            Update object j according to Equation 5.13
          end
        end
      end
    end
  end
  Return $Allocation\_vector$
end

MOMVO integrates an archive of a fixed capacity to preserve the best Pareto dominated solutions obtained throughout the process. At each step, a leader is elected from the archive, which is subjected to the exploration and exploitation steps.

The steps in MVO are inspired by the white-hole, black-hole and worm-hole concepts in cosmology. During the white-hole propagation, objects from an inferior universe are replaced with the corresponding object in the best universe. To enable ex-

ploitation, wormhole tunnels are formed between each universe and the best universe. The exploitation process is guided by two variables: Wormhole Existence Probability (WEP) and Travelling Distance Ratio (TDR). The information is exchanged between j^{th} objects of i^{th} universe and best universe X through wormhole tunnels as follows:

$$x_i^j = \begin{cases} \begin{cases} X_j + TDR * ((ub_j - lb_j) * r4 + lb_j), & \text{if } r3 < 0.5 \\ X_j - TDR * ((ub_j - lb_j) * r4 + lb_j), & \text{if } r3 \geq 0.5 \end{cases}, & \text{if } r2 < WEP \\ x_i^j, \text{if } r2 \geq WEP \end{cases} \tag{5.13}$$

where $r3, r4$ *are random numbers* $\in [0, 1]$, $[lb_j, ub_j]$ defines the lower and upper bounds of the j^{th} object. The adapted MOMVO-based Node Selection algorithm is detailed in Algorithm 5.4. Each universe in the algorithm denotes a potential mapping. Thus the dimension of each universe is $|Evict\_List|$ and the j^{th} object value denotes the index of a potential node from $Feasible\_Node$ that can hold the j^{th} pod.

5.3 EXPERIMENTAL DESIGN AND SETUP

The proposed TIARM system was evaluated by developing a proof-of-concept using Golang and conducting experiments on the real-time GKE public Cloud platform. With TIARM, users can set different values for threshold (generally for CPU utilization) to dynamically trigger rescheduling operations.

Datacenter operators can interact with the k8s cluster using the *kubectl* CLI. Every operation on the cluster is communicated to the API Server component which authorizes any operations on the cluster and updates the persistent *etcd* store. All interactions with the cluster by entities such as the scheduler, pass through the API Server (refer Figure 5.6). In the proof-of-concept implementation, the rescheduler was incorporated into the scheduler pod and a descheduler pod was deployed to perform the container selection and eviction.

The TIARM System Monitoring agent works in conjunction with the open-source Prometheus monitoring tool to control the rescheduling activities.

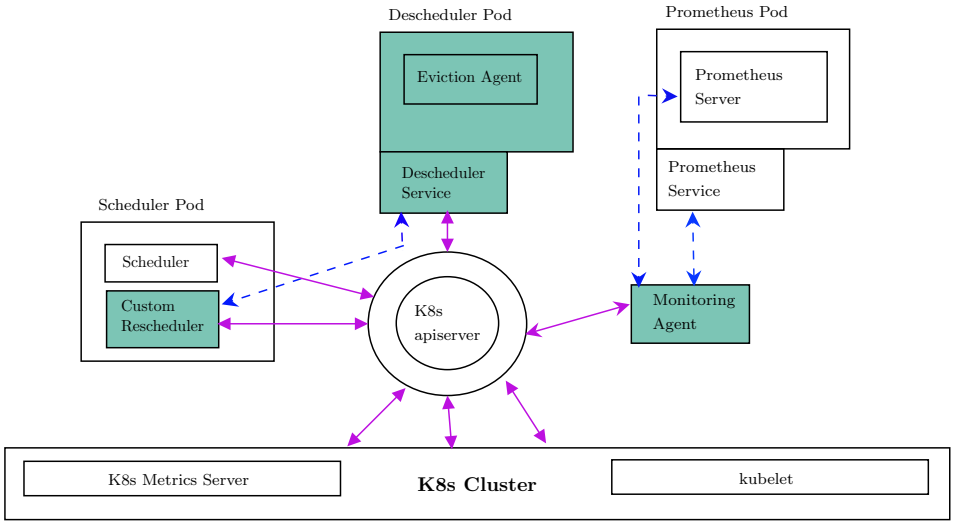


Figure 5.6: Communication between different entities in TIARM. All interactions to the cluster are directed to the API Server. The shaded components represent the extensions introduced in TIARM.

Type	vCPU	Memory
f1-micro	0.2 vCPU	0.60 GB memory
g1-small	0.5 vCPU	1.70 GB memory
n1-standard-1	1 vCPU	3.75 GB memory
n1-standard-2	2 vCPU	7.5 GB memory
n2-standard-2	2 vCPU	8 GB memory
n1-standard-4	4 vCPU	15 GB memory
n2-standard-4	4vCPU	16GB memory

Table 5.1: Configuration of VM instances

The GKE offers diverse types of VM instances. Some configurations of VM instances offered by the GCP are provided in Table 5.1. n1-standard-1 and n1-standard-2 instances were used in the experiments. K8s clusters consist of different node pools. VMs in each node pool are homogeneous, but they can vary from pool to pool. Each node VM instance runs the Container-Optimized Ubuntu OS image. Each VM instance has a boot disk size of 100 GB. The k8s cluster master node runs Docker version 1.12.8-gke.10.

The values of parameters used for the MOMVO parameters are provided in Table 5.2. The other parameters were assigned values as mentioned in the research by Mirjalili et al. (2017).

Parameter	Value
Number of Iterations	1000
Number of universes	50
Archive Size	200

Table 5.2: Parameter settings for MOMVO algorithm

5.3.1 Microservice Application Deployment

Three microservice web applications are deployed using containers to evaluate and validate the performance of the TIARM system.

The HipsterShop [1] is a 10-tier e-commerce web application which enables users to search items, put items into the cart and purchase the items. The HipsterShop application involves multiple microservice components frontend to enable user login, cartservice for managing and storing items in user carts, currencyservice to handle monetary conversions, shippingservice to estimate item shipping costs and etc. HipsterShop features a polyglot application with microservice components written in Go, Python, NodeJS, Java and C#.

Descartes Teastore is a reference microservice application (Eismann et al. 2018) that emulates an online store for selling tea and tea supplies. The application involves five microservice components all of which communicate with a registry microservice. Users are authorized by the Authentication Service. The WebService provides a UI and displays details of different products fetched from the Image and Persistence microservices. The application also includes a Recommender service to provide user recommendations.

Istio BookInfo is a sample microservice application consisting of four individual microservices [2]. The application emulates a cataloging web application where users can browse a database of books and related information along with their reviews. The productpage microservice collects information from the details and reviews services and displays them. Books rating information is provided by the ratings microservice. The review microservice is maintained in different versions ranging from one to three.

[1] https://github.com/GoogleCloudPlatform/microservices-demo
[2] https://istio.io/docs/examples/bookinfo/

5.3.2 Performance Metrics

The efficiency of the proposed system was evaluated by measuring two metrics: throughput and response time.

1. Throughput: The throughput of a system quantifies the useful amount of work done by the system. Microservice throughput is the number of service requests processed by the microservice per unit time. The throughput of a microservice is largely determined by several factors such as communication delays.

2. Response Time: The QoS of container orchestration systems for microservice-based applications is generally quantified using the average response time. User-perceived response time represents the time taken from the instant the user sends the request to the instant when the user receives a response from the microservice.

The experiments were designed to address the following questions:

- How do the values of the overload and underload thresholds affect the performance of the descheduling module?
- How do the weights in the weight vector affect the overall performance of the proposed system? What are the desirable weights to be used in the calculation of the objective value, $O.V$?
- Is a correlation-based Node-Selection approach recommended over anti-correlation based Node Selection approach?
- How does the proposed MOMVO-based re-scheduling approach perform when compared to other baseline approaches? The proposed Node Selection approach is compared with a heuristic and NSGA-II metaheuristic-based approaches.
- How does the proposed framework handle diverse microservice applications?

5.4 EXPERIMENTAL RESULTS AND ANALYSIS

Experimental data was generated for different workloads on real-time Cloud environments. The obtained results and detailed analysis is presented in this Section.

Target	Parameter	Value
Examine the impact of underload threshold	ULT	$\{20\%, 30\%, 40\%\}$
Examine the impact of overload threshold	OLT	$\{70\%, 80\%, 90\%\}$

Table 5.3: Experiment parameters

5.4.1 Impact of Upper and Lower threshold values

In order to examine and observe the impacts of the threshold values, different values were set for the upper and lower threshold (Piraghaj et al. 2015) (as shown in Table 5.3) that control the nodes which are subjected to rescheduling activities. The relative number of pods evicted were analyzed in each scenario.

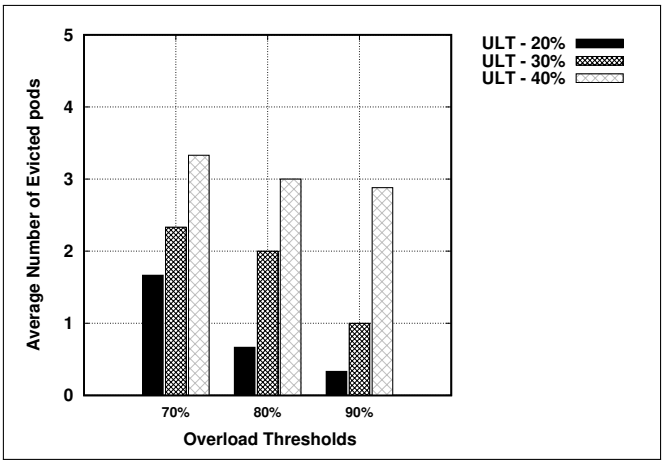


Figure 5.7: Performance Impacts of Underload Threshold (ULT) and Overload Threshold (OLT)

From Figure 5.7, it was observed that increasing the underload threshold value increases the probabilites of pods to be evicted. Higher values lead to more nodes being identified as 'underloaded' and pods from all these nodes are evicted, leading to more number of container migrations.

For higher overload threshold values, lesser number of nodes are subjected to the rescheduling process. On the other hand, it was also observed that when the values of the overload thresholds were high, more number of containers suffered from CPU throttling on the nodes that were excluded from the rescheduling process. This led to higher chances of QoS violations. The most efficient thresholds for TIARM are $\{80\%, 30\%\}$.

5.4.2 Impact of weight vector in the weighted sum objective value

The descheduler in TIARM uses an objective value (Equation 5.3) that involves a weight vector. For analyzing the impact of the weight vector, different combinations of w_1, w_2 were tested, as shown in Table 5.4. Figure 5.8 summarizes the response time and throughput obtained under different configurations. Figures 5.8a-5.8b depict the

Configuration	w_1	w_2
W1:W2->3:7	0.3	0.7
W1:W2->4:6	0.4	0.6
W1:W2->5:5	0.5	0.5
W1:W2->6:4	0.6	0.4
W1:W2->7:3	0.7	0.3

Table 5.4: Different combinations of w_1, w_2 in Equation 5.3

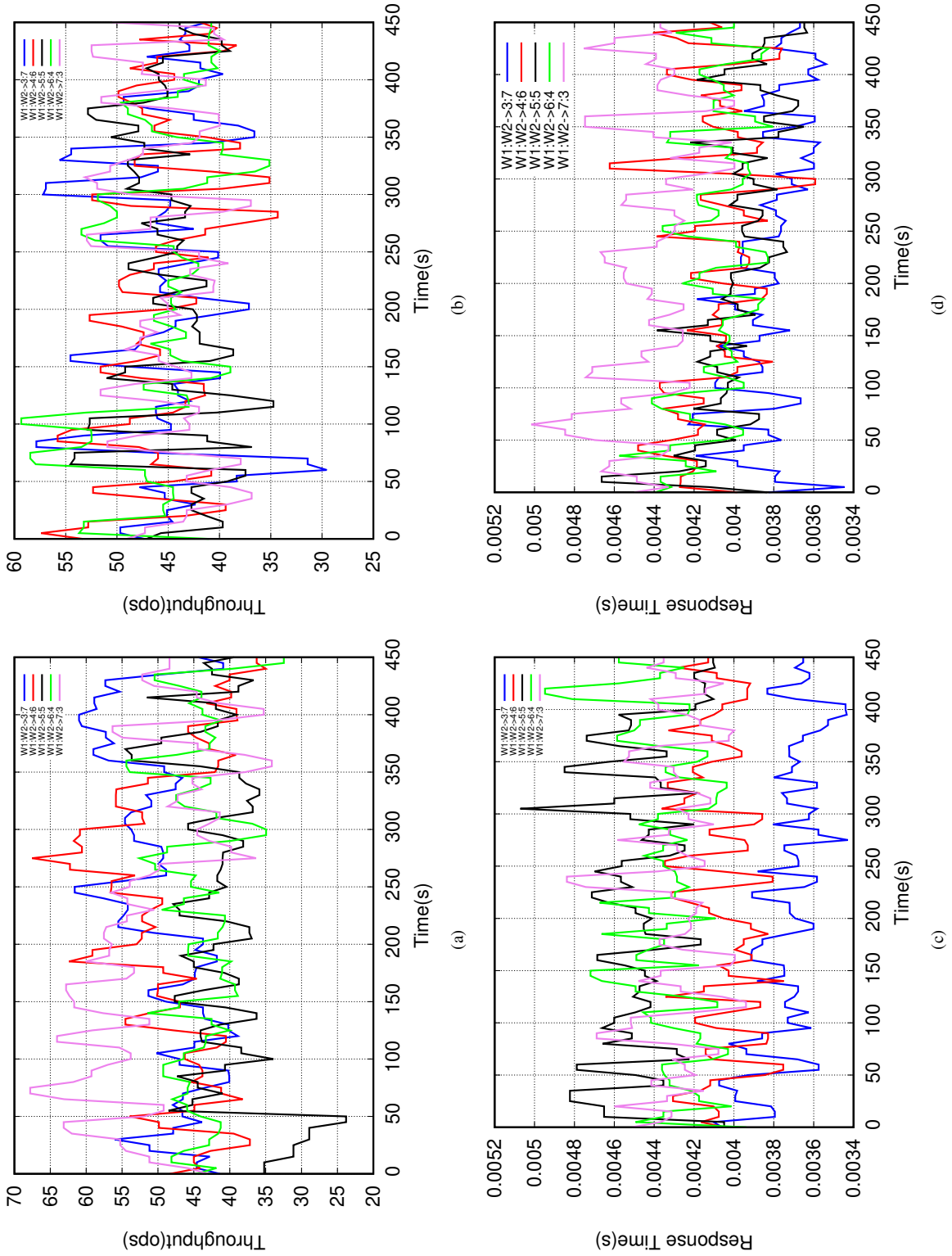


Figure 5.8: Comparison of throughput and response time for varying weight vector values

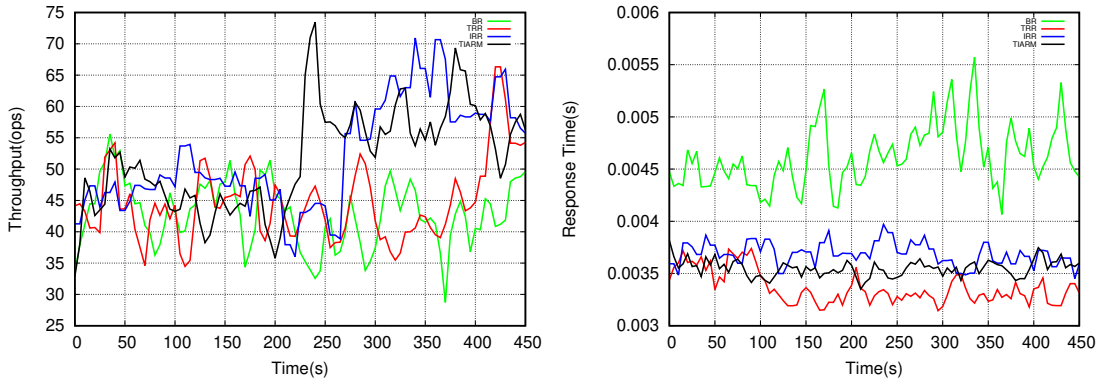
throughput values before and after rescheduling strategy is effected. For the values of $w_1 = 0.3$, the CPU throttling value increases after rescheduling. In all other cases the throttling decreases, with the highest decrease observed when $w_1 = 0.7$ and $w_2 = 0.3$. However, as interacting entities are spread across different nodes, more time is spent in communication leading to an increase in the average response time as inferred from Figures 5.8c and 5.8d. When $w_1 = 0.4$ and $w_2 = 0.6$, the throttling decreases, but the changes in response time and throughput are relatively smaller. At $w_1 = 0.5$ and $w_2 = 0.5$, the response time decreases by 10.7%, accompanied by a 9.21% increase in the throughput, implying that both the throttling level and interaction factor values contribute equally to the determination of the application response time.

5.4.3 Analysis of varying rescheduling strategies

To reveal the performance effectiveness of TIARM, three baseline algorithms were adopted as follows:

- Bare Rescheduler (BR): This rescheduler neglects any rescheduling actions and does not employ any container migrations. This can be considered to be equivalent to a system with no rescheduling features.
- Throttling-based Random Rescheduler (TRR): This rescheduling system performs container selection based on the CPU Throttling values and ignores the interaction-factor values. Evicted pods are then placed on randomly selected node instances.
- Interaction-aware Random Rescheduler (IRR): IRR performs container selection based on the Interaction-factor values without any regard for the CPU throttling values. In the next phase, the evicted pods are placed on nodes selected at random from the feasible set of nodes.

TRR and IRR are variants of TIARM that vary in the container selection and node selection techniques applied. The rescheduling strategies were varied and compared with the proposed TIARM strategy. Figures 5.9a and 5.9b provide the throughput and response time respectively. An inference of primary importance is that applying any rescheduling strategy improves the performance attributes, when compared to a system



(a) Comparison of application throughput for varying rescheduling strategies

(b) Comparison of application response time for varying rescheduling strategies

Figure 5.9: Performance comparison of BR, IRR, TRR and TIARM rescheduling strategies

with no support for re-scheduling. It is observed that the TRR, IRR and TIARM policies result in 3.3%, 8.5% and 10.7% decrease in the response time, respectively. However, the IRR policy increases the throughput only by 5.7%, whereas the TIARM increases the throughput by 9.2%.

5.4.4 Performance comparison of anti-correlated workloads and correlated workloads

TIARM designates migrating containers to nodes with resource utilization that are highly anti-correlated to that of the containers. An alternative technique prefers nodes that are highly co-related. To analyze and justify the use of anti-correlation, migrating containers were first placed on highly co-related nodes and then on anti-corelated nodes, while observing the number of pods that go into 'Pending' State. This state of k8s pods indicate that no suitable node was identified to run the container and thus waits till the next node becomes available.

It is clearly observed from Figure 5.10, that directing pods to the nodes with highest correlation starves the dominant resources available on the nodes, leading to more pods stuck in the 'Pending' state. Placing pods on nodes that are anti-correlated ensures that the resource needs of the pods are complementary to that of the nodes, thereby increasing the resource utilization of the nodes in the system.

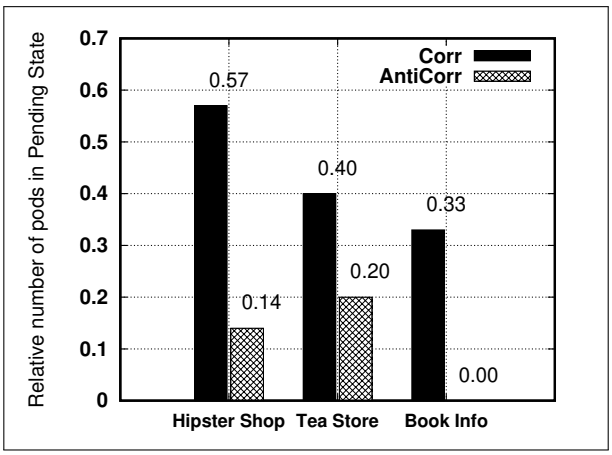


Figure 5.10: Comparison of 'Pending' pods using the anti-correlated and co-related strategy

5.4.5 Analysis of varying node selection strategies

Non-Dominated Sorting Algorithm (NSGA-II) (Deb et al. 2002) is a well-known meta-heuristic for multi-objective optimization. The node selection strategy in TIARM was varied to use NSGA-II (represented as NSGA2), instead of the MOMVO. Another node selection strategy, AntiCorr, that tried to minimize only Anti-correlation was also implemented and evaluated.

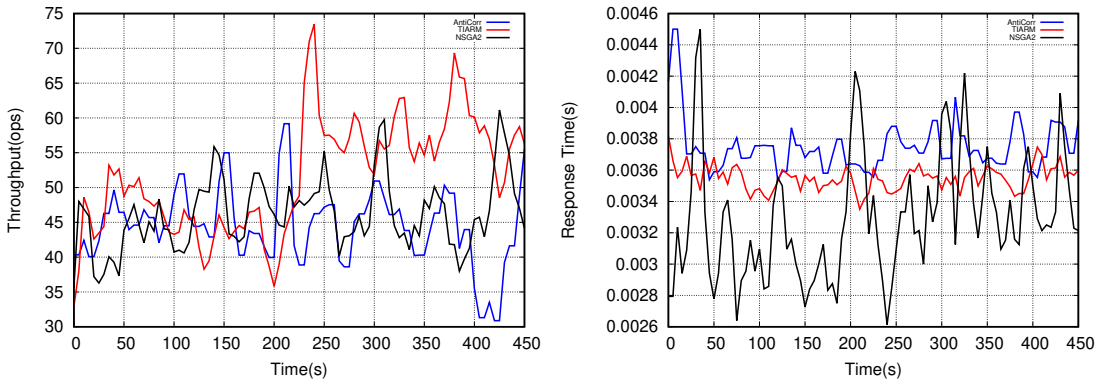


Figure 5.11: Analysis of different Node Selection Strategies

Figure 5.11 summarizes the performance results of the different node selection strategies. The AntiCorr policy has an adverse effect, by decreasing the throughput of the system. Both NSGA2 and MOMVO strategies improve the performance of the system. The NSGA2 decreases the response time by 11% and increases the throughput by 5.8%. Owing to the better diversity of solutions generated by the MOMVO, TIARM

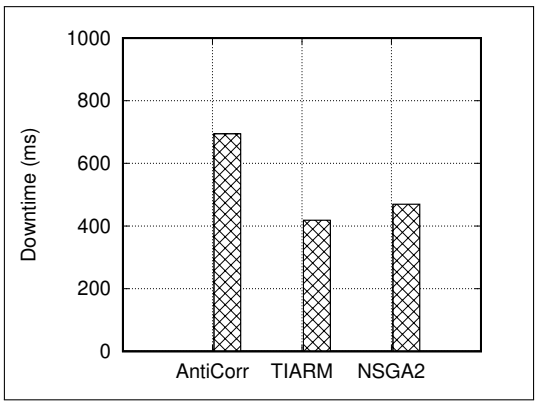


Figure 5.12: Downtime values for varying node selection Strategies

performs better, decreasing the response time by 13.9% and increasing the throughput by 16.3%. The average values of downtime obtained using the dfferent Node Selection strategies is illustrated in Figure 5.12. The variation of the objective function values across the generations is depicted in Figure 5.13.

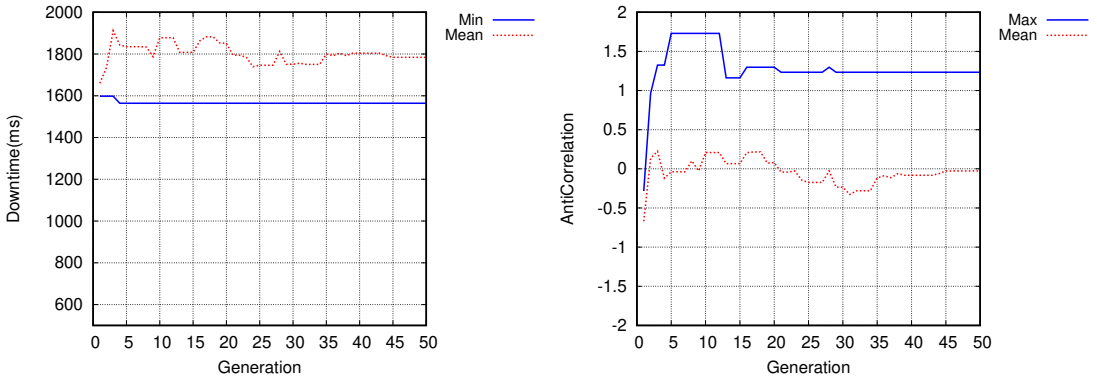


Figure 5.13: Values of the two objective functions across different generations for MOMVO

5.4.6 Efficient Resource Management for various microservice applications

In order to analyze the efficiency of the TIARM system, different real-world microservice applications, discussed in Section 5.3.1 were deployed and subjected to rescheduling. Figure 5.14 illustrates the performance of the system where HipsterShop microservice application is deployed. Figures 5.15 and 5.16 depicts the scenario where TIARM re-schedules different components of the Istio BookInfo and Descartes Teastore microservice applications respectively. In all scenarios, TIARM reduces the response time while slightly enhancing the application throughput.

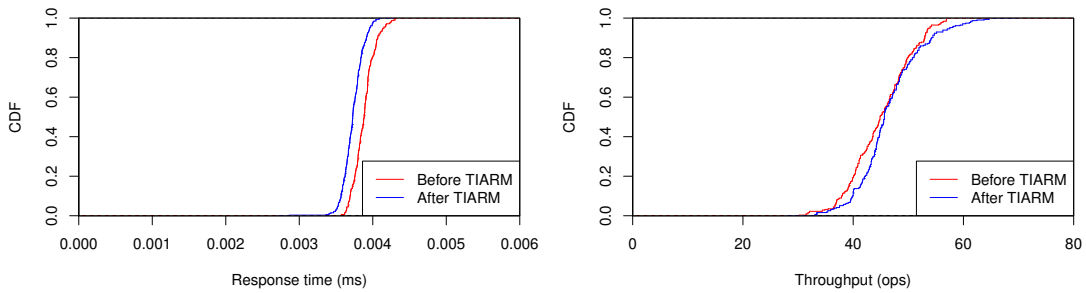


Figure 5.14: CDF of Throughput and Response Time for HipsterShop microservice application

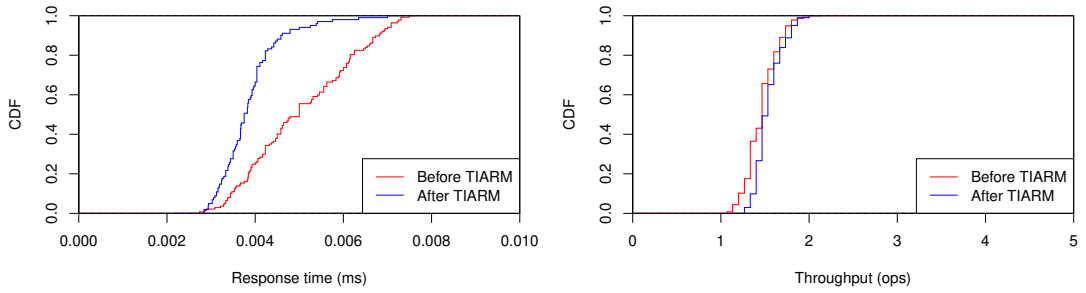
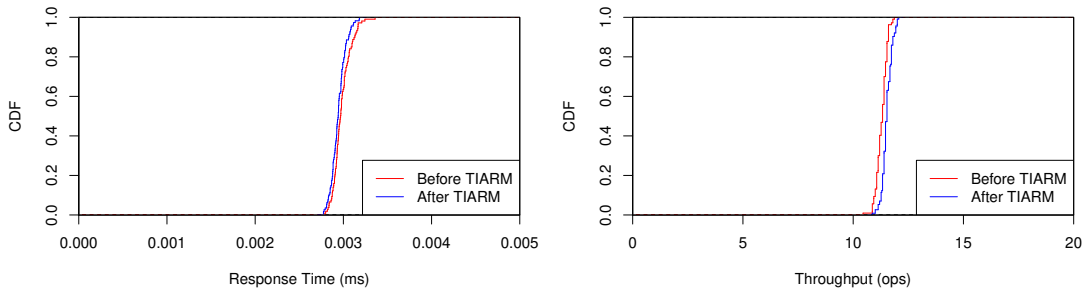


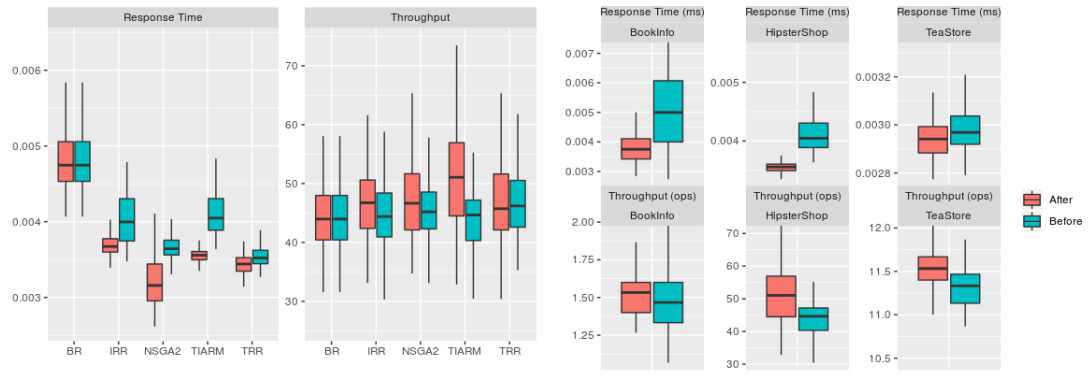
Figure 5.15: CDF of Throughput and Response Time for BookInfo microservice application



(a) (b)

Figure 5.16: CDF of Throughput and Response Time for TeaStore microservice application

The different experimental results are summarized in Figures 5.17a and 5.17b. Figure 5.17a compares the different re-scheduling strategies against the TIARM and Figure 5.17b presents the TIARM performance when different microservice applications are



(a) Throughput and Response Time Comparison for different rescheduling strategies

(b) Throughput and Response Time Comparison for different microservice applications using TIARM

Figure 5.17: Summary of Performance Analysis

considered. It is inferred that significantly better performance is obtained in the case of TIARM.

5.5 SUMMARY

The inherently dynamic microservice ecosystems entail adaptations at runtime. One technique that can be employed for this is microservice re-scheduling. Existing works target re-scheduling in hypervisor-based systems, while ignoring the influence of configuration parameters of container-based microservices. In an effort to address these challenges, in this chapter, a novel microservice re-scheduling framework, TIARM, is presented. TIARM proactively performs re-scheduling activities whilst ensuring timely service responses. Based on periodic monitoring of the performance attributes, the framework schedules container migrations. Considering the exponentially large solution space, a metaheuristic approach based on Multi Verse Optimization is developed to generate the near-optimal mapping of microservices to the datacenter resources. Experimental results indicate that our framework provides superior performance with a reduction of upto 13.97% in the average response time, when compared to systems with no support for re-scheduling.

CHAPTER 6

CONCLUSIONS AND FUTURE SCOPE

Distributed Cloud environments are now resorting to Cloud applications composed of heterogeneous microservices. This transition to microservices introduces a wide range of infrastructural orchestration challenges. In this research work, two orchestration challenges, namely microservice allocation and microservice re-scheduling have been addressed.

A robust interaction-aware deployment strategy, called IntMA, to deploy the microservice application modules in a balanced manner is presented in this research. The deployment strategy ensures that interactions between the physical nodes are kept to the minimum, thereby ensuring that the inter-node communication latencies do not adversely impact the user response time and throughput. Instead of considering the dependencies, the run-time interaction pattern is captured to identify the frequently interacting entities. The microservice components which frequently interact with each other are considered for possible deployment on the same node. The NP-hard microservice allocation problem is also modeled as a binary QPP. The effectiveness of the proposed approach is evaluated on the Google Cloud Platform, using different microservice reference applications. Experimental results indicate that the proposed approach improves the response time and throughput of the microservice-based systems.

The dynamism of microservice ecosystems necessitates runtime adaptations and microservices re-scheduling to avoid performance degradation. In this research work, the relationship between the container configuration metrics and application performance

is investigated. Based on this relationship, a re-scheduling framework, TIARM, that encompasses policies for the different phases, namely host node identification, container selection and destination node selection, is presented. The framework incorporates a component that performs periodic monitoring and triggers re-scheduling activities based on threshold-based rules. The de-scheduling phase first selects the containers for migration based on a Multi-Criteria Decision Making (MCDM) method and terminates them on the current node. The re- scheduling phase includes an initial stage where the containers are resized according to their previous resource usage status and re-deployed onto nodes selected by the MOMVO-based selection strategy. Extensive experiments were conducted to examine the practical feasibility of the proposed framework in real Cloud environments. The results indicate that adopting the re-scheduling approach succeeds in reducing the average response time by upto 13.9%.

6.1 FUTURE RESEARCH DIRECTIONS

In this section, an insight into the promising pathways for future research in resource orchestration of containerized environments is provided.

6.1.1 Augmenting with autonomic capabilities

In order to ensure dynamic resource provisioning that can handle unprecedented fluctuations in Cloud environments, resource orchestration frameworks must be augmented with autonomic capabilities. Techniques based on the MAPE control loop (Martin et al. 2020) may be integrated in resource orchestration frameworks to facilitate adaptive control over the managed resources. Autonomic loops can also be used to realise autoscaling and elasticity in containerized environments.

6.1.2 Exploring the impact of resource heterogeneity

Cloud computing environments generally offer diverse resources that vary in their attributes. The resources may differ at the infrastructure and/or the platform level. In order to ensure resource usage efficiency, the multiplicities and heterogeneities of resources in the Cloud environments must be considered. Additional resources such as network and storage, must be considered to devise resource orchestration strategies that

are more comprehensive. For instance, the diversity in container engines supported by the nodes may restrict the potential nodes that are available for the re-scheduling phase.

6.1.3 Incorporating Machine Learning Techniques

The allocation and re-scheduling strategies may be enhanced by adopting machine learning techniques. For instance, microservice requests may be clustered using clustering algorithms to determine sets of requests to be placed on similar sets of nodes. Additionally, the use of machine learning algorithms to capture the microservice interaction pattern and the impacts of container configuration parameters on the QoS values must also be investigated.

6.1.4 Investigating additional optimization goals

Resource orchestration frameworks may further be optimized for energy-efficiency and other additional QoS parameters. Some of the possible optimization goals are as follows:

6.1.4.1 Energy efficiency

Resource orchestration strategies may be devised to optimize the energy efficiency of multi-container systems. Various techniques such as the brownout paradigm (Xu and Buyya 2019) can be adopted to improve the energy efficiency. The brownout approach involves temporarily disabling the optional components of a system thus ensuring that the system is responsive under varying workload conditions. The overall activities of a brownout-based system are controlled by a control knob called the dimmer. Further, the integration of resource allocation strategies with energy-aware features must be studied.

6.1.4.2 Cost models

Cloud resource pricing is an important concern for Cloud providers. Transparent cost models enable providers to estimate the actual cost of the Cloud resources (Sharma et al. 2014). Cost models must be able to quantify the monetary value of execution. In this context, the efficiency of finance-based models inspired by the option pricing theory must be investigated. Cloud pricing strategies devised must satisfy customer service requirements while also ensuring profit gains for the providers.

6.1.4.3 Security features

A limitation of the container-based virtualization systems is the reduction in the level of security of the system (Fernandez and Brito 2019). To overcome this, resource orchestration mechanisms may be integrated with additional mechanisms such as cryptographic algorithms and access control mechanisms. Other mechanisms may include encrypting the image layer and securing the communications in the network (Casalicchio and Iannucci 2020). Software Guard Extensions (SGX) enabled containers (Vaucher et al. 2018) may also be employed to enhance the level of security in Cloud environments.

6.1.5 Integration with Serverless Computing and other emerging distributed computing environments

The features of microservice architectures deem it suitable for developing IoT applications. The micoservice IoT based applications may be deployed in lightweight environments such as the Fog and Edge. Resource orchestration strategies must be customized to better suit the needs of the distributed environment in which the microservices are deployed. Serverless computing paradigms such as Function as a Service (FaaS), also deploy applications designed as microservices. To better harness the benefits of FaaS platforms, advanced resource orchestration strategies that can overcome the perfomance isolation challenges must be developed.

6.1.6 Integration with Blockchain technology

Blockchain systems employ consensus mechanisms to ensure access control and security. Containerized environments may leverage the facilities of blockchain systems to enhance the level of security. Smart contracts may also be implemented as microservices that interact with each other through API gateways (Tonelli et al. 2019, 2018). The blockchain may also be used to support decentralized resource orchestration in microservice environments. For instance, Xu et al. (2017) adopted the blockchain to perform resource management in Cloud environments. Each block maintains details of the resources allocated to each user. This approach may be further extended and adapted to tackle the complexities of microservice-based systems.

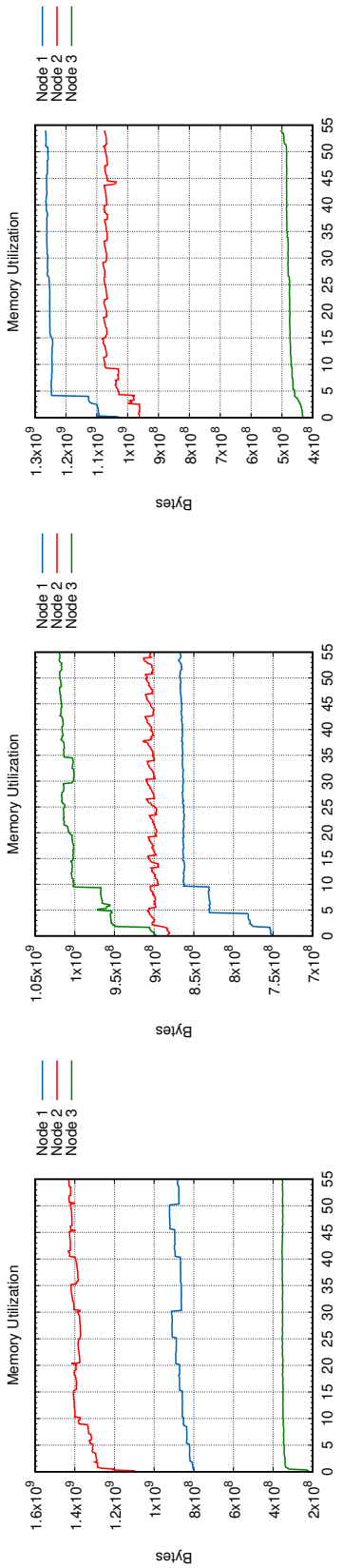
Appendices

Appendix A

RESOURCE USAGE ANALYSIS FOR WORKLOAD MICROSERVICE APPLICATIONS DEPLOYED USING DIFFERENT SCHEDULING POLICIES

In order to study the impact of the different scheduling policies on the Cloud data center resource utilization level, the CPU and memory usage of the different workload microservice applications were studied.

Sock Shop application and Istio BookInfo application are memory-intensive applications. So, we analyzed the memory utilization across nodes for the three different deployment policies. Figures A.1a, A.1b and A.1c give the memory usage per-node for the default scheduler, IntRR scheduler and the IntMA scheduler, respectively. It is observed that the memory load is better balanced when the application is deployed using IntMA. For the large microservice application, IntRR also improves the overall average memory utilization. For the Istio BookInfo application, the memory usage across nodes is given in Figures A.2a, A.2b and A.2c corresponding to the the default scheduler, IntRR scheduler and the IntMA scheduler, respectively. The IntMA scheduler exhibits similar trends by attaining a better balanced distribution of the memory load, thereby improving the average memory utilization. Hipster Shop application is a CPU-intensive application. Hence, we analyzed the CPU usage across the nodes for the default, IntRR and IntMA scheduler in Figures A.3a, A.3b and A.3c respectively. The CPU load is more equally distributed across the nodes in the IntMA and IntRR scheduler cases.

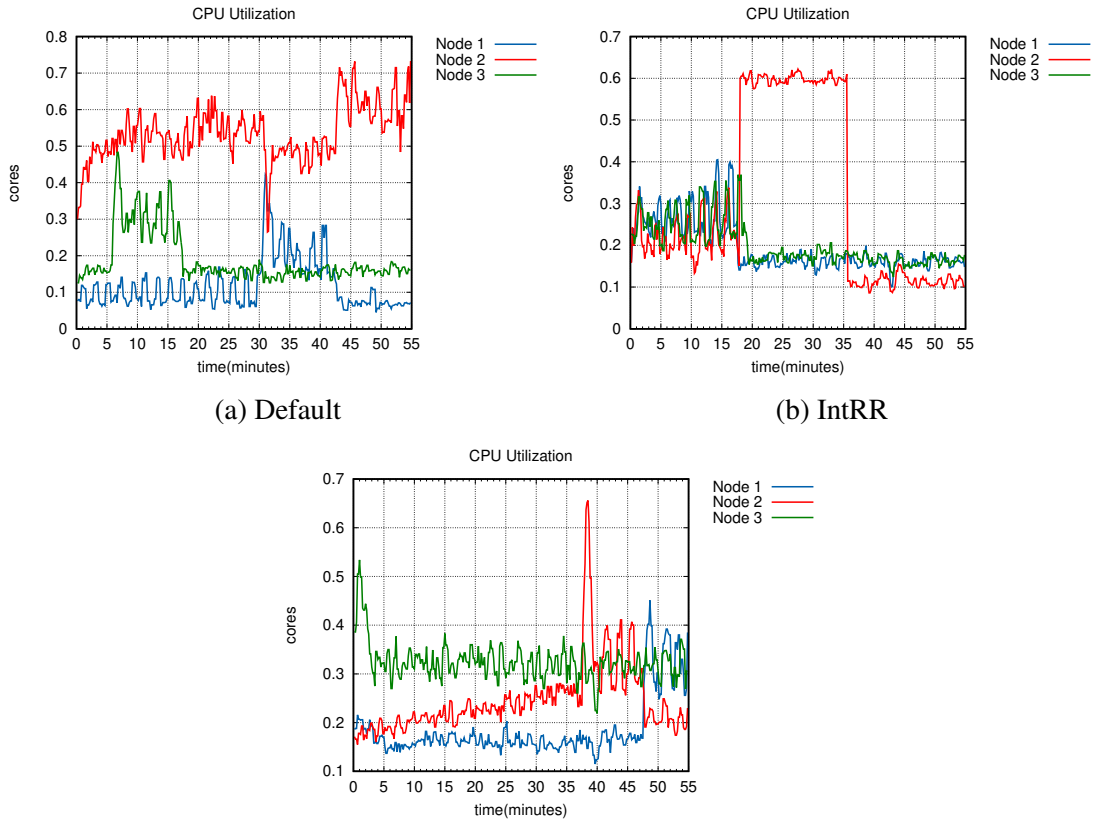


(a) Default (b) IntRR (c) IntMA

Figure A.1: Memory usage values for Sock Shop microservice application

(a) Default (b) IntRR (c) IntMA

Figure A.2: Memory usage values for Istio Book Info microservice application



(a) Default

(b) IntRR

(c) IntMA

Figure A.3: CPU usage values for Hipster Shop microservice application

Thus, it is concluded that adopting the proposed scheduling algorithm does not impose much additional overheads in terms of resource usage on the various nodes in the Cloud environment.

BIBLIOGRAPHY

Abeysinghe, A. (2016). "Scope versus size: a pragmatic approach to microservice architecture." Technical report, Solutions Architecture, WSO2.

Adhikari, M. and Srirama, S. N. (2019). "Multi-objective accelerated particle swarm optimization with a container-based scheduling for internet-of-things in cloud environment." *Journal of Network and Computer Applications*, 137, 35–61.

Almeida, W. H. C., de Aguiar Monteiro, L., Hazin, R. R., de Lima, A. C. and Ferraz, F. S. (2017). "Survey on microservice architecture-security, privacy and standardization on cloud computing environment." *ICSEA 2017*, 210.

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

Asik, T. and Selcuk, Y. E. (2017). "Policy enforcement upon software based on microservice architecture." In *Software Engineering Research, Management and Applications (SERA), 2017 IEEE 15th International Conference on*, IEEE, 283–287.

Bakshi, K. (2017). "Microservices-based software architecture and approaches." In *Aerospace Conference, 2017 IEEE*, IEEE, 1–8.

Balalaie, A., Heydarnoori, A. and Jamshidi, P. (2014). "On micro-services architecture." *International Journal of Open Information Technologies*, 2(9), 24–27.

Baylov, K. and Dimov, A. (2017). "Reference architecture for self-adaptive microser-

vice systems." In *International Symposium on Intelligent and Distributed Computing*, Springer, 297–303.

Beloglazov, A. and Buyya, R. (2012). "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers." *Concurrency and Computation: Practice and Experience*, 24(13), 1397–1420.

Bermejo, B., Juiz, C. and Guerrero, C. (2019). "Virtualization and consolidation: a systematic review of the past 10 years of research on energy and performance." *The Journal of Supercomputing*, 75(2), 808–836.

Bittencourt, L. F., Goldman, A., Madeira, E. R., da Fonseca, N. L. and Sakellariou, R. (2018). "Scheduling in distributed systems: A cloud computing perspective." *Computer science review*, 30, 31–54.

Bogner, J., Wagner, S. and Zimmermann, A. (2017). "Automatically measuring the maintainability of service-and microservice-based systems: a literature review." In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, ACM, 107–115.

Bogner, J., Zimmermann, A. and Wagner, S. (2018). "Analyzing the relevance of soa patterns for microservice-based systems." In *Proceedings of the 10th Central European Workshop on Services and their Composition (ZEUS'18). CEUR-WS. org*.

Bonér, J., Farley, D., Kuhn, R. and Thompson, M. (2014). "The reactive manifesto." `https://www.reactivemanifesto.org/` (15 May, 2020).

Buyya, R., Rodriguez, M. A., Toosi, A. N. and Park, J. (2018). "Cost-efficient orchestration of containers in clouds: A vision, architectural elements, and future directions." *Journal of Physics: Conference Series*, 1108(1), 012001.

Buzachis, A., Galletta, A., Celesti, A., Carnevale, L. and Villari, M. (2019). "Towards osmotic computing: a blue-green strategy for the fast re-deployment of microser-

vices." In *2019 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 1–6.

Caprara, A. (2008). "Constrained 0–1 quadratic programming: Basic approaches and extensions." *European Journal of Operational Research*, 187(3), 1494–1503.

Carnevale, L., Galletta, A., Celesti, A., Fazio, M., Paone, M., Bramanti, P. and Villari, M. (2017). "Big data HIS of the IRCCS-ME future: The osmotic computing infrastructure." In *Cloud Infrastructures, Services, and IoT Systems for Smart Cities*, Springer, 199–207.

Casalicchio, E. and Iannucci, S. (2020). "The state-of-the-art in container technologies: Application, orchestration and security." *Concurrency and Computation: Practice and Experience*, e5668.

Cito, J., Leitner, P., Fritz, T. and Gall, H. C. (2015). "The making of cloud applications: An empirical study on software development for the cloud." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 393–403.

de Camargo, A., Salvadori, I., Mello, R. d. S. and Siqueira, F. (2016). "An architecture to automate performance tests on microservices." In *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*, ACM, 422–429.

de Santana, C. J. L., de Mello Alencar, B. and Prazeres, C. V. S. (2019). "Reactive microservices for the internet of things: a case study in fog computing." In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 1243–1251.

Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T. (2002). "A fast and elitist multiobjective genetic algorithm: Nsga-ii." *IEEE transactions on evolutionary computation*, 6(2), 182–197.

Derakhshanmanesh, M. and Grieger, M. (2016). "Model-integrating microservices: A vision paper." In *Software Engineering (Workshops)*, 142–147.

Di Francesco, P. (2017). "Architecting microservices." In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, IEEE, 224–229.

Diepenbrock, A., Rademacher, F. and Sachweh, S. (2017). "An ontology-based approach for domain-driven design of microservice architectures." *INFORMATIK 2017*.

Docker, I. (2020). "Docker: Empowering app development for developers." `https://www.docker.com/` (1 May, 2020).

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2017). "Microservices: yesterday, today, and tomorrow." *Present and ulterior software engineering*, 195–216.

Düllmann, T. F. (2017). "Performance anomaly detection in microservice architectures under continuous change." Master's thesis, University of Stuttgart.

Düllmann, T. F. and van Hoorn, A. (2017). "Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches." In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ACM, 171–172.

Eismann, S., Kistowski, J., Grohmann, J., Bauer, A., Schmitt, N., Herbst, N. and Kounev, S. (2018). "Teastore: A micro-service reference application for cloud researchers." In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 11–12.

Esposito, C., Castiglione, A. and Choo, K.-K. R. (2016). "Challenges in delivering software in the cloud as microservices." *IEEE Cloud Computing*, 3(5), 10–14.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*, Addison-Wesley Professional.

Familiar, B. (2015). *Microservices, IoT and Azure: Leveraging DevOps and Microservice Architecture to deliver SaaS Solutions*, Apress.

Fano, A. and Gershman, A. (2002). "The future of business services in the age of ubiquitous computing." *Communications of the ACM*, 45(12), 83–87.

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

Fernandez, G. P. and Brito, A. (2019). "Secure container orchestration in the cloud: policies and implementation." In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 138–145.

Fernández Villamor, J. I., Iglesias Fernandez, C. A. and Garijo Ayestaran, M. (2010). "Microservices: lightweight service descriptions for rest architectural style." In *2nd International Conference on Agents and Artificial Intelligence, ICAART 2010*, INSTICC, Institute for Systems and Technologies of Information, Control and Communication.

Filip, I.-D., Pop, F., Serbanescu, C. and Choi, C. (2018). "Microservices scheduling model over heterogeneous cloud-edge environments as support for iot applications." *IEEE Internet of Things Journal*, 5(4), 2672–2681.

Florio, L. (2015). "Decentralized self-adaptation in large-scale distributed systems." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 1022–1025.

Fritzsch, J., Bogner, J., Wagner, S. and Zimmermann, A. (2019). "Microservices migration in industry: intentions, strategies, and challenges." In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 481–490.

Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J. and Montesi, F. (2016). "Self-reconfiguring microservices." In *Theory and Practice of Formal Methods*, Springer, 194–210.

Gadea, C., Trifan, M., Ionescu, D. and Ionescu, B. (2016). "A reference architecture for real-time microservice api consumption." In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, ACM, 1–6.

Geisriegler, M., Kolodiy, M., Stani, S. and Singer, R. (2017). "Actor based business process modeling and execution: A reference implementation based on ontology models

and microservices." In *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on*, IEEE, 359–362.

George, F. (2013). "Microservice architecture: A personal journey of discovery." `http://www.baselinemag.com/enterprise-apps/walmart-embraces-microservices-to-get-more-agile.html` (15 May, 2020).

Github, I. (2020). "Sample cloud-native application with 10 microservices showcasing kubernetes, istio, grpc and opencensus." `https://github.com/GoogleCloudPlatform/microservices-demo` (1 May, 2020).

Goldschmidt, T., Hauck-Stattelmann, S., Malakuti, S. and Grüner, S. (2018). "Container-based architecture for flexible industrial control applications." *Journal of Systems Architecture*, 84, 28–36.

Gonzalez, N. M., de Brito Carvalho, T. C. M. and Miers, C. C. (2017). "Cloud resource management: towards efficient execution of large-scale scientific applications and workflows on complex infrastructures." *Journal of Cloud Computing*, 6(1), 13.

Google, I. (2020). "Google cloud platform: Cloud computing services." `https://cloud.google.com/` (1 May, 2020).

Grafana, A. (2020). "Grafana-the open platform for analytics and monitoring." `https://grafana.com/` (1 May, 2020).

Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L. and Di Salle, A. (2017a). "Microart: A software architecture recovery tool for maintaining microservice-based systems." In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, IEEE, 298–302.

Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L. and Di Salle, A. (2017b). "Towards recovering the software architecture of microservice-based systems." In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, IEEE, 46–53.

Guerrero, C., Lera, I. and Juiz, C. (2018a). "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture." *Journal of Grid Computing*, 16(1), 113–135.

Guerrero, C., Lera, I. and Juiz, C. (2018b). "Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications." *The Journal of Supercomputing*, 74(7), 2956–2983.

Harms, H., Rogowski, C. and Lo Iacono, L. (2017). "Guidelines for adopting frontend architectures and patterns in microservices-based systems." In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 902–907.

Haselböck, S. and Weinreich, R. (2017). "Decision guidance models for microservice monitoring." In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, IEEE, 54–61.

Hassan, S., Ali, N. and Bahsoon, R. (2017). "Microservice ambients: An architectural meta-modelling approach for microservice granularity." In *Software Architecture (ICSA), 2017 IEEE International Conference on*, IEEE, 1–10.

Hassan, S., Bahsoon, R. and Kazman, R. (2020). "Microservice transition and its granularity problem: A systematic mapping study." *Software: Practice and Experience*, 50(9), 1651–1681.

Hedengren, J. D., Shishavan, R. A., Powell, K. M. and Edgar, T. F. (2014). "Nonlinear modeling, estimation and predictive control in apmonitor." *Computers & Chemical Engineering*, 70, 133–148.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., Schulte, S. and Wettinger, J. (2017). "Performance engineering for microservices: Research challenges and directions." In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ACM, 223–226.

Henry, A. and Ridene, Y. (2020). "Assessing your microservice migration." In *Microservices*, Springer, 73–107.

Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K. and Sekar, V. (2016). "Gremlin: systematic resilience testing of microservices." In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, IEEE, 57–66.

Hevner, A. R., March, S. T., Park, J. and Ram, S. (2004). "Design science in information systems research." *MIS quarterly*, 75–105.

Heyman, J., Byström, C., Hamrén, J. and Heyman, H. (2020). "An open source load testing tool.." `https://locust.io/` (1 May, 2020).

Hilman, M. H., Rodriguez, M. A. and Buyya, R. (2020). "Multiple workflows scheduling in multi-tenant distributed systems: A taxonomy and future directions." *ACM Computing Surveys (CSUR)*, 53(1), 1–39.

HoseinyFarahabady, M., Lee, Y. C., Zomaya, A. Y. and Tari, Z. (2017). "A qos-aware resource allocation controller for function as a service (faas) platform." In *International Conference on Service-Oriented Computing*, Springer, 241–255.

Istio, A. (2020). "Bookinfo application." `https://istio.io/docs/examples/bookinfo/` (1 May, 2020).

JaegerTracing (2020). "Jaeger: open source, end-to-end distributed tracing." `https://www.jaegertracing.io/` (1 May, 2020).

Kaewkasi, C. and Chuenmuneewong, K. (2017). "Improvement of container scheduling for docker using ant colony optimization." In *Knowledge and Smart Technology (KST), 2017 9th International Conference on*, IEEE, 254–259.

Kampars, J. and Pinka, K. (2017). "Auto-scaling and adjustment platform for cloud-based systems." In *Proceedings of the 11th International Scientific and Practical Conference. Volume II*, volume 52, 57.

Kang, D.-K., Choi, G.-B., Kim, S.-H., Hwang, I.-S. and Youn, C.-H. (2016a). "Workload-aware resource management for energy efficient heterogeneous docker containers." In *Region 10 Conference (TENCON), 2016 IEEE*, IEEE, 2428–2431.

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

Karhula, P., Janak, J. and Schulzrinne, H. (2019). "Checkpointing and migration of iot edge functions." In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, ACM, 60–65.

Karwowski, W., Rusek, M., Dwornicki, G. and Orłowski, A. (2017). "Swarm based system for management of containerized microservices in a cloud consisting of heterogeneous servers." In *International Conference on Information Systems Architecture and Technology*, Springer, 262–271.

Katie, C. (2019). "Gartner projects cloud services industry to grow exponentially through 2022." `https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g` (15 May, 2020).

Katuwal, K. (2016). "Microservices: A flexible architecture for the digital age version 1.0." In *American Journal of Computer Science and Engineering*, Open Science, 20–24.

Khan, A. (2017). "Key characteristics of a container orchestration platform to enable a modern application." *IEEE Cloud Computing*, 4(5), 42–48.

Khan, M. A., Paplinski, A., Khan, A. M., Murshed, M. and Buyya, R. (2018). "Dynamic virtual machine consolidation algorithms for energy-efficient cloud resource management: a review." In *Sustainable Cloud and Energy Services*, Springer, 135–165.

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

Kiss, T., Kacsuk, P., Kovacs, J., Rakoczi, B., Hajnal, A., Farkas, A., Gesmier, G. and Terstyanszky, G. (2017). "Micado-microservice-based cloud application-level dynamic orchestrator." *Future Generation Computer Systems*, 1–10.

Kleehaus, M., Uludag, O., Schäfer, P. and Matthes, F. (2018). "MICROLYZE: A framework for recovering the software architecture in microservice-based environments." In *30th International Conference on Advanced Information Systems Engineering (CAISE Forum), Tallin, Estonia.*

Klock, S., Van Der Werf, J. M. E., Guelen, J. P. and Jansen, S. (2017). "Workload-based clustering of coherent feature sets in microservice architectures." In *Software Architecture (ICSA), 2017 IEEE International Conference on*, IEEE, 11–20.

Kratzke, N. and Quint, P.-C. (2017). "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study." *Journal of Systems and Software*, 126, 1–16.

Krauter, K., Buyya, R. and Maheswaran, M. (2002). "A taxonomy and survey of grid resource management systems for distributed computing." *Software: Practice and Experience*, 32(2), 135–164.

Kubernetes, A. (2020a). "Kubernetes: Production-grade container orchestration." `https://kubernetes.io/` (1 May, 2020a).

Kubernetes, A. (2020b). "Pod lifecycle." `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/` (1 May, 2020b).

Kukade, P. P. and Kale, G. (2015). "Auto-scaling of micro-services using containerization." *International Journal of Science and Research (IJSR)*, 4(9), 1960–1963.

Lange, P. D., Nicolaescu, P., Derntl, M., Jarke, M. and Klamma, R. (2016). "Community application editor: collaborative near real-time modeling and composition of microservice-based web applications." In *Lecture Notes in Informatics- Modellierung 2016-Workshopband*, Gesellschaft für Informatik eV, 123–127.

Lera, I., Guerrero, C. and Juiz, C. (2018). "Availability-aware service placement policy in fog computing based on graph partitions." *IEEE Internet of Things Journal*, 6(2), 3641–3651.

Lewis, J. (2012). "Micro services - java, the unix way." `http://2012.33degree.org/talk/show/67` (15 May, 2020).

Lin, M., Xi, J., Bai, W. and Wu, J. (2019). "Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud." *IEEE Access*, 7, 83088–83100.

Long, K. B., Yang, H. and Kim, Y. (2017). "Icn-based service discovery mechanism for microservice architecture." In *Ubiquitous and Future Networks (ICUFN), 2017 Ninth International Conference on*, IEEE, 773–775.

López, M. R. and Spillner, J. (2017). "Towards quantifiable boundaries for elastic horizontal scaling of microservices." In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, ACM, 35–40.

Ma, L., Yi, S., Carter, N. and Li, Q. (2018). "Efficient live migration of edge services leveraging container layered storage." *IEEE Transactions on Mobile Computing*.

Ma, L., Yi, S. and Li, Q. (2017). "Efficient service handoff across edge servers via docker container migration." In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ACM, 11.

MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., Metz, R. and Hamilton, B. A. (2006). "Reference model for service oriented architecture 1.0." *OASIS standard*, 12, 18.

Mahdhi, T. and Mezni, H. (2018). "A prediction-based vm consolidation approach in iaas cloud data centers." *Journal of Systems and Software*, 146, 263–285.

Martin, J. P., Kandasamy, A. and Chandrasekaran, K. (2020). "Mobility aware autonomic approach for the migration of application modules in fog computing environment." *Journal of Ambient Intelligence and Humanized Computing*, 1–20.

Mayer, B. and Weinreich, R. (2017). "A dashboard for microservice monitoring and management." In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, IEEE, 66–69.

Mazzara, M., Khanda, K., Mustafin, R., Rivera, V., Safina, L. and Sillitti, A. (2016). "Microservices science and engineering." In *International Conference in Software Engineering for Defence Applications*, Springer, 11–20.

Meinke, K. and Nycander, P. (2015). "Learning-based testing of distributed microservice architectures: Correctness and fault injection." In *International Conference on Software Engineering and Formal Methods*, Springer, 3–10.

Messina, A., Rizzo, R., Storniolo, P. and Urso, A. (2016). "A simplified database pattern for the microservice architecture." In *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, 35–40.

Meyer, B. (1988). *Object-oriented software construction*, volume 2, Prentice hall, New York.

Mirjalili, S., Jangir, P., Mirjalili, S. Z., Saremi, S. and Trivedi, I. N. (2017). "Optimization of problems with multiple objectives using the multi-verse optimization algorithm." *Knowledge-Based Systems*, 134, 50–71.

Mirjalili, S., Mirjalili, S. M. and Hatamlou, A. (2016). "Multi-verse optimizer: a nature-inspired algorithm for global optimization." *Neural Computing and Applications*, 27(2), 495–513.

Montesi, F. and Weber, J. (2016). "Circuit breakers, discovery, and api gateways in microservices." *arXiv preprint arXiv:1609.05830*.

Nadareishvili, I., Mitra, R., McLarty, M. and Amundsen, M. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture*, O'Reilly Media, Inc.

Naily, M. A., Setyautami, M. R. A., Muschevici, R. and Azurat, A. (2017). "A framework for modelling variable microservices as software product lines." In *International Conference on Software Engineering and Formal Methods*, Springer, 246–261.

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

Netaji, V. K. and Bhole, G. (2019). "Optimal container resource allocation in cloud architecture: A new hybrid model." *Journal of King Saud University-Computer and Information Sciences*.

Netto, H. V., Lung, L. C., Correia, M., Luiz, A. F. and de Souza, L. M. S. (2017). "State machine replication in containers managed by kubernetes." *Journal of Systems Architecture*, 73, 53–59.

Newman, S. (2015). *Building microservices: designing fine-grained systems*, O'Reilly Media, Inc.

O'Connor, R. V., Elger, P. and Clarke, P. M. (2017). "Continuous software engineering-a microservices architecture perspective." *Journal of Software: Evolution and Process*, 29(11).

Pallewatta, S., Kostakos, V. and Buyya, R. (2019). "Microservices-based iot application placement within heterogeneous and resource constrained fog computing environments." In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 71–81.

Papazoglou, M. P. (2003). "Service-oriented computing: Concepts, characteristics and directions." In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, IEEE, 3–12.

Peinl, R. and Holzschuher, F. (2015). "The docker ecosystem needs consolidation.." In *CLOSER*, 535–542.

Peinl, R., Holzschuher, F. and Pfitzer, F. (2016). "Docker cluster management for the cloud-survey results and own solution." *Journal of Grid Computing*, 14(2), 265–282.

Petrasch, R. (2017). "Model-based engineering for microservice architectures using enterprise integration patterns for inter-service communication." In *Computer Science*

and Software Engineering (JCSSE), 2017 14th International Joint Conference on, IEEE, 1–4.

Piraghaj, S. F., Dastjerdi, A. V., Calheiros, R. N. and Buyya, R. (2015). "A framework and algorithm for energy efficient container consolidation in cloud data centers." In *2015 IEEE International Conference on Data Science and Data Intensive Systems*, IEEE, 368–375.

Pozdniakova, O. and Mazeika, D. (2017). "Systematic literature review of the cloud-ready software architecture." *Baltic Journal of Modern Computing*, 5(1), 124.

Prometheus, A. (2020). "Prometheus-monitoring system & time series database." `https://prometheus.io/` (1 May, 2020).

Puliafito, C., Vallati, C., Mingozzi, E., Merlino, G., Longo, F. and Puliafito, A. (2019). "Container migration in the fog: A performance evaluation." *Sensors*, 19(7), 1488.

Rademacher, F., Sachweh, S. and Zündorf, A. (2017). "Differences between model-driven development of service-oriented and microservice architecture." In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, IEEE, 38–45.

Rajagopalan, S. and Jamjoom, H. (2015). "App-bisect: Autonomous healing for microservice-based apps." In *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, USENIX Association, 16–16.

Rajagopalan, S., Nagpurkar, P., Eilam, T., Jamjoom, H., Lev-Ran, E., Bortnikov, V. and Budinsky, F. (2016). "Opportunities & challenges in adopting microservice architecture for enterprise workloads." In *Technical Session Presentation, USENIX Association*.

Rattihalli, G. (2018). "Exploring potential for resource request right-sizing via estimation and container migration in apache mesos." In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 59–64.

Research and Markets (2018). "$1.8 billion cloud microservices market - global forecast to 2023." `https://www.prnewswire.com/news-releases/1-8-billion-cloud-microservices-market---global-forecast-to-2023--300665538.html` (15 May, 2020).

Richardson, C. (2017). "Microservice patterns." `microservices.io/patterns/index.html` (1 May, 2020).

Rodriguez, M. and Buyya, R. (2020). "Container orchestration with cost-efficient autoscaling in cloud computing environments." In *Handbook of Research on Multimedia Cyber Security*, IGI Global, 190–213.

Rodriguez, M. A. and Buyya, R. (2018). "Containers orchestration with cost-efficient autoscaling in cloud computing environments." *arXiv preprint arXiv:1812.00300*.

Rodriguez, M. A. and Buyya, R. (2019). "Container-based cluster orchestration systems: A taxonomy and future directions." *Software: Practice and Experience*, 49(5), 698–719.

Rosenberg, D., Boehm, B., Wang, B. and Qi, K. (2017). "Rapid, evolutionary, reliable, scalable system and software development: the resilient agile process." In *Proceedings of the 2017 International Conference on Software and System Process*, ACM, 60–69.

Rotter, C., Illés, J., Nyíri, G., Farkas, L., Csatári, G. and Huszty, G. (2017). "Telecom strategies for service discovery in microservice environments." In *Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on*, IEEE, 214–218.

Rusek, M., Dwornicki, G. and Orłowski, A. (2016). "A decentralized system for load balancing of containerized microservices in the cloud." In *International Conference on Systems Science*, Springer, 142–152.

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M. and Al-Hammadi, Y. (2016). "The evolution of distributed systems towards microservices architecture." In *Internet Technology and Secured Transactions (ICITST), 2016 11th International Conference for*, IEEE, 318–325.

Savchenko, D. and Radchenko, G. (2015). "Microservices validation: Methodology and implementation." In *CEUR Workshop Proceedings. Vol. 1513: Proceedings of the 1st Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists (Ural-PDC 2015).-Yekaterinburg, 2015.*, 235–240.

Savchenko, D. I., Radchenko, G. I. and Taipale, O. (2015). "Microservices validation: Mjolnirr platform case study." In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, IEEE, 235–240.

Schermann, G., Schöni, D., Leitner, P. and Gall, H. C. (2016). "Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies." In *Proceedings of the 17th International Middleware Conference*, ACM, 12.

Selimi, M., Cerdà-Alabern, L., Sánchez-Artigas, M., Freitag, F. and Veiga, L. (2017). "Practical service placement approach for microservices architecture." In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE Press, 401–410.

Sharma, B., Thulasiram, R. K., Thulasiraman, P. and Buyya, R. (2014). "Clabacus: a risk-adjusted cloud resources pricing model using financial option theory." *IEEE Transactions on Cloud Computing*, 3(3), 332–344.

Sharma, V., Srinivasan, K., Jayakody, D. N. K., Rana, O. and Kumar, R. (2017). "Managing service-heterogeneity using osmotic computing." *arXiv preprint arXiv:1704.04213*.

Sharma, Y., Si, W., Sun, D. and Javadi, B. (2019). "Failure-aware energy-efficient vm consolidation in cloud computing systems." *Future Generation Computer Systems*, 94, 620–633.

Shaw, R., Howley, E. and Barrett, E. (2018). "A predictive anti-correlated virtual machine placement algorithm for green cloud computing." In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, IEEE, 267–276.

Sousa, G., Rudametkin, W. and Duchien, L. (2016). "Automated setup of multi-cloud environments for microservices applications." In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, IEEE, 327–334.

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

Stroustrup, B. (1994). *The design and evolution of C++*, Pearson Education India.

Stubbs, J., Moreira, W. and Dooley, R. (2015). "Distributed systems of microservices using docker and serfnode." In *Science Gateways (IWSG), 2015 7th International Workshop on*, IEEE, 34–39.

Sundar, A. (2017). "An insight into microservices testing strategies." Technical report, Infosys.

Tai, S. (2016). "Continuous, trustless, and fair: Changing priorities in services computing." In *European Conference on Service-Oriented and Cloud Computing*, Springer, 205–210.

Tao, Y., Wang, X., Xu, X. and Chen, Y. (2017). "Dynamic resource allocation algorithm for container-based service computing." In *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*, IEEE, 61–67.

Tasgetiren, M. F., Sevkli, M., Liang, Y.-C. and Gencyilmaz, G. (2004). "Particle swarm optimization algorithm for single machine total weighted tardiness problem." In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, volume 2, IEEE, 1412–1419.

Tellago, B. (2016). "Microservices architecture in the enterprise : A research study and reference architecture." Technical report, Tellago Incorporation.

Thalheim, J., Rodrigues, A., Akkus, I. E., Bhatotia, P., Chen, R., Viswanath, B., Jiao, L. and Fetzer, C. (2017). "Sieve: actionable insights from monitored metrics in distributed systems." In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ACM, 14–27.

Thomas, D., Sagar, K. and Vincent, C. (2019). "Make no mistake, the cloud continues to accelerate (etr research)." `https://etr.plus/articles/cloud-continues-to-accelerate` (15 May, 2020).

Tizzei, L. P., Nery, M., Segura, V. C. and Cerqueira, R. F. (2017). "Using microservices and software product line engineering to support reuse of evolving multi-tenant saas." In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, ACM, 205–214.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. and Edmonds, A. (2015). "An architecture for self-managing microservices." In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, ACM, 19–24.

Tonelli, R., Lunesu, M. I., Pinna, A., Taibi, D. and Marchesi, M. (2019). "Implementing a microservices system with blockchain smart contracts." In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, IEEE, 22–31.

Tonelli, R., Pinna, A., Baralla, G. and Ibba, S. (2018). "Ethereum smart contracts as blockchain-oriented microservices." In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, 1–2.

Torkura, K. A., Sukmana, M. I., Cheng, F. and Meinel, C. (2017). "Leveraging cloud native design patterns for security-as-a-service applications." In *Smart Cloud (SmartCloud), 2017 IEEE International Conference on*, IEEE, 90–97.

Tsai, W.-T. (2005). "Service-oriented system engineering: a new paradigm." In *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05).*, IEEE, 3–8.

Ulander, D. (2017). "Software architectural metrics for the scania internet of things platform: From a microservice perspective." Master's thesis, Uppsala Universitet.

Van Eyk, E., Grohmann, J., Eismann, S., Bauer, A., Versluis, L., Toader, L., Schmitt, N., Herbst, N., Abad, C. and Iosup, A. (2019). "The SPEC-RG reference architecture for FaaS: From microservices and containers to serverless platforms." *IEEE Internet Computing*.

Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uă, A. and Iosup, A. (2018). "Serverless is more: From paas to present cloud computing." *IEEE Internet Computing*, 22(5), 8–17.

Vaucher, S., Pires, R., Felber, P., Pasin, M., Schiavoni, V. and Fetzer, C. (2018). "Sgx-aware container orchestration for heterogeneous clusters." In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 730–741.

Versteden, A. and Pauwels, E. (2016). "State-of-the-art web applications using microservices and linked data." In *4th Workshop on Services and Applications over Linked APIs and Data(SALAD)*.

Villari, M., Celesti, A. and Fazio, M. (2017). "Towards osmotic computing: Looking at basic principles and technologies." In *Conference on Complex, Intelligent, and Software Intensive Systems*, Springer, 906–915.

Villari, M., Fazio, M., Dustdar, S., Rana, O. and Ranjan, R. (2016). "Osmotic computing: A new paradigm for edge/cloud integration." *IEEE Cloud Computing*, 3(6), 76–83.

Vinoski, S. (1997). "CORBA: integrating diverse applications within distributed heterogeneous environments." *IEEE Communications magazine*, 35(2), 46–55.

Vizard, M. (2015). "Walmart embraces microservices to get more agile." `http://www.baselinemag.com/enterprise-apps/walmart-embraces-microservices-to-get-more-agile.html` (15 May, 2020).

Wan, X., Guan, X., Wang, T., Bai, G. and Choi, B.-Y. (2018). "Application deployment using microservice and docker containers: Framework and optimization." *Journal of Network and Computer Applications*, 119, 97–109.

Wang, J. V., Cheng, C.-T. and Tse, C. K. (2019a). "A thermal-aware vm consolidation mechanism with outage avoidance." *Software: Practice and Experience*, 49(5), 906–920.

Wang, J. V., Ganganath, N., Cheng, C.-T. and Chi, K. T. (2019b). "Bio-inspired heuristics for vm consolidation in cloud data centers." *IEEE Systems Journal*.

Weaveworks, I. (2017). "Socks shop: Microservices demo." `microservices-demo.github.io/` (1 May, 2020).

Wen, Z., Lin, T., Yang, R., Ji, S., Romanovsky, A., Lin, C., Xu, J. et al. (2019). "Ga-par: Dependable microservice orchestration framework for geo-distributed clouds." *IEEE Transactions on Parallel and Distributed Systems*.

Witanto, J. N., Lim, H. and Atiquzzaman, M. (2018). "Adaptive selection of dynamic vm consolidation algorithm using neural network for cloud resource management." *Future Generation Computer Systems*, 87, 35–42.

Wizenty, P., Sorgalla, J., Rademacher, F. and Sachweh, S. (2017). "Magma: build management-based generation of microservice infrastructures." In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, ACM, 61–65.

Wu, A. (2017). "Taking the cloud-native approach with microservices." Technical report.

Xu, C., Wang, K. and Guo, M. (2017). "Intelligent resource management in blockchain-based cloud datacenters." *IEEE Cloud Computing*, 4(6), 50–59.

Xu, M. and Buyya, R. (2019). "Brownoutcon: A software system based on brownout and containers for energy-efficient cloud computing." *Journal of Systems and Software*, 155, 91–103.

Xu, M., Toosi, A. N. and Buyya, R. (2018). "ibrownout: An integrated approach for managing energy and brownout in container-based clouds." *IEEE Transactions on Sustainable Computing*, 4(1), 53–66.

Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., Kong, J. and Jue, J. P. (2019). "All one needs to know about fog computing and related edge computing paradigms: A complete survey." *Journal of Systems Architecture*.

Zdun, U., Navarro, E. and Leymann, F. (2017). "Ensuring and assessing architecture conformance to microservice decomposition patterns." In *International Conference on Service-Oriented Computing*, Springer, 411–429.

Zhang, Y., Xu, K., Wang, H., Li, Q., Li, T. and Cao, X. (2018). "Going fast and fair: Latency optimization for cloud-based service chains." *IEEE Netw*, 32(2), 138–143.

Zheng, T., Zheng, X., Zhang, Y., Deng, Y., Dong, E., Zhang, R. and Liu, X. (2019). "SmartVM: a sla-aware microservice deployment framework." *World Wide Web*, 22(1), 275–293.

Zhong, Z. and Buyya, R. (2020). "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources." *ACM Transactions on Internet Technology (TOIT)*, 20(2), 1–24.

Zimmermann, O. (2016). "Microservices tenets: agile approach to service development and deployment." *Computer Science-Research and Development*, 32(3), 301–310.

Zúñiga-Prieto, M., Insfran, E., Abrahão, S. and Cano-Genoves, C. (2017). "Automation of the incremental integration of microservices architectures." In *Complexity in Information Systems Development*, Springer, 51–68.

RESEARCH OUTCOMES

PUBLICATIONS

Journal Papers

1. Joseph, C. T. & Chandrasekaran, K. (2019). Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. Wiley Software Practice and Experience, 49(10), 1448-1484. (DOI: https://doi.org/10.1002/spe.2729, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2729) **[SCI Indexed]**

2. Joseph, C. T. & Chandrasekaran, K. (2020). IntMA: Dynamic Interaction-Aware Resource Allocation for Containerized Microservices. Elsevier Journal of Systems Architecture. (DOI: https://doi.org/10.1016/j.sysarc.2020.101785, URL: https://www.sciencedirect.com/science/article/pii/S1383762120300758?via%3Dihub) **[SCI Indexed]**

3. Joseph, C. T. & Chandrasekaran, K. (2021). Nature-Inspired Resource Management and Dynamic Rescheduling of Microservices in Cloud Datacenters, Wiley Concurrency and computation: practice and experience (Accepted) **[SCI Indexed]**.

Conference Papers

1. Joseph, C. T. & Chandrasekaran, K. (2019). A Probe into the Technological Enablers of Microservice Architectures. Integrated Intelligent Computing, Communication and Security, vol. 771, 493-506. Springer, Singapore. (DOI: https://doi.org/10.1007/978-981-10-8797-4_50, URL: https://link.springer.com/chapter/10.1007/978-981-10-8797-4_50)

BIODATA

Name:	Christina Terese Joseph
Date of Birth:	3rd July, 1990
Gender:	Female
Marital Status:	Married
Father's Name:	K. S. Joseph
Mother's Name:	Shirley Joseph
Address:	Pottakka House, Azhakam, Kodakara P.O.. Thrissur, Kerala, PIN: 680684.
E-mail:	xtina1232@gmail.com
Mobile:	9497482293
Qualification:	B.Tech in Computer Science and Engineering (Rajagiri School of Engineering and Technology, Kerala) M.Tech in Computer Science and Engineering (Rajagiri School of Engineering and Technology, Kerala)
Areas of Interest:	Distributed Computing, Cloud Computing, Resource Management in Distributed Systems