Microservices-based Internet of Things Applications Placement in Fog Computing Environments

Samodha Pallewatta

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE, AUSTRALIA

February 2023

Microservices-based Internet of Things Applications Placement in Fog Computing Environments

Samodha Pallewatta
Principal Supervisor: Prof. Rajkumar Buyya
Co-Supervisor: Prof. Vassilis Kostakos

Abstract

The Internet of Things (IoT) paradigm is rapidly improving various application domains such as healthcare, smart city, Industrial IoT (IIoT), and intelligent transportation by interweaving sensors, actuators and data analytics platforms to create smart environments. Initially, the cloud-centric IoT was introduced as a viable solution for processing and storing massive amounts of data generated by IoT devices. However, with rapidly increasing data volumes, data transmission from geo-distributed IoT devices to the centralised Cloud incurs high network congestion and high latency. Thus, cloud-centric IoT often fails to satisfy the Quality of Service (QoS) requirements of latency-sensitive and bandwidth-hungry IoT application services. *Fog computing* paradigm extends cloud-like services towards the edge of the network, thus offering low latency service delivery. However, Fog nodes are distributed, heterogeneous and resource-constrained, creating the need to utilise both Fog and Cloud resources to execute IoT applications in a QoS-aware manner.

Meanwhile, *MicroService Architecture* (MSA) has emerged as a powerful application architecture capable of satisfying the development and deployment needs of rapidly evolving IoT applications. The fine-grained modularity of microservices, their independently deployable and scalable nature, along with the lack of centralised management, demonstrate immense potential in harnessing the power of distributed Fog and Cloud resources to meet the QoS requirements of IoT applications. Furthermore, the loosely coupled nature of microservices enables the dynamic composition of distributed microservices to achieve diverse performance requirements of IoT applications while utilising distributed computing resources. To this end, efficient placement of microservices plays a vital role, and scalable placement techniques can use MSA characteristics to harvest the full potential of the Fog computing paradigm.

This thesis investigates novel placement techniques and systems for microservices-based IoT applications in Fog computing environments. Proposed approaches identify MSA characteristics to overcome challenges within the Fog computing environments and make use of them to fulfil heterogeneous QoS requirements of IoT application services in terms of service latency, budget, throughput and reliability while utilising Fog and Cloud resources in a balanced manner. This thesis advances the state-of-the-art in Fog computing by making the following key contributions:

1. A comprehensive taxonomy and literature review on the placement of microservices-based IoT applications considering different aspects, namely modelling microservices-based applications, creating application placement policies, microservice composition, and performance evaluation, in Fog computing environments.

2. A distributed placement technique for scalable deployment of microservices to minimise the latency of the application services and network usage due to IoT data transmission.

3. A robust placement technique for batch placement of microservices-based IoT applications, where the technique considers the placement of a set of applications simultaneously to optimise the QoS satisfaction of application services in terms of makespan, budget and throughput while dynamically utilising Fog and Cloud resources.

4. A reliability-aware placement technique for proactive redundant placement of microservices to improve reliability satisfaction in a throughput and cost-aware manner.

5. A software framework for microservices-based IoT application placement and dynamic composition across federated Fog and Cloud computing environments.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Samodha Pallewatta, February 2023

Preface

Main Contributions

This thesis research has been carried out in Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2-6 and are based on the following publications:

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Placement of Microservices-based IoT Applications in Fog Computing: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)*, Volume 55, No. 14s, Article 321, ISSN: 0360-0300, December 2023.

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Microservices-based IoT Application Placement within Heterogeneous and Resource Constrained Fog Computing Environments", *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, Pages: 71-81, Auckland, New Zealand, December 2-5, 2019.

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "QoS-aware placement of microservices-based IoT applications in Fog computing environments", *Future Generation Computer Systems (FGCS)*, Volume 131, Pages: 121-136, ISSN: 0167-739X, June 2022.

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Reliability-aware Proactive Placement of Microservices-based IoT Applications in Fog Computing

Environments", *IEEE Transactions on Mobile Computing (TMC)*, (revision, August 2023).

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "MicroFog: A Framework for Scalable Placement of Microservices-based IoT Applications in Federated Fog Environments", *Journal of Systems and Software*, (revision, June 2023).

Supplementary Contributions

During the Ph.D. candidature, I have also contributed to the following collaborative works (this thesis does not claim them as its contributions):

- Redowan Mahmud, **Samodha Pallewatta**, Mohammad Goudarzi, and Rajkumar Buyya, "IFogSim2: An Extended iFogSim Simulator for Mobility, Clustering, and Microservice Management in Edge and Fog Computing Environments", *Journal of Systems and Software (JSS)* Volume 190, ISSN: 0164-1212, August 2022.

Acknowledgements

I would like to thank my supervisors, Professor Rajkumar Buyya and Professor Vassilis Kostakos, for giving me the opportunity to pursue my PhD under their guidance. I am grateful for their invaluable support, encouragement, and guidance throughout my candidature. I would like to express my sincere gratitude to them for providing me with great opportunities to grow as a researcher and helping me navigate the ups and downs of the challenging PhD journey. I would also like to express my gratitude to my PhD advisory committee members, Professor Alistair Moffat and Professor Shanika Karunasekera, for their valuable comments and suggestions.

I would also like to thank all the past and current members of the CLOUDS Laboratory at the University of Melbourne. In particular, I thank Dr. Mohammad Goudarzi, Dr. Redowan Mahmud, Dr. Maria Rodriguez, Dr. Sara Kardani, Dr. Shashikant Ilager, Dr. Muhammad Hilman, Dr. Muhammed Tawfiqul Islam, Dr. TianZhang He, Zhiheng Zhong, Amanda Jayanetti, Anupama Mampage, Rajeev Muralidhar, Kwangsuk Song, Jie Zhao, Ming Chen, Siddharth Agarwal, Tharindu Bandara, Thanh-Hoa Nguyen, Yulun Huang, Zhiyu Wang, Kalyani Pendyala, Duneesha Fernando, Jayath Seneviratne, Chun Wei Lim, and Thakshila Mohottige for their support.

I would also like to thank friends and family, Shashi Jayaweera, Anjana Perera, Manjula Geeganaarachichi, Punsala Manage and Upendra Keerthilatha, for their support and companionship.

I acknowledge the University of Melbourne for providing me with the scholarship and resources to pursue my doctoral studies. My research is also supported by a Discovery Project grant from the Australian Research Council (ARC) awarded to my principal supervisor.

I would like to extend my sincere thanks to the past and present admin staff of the School of Computing and Information Systems for their support.

I would like to thank my parents, Bandula Pallewatta and Jayanthika Chithrangani, and my brother Samudaya Pallewatta for their unconditional love and support. My sincerest gratitude goes to my parents for always encouraging me and believing in me throughout this challenging journey. Reaching this amazing goal would not have been possible without their endless support.

Samodha Pallewatta
February 2023, Melbourne, Australia

Contents

List of Figures

List of Tables

List of Acronyms

IoT Internet of Things

IIoT Industrial IoT

MSA MicroService Architecture

FAPP Fog Application Placement Problem

SOA Service Oriented Architecture

CPS Cyber-Physical Systems

SLA Service Level Agreement

QoS Quality of Service

QoE Quality of Experience

MEC Mobile Edge computing

VM Virtual Machines

CoAP Constrained Application Protocol

LAN Local Area Network

WAN Wide Area Network

WLAN Wireless Local Area Network

RAN Radio Access Network

SDN Software-Defined Networking

NFV Network Function Virtualization

DAG Directed Acyclic Graph

BoT Bag of Tasks

FIFO First In First Out

ILP Integer Linear Programming

MILP Mixed Integer Linear Programming

GA Genetic Algorithm

PSO Particle Swarm Optimisation

ACO Ant Colony Optimisation

S-CLPSO Set-based Comprehensive Learning Particle Swarm Optimisation

AHP Analytical Hierarchy Process

ML Machine Learning

Chapter 1
Introduction

The Internet of Things (IoT) paradigm is gaining immense popularity due to its significance in technical, social and economic aspects [1]. IoT transforms everyday objects and infrastructure into intelligent entities that can interact with each other without human intervention, which has resulted in its expansion to a wide range of services, including healthcare, transportation, industrialisation, and agriculture. IoT generates enormous quantities of dynamic data composed of various data types to be processed, analysed and stored. Cloud computing was initially identified as a viable solution for hosting such IoT services, giving rise to cloud-centric IoT [2]. However, due to the exponential increase in connected devices, raw data transmission towards centralised Cloud data centres increases network congestion and latency, thus reducing the feasibility of cloud-centric IoT analytics. As a solution, a novel distributed computing paradigm called Fog computing is introduced, bringing data processing and storage closer to the end-user, hence supporting latency-critical and bandwidth-consuming IoT services.

Meanwhile, MicroService Architecture (MSA) emerged as a cloud-native application architecture style, enabling the development and deployment of highly reliable and scalable software systems that can undergo frequent updates and deployments [3, 4]. Microservices-based IoT applications are gaining tremendous momentum due to their potential to improve the performance of IoT services deployed within distributed computing environments [5]. According to the market research conducted by *The International Market Analysis Research and Consulting Group (IMARC Group)*, the global microservice architecture market is expected to reach US$ 6.84 Billion by 2027 with a Compound Annual Growth Rate (CAGR) of 15.70% during 2022-2027 [6]. Compared to previous application architectures such as monolithic architecture and Service Oriented Architec-

ture (SOA) realised through web services, the true potential of MSA as a cloud-native application architecture lies in its loosely coupled nature, which enables containerised deployment, dynamic composition, and load-balancing across federated multi-fog and multi-cloud environments with the support of other cloud-native technologies like container orchestrators (i.e., Kubernetes, Docker Swarm) and service mesh technologies (i.e., Istio, Linkerd). Microservices can be independently and dynamically deployed and horizontally scaled across hybrid environments while maintaining seamless connectivity among interacting microservices under dynamic conditions. This paved the way for the convergence of Fog computing, IoT and microservices, thus resulting in the introduction of novel paradigms such as Osmotic Computing [7] that focus on dynamic placement and deployment of microservices across federated Fog-Cloud environments. Thus, within geo-distributed and heterogeneous Fog environments, the placement of microservices-based applications remains one of the most critical and challenging areas. The placement algorithms benefit from awareness of the MSA-related characteristics so that they can adequately utilise the strengths of the application architecture to overcome the challenges and limitations of Fog computing environments.

To this direction, this thesis addresses the problem of optimal placement of microservices-based IoT applications within Fog computing environments. It investigates novel placement approaches to utilise MSA characteristics to overcomes the challenges of Fog computing and improve the QoS of IoT application services. This is achieved by conducting a comprehensive background survey and developing a taxonomy of the state-of-the-art to identify multiple aspects related to solving the placement problem of microservies-based IoT applications within Fog computing environments. Moreover, a set of placement algorithms are developed to improve the QoS satisfaction of the IoT application services in terms of service latency, deployment cost, throughput and reliability. Finally, a software framework is designed and implemented to enable the placement and dynamic composition of microservices across multi-fog multi-cloud environments.

Proposed placement algorithms and software framework are evaluated using three methodologies; numerical evaluations to determine the efficiency of the proposed algorithms based on convergence and complexity analysis, discrete event-driven simulations to evaluate placement approaches using simulated Fog environments and practical im-

plementations to create prototype systems to evaluate proposed software framework. In this thesis, simulations are conducted using the iFogSim simulation toolkit [8, 9]. To this end, we extend iFogSim and implement microservice orchestration-related features [8]. For practical evaluations presented in this thesis, we create the prototype using cloud-native technologies such as Docker, Kubernetes and Istio.

1.1 Background

This section discusses fundamental concepts related to the research problem addressed in the thesis.

1.1.1 Fog Computing

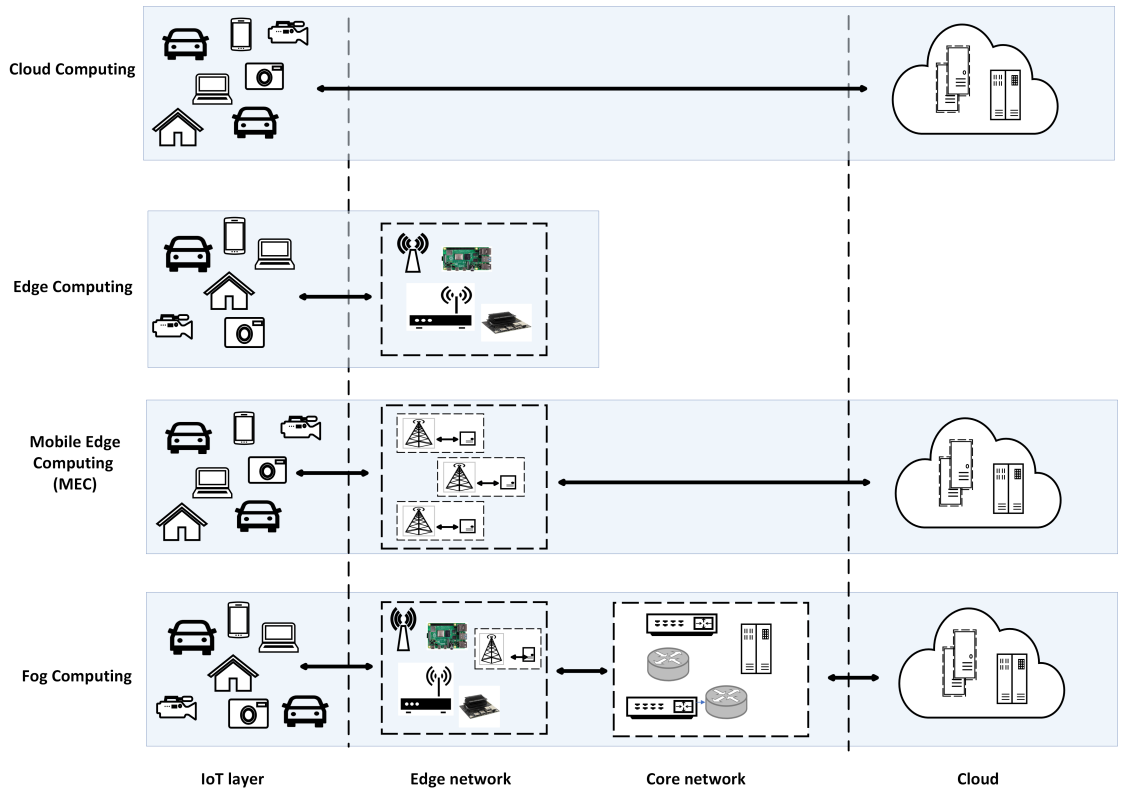


Figure 1.1: Fog computing and related paradigms

For many years Cloud computing has been one of the leading facilitators of IoT that offers on-demand services to aggregate, process and store the data generated by IoT devices. As Cloud data centres are located multiple hops away from IoT devices, data analytics in the Cloud results in higher latency values due to the extended data transmission delay. This considerably degrades the performance of IoT application services with low latency requirements. With IoT applications growing into a vital aspect of modern living, the number of IoT devices has increased exponentially. According to the estimates by International Data Corporation, 41.6 billion IoT devices will generate 79.4 zettabytes of data in 2025 [10]. Transmitting such a large amount of data towards Cloud would add a substantial load to the network resulting in severe network congestion.

To address these limitations of the cloud-centric IoT model, Fog computing is introduced to extend cloud-like services towards the edge of the network [11]. The Fog computing paradigm was first introduced by Cisco in 2012 as a platform to support the unique requirements of IoT, such as low latency, location awareness, mobility support, and geo-distribution [12]. To this end, Fog computing introduces an intermediate layer between IoT devices and Cloud data centres [11], which is organised in a multi-tier architecture. It exploits computation, storage and networking resources that reside within the path connecting end devices to the Cloud data centres [13]. Thus, Fog resources consist of a diverse set of devices (i.e., smart routers and switches, personal computers, edge servers, Raspberry Pi devices, micro-datacentres, cloudlets, etc.). Consequently, Fog computing provides data processing in the proximity of the data sources, thus supporting low service delivery times. Furthermore, distributed data processing at the network edge minimises the amount of data sent towards the Cloud. This reduces network congestion and lowers the burden on Cloud data centres.

Compared to the Cloud data centres, Fog nodes are distributed and resource-constrained. To overcome the resource limitations, the Fog computing paradigm maintains federated Fog computing architectures, where distributed Fog resources collaborate to satisfy client requirements [14]. Moreover, the Fog computing layer maintains a seamless connection with the Cloud [15] so that computation-intensive tasks can be carried out using Cloud resources. In our view, paradigms like Edge Computing and Mobile Edge Computing (MEC) utilise only the Edge resources residing within the closest layer to the IoT

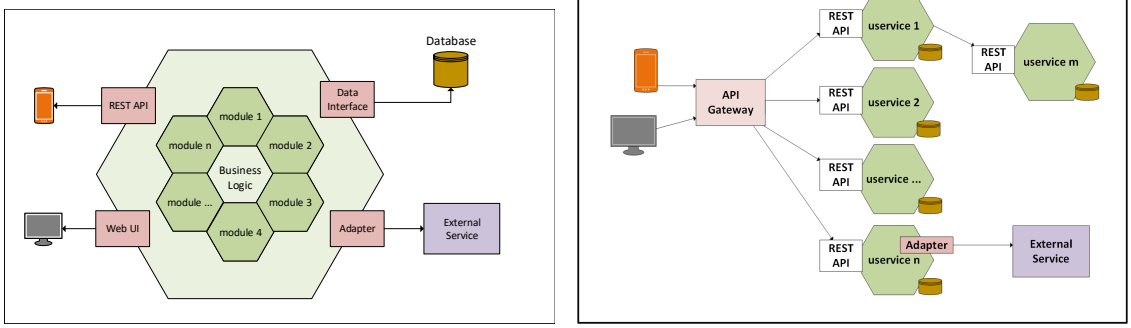
devices. In contrast, Fog extends this concept further to include resources at different computing layers with the integration of Cloud data centres when necessary (see Figure 1.1) to provide IaaS, PaaS and SaaS services in the proximity of the data sources (although some works use these terms interchangeably).

Considering the benefits and potential of Fog computing, major Cloud service providers like Amazon, Google, Microsoft, Oracle, and IBM have started extending their infrastructure to support Fog services [16]. Moreover, hardware manufacturers such as Cisco, Dell and Intel are developing devices to be used as Fog computing nodes. Furthermore, companies like VMware, FogHorn Systems, and SONM are building software platforms and compute stacks to build, run, and manage Fog computing-based IoT solutions [17]. To further increase the momentum of Fog adaptation, telecom providers such as Telstra and Telefonica have commenced developing prototypes to provide computing at the network edge through mini-data centres. With such rapid advancement, the Fog market is expected to reach USD 155.90 billion by 2030 [18].

1.1.2 Microservice Architecture (MSA)

An enterprise application usually consists of a server-side application that implements its domain-specific business logic, and client-side user interfaces supporting different clients such as desktop browsers and/or mobile browsers, and a database to persist the data. Moreover, the application may connect with other third-party applications (through web services or message brokers) and expose APIs for third parties to consume. The design and development of such applications follow different architectural patterns depending on the complexity of the business domain. Figure 1.2 presents a general representation of an application developed using Monolithic and Microservice architecture.

In monolithic architecture, the server-side application is a single logical executable developed either as a single process or a modular monolith where modules invoke each other through method/function calls at the programming language level [19]. Using monolithic architecture is advantageous during the early stages of an application or if the application is quite simple with only a few core functionalities. But as the appli-



(a) Monolithic application architecture

(b) Microservice application architecture (MSA)

Figure 1.2: Application architecture (monolithic vs microservices)

cations grow, they tend to outgrow the monolithic architecture [20]. As all modules of the application are combined into a single unit, the coupling among them increases, and the application becomes too complex to understand, inflexible to change, incurs high deployment delays and lacks scalability.

MSA aids in overcoming the limitations of the monolithic architecture within rapidly growing software ecosystems. Martin Fowler defines MSA as ”an approach to developing a single application as a suite of small services, each running as independent processes and communicating with lightweight mechanisms, often an HTTP resource API” [21]. Microservices are independently deployable and scalable units designed around business logic adhering to the single responsibility principle [22]. Moreover, microservices are loosely connected components that can be easily integrated to create complex applications. Due to fine-grained modularity, each microservice can be deployed on hardware that best matches its resource requirements (i.e., CPU-intensive, memory-intensive, I/O-intensive, etc.) and can be deployed and scaled independently according to the load on each service. Unlike monolithic architecture, MSA provides better fault isolation, as a fault in a particular service only affects that service. Furthermore, the independent scalability of microservices improves redundant deployment to achieve fault tolerance. New technologies that best suit the microservices can be easily adapted as services are loosely coupled. This mitigates the need to stick to a single technology stack and provides the flexibility to evolve with technologies. Table 1.1 summarises and compares the characteristics of the two architectures.

Table 1.1: Comparison between monolithic and microservice architecture

Monolithic Architecture	Microservice Architecture
Server-side application is a single logical executable	Server-side consists of independently deployable, multiple microservices
Modules communicate through language level method invocations	Inter-process communication through REST APIs or lightweight messaging
Low flexibility	Flexible - different programming languages and technologies can be used based on microservice requirements
Less support for scaling	Highly scalable (independently deployable and scalable microservices)
Longer time from development to deployment	Supports rapid and agile development and deployment
Unreliable due to single point of failure	Higher resilience to failure due to failure isolation and redundancy support

These characteristics of MSA have established it as the backbone of the cloud-native application architecture. According to the cloud-native architecture, microservices are packaged as self-contained, lightweight containers (such as Docker) and managed through container orchestration tools (such as Kubernetes) to utilise elasticity, scale and resiliency provided by computing architectures like Cloud and Fog. Moreover, MSA uses service mesh technologies (such as Istio) to manage microservice composition-related cross-cutting concerns such as service discovery, load balancing among horizontally scaled microservices, distributed monitoring, and security. Thus, MSA has emerged as the leading enabler for scalable application execution in distributed, multi-cloud environments, thus making MSA a strong candidate for Fog applications [23].

1.1.3 Microservices-based IoT Applications and Fog Computing

This section highlights how the main characteristics of MSA, IoT applications and Fog computing paradigm fit perfectly together, giving rise to microservices-based IoT applications for Fog environments. Figure 1.3 lists these characteristics and groups them to show their relationships.

IoT application characteristics can be categorised into three main groups; Design and Development, Deployment and Management, and Service Characteristics. Rapid design, development needs and interoperability, and service reusability caused by the IoT ecosystem's complexity can be satisfied by adopting a software architecture that

supports higher flexibility to change, reduced time to market, and collaboration among multiple development teams. MSA can enable these requirements with the fine-grained modular design and loosely coupled nature, which perfectly conforms with agile design and development principles. From a deployment and maintenance perspective, IoT applications must maintain rapid deployment cycles with minimum service disruptions and support the dynamic workload while maintaining service availability and resilience. These requirements can be successfully satisfied using MSA [24–26]. MSA decomposes large and complicated applications into independently deployable and scalable modules that can be conveniently packaged into lightweight containers with considerably lower startup times and rapid deployment and migration support, which result in higher service availability and reliability under dynamic conditions. The decomposition of the application into small independent units allows only the updated or newly developed microservices to be re-deployed and just the performance-affected microservices to be scaled within each deployment cycle. Moreover, this enables a proper balance between horizontal and vertical scalability where microservices that are harder to scale horizontally (i.e., services that use relational databases) can be vertically scaled while the rest can be horizontally scaled [27]. Thus, MSA meets the scalability, maintainability, extensibility, and interoperability requirements of large and complex IoT software systems [28]. As a result, MSA is increasingly adopted for IoT application development in many areas such as smart cities, smart healthcare, IIoT [29–31].

In contrast to Cloud data centres, Fog environments consist of distributed, heterogeneous, resource-constrained devices. Thus, deploying a monolithic application onto such devices is less feasible due to their resource demand. The scalability of such applications is also limited by the said characteristic of Fog devices. Fine-grained microservices match such environments better due to their modular, loosely coupled nature, which makes the resource requirements of each microservice small enough to be satisfied by the distributed resources. Independent deployability and scalability of such microservices enable them to adjust to dynamic conditions (i.e., device failures, mobility, workload changes, etc.) while utilising limited resources. Moreover, these characteristics support dynamic and fast migration and composition of microservices across distributed resources, thus improving deployment flexibility. Thus, MSA demonstrates

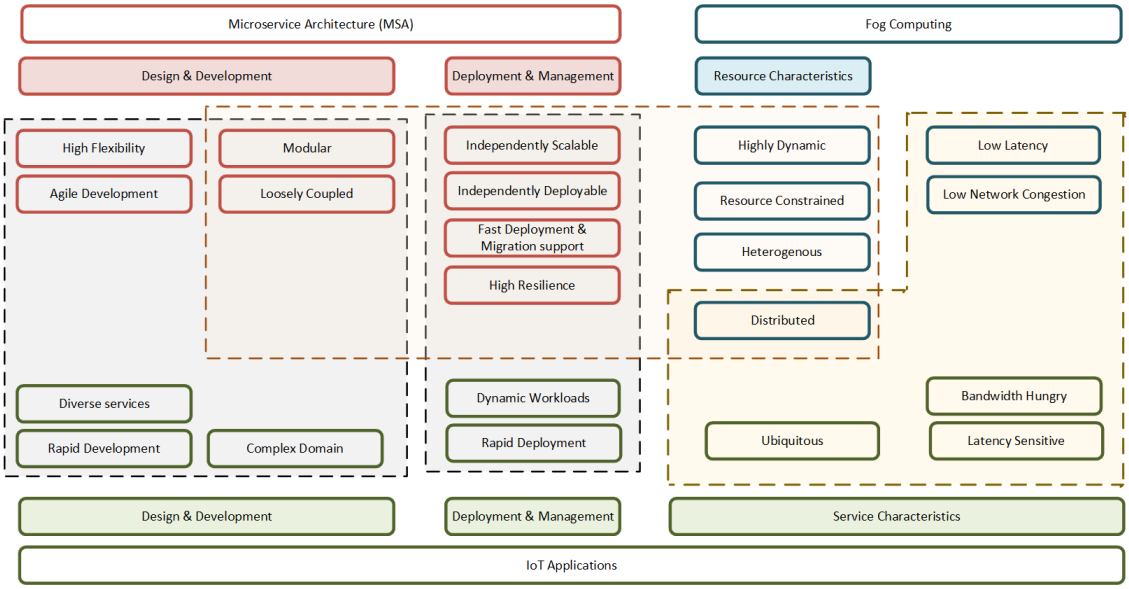


Figure 1.3: IoT applications, Microservice architecture and Fog computing

the potential to utilise Fog environments and achieve a balance between federated Fog and Cloud usage to improve application performance and meet QoS requirements.

1.2 Problem Definition

Microservices-based IoT Application Placement within Fog computing environments falls under Fog Application Placement Problem (FAPP) addressed in works such as [32–35]. FAPP addresses application deployment and maintenance within Fog environments, where services with agreed Service Level Agreements (SLA) are deployed onto federated Fog and Cloud resources for the shared use by the application users. To this end, FAPP considers factors such as horizontal and vertical scalability, request load balancing, ubiquitous access, location awareness, and fault tolerance to produce application deployments that meet the required performance of the application services [32, 34].

Based on the definition of FAPP, we define "Microservices-based IoT applications placement in Fog environments" as follows:

Let A be a microservices-based IoT application where A consists of a set of inde-

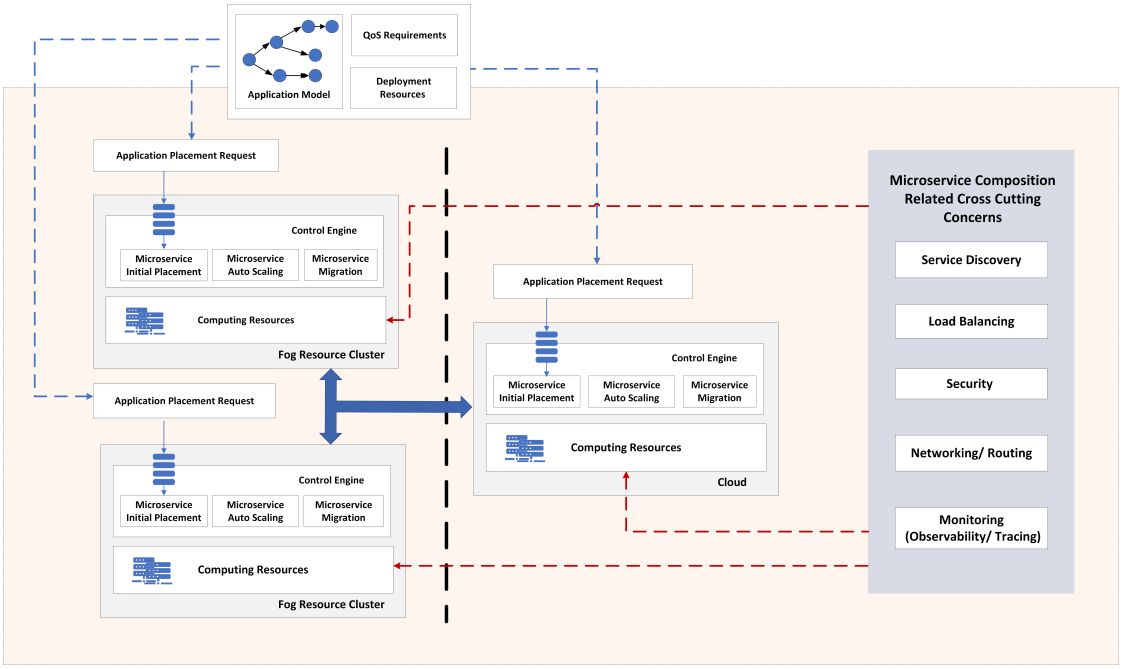


Figure 1.4: Microservices-based Fog application placement

pendently deployable and scalable microservices (M_A) and a set of services provided to the application users (S_A). We use the term "service" to denote end-user-requested business functionalities, which can be either an atomic service (consisting of only a single microservice) or composite services (composed of multiple microservices). As the microservices are granular with well-defined business boundaries, they ($m \in M_A$) can communicate using lightweight protocols to create composite services ($s \in S_A$) requested by the end-users. Figure 1.5 presents example use cases of microservice composition highlighting various different patterns. Placement of such applications includes mapping these microservices to distributed Fog and Cloud resources such that the requirements (i.e., resource requirements of the microservices, QoS requirements of the services, etc.) are ensured while maintaining seamless connectivity across distributed microservices to create composite services. Figure 1.4 provides a motivation scenario to elaborate this further by demonstrating the event flow of the Fog application placement process. The depicted event flow contains distributed control engines receiving application placement requests from application providers and processing them for placement across Fog device clusters and Cloud data centres using efficient and ro-

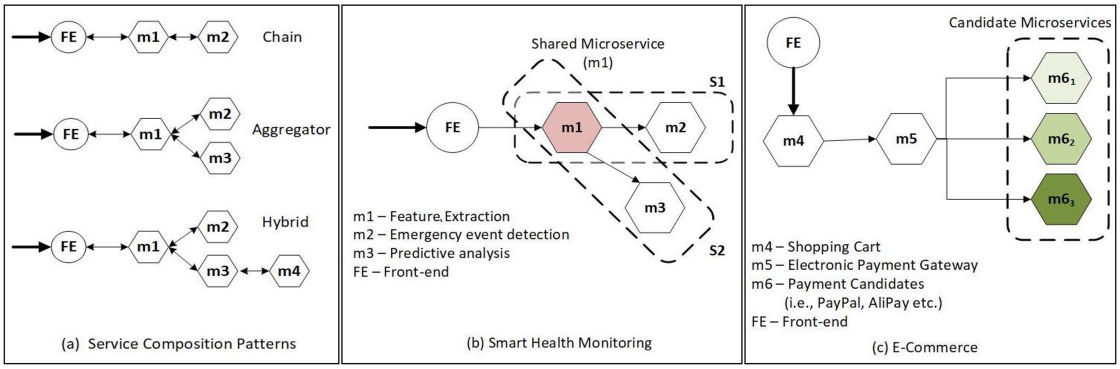


Figure 1.5: Example use cases of microservice composition

bust placement techniques. Placement policies consider multiple aspects such as service quality expectations of the application users, SLAs negotiated between application service providers and infrastructure providers (i.e., service latency, throughput, cost of deployment, reliability) [36–38], and resource offerings of Fog and Cloud infrastructure providers and their revenue [39, 40] to makes the placement decisions. Moreover, placement policies incorporate knowledge about the application model [41, 42] and microservice composition-related features [38, 43, 44] to improve placement approaches to enhance the performance of the applications.

1.2.1 Challenges of Fog Application Placement

The challenges of microservices-based application placement in Fog computing environments are discussed below:

• *Distributed, resource-constrained, and heterogeneous Fog resources*: Fog nodes are deployed in a geo-distributed manner closer to the edge of the networks. While this facilitates the location-aware ubiquitous access required by IoT applications, application placement across distributed computing resources is challenging. Moreover, Fog nodes consist of a large variety of devices (i.e., Raspberry pi, edge servers, nano data centres, cloudlets, etc.) having heterogeneous resources, communication standards, operating systems, etc. Furthermore, their processing power, storage and networking capabilities are limited compared to Cloud resources. Thus, microservices-based Fog application placement approaches have to address challenges such as distributed placement of ap-

plication microservices, latency and failures in communication among distributed microservices and their dynamic composition through efficient service discovery and load balancing mechanisms.

• *Heterogeneous IoT application services*: IoT application services can consist of multiple microservices with heterogeneous resource requirements in terms of processing power, memory, storage, etc. Moreover, these services can have heterogeneous QoS requirements (i.e., latency, throughput, reliability, etc.). Thus, optimal use of limited Fog resources in a QoS-aware manner is a significant challenge where batch placement approaches need to be developed. With batch placement approaches, a set of applications are considered for placement simultaneously, which helps to prioritise services dynamically based on their QoS requirements. While MSA enables these heterogeneous characteristics to be captured at a very granular level providing more scope for improving placement QoS-aware placements, the complex interaction patterns among microservices become a significant challenge to overcome. To this end, developing efficient and intelligent batch placement approaches that dynamically utilise Fog and Cloud resources is an important challenge to satisfy diverse application service requirements.

• *Uncertain failures and lack of dependability*: Even though Fog computing reduces service latency and network congestion, Fog environments have lower dependability compared to Cloud servers. Fog nodes are highly prone to hardware and software failures of the nodes, network failures, and power failures, which ultimately result in service unavailability. Due to the latency-sensitive nature of the services deployed within Fog environments, it is challenging to maintain service availability and reliability requirements. MSA aims to improve failure resistance through the distributed deployment of modular microservices, thus enabling fault isolation. However, it increases the possible points of failure. Also, the complex interaction patterns among microservices result in cascading and correlated failures that degrade the performance of composite IoT services. Thus, accurate reliability modelling in the context of Fog and MSA, and developing reliability-aware placement approaches are important challenges to achieving the performance requirements of the IoT application services.

• *Interoperability and federation:* As Fog environments are distributed and resource-constrained, Fog node clusters interact with adjacent Fog clusters (provided by multiple

Fog infrastructure providers) or Cloud data centres to execute application services. To enable such application placement, it is necessary to develop techniques to run placement algorithms across multiple Fog and Cloud environments that are geographically separated and provided by multiple infrastructure providers. Moreover, applications developed using MSA require microservice composition-related cross-cutting functions (i.e., service discovery, load balancing, security, networking and monitoring, etc.) to be extended across federated Fog-Cloud environments to maintain seamless connectivity among interacting microservices. Although the concept of Fog federation offers better application performance, its adaptation for application placement is hindered by the lack of frameworks and standards addressing these challenges to enable multi-fog multi-cloud integration.

1.3 Research Questions and Objectives

In smart systems, a large number of geo-distributed IoT devices and application users interact with Fog and Cloud environments to access data analytics services provided by microservices-based IoT applications placed within them. Application placement within Fog environments is mainly determined by service quality expectations of the application users, SLAs negotiated between application service providers and infrastructure providers, and resource offerings of Fog and Cloud infrastructure providers. Thus, this thesis investigates the placement of microservices-based IoT applications from the perspective of different entities interacting with IoT systems. The objective of this thesis is to create algorithms and systems for optimal placement of microservices to meet the QoS requirements of IoT applications within Fog computing environments. To achieve these objectives, we solve the application placement problem by addressing the following research questions:

- Q1. *How to utilise microservice characteristics for the efficient placement of IoT applications across resource-constrained and heterogeneous Fog resources?* Rapid growth in the number of connected devices and IoT application users increases the demand for Fog resources to satisfy the stringent latency requirements of latency-critical IoT services and reduce the burden on core-network caused by bandwidth-hungry

services. However, this is limited by the resource-constrained and heterogeneous nature of the distributed Fog devices. Hence, it is vital to exploit the MSA characteristics that align with Fog computing architecture. To this end, distributed placement approaches need to be developed to use the independently deployable and scalable nature of the microservices, along with their lack of centralised management to dynamically deploy horizontally scaled microservices, thus improving the utilisation of distributed Fog resources. Furthermore, this requires service discovery and load-balancing approaches to enable the dynamic composition of microservices deployed across distributed Fog resources.

- Q2. *How to utilise the fine-grained nature of the microservices to improve the QoS satisfaction of heterogeneous IoT application services?* As the number and diversity of IoT application services grow, the optimal use of Fog resources becomes a critical challenge. MSA decomposes applications into granular components that communicate to create composite services with heterogeneous QoS requirements. Thus, batch placement approaches can be developed to dynamically place microservices across Fog and Cloud resources based on the QoS requirements defined at the composite service level. Such placement approaches need to consider complex interaction patterns among microservices and multiple QoS parameters such as makespan, budget and throughput and facilitate dynamic placement across Fog and Cloud to ensure optimum usage of computing and network resources, which makes the placement problem more complex. Hence, developing efficient and robust placement algorithms that can traverse large solution spaces is necessary to reach resultant placements with improved QoS satisfaction.

- Q3. *How to improve the reliability of mission-critical IoT application services through the proactive redundant placement of microservices ?* IoT applications such as smart healthcare, intelligent transportation and IIoT provide highly safety-critical and mission-critical services with high-reliability requirements. Due to their low or ultra-low latency requirements, proactive fault-tolerant methods are required to improve service availability, especially considering the low dependability of Fog resources. Due to their independently scalable nature, microservices can support

reliability-aware redundant placements. However, factors such as complex interaction patterns among microservices, resource limitations in Fog nodes, and increased deployment cost due to redundant deployments restrain such placements and increase the complexity of the placement problem. Thus, it is essential to develop robust placement algorithms that can use the proactive redundant placement of microservices to ensure the reliability requirements of mission-critical services while adhering to the above constraints.

- Q4. *How to utilise federated Fog and Cloud resources through distributed placement and dynamic composition of microservices?* Federated Fog Computing can overcome the resource limitations of Fog devices and provide ubiquitous access to IoT service users. Furthermore, due to the loosely coupled nature of microservices, it is possible to place them across distributed computing resources, thus reaping the benefits of Fog federation. However, existing literature lacks frameworks that can enable scalable microservice placement and dynamic composition across multiple Fog and Cloud clusters. To this end, the development of scalable and extensible Fog computing frameworks is essential to support the integration and execution of placement algorithms in a distributed manner within multi-cloud multi-fog environments. Moreover, such frameworks need to facilitate dynamic microservice composition (i.e., service discovery, load balancing) among microservices that span multiple Fog and Cloud clusters to improve the QoS of the applications placed within federated Fog environments.

1.4 Thesis Contributions

This thesis makes the following contributions to address the research problems mentioned above:

1. It proposes different taxonomies on the placement of microservices-based IoT applications in Fog computing environments and reviews the existing placement approaches.

2. It investigates a placement approach that utilises the independently deployable

and scalable nature of the microservices to minimise the latency and network usage of IoT applications while ensuring optimum usage of heterogeneous and resource-constrained Fog devices (addresses Q1).

- A distributed placement technique for processing application placement requests.
- A microservice placement algorithm to place latency-critical and bandwidth-hungry microservices as close as possible to the network edge by efficiently utilising the horizontal scalability of microservices.
- A Fog node architecture to support decentralised placement along with service discovery and load balancing.

3. It proposes a batch placement technique for QoS-aware placement of heterogeneous IoT applications to satisfy makespan, budget and throughput requirements of application services while ensuring optimum resource usage through collaboration among Fog and Cloud resources (addresses Q2).

   - Formulation of Fog application placement problem as a Lexicographic Combinatorial Optimisation Problem considering QoS satisfaction (in terms of makespan, budget, and throughput) as the primary objective and optimum resource usage as the secondary objective.
   - Exploitation of microservice fine-granularity by incorporating throughput-aware horizontal scalability and service-level QoS definitions into the problem formulation.
   - An improved meta-heuristic technique based on Set-based Comprehensive Learning Particle Swarm Optimisation (S-CLPSO) for batch placement of IoT applications.
   - Multiple heuristic techniques to improve the convergence speed of the meta-heuristic algorithm and avoid premature convergence to reach the global optimum solution.

4. It puts forward a batch placement technique for proactive redundant placement of microservices to meet reliability requirements of mission-critical IoT application

services under resource constraints of the Fog devices (addresses Q3).

- It models the reliability of microservices-based application services as *k out of n* serial-parallel systems and formulate the placement problem and capture reliability, throughput requirements, and cost at the composite service level.
- The problem formulation captures both independent and correlated failures within Fog environments.
- A hierarchical placement approach consisting of two optimised meta-heuristic algorithms (based on S-CLPSO and NSGA-II) to place microservice replicas within Fog environments proactively.
- Multiple approaches, including novel heuristic techniques and reliability-aware fitness functions, to improve the meta-heuristic algorithms to achieve faster convergence to reach the optimum solution.

5. It designs and develops a framework for scalable placement of microservices within federated Fog computing environments (addresses Q4).

- A scalable and extensible framework for deploying and managing microservices-based IoT applications within federated Fog and Cloud environments.
- A configurable software component (developed as a microservice) for processing application placement requests.
- Deployment architectures for the major components of the framework to ensure their scalable and fault-tolerant deployment across federated Fog and Cloud environments.
- Implementation and integration of placement algorithms with the software framework using practical implementation.
- A proof-of-concept prototype of the framework and evaluation of the framework's main features within a multi-fog multi-cloud setup.

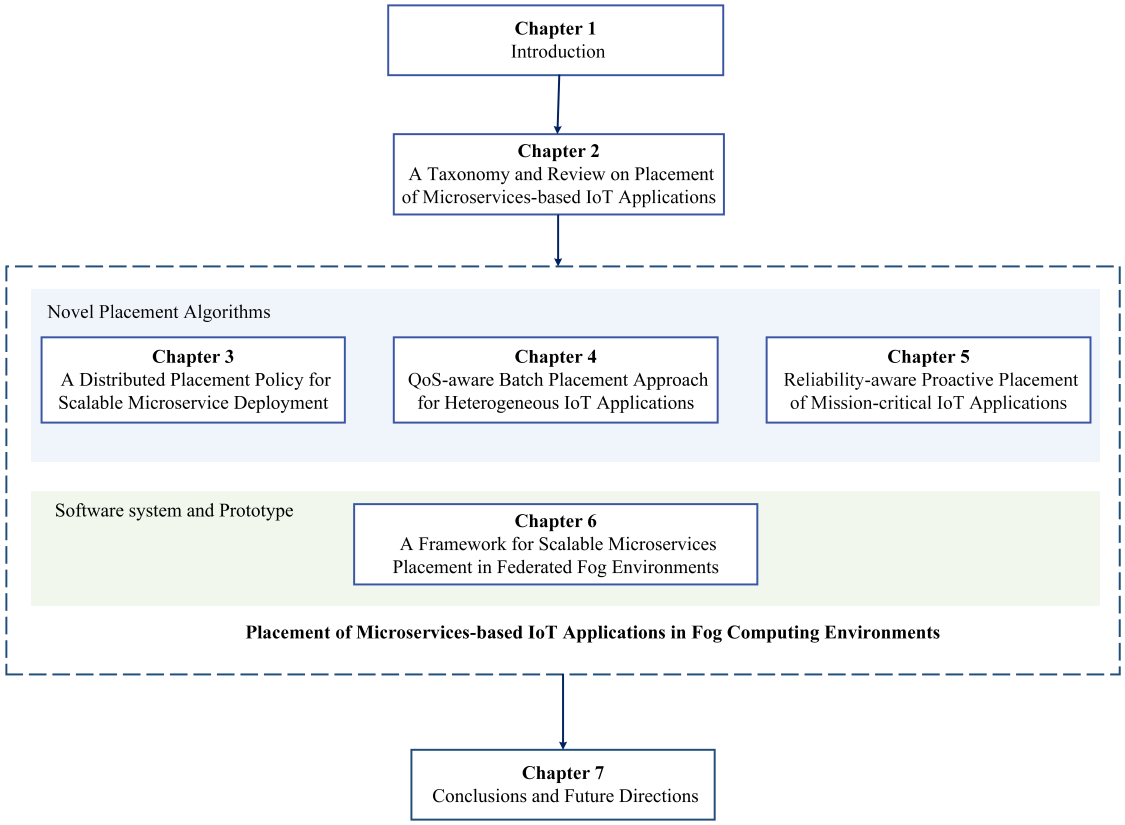


Figure 1.6: The thesis structure

1.5 Thesis Organization

The structure of this thesis is shown in Figure 1.6. The remaining part of this thesis is organized as follows:

- Chapter 2 presents a taxonomy and literature review on the placement of microservices-based IoT applications in Fog computing environments. This chapter is derived from:

  - **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Placement of Microservices-based IoT Applications in Fog Computing: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)*, Volume 55, No. 14s, Article 321, ISSN: 0360-0300, December 2023.

- Chapter 3 presents a distributed placement policy for scalable placement of mi-

croservices across Fog and Cloud computing resources to minimise the service latency and overall network usage. This chapter is derived from:

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Microservices-based IoT Application Placement within Heterogeneous and Resource Constrained Fog Computing Environments", *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, Pages: 71-81, Auckland, New Zealand, December 2-5, 2019.

- Chapter 4 presents an efficient batch placement policy for QoS-aware placement of IoT application services with heterogeneous QoS requirements in terms of makespan, budget and throughput. This chapter is derived from:

  - **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "QoS-aware placement of microservices-based IoT applications in Fog computing environments", *Future Generation Computer Systems (FGCS)*, Volume 131, Pages: 121-136, ISSN: 0167-739X, June 2022.

- Chapter 5 presents a technique for proactive redundant placement of microservices to improve the reliability satisfaction of mission-critical IoT application services in a throughput and cost-aware manner. This chapter is derived from:

  - **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Reliability-aware Proactive Placement of Microservices-based IoT Applications in Fog Computing Environments", *IEEE Transactions on Mobile Computing (TMC))*, 2022 (revision, August 2023).

- Chapter 6 proposes a software framework to enable placement and dynamic composition of microservices across multi-fog multi-cloud environments to harvest distributed and resource constrained Fog resources. This chapter is derived from:

  - **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "MicroFog: A Framework for Scalable Placement of Microservices-based IoT Applications in Federated Fog Environments", *Journal of Systems and Software*, 2023 (revision, June 2023).

- Chapter 7 concludes the thesis by summarising the findings and identifies future research directions.

Chapter 2

A Taxonomy and Review on Placement of Microservices-based IoT Applications

This chapter investigates the existing techniques in Fog computing for the placement of microservices-based IoT applications and reviews them under four aspects, namely, modelling microservice architecture, designing placement policies, incorporating microservice composition-related features, and performance evaluation. After conducting an in-depth literature analysis, separate taxonomies for each aspect of placement of microservices-based IoT applications in Fog computing environments are proposed. Next, a comprehensive survey of existing approaches is conducted according to the proposed taxonomies. Finally, the research gaps are identified and discussed in detail for further improvement of the application placement in Fog computing environments.

2.1 Introduction

Microservice architecture is becoming exceedingly popular for the design and development of large-scale IoT applications, and the features of the architecture (loosely coupled nature, fine granularity, extensibility, cohesiveness, scalability, etc.) demonstrate immense potential to improve the performance of the applications through their efficient placement within federated Fog-Cloud environments. As a cloud-native application architecture, MSA supports dynamic composition of loosely coupled microservice that

This chapter is derived from:

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Placement of Microservices-based IoT Applications in Fog Computing: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)*, Volume 55, No. 14s, Article 321, ISSN: 0360-0300, December 2023.

can be scaled up/down across geo-distributed Fog environments and distributed across Fog and Cloud data centres to meet the throughput demand of the applications in a QoS-aware manner. Thus, designing efficient and scalable algorithms for the "Microservices-based IoT applications placement in Fog environments" is a vital and challenging research area.

To develop efficient placement approaches, it is paramount to incorporate the characteristics and features of the MSA into the placement problem formulation and also carry out extensive evaluations. However, very few initiatives have been taken to identify different aspects related to designing novel Fog placement policies for microservices-based applications. To this end, we identify four aspects related solving the microservices-based Fog application placement problem, as follows:

• ***Accurate modelling of MSA***: The precise formulation of the placement problem depends on accurate modelling of the application architecture, such that all essential characteristics of the architecture are properly abstracted and captured to support the placement of an extensive range of IoT applications developed following the said architecture. When considering MSA, the fine-grained nature of the microservices and their complex interaction patterns set MSA apart from other application architectures and add novel challenges to the placement problem formulation. Thus, to utilise MSA's full potential and overcome related challenges, accurate modelling of the applications is of paramount importance. This includes the correct depiction of microservice granularity (i.e., number of microservices, microservice heterogeneity and their invocation patterns), service composition (i.e., number of microservices per service and their dataflow patterns), and application composition (i.e., number of heterogeneous services per application and advanced interactions like shared/candidature/3rd party microservices).

•***Developing microservices placement policy***: Developing a novel placement policy includes problem formulation, which contains optimisation metrics, objectives, constraints, etc., and, based on that creating an efficient algorithm to reach an optimum placement. Compared to other application models, the loosely coupled nature and the resultant complex interaction patterns, along with higher dynamism, affects different aspects of the problem formulation, such as QoS granularity (i.e., latency requirements can be defined per composite service or among interacting microservices), QoS-awareness

(i.e., handling competing QoS requirements among multiple composite services in a QoS-aware manner), incorporating the horizontally and vertically scaled placement of microservices across federated Fog, and Cloud environments and load balancing requests among dynamically composed distributed instances [38, 45, 46] to utilise limited resources better. This provides opportunities to create efficient redundant placements, optimum request routing-based placements, locality-aware placements, resource contention-aware batch placements, etc., to meet the expected performance of the services. While these aspects increase the complexity of the placement process, their incorporation in placement policies contributes to optimum and balanced utilisation of distributed, heterogeneous and resource-constrained Fog devices and resource-rich, centralised Cloud resources to improve the performance of heterogeneous IoT application services.

• ***Microservice composition***: The composition of granular microservices to create composite services becomes a critical challenge affecting the application's performance. The main challenges related to this include cross-cutting functions such as service discovery, load balancing, monitoring, networking etc., that come with distributed placement, scalability, elasticity, migration, redundant deployment and failures of microservices. Furthermore, as a cloud-native application architecture that can be extended to the distributed Fog layer, these composition-related functions need to support higher flexibility and dynamism across distributed environments, which separates microservice composition from previous web services-oriented SOA. Thus, container orchestration or choreography frameworks and service mesh technologies are introduced to handle the dynamic changes and maintain interconnections among microservices minimising adverse effects on service performance.

Moreover, knowledge about their capabilities is essential in developing placement policies and, in turn, is vital in creating evaluation platforms (simulators and real-world test beds) to evaluate the performance of the developed policies. To this end, factors such as load-balancing policies, overheads due to dynamic service discovery, ability to have service discovery and load-balancing across multiple computing environments needs to be incorporated into placement problem formulation [43, 47, 48] to achieve efficient placements. Moreover, the evaluation of placement policies should be carried out with

the accurate implementation of these cross-cutting concerns to properly evaluate the placement policies.

• ***Performance evaluation***: Accurate evaluation of placement policies depends on the used Fog framework/platform and the workloads. Due to the lack of commercial Fog service providers providing Fog platforms such as IaaS, PaaS or SaaS, the evaluation of the novel placement approaches is mainly handled by simulations or small-scale Fog computing frameworks developed by researchers. Furthermore, with microservices architecture, simulators and frameworks need to be extended further with container orchestration/choreography support, MSA-related cross-cutting function support through service mesh technologies, distributed monitoring, dynamic placement policy integration, etc. Moreover, the workloads used for evaluations should adequately capture the complex characteristics of the MSA and large-scale IoT applications to achieve accurate evaluations.

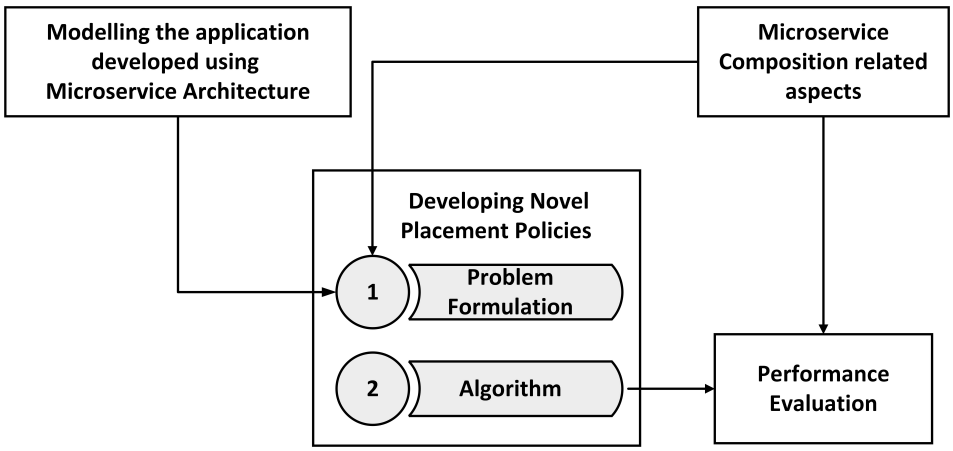


Figure 2.1: Main challenges related to designing novel Fog placement policies for microservice-based IoT applications and their relationships.

Figure 2.1 depicts the relationships among these different aspects to emphasise the importance of their collective consideration to produce efficient placement approaches for microservices-based IoT applications. Thus, it is essential to analyse existing works that focus on microservices placement in Fog environments to create a comprehensive picture of the current status of research and what research gaps remain to be addressed. To this end, we create a taxonomy based on the four aspects discussed above (see Figure

2.2).

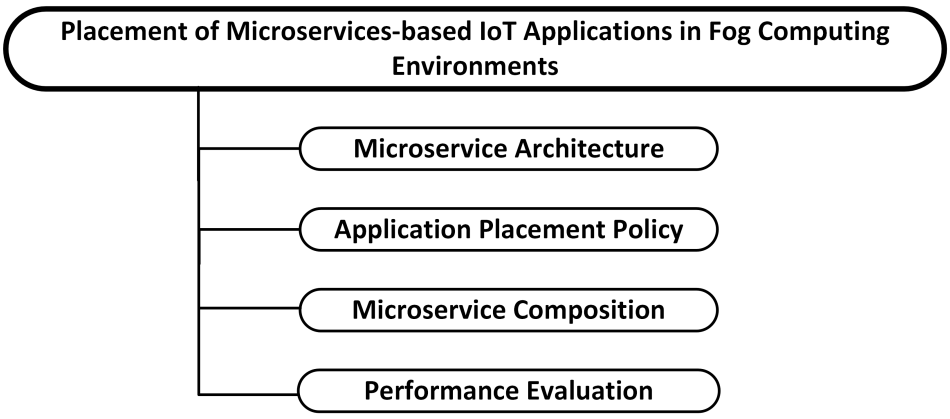


Figure 2.2: Taxonomy for microservices-based IoT applications placement

The rest of this chapter is organised as follows. Section 2.2 presents a qualitative analysis of existing surveys and compares them with ours. The proceeding sections introduce and discuss detailed taxonomies under each aspect identified in our high-level taxonomy: Section 2.3 introduces the taxonomy on modelling IoT applications developed according to MSA and analyses existing works, Section 2.4 presents the taxonomy for placement policy design for microservices-based applications and discusses the current works accordingly, Section 2.5 introduces the taxonomy for microservice composition, and Section 2.6 presents the taxonomy for performance evaluation. Section 2.7 concludes this chapter.

2.2 Related Surveys

We analyse related surveys belonging to three main areas of research; surveys on Edge and Fog computing, Osmotic computing, and MSA, and conduct a qualitative comparison of the aspects covered by each of these surveys to define our contribution. Table 2.1 presents a comparison of the features covered in the surveys.

Existing surveys on Edge and Fog computing cover a wide range of research areas, such as resource management, application placement, and application/task scheduling. Surveys on Fog resource management mainly focus on the architecture and character-

Table 2.1: Summary of existing surveys

	Research Areas						Discussion on Application Model			MSA specific discussions			Placement Evaluation	
		Edge/Fog Computing					Depth							
Work	MSA	Resource Management	Application Placement	Application/ Task Scheduling	Osmotic Computing	Available	High Level Taxonomy	Architecture Specific Taxonomy	Modelling Application	Microservice Composition	Relates application model to placement problem formulation	Relates microservice composition to placement problem formulation	Fog Platforms	Microservice Workloads
[49]	o	✓	Δ	o	o	o	o	o	o	o	o	o	o	o
[50]	o	✓	o	✓	o	Δ	Δ	o	o	o	o	o	o	o
[51]	Δ	o	✓	o	o	✓	✓	o	o	o	o	o	o	o
[52]	o	Δ	✓	o	o	✓	✓	o	o	o	o	o	o	o
[53]	o	o	Δ	✓	o	✓	✓	o	o	o	o	o	Δ	o
[34]	o	o	✓	o	o	Δ	o	o	Δ	0	Δ	o	o	o
[35]	o	o	✓	o	o	✓	✓	o	o	o	o	o	Δ	o
[54]	o	Δ	Δ	Δ	o	✓	✓	o	o	o	o	o	o	o
[22]	✓	o	Δ	o	Δ	✓	o	Δ	Δ	Δ	o	o	o	o
[28]	✓	o	o	o	o	✓	o	o	Δ	Δ	o	o	o	o
[55]	✓	o	o	o	o	✓	o	Δ	Δ	Δ	o	o	o	o
[56]	✓	o	o	o	✓	Δ	o	o	o	Δ	o	o	o	o
[57]	✓	o	o	o	✓	o	o	o	o	o	o	o	o	o
Our	✓	o	✓	o	Δ	✓	o	✓	✓	✓	✓	✓	✓	✓

✓: Discussed, Δ: Partially Discussed, o: Not Discussed

istics of underlying Fog infrastructure (i.e., virtualisation, tenancy, etc.) and algorithms used in administrative operations, including device discovery, monitoring, performance benchmarking, load distribution, and auto-scaling. Fog application placement-related surveys comprehensively analyse the broad range of research work that maps applications to Fog resources to meet their non-functional requirements. In contrast, surveys on Fog application/task scheduling discuss distributing and sequencing ephemeral tasks (i.e., independent tasks or dependent tasks of the application arranged as workflows) for execution within Fog resources. These surveys also discuss works related to the computation-offloading problem in Edge and Fog computing. Thus, scheduling problem considers tasks or workflows with ephemeral life cycles that are deployed for use by a particular user, whereas Fog application placement considers deploying and managing applications with perpetual life cycles with application services accessed by a large number of users.

Hong et al. [49] focus on resource management and, thus, study and classify the architecture of Edge and Fog platforms and algorithms designed for resource management. Their discussions on application placement are limited to analysing how static and dynamic characteristics of the resources are incorporated into placement algorithms for resource allocation. Hence, they do not focus on application architecture-related

characteristics and identifying their effect on efficient placement. Jamil et al. [50] propose a taxonomy of optimisation metrics and algorithms used in resource allocation and task scheduling in Fog computing. However, they do not characterise existing works based on the application models or architectures and fail to analyse optimisation characteristics with respect to them. Brogi et al. [34], Salaht et al. [35], Mahmud et al. [52], and Islam et al. [51] discuss Fog Application Placement Problem (FAPP). In Brogi et al. [34], the main focus is on analysing placement algorithms based on the methodologies of solving the placement problem, their constraints, and optimisation metrics. Their discussion on application models is limited to modelling application module dependencies as constraints of the placement problem formulation. Salaht et al. [35], and Mahmud et al. [52] provide a high-level taxonomy on covering the breadth of Fog application placement. Thus, their classifications of the application models are limited to high-level taxonomies that categorise existing works under monolithic architecture and distributed architectures such as modular and microservices. Islam et al. [51] also provide a high-level taxonomy that classifies applications into monoliths and distributed architecture and limits the discussion on MSA to providing future research directions for microservices-based application placement. Thus, the above works do not capture MSA-related characteristics and challenges related to solving the placement problem of microservices-based IoT applications.

Goudarzi et al. [53] also focus on providing a high-level taxonomy to give a comprehensive and broad overview of IoT application scheduling in Fog environments where the authors discuss general aspects related to scheduling, such as application structure, environmental architecture, optimisation characteristics, decision engine characteristics, and performance evaluation of algorithms. As their survey focuses on providing a high-level taxonomy of each aspect, the taxonomy of the application architecture is limited to a high-level categorisation of applications (monolithic, independent, loosely coupled, etc.). This work introduces microservices architecture as an example of loosely-coupled application structure but does not carry out further discussion on the characteristics of the architecture. An in-depth analysis of MSA is out of scope for this survey, microservice composition-related cross-cutting functionalities (i.e., load-balancing, service-discovery, networking, distributed monitoring and tracing, etc.), tools enabling dynamic

composition (i.e., container orchestrators, service meshes, etc.), incorporation of MSA related features into problem formulation and algorithm evaluation (i.e., Fog computing platforms that support dynamic microservice composition, microservice workloads) are not discussed in their work. Moreover, as the main focus is on scheduling problem, FAPP-related aspects such as throughput-aware horizontal scaling, redundant placements, load-balancing, location-awareness to support distributed users, etc., are not discussed in their work.

MSA-related surveys mainly focus on development, operational phase concerns [22, 55] and challenges related to adaptation of MSA for application development [28]. Joseph et al. [22] provide a broad taxonomy using research work that captures development aspects related to MSA, such as modelling, architectural patterns, maintenance, testing and quality assurance, along with operational aspects, including placement, migration, service discovery and load balancing. Although the work in [22] discusses Osmotic computing and Edge/Fog computing as distributed computing paradigms employing MSA, a categorisation of existing works within these paradigms is not performed. [55] also provides a taxonomy covering all aspects of the microservices lifecycle. Taxonomy provided in [55] is helpful for application developers adapting MSA to design and implement their applications, whereas our survey focuses on large-scale placement of multiple diverse microservices-based applications within distributed Fog environments. Thus, in contrast to [55], we explore how to incorporate microservice characteristics into the Fog application placement problem to generate optimum placements. Razzaq et al. [28] study existing research to identify MSA-related software architectural styles, patterns, models, and reference architectures adapted by IoT systems. However, the placement aspects of such applications are out of the scope of their study. Thus, microservices-related surveys mainly focus on the design, development and maintenance aspects of MSA in general without focusing on challenges related to their placement and deployment within distributed computing paradigms such as Edge and Fog computing.

Osmotic computing-related surveys focus on detailing the concepts and features of the Osmotic computing paradigm along with challenges and future directions [7, 58]. Androcec et al. [57] and Neha et al. [56] conduct a systematic review to capture the

current status of Osmotic computing by analysing applications that follow the Osmotic computing principles, Osmotic computing-related topics addressed and their level of maturity. While these works highlight the concepts and potential of Osmotic computing, they do not analyse existing works based on placement-related aspects, including features and challenges related to modelling, placement, service composition, and evaluation with respect to MSA.

Table 2.1 compares existing surveys with our work by presenting a summary of the key aspects covered. Developing efficient placement approaches for microservices-based applications requires modelling applications to capture MSA-related characteristics, placement policy creation by incorporating the application model-specific features and microservice composition-related characteristics to the placement problem formulation, and proper evaluation of the generated policies using Fog platforms that support microservice composition and microservices-based workloads. Based on the qualitative comparison, existing surveys fail to provide a thorough taxonomy to capture the above. In this work, we propose taxonomies for modelling applications based on MSA, placement policy creation, microservice composition, and performance evaluation and discuss their relationships. Moreover, we identify research gaps with respect to each aspect and propose future research directions.

2.3 Microservice Architecture

To utilise the capabilities of MSA and overcome the challenges of the architecture within Fog environments, proper modelling of the applications is of vital importance so that the placement algorithms can capture all aspects of the placement problem. To this end, Figure 2.3 presents the taxonomy for modelling applications based on MSA where we analyse microservices-based application modelling at multiple levels; granularity at microservice level (***Granularity***), the composition of microservices at the service level (***Service Composition***) and composition of services at the application level (***Application Composition***). Table 2.2 maps the existing works to the proposed taxonomy based on how each research work models the microservices-based applications.

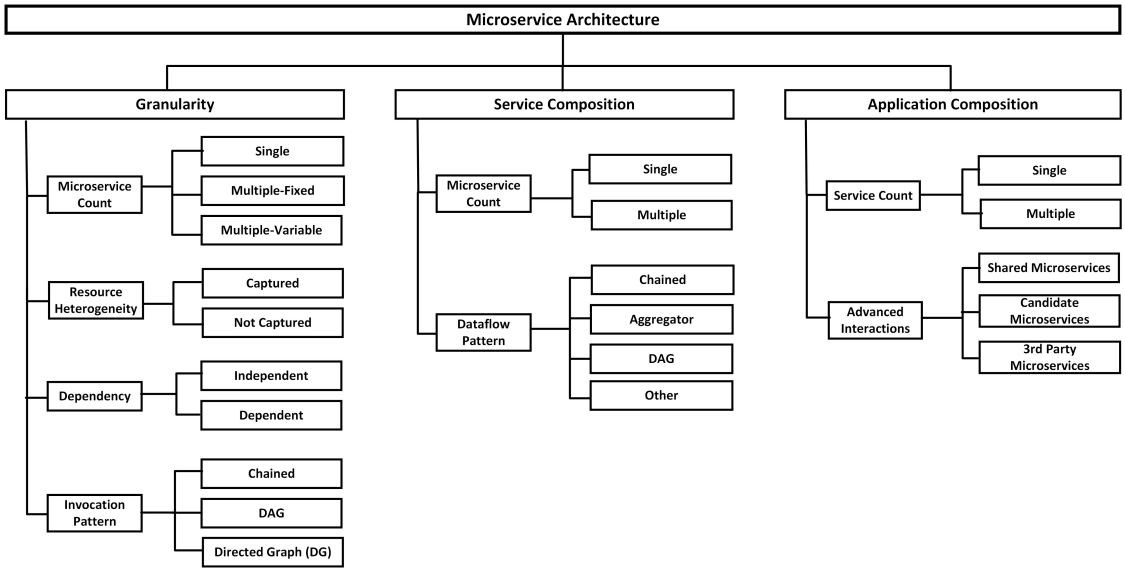


Figure 2.3: Taxonomy for modelling of microservice architecture for placement problem formulation

2.3.1 Granularity

Granularity is one of the most important and challenging aspects of MSA. The fine-grained nature of the microservices allows application services to be depicted as a collection of small components communicating together to perform a certain end-user-requested service. While this allows QoS-improved placement within resource-constrained Fog devices and dynamic movement between federated Fog-Cloud environments following the Osmotic computing paradigm, it introduces complexities in interaction patterns among the microservices. Thus, when modelling the application placement problem, the level of granularity should be captured accurately to overcome the challenges while utilising the advantages introduced by the granular design.

Hence, we analyse the microservice granularity within IoT applications as follows:

1. *Microservice count*: MSA decomposes the application into a set of microservices, increasing the complexity of the placement problem as the number of microservices increases. Thus, existing works capture different levels of granularity based on the number of microservices in the modelled IoT applications. Each application modelled in [61, 67, 68, 72] consists of a single microservice such that the microservice is

designed to perform a specific task requested by the end-user. As an example scenario, [61] introduces an object detection application used by autonomous cars for the detection of other vehicles, pedestrians, road signs etc., that consists of a single microservice for object detection service. To avoid the complexities introduced by having a large number of interconnected microservices, [40] simplifies the placement problem by designing the placement algorithm to handle applications with a fixed number of microservices. The work in [40] proposes the placement algorithm for applications consisting of two microservices: a high throughput microservice that receives data and pe-process it to reduce the throughput, and a low throughput microservice which process the data sent from the first microservice. Works such as [32, 36, 42, 45, 46] remove this constraint and model the application as a collection of any number of microservices, thus providing robust placement algorithms that capture the problem-domain dependent granularity levels of MSA more accurately.

2. *Resource Heterogeneity*: One of the advantages of decomposing applications into microservices is to achieve functional separation where the application is divided into separate modules following the "separation of concern" design pattern. This results in the separation of microservices based on their resource requirements as well (i.e., CPU, GPU, RAM, storage etc.). It's especially advantageous in Edge/-Fog environments where resources are heterogeneous (i.e., Raspberry Pi [1], Jetson Nano [2], Dell PowerEdge XR12 [3], Lenovo ThinkEdge SE50 [4], etc.) and resource-constrained unlike in Cloud environments. Works such as [40, 46, 69] demonstrate this by modelling microservices within the same application to have heterogeneous resource requirements in terms of multiple resource parameters such as RAM, CPU, storage and bandwidth. Works like [65, 75] extend this to include GPU as well, where microservices with GPU requirements may have to be moved to different Fog locations or Fog service providers based on the GPU availability. While the above works represent resource requirements as a vector, some

[1]https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/
[2]https://developer.nvidia.com/embedded/jetson-nano
[3]https://www.dell.com/en-au/work/shop/cty/pdp/spd/poweredge-xr12/aspexr12_vi_vp
[4]https://psref.lenovo.com/syspool/Sys/PDF/ThinkEdge/ThinkEdge_SE50/ThinkEdge_SE50_Spec.pdf

works like [32, 36] simplify the representation by introducing the scalar parameter "Resource Units", with the possibility of extending it to include multiple resource types.

3. *Dependency among microservices*: Microservices are developed as independently deployable units with well-defined business boundaries such that their functionality is exposed to the outside through open interfaces. This allows microservices within applications to communicate easily with each other to create composite services. Works like [61, 62, 64] represent microservices as independent entities without any interconnections among them. In contrast, works such as [36, 59, 60] model them to have dependencies, where microservices communicate with each other through lightweight communication protocols such as REST APIs and message brokers, creating a plethora of IoT services.

4. *Invocation pattern*: The higher level of openness supported by microservices results in different invocation patterns among them, where client microservices invoke other microservices to perform functions required to complete the composite services. Works such as [38, 45, 59] model the invocation relationship as a chained pattern. Works presented in [46, 69, 73] use a more general representation of microservice invocation by modelling it using DAG representation. Works like [41] and [36] model the invocation as directed graphs where the interactions between microservice are depicted using many-to-many consumption relationships.

2.3.2 Service Composition

With granularity comes the concept of composite services, where microservices interact to create services that perform a certain task and provide an output to the user. In this section, we analyse and categorise aspects related to service composition.

1. *Microservice count*: The granularity of the microservices creates end-user services with varying numbers of microservices: "atomic services" consisting of a single microservice and "composite services" that consist of multiple interconnected microservices. Works such as [61, 64] represent each service by a single microservice

that receives requests from the end-user front-end, completes a task and provides the result back to the user without interacting with any other microservices. Other works like [38, 46, 73] model services with multiple microservices that interact together to perform tasks. The work presented in [38] describes a smart city application with a smart policing service used by the police to identify suspects where the service is a composition of three microservices, and [46, 60] model a smart healthcare application with an emergency notification service where multiple microservices interact to detect abnormalities in ECG data streams and raise emergency alarms in real-time.

2. *Dataflow Pattern*: Due to different interaction patterns among microservices, the data flow within composite services can take many forms. Works such as [40, 59, 63] consider the chained composition of microservices, whereas [46, 69] model the services considering chained, aggregator and hybrid dataflow patterns to create composite services. Works such as [65, 66] model dataflow among microservices as DAGs, assuming the absence of cyclic data flows while [44, 70] include cyclic dataflows and represent the dataflows using DGs. The work presented in [74] models the interaction pattern using an Undirected Weighted Graph (UWG) where edges are represented using interaction weights irrespective of the direction of communication.

2.3.3 Application Composition

As the capabilities of IoT applications improve rapidly, they quickly evolve into complex applications covering large business domains due to the flexibility of design and development using MSA. We analyse the aspects related to this as follows:

1. *Service count*: With the increase in connected devices and the generation of diverse data, IoT applications have evolved to provide many services to users. Thus, microservices-based IoT applications are modelled as a composition of one or more services. Works such as [40, 61] define applications with single services, whereas [41, 63, 66] model applications with multiple end-user services with heterogeneous QoS requirements. Works like [46, 60] model a smart health care ap-

plication with two primary services; emergency alarm generation service with stringent latency requirements and a latency tolerant long-term analysis service. Modelling the applications with multiple services allows the placement problem to capture competing QoS requirements within applications and paves the way to propose placement policies that can utilise Fog-Cloud resources more efficiently.

2. *Advanced Interactions*: Due to the fine-grained nature of the microservices, along with well-defined functional boundaries and lightweight communication methods, advanced interactions among microservices are possible. The work in [46] models applications where some microservices (i.e., feature extraction, data cleaning etc.) are part of multiple composite services that have heterogeneous latency requirements. The paper in [45] models the services to have candidate microservices (i.e., online payment gateways) depending on the payment method used by the user. Thus, the application is modelled to have alternative data paths. The work proposed in [72] handles the concept of having alternative paths by defining control structures for each composite service through conditional branching within some of the microservices. Another advanced capability of microservices is using third-party microservices through the use of APIs, which is modelled in the [72] where composite services are created by combining microservices developed by multiple developers, and provided as services by multiple computing service providers.

2.3.4 Research Gaps

Based on the analysis of existing works presented in Table 2.2, we have identified the following gaps related to microservices-based application modelling.

1. Existing works show shortcomings in several aspects in capturing the granularity of the MSA, such as lack of use in generalised invocation patterns (i.e., directed graphs) and capturing resource heterogeneity in different microservices within the same application (i.e., use of GPU, databases etc.). Many works use a chained invocation of microservices without considering complex dependency patterns that can occur due to the openness of microservice design. For capturing resource het-

erogeneity, most works consider one or more parameters such as CPU, RAM and storage. However, GPU or TPU requirements are scarcely considered. With the rise of EdgeAI workloads in IoT, such parameters and the constraints imposed by them play an important role in the placement logic.

2. In service composition, the data flow pattern is not adequately defined and utilised in the placement process of the majority of the works. Many works define chained data flow or acyclic data flows, disregarding cyclic data flows, which affect the end-to-end latency calculations of the services.

3. Under application composition, most works model each application to have a single composite service. Thus heterogeneity in QoS requirements between different composite services within the same IoT application is not appropriately captured. This also hinders the application from having complex interaction patterns and data flow patterns within the applications.

4. In modelling the applications, most works do not consider complex interaction patterns of microservices, including shared microservices among different composite services, candidate microservices and third-party microservices. The placement of shared microservices has to consider multiple, heterogeneous composite services that they are part of and place them so that all non-functional requirements of the services are satisfied under resource contention. For efficient placement of the candidate services, knowledge of their demand and usage is vital. The use of third-party microservices in applications results in security, reliability and performance challenges (i.e., latency, availability, transaction cost etc.) that are out of the control of the application developers, which need to be accounted for during the placement phase to improve performance. Most of the works that consider these only analyse the effect of a single pattern, thus failing to capture the impact of multiple ones.

2.4 Application Placement Policy

MSA introduces novel aspects (i.e., QoS granularity, scalability, lightweight/ independent deployment, etc.) that can be utilised for better placement of the IoT applications while also giving rise to novel challenges (i.e., microservice dependencies, interactions, cascading failures etc.). We consider these MSA-specific effects in addressing the application placement problem and create a novel taxonomy as shown in Figure 2.4. Current works are mapped to the taxonomy to identify gaps and possible improvements (see Table 2.3).

2.4.1 Placement Mode

Placement mode represents the number of placement requests processed by the placement engine during each execution of the placement algorithm. Placement modes are categorised into two groups: sequential mode, where the placement algorithm queues the placement requests to process one after the other, and batch mode, where a set of applications are considered for placement simultaneously. In the works such as [38, 60, 61], the placement engine uses a First In First Out (FIFO) queue to store and process application placement requests sequentially. Some works, like [40, 42] prioritise the applications in the queue based on their resource requirements, whereas [36] and [67] prioritise them based on the deadlines of the applications and order them for sequential placement. Placement engines in [32, 46, 66] are designed using meta-heuristic algorithms (i.e., genetic algorithm, particle swarm optimisation) to handle the complexities of the batch placement and navigate the larger solution space successfully. The work presented in [65] employs a batch placement policy based on a heuristic algorithm to maximise the placement of IoT applications within a multi-domain federated Fog ecosystem under locality constraints imposed on microservices. Some works, such as [47, 48, 64] do not properly define placement mode and carry out the evaluations using the placement of a single application.

2.4.2 Placement Perspective

The placement perspective is identified based on the optimization parameters/objectives considered during the placement and from whose viewpoint they are addressed from. Works like [40, 42, 44] define the placement problem from the perspective of the Fog infrastructure provider, thus aiming to maximise the IaaS provider's revenue. Other works such as [38, 45, 63] define the placement problem from the application provider perspective, where application providers expect the satisfaction of non-functional requirements (i.e., latency, throughput, reliability etc.) of the services provided by their application while ensuring budget constraints related to Fog-Cloud deployment. Some works, such as [41, 59] address this from a user perspective where the user sends a request for an application or service along with performance requirements, and the placement algorithm ensures the availability of the requested service under requested constraints (i.e., latency, throughput etc.) to satisfy the user expectations.

2.4.3 Placement Parameters

In this section, we analyse the characteristics of the parameters considered by the placement policy.

1. *Parameters:* IoT application services have multiple heterogeneous QoS parameters (i.e., latency [38, 43, 45, 70, 74], cost [38, 46, 67], throughput [44, 70, 72], energy [48], reliability [37] etc.) negotiated with the Edge/Fog infrastructure provider in the form of SLA. Moreover, placement decisions made from the infrastructure provider perspective consider the maximisation of the revenue of the IaaS provider. Works such as [40], and [42] formulate the placement problem to maximise the revenue of the Fog provider, whereas [44, 65] consider a federate Fog environment where each provider focuses on minimising the total deployment cost while minimising the number of resources rented from external Fog infrastructure providers and the Cloud. Resource utilisation is another parameter considered in current research where [32, 72, 74] consider optimisation of computation resource usage, [73] optimises network resource usage, while [39, 46] consider both. Placement policies focus on optimising one or more of the above-mentioned QoS pa-

Table 2.2: Analysis of existing literature based on the taxonomy for modelling of microservice architecture

Work	Granularity				Service Composition			Application Composition		
	Microservice count	Resource Heterogeneity	Dependency	Invocation pattern	Microservice count	Dataflow pattern	Service count	Advanced Interactions		
								Shared	Candidate	3rd Party
[59]	Multi-V	Captured (P_C)	Dependent	Ch	Multiple	Ch	NA	-	-	-
[36]	Multi-V	Captured (RU)	Dependent	DG	Multiple	ND	Single	-	-	-
[41]	Multi-V	Captured (P_C)	Dependent	DG	Multiple	NA	Multiple	-	-	-
[40]	Multi-F	Captured (R,P_C,S)	Dependent	Ch	Multiple	Ch	Single	-	-	-
[60]	Multi-V	Captured (R,P_C,S,B)	Dependent	DAG	Multiple	NA	Multiple	✓	-	-
[32]	Multi-V	Captured (RU)	Dependent	DAG	NA	NA	NA	-	-	-
[42]	Multi-V	Captured (R,P_C,S,B)	Dependent	DAG	Multiple	DAG	Single	-	-	-
[61]	Single	NC	Independent	NA	Single	NA	Single	-	-	-
[37]	Mult-V	Captured (P_C, R)	Dependent	ND	ND	ND	ND	-	-	-
[62]	ND	Captured (RU)	Independent	NA	ND	NA	ND	-	-	-
[63]	Multi-V	Captured (P_C)	Dependent	ND	Multiple	Ch	Multiple	✓	-	-
[64]	Single	Captured (P_C)	Independent	NA	Single	NA	Single	-	-	-
[65]	Multi-V	Captured(R,P_C,P_G,S)	Dependent	DAG	Multiple	DAG	ND	-	-	-
[45]	Multi-V	Captured (P_C)	Dependent	Ch	Multiple	Ch	Single	-	✓	-
[38]	Multi-V	Captured (R,P_C,S)	Dependent	Ch	Multiple	Ch	Single	-	-	-
[48]	Multi-V	NC	Dependent	Ch	Multiple	Ch	ND	-	-	-
[47]	Multi-V	Captured (P_C)	Dependent	Ch	Multiple	Ch	ND	-	-	-
[66]	Multi-V	Captured (P_C)	Dependent	DAG	Multiple	DAG	Multiple	-	-	
[44]	Multi-V	Captured (P_C,R,S)	Dependent	DG	Multiple	DG	ND	-	-	-
[67]	Single	NC	Independent	NA	Single	NA	Single	-	-	-
[68]	Single	NC	Independent	NA	Single	NA	ND	-	-	-
[43]	Multi-V	Captured (R)	Dependent	Ch	Multiple	Ch	Multiple	✓	-	-
[69]	Multi-V	Captured (R,P_C,S)	Dependent	DAG	Multiple	Ch, Ag, H	Multiple	✓	-	-
[70]	Multi-V	Captured (P_C, R)	Dependent	DG	Multiple	DG	Single	-	✓	✓
[71]	Multi-V	Captured (P_C, R, B)	Dependent	DAG	Multiple	DAG	ND	-	-	-
[72]	Single	Captured (R,B)	Independent	NA	Single	NA	ND	-	-	-
[39]	Multi-V	Captured (P_C, R, B)	Dependent	DAG	Multiple	DAG	ND	-	-	-
[46]	Multi-V	Captured (R,P_C,S)	Dependent	DAG	Multiple	Ch, Ag, H	Multiple	✓	-	-
[73]	Multi-V	NC	Dependent	DAG	Multiple	DAG	ND	-	-	-
[74]	Multi-V	Captured (P_C,R)	Dependent	NA	Multiple	UWG	NA	-	-	-
[75]	Multi-V	Captured (P_G)	Dependent	DAG	Multiple	DAG	Single	-	-	-
[76]	Multi-V	Captured (RU)	Dependent	NA	Multiple	DG	ND	-	-	-

Mult-F:Multiple-Fixed, Mult-V:Multiple-Variable, Ch:Chain, Ag:Aggregator, H:Hybrid, DAG:Directed Acylclic Graph, DG:Directed Graph, UWG: Undirected Weighted Graph, R:RAM, P_C:Processing power (CPU), P_G:Processing power (GPU), S:Storage, B:Bandwidth, RU:Resource Units, NC:Not Captured, ND: Not Defined, NA:Not Applicable

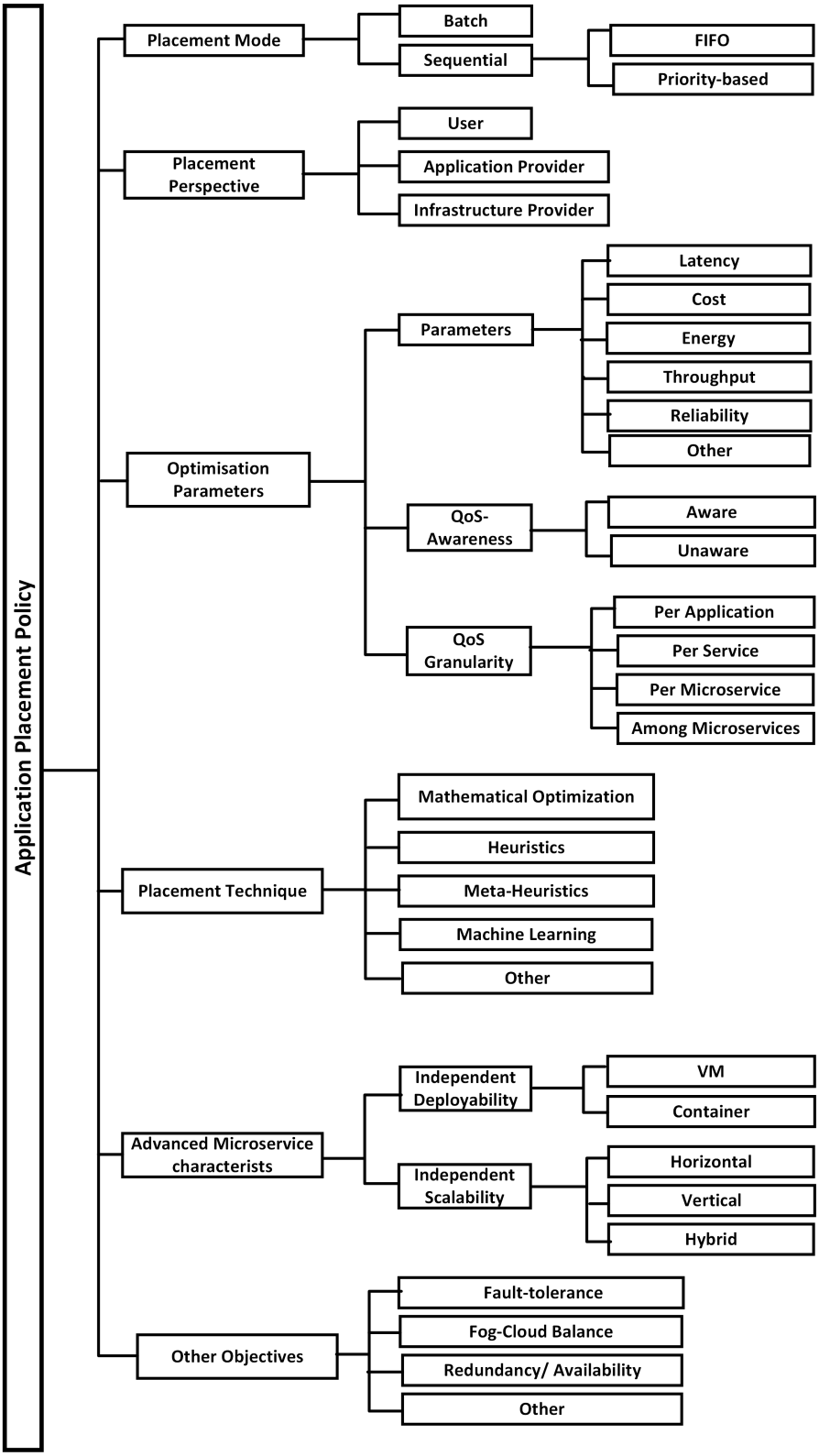


Figure 2.4: Taxonomy for placement policies designed for microservices-based applications

Table 2.3: Analysis of existing literature based on the taxonomy for application placement policy

Work	Placement	Placement	Placement Parameters			Placement	Adv μservice characteristics		Other Objectives
	Mode	Perspective	Parameters	QoS-awareness	QoS Granularity	Techniques	Ind. Deployability	Ind. Scalability	
[59]	Seq-FIFO	UP	Latency	No	per microservice/s (Latency)	Heuristic - Greedy	VMs	NC	-
[36]	Seq-P	AP	Latency, Availability	Yes (Latency)	per application (Latency)	Heuristic	Containers	H	-
[41]	Seq-ND	UP	Latency	No	per service (Latency)	Heuristic	Containers	H	F-C Balance (D)
[40]	Seq-P	IP	Revenue, Latency Throughput	Yes (Latency, Throughput)	among microservices (Throughput) per application (Latency)	Heuristic-Greedy	Containers	NC	F-C Balance (S)
[32]	Batch	AP	Latency, RU_C	No	ND	Meta-Heuristic	Containers	H	Service Spread
[60]	Seq-FIFO	UP	Latency	Yes	per service (Latency)	Heuristic	Containers	H+V	F-C Balance (S)
[47]	ND	AP	Latency, Cost	No	per service	Meta-Heuristic	Containers	H+V	Redundancy
[37]	Seq-ND	AP	Cost, Reliability Latency	Yes (All)	per application (All)	Meta-Heuristic	ND	H	-
[48]	ND	AP	Latency, Energy	No	NC	Meta-Heuristic	Containers	H	Redundancy
[64]	ND	UP	Latency	Yes (Latency)	per microservice (Latency)	Machine Learning	Containers	H	Proactive Scaling
[62]	Batch	UP	Energy, Latency Cost	Yes (Latency)	per microservice (Latency)	Mathematical Programming (Lagrangian Multiplier)	Containers	H	Fault-tolerance
[66]	Batch	AP	Latency, Cost	Yes (Latency)	per service (Latency)	Meta-Heuristic	ND	NC	Fault-tolerance
[63]	Seq-ND	AP	Energy, Latency	No	NC	Meta-Heuristic	VMs	NC	-
[65]	Batch	IP	Revenue, Latency, Throughput	Yes (Latency, Throughput)	among microservices (Latency, Throughput)	Heuristic	Containers	NC	Locality-awareness
[42]	Seq-P	IP	Revenue, Latency Throughput	Yes (Throughput, Latency)	among microservice (Throughput) per application (Latency)	Heuristic-Greedy	Containers	NC	F-C Balance (D)
[61]	Seq-FIFO	AP/UP	Latency, Cost	Yes (Latency)	per application (Latency)	Reinforcement Learning	Containers	NC	Mobility-awareness
[44]	Seq-ND	IP	Latency, Throughput	Yes (All)	among microservices (All)	Heuristic	Containers	H	F-C Balance (D)
[67]	Seq-P	UP+AP	Latency, Cost	Yes (Latency)	per microservice (Latency)	Heuristic - Greedy	Containers	NC	F-C Balance (D)
[69]	Seq: ND	AP	Latency, Cost	Yes (Cost)	among microservices (Latency) per application (Cost)	Heuristic - Greedy	Containers	H	-
[72]	Seq-ND	AP	Latency, RU_C Throughput	Yes (Throughput)	per microservice (Throughput, Latency)	Analytical Hierarchy Process (AHP)	Containers	NC	-

Work	Placement	Placement	Placement Parameters			Placement	Adv μservice characteristics		Other Objectives
	Mode	Perspective	Parameters	QoS-awareness	QoS Granularity	Techniques	Ind. Deployability	Ind. Scalability	
[71]	ND	IP+UP	Latency, Throughput Cost, RU_C, RU_N	Yes (Latency, Throughput)	per application (Latency, Throughput)	Approximate Algorithm + Deep Reinforcement Learning	Containers	H	Load-awareness Resource contention
[46]	Batch	AP	Latency, Cost Throughput, RU_C, RU_N	Yes (All)	per service (All)	Meta-Heuristic	Containers	H+V	F-C Balance (D)
[73]	Batch	UP	Latency, RU_N	Yes (Latency)	per service (Latency)	Meta-Heuristic	Containers	H	Location-awareness
[74]	ND	AP	Latency, RU_C	No	NC	Deep Reinforcement Learning + Heuristic	Containers	H	Load-awareness
[38]	Seq-FIFO	AP	Cost, Latency	Yes (Latency)	per service (All)	Mathematical Programming (Branch and Bound)	Containers	H+V	-
[45]	Seq-FIFO	AP	Latency	No	per service/application (Latency)	Monte carlo + Meta-heuristic	Containers	H	Availability
[43]	Batch	IP	Latency	No	per service (Latency)	Mathematical Programming (Gurobi MILP solver)	ND	H+V	Optimal routing
[70]	Seq-ND	UP	Latency, Throughput	Yes (Latency, Throughput)	among microservices (Latency, Throughput)	Mathematical Programming	VMs	H	Redundancy
[39]	ND	IP+UP	Latency, Throughput Cost, RU_C, RU_N	Yes (Latency, Throughput)	per application (Latency, Throughput)	Approximate Algorithm + Deep Reinforcement Learning	Containers	H	Load-awareness Resource contention
[75]	ND	UP	Latency, Throughput	Yes (Latency)	per service (Latency)	Heuristic-Greedy	ND	H	Resource contention
[77]	Batch	IP + UP	Latency, Energy Throughput	Yes (Latency, Throughput)	among microservices (Latency, Throughput)	Meta-heuristic	Containers	H	Fault-tolerance
[76]	Batch	UP	Latency	No	among microservices	Meta-Heuristic	Containers	NC	-

FIFO:First-In-First-Out, P:Prioritised, UP:User Perspective, AP:Application provider Perspective, IP:Infrastructure provider Perspective, RU_C:Computation Resource Utilisation, RU_N:Network Resource Utilisation, ND:Not Defined, NC:Not Considered, H:Horizontal, V:Vertical, F-C Balance(D):Dynamic Fog-Cloud Balance, F-C Balance(S):Static Fog-Cloud Balance

rameters. Some works, such as [41, 59, 60] formulate the placement problem considering the satisfaction of a single parameter and focus on minimising the latency of the services deployed within Fog environments. Other works like [37, 62, 66] focus on multiple parameters to reach a trade-off between conflicting parameters, such as latency, cost, energy etc., by formulating the placement problem as a multi-objective optimisation problem.

2. *QoS-awareness:* Reduction of service latency and core network congestion are two of the main objective of Edge/Fog computing paradigms. As a result, works like [41, 45, 47] aim to place the microservices as close as possible to the user to reduce the overall latency of the application. While this is done in a QoS-unaware manner, assuming that all services require low latency, other research works such as [37, 46, 65, 73] consider the QoS requirements of the IoT services before placing them. Such approaches highlight one of the main limitations of Edge/Fog computing which is its resource-constrained nature compared to the Cloud data centres. Thus, they aim to capture the QoS heterogeneity of the IoT services and prioritise them based on their QoS requirements (i.e., stringent latency requirements over latency-tolerant services [65, 73], stringent budget constraints over higher budget availability [37, 46], throughput-aware placement/scaling of microservices [40, 46]) and achieve a proper balance between Fog and Cloud resource usage in a QoS-aware manner. MSA further enhances this behaviour due to the ease of moving independently deployable microservices across Fog and Cloud data centres dynamically.

3. *QoS Granularity:* Because of the fine-grained nature of the microservices and their possible composition patterns, QoS parameters are defined at different levels of granularity. Among existing works, QoS parameters are defined at three primary levels: microservice level, composite-service level and application level. Research works like [40, 42] define the throughput requirements at the microservice level, where it is presented as the bandwidth requirement between interacting microservices, whereas [59, 62, 64] define latency requirements per each microservice. Several works such as [41, 46, 47] define latency, throughput, cost etc., per

each composite service where microservices-based applications can consist of one or more such services. Works such as [42, 61] define latency requirement per each application, assuming that the application performs a single function/service requested by the end-user. Most of the works define all QoS parameters at a single level, whereas [40, 42] use different granularity levels based on the parameter (i.e., throughput at the microservice level and latency at the application level).

2.4.4 Placement Techniques

Different placement techniques are chosen to implement the placement policies based on the complexity of the modelled microservices-based applications, placement mode, optimisation parameters and their granularity, and the dynamism of the considered scenario (in terms of distributed resources, workload, failures etc.). Works such as [38, 43, 62, 70] use mathematical programming to solve microservices-based application placement in Fog: [43] formulates the problem using Mixed Integer Linear Programming (MILP) and solve it using Gurobi MILP solver [5], [62] uses the Lagrangian multipliers to solve the optimisation problem under multiple constraints including network QoS, price, and resource usage. Several works, including [41, 59, 60] propose heuristic placement algorithms to optimise a single placement parameter such as latency. Heuristic approaches are proposed by works such as [36, 40, 65] that consider multiple parameters as well. Here, [40], and [65] formulate the placement problem as an Integer Linear Programming problem to solve it using greedy heuristic algorithms. [32, 66, 73] that try to handle batch placement scenarios and multiple placement parameters, propose placement policies based on evolutionary meta-heuristic algorithms such as Genetic Algorithm (GA), Particle Swarm Optimisation (PSO), Ant Colony Optimisation (ACO) to navigate large solution space efficiently. With increased computation power available for placement engines, algorithms are moving towards Machine Learning (ML): [64] uses ML-based forecasting models to make predictive auto-scaling decisions, and [39, 61, 74] use Reinforcement Learning based approaches to tackle the highly dynamic nature of Edge/Fog environments and the microservices. Moreover, existing works use other techniques

[5] https://www.gurobi.com/products/gurobi-optimizer/

such as Analytical Hierarchy Process (AHP) [72], which is a powerful decision analysis method used to prioritise and balance multiple criteria, and computation algorithms such as Monte Carlo method [45] used to capture uncertainties in the placement problem introduced by the MSA (i.e., candidate microservices, 3rd party microservices, etc.).

2.4.5 Advanced Microservice Characteristics

The popularity of the MSA for deployment within highly distributed Fog environments stems from two primary microservices-related characteristics, their independently deployable and scalable nature.

1. *Independent Deployability:* Decomposing monolithic applications into a collection of microservices can improve deployment-related aspects such as fault tolerance and reliability due to the independently deployable nature of the microservices. This is further supported by containerisation, which provides a lightweight method for deploying microservices compared to the earlier used Virtual Machines(VMs). The work in [59] proposes a deployment scenario where all microservices of a user-requested service are mapped onto a single VM. The vast majority of existing works [32, 36, 48, 65] improve the deployment through the use of containers (i.e., Docker), which allows rapid deployment by providing microservices with lightweight, isolated environments. The use of containers enables fast deployment of microservices while mitigating single point of failure using the distributed placement of microservices. Works such as [60, 62] highlight the importance of using container technology due to their faster spin-up times and analyse resultant performance improvements under dynamic conditions. Works such as [38, 46] consider the cost of the containerised microservices by adapting the pricing models used in container platforms provided by the commercial Cloud providers (i.e., Amazon Fargate, Azure Containers).

2. *Independent Scalability:* As microservices are independently deployable units with well-defined business boundaries, each microservice can be independently scaled to meet the throughput requirements. This is especially advantageous in Edge/-Fog computing scenarios where devices are heterogeneous in their resource ca-

pacities, workloads also vary rapidly with the popularity of the services, mobility of the users etc. Scalability of the microservices is used in multiple ways where some works consider only horizontal scalability while others use a combination of horizontal and vertical scalability to satisfy throughput requirements. The paper in [32] uses horizontal scalability of the microservices to spread instances of each microservices uniformly across the Fog landscape to support distributed access by users. Works such as [46, 60] consider resource heterogeneity of the Fog devices and use a hybrid approach of vertical and horizontal scalability to meet the throughput requirement. Current works use the scalability of the microservices for multiple purposes. Above works focus on satisfying the overall throughput requirement while utilising resource-constrained Edge/Fog devices. Meanwhile, the work presented in [73] uses horizontal scalability to incorporate location awareness, whereas [45, 62] use horizontal scalability to achieve redundancy which improves availability and fault-tolerance of the application.

2.4.6 Other Placement Objectives

One of the main advantages of using MSA for Fog application development is the ability to easily utilise both Fog and Cloud resources to meet application requirements. This concept is also put forth in Osmotic computing, which highlights the importance of dynamically moving microservices between Cloud and Fog and achieving an equilibrium such that the non-functional requirements of the services are met. In some works such as [40, 60], this is handled in a static manner where microservices to be placed on Cloud are predefined per each application based on the deadline requirement of the composite services each microservice belongs to. Works such as [41, 42, 44, 46] handle this in a dynamic way: [42] partitions the application based on the throughput requirement to place microservices with higher throughput requirements in the Fog and rest in the Cloud, [41] analyses the popularity of the microservices based on the user requests to move less popular ones to the Cloud, [44] achieves balance by considering the cost of deployment in Fog and Cloud, [46] formulates a multi-objective problem to achieve a trade-off between Fog device usage and network usage to achieve a balance between

Fog and Cloud resource usage dynamically.

Locality or location awareness is another aspect that comes with IoT applications due to data security and latency requirements. Sensitive IoT data (i.e., healthcare, security camera footage etc.) can have constraints on geographical locations for the processing, which requires certain microservices to be placed within certain regions or within certain Fog service providers in federated Fog computing environments. The paper in [65] adds locality constraint to their problem formulation and handles it using a heuristic approach. With the distributed nature of the Edge/Fog resources and the users, location awareness can be used to minimise network resource usage and delay [32, 73]. The work presented in [73] utilises this concept where the user's location is used to select the microservice instances such that the requests are routed to the closest instances. The paper in [32] introduces a metric called service spread which evenly distributes microservices across the Fog landscape to improve performance when users are uniformly distributed.

Modularity and scalability of microservices pave the way for efficient redundant placement to improve the availability and fault-tolerance of the services. The work presented in [45] proposes a redundant placement policy for composite services consisting of chained microservices, considering the uncertainty of requests and heterogeneity of the resources. This work aims to reduce the outage time under failures through redundant placement of microservices. To address the challenge of cascading failures that occur in MSA, [66] proposes a fault tolerance method for applications deployed using the API gateway pattern where the API gateway acts as the access point for requests coming from the users. This work deploys API gateways within cache-enabled edge nodes so that data related to service requests can be cached proactively in case downstream microservices become unavailable due to failures until the microservice is redeployed using a reactive fault-tolerance approach.

Due to the horizontal scalability of microservices, proper load balancing and routing approaches are required to ensure performance requirements. To address this, some works such as [43, 47, 48] formulate the application placement problem as a combination of microservice replica placement and determining the optimal data flow path of composite services. The work in [47] uses minimisation of service latency and cost (both computation and data transfer costs) to achieve the optimum level of service replication

and identification of the request path that minimises latency of the composite services. The paper in [48] considers latency and power consumption as optimisation parameters to achieve this. The work presented in [43] extends this further by incorporating SDN controller placement as part of the problem, where SDN controllers are used for service discovery and determining data flow.

Deploying multiple lightweight containers onto the same Fog device can result in resource contention depending on the resource requirements of each container instance. To overcome this, several works like [39, 71, 75] propose online algorithms to detect shared-resource contentions and afterwards dynamically adjust resource allocations or migrate microservice instances to other idle nodes to maintain the required level of performance. These works capture different types of resource contentions, including I/O [39], GPU global memory bandwidth [75], and computation capacity [71] contentions.

2.4.7 Research Gaps

Based on the analysis of existing works presented in Table 2.3, we identified the following gaps related to microservices-based application placement.

1. QoS-granularity and QoS-awareness, together with proper placement mode selection, can improve performance, especially when microservices-based IoT applications are considered. As IoT applications grow in complexity to provide many services with heterogeneous QoS requirements, the granularity of the microservices paves the way for per-service QoS definitions. Together with batch placement or sequential placement with QoS-aware prioritisation, this approach is rarely explored in existing works. The following shortcomings are apparent in existing works;

    - QoS-granularity: Most of the works define QoS requirements among interacting microservices (i.e., latency and throughput requirements between two microservices) which becomes less feasible due to composite services with complex interaction patterns and their agile evolution as applications evolve. As a result, existing works scarcely capture the QoS heterogeneity of the composite services within the same application. Thus, MSA-related scenarios like

competing requirements that exist due to shared microservices are not handled.

- QoS-awareness: Many works do not define QoS requirements and capture the heterogeneity but try to minimise overall latency, cost etc. This approach hinders the proper utilisation of limited Edge/Fog resources and limits achieving equilibrium between Fog-Cloud resource usage.

The above gaps are tightly coupled with how the microservices-based application is modelled, and overcoming them requires the proper capturing of MSA as described in Section 2.3.

2. In existing works, there's scope for incorporating advanced microservice characteristics such as the independently deployable and scalable nature of the microservices. Although many of the works use container technology for microservice deployment, they lack proper utilisation of the fast spin-up time, lightweight deployment capabilities and related cost models, whereas, for independent scalability, only a very few works explore the combination of horizontal and vertical scalability to support throughput requirements of the composite services. This can further consider the scalability constraints of different microservices (i.e., microservices with databases).

3. Placement objectives of the current works lack emphasis on the following: security challenges related to microservice, utilising microservices for improvement of reliability and fault tolerance, dynamic microservice placement under federated Fog architectures, dynamic Fog-Cloud balance, mobility-aware placement, consideration for load uncertainty, etc.

2.5 Microservice Composition

The complex interaction patterns among microservices, their ability to independently scale up/down to maintain performance and distributed deployment of microservice instances across networked devices are supported through microservice composition

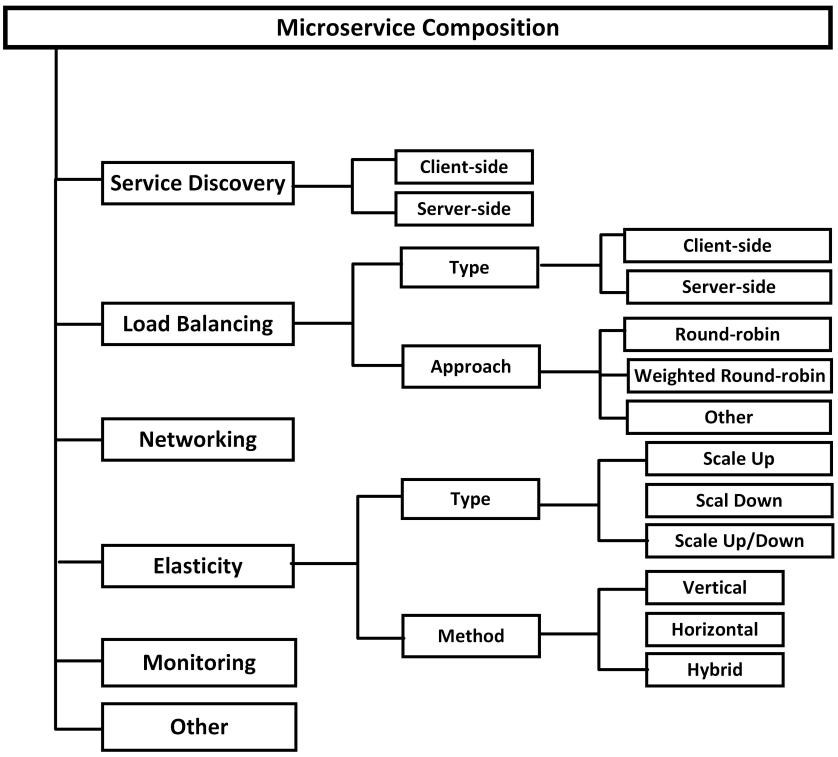


Figure 2.5: Taxonomy for microservice composition

mechanisms. Figure 2.5 presents the taxonomy for essential aspects of the successful composition of microservices within distributed environments. For this analysis, we use both Edge/Fog frameworks designed for microservice deployment [78, 79] and conceptual frameworks/simulators used in formulating and solving placement problems [38, 69] as demonstrated in Table 2.4. We analyse the main functions related to microservice composition in the sections below.

2.5.1 Service Discovery

Dynamic placement algorithms proposed to handle microservice placement [60, 74, 79] define service discovery mechanisms so that changes (i.e., scale up/down, failures) in microservice instances are made know to other microservices that interact with them. Works such as [8, 60] use a client-side service discovery pattern where each client microservice queries a dynamically updated service registry to determine the available

Table 2.4: Analysis of existing literature based on the taxonomy for microservice composition

Work	Service	Load Balancing		Networking	Elasticity		Monitoring	Other
	Discovery	Type	Approach		Type	Method		
[78]	S-S	S-S	Round Robin	✓ (Kubernetes)	U-D	H	✓ (Prometheus)	-
[60]	C-S	C-S	Weighted Round Robin	-	U	H+V	-	-
[38]	NC	C-S	Round Robin	-	-	-	-	-
[46]	NC	C-S	Weighted Round Robin	-	-	-	-	-
[44]	S-S	S-S	Custom	✓ (Flannel + Istio)	-	-	✓ (Prometheus)	-
[74]	S-S	S-S	Custom	✓ (Kubernetes + Istio)	U-D	H	✓ (Prometheus)	-
[47]	ND	S-S	Custom	-	-	-	-	
[66]	S-S	NC	-	-	-	-	-	Fault-tolerance
[69]	ND	C-S	Round Robin	-	-	-	-	-
[79]	S-S	S-S	Weighted Round Robin	✓ (Kubernetes + KubeEdge)	-	-	✓ (Custom)	-
[8]	C-S	C-S	Weighted Round Robin	✓	U-D	H+V	-	-
[80]	S-S	S-S	ND	✓ (MQTT Broker)	U-D	H	✓ (Custom)	Fault-tolerance Security
[81]	S-S	S-S	ND	✓ (Kubernetes + CNI)	U-D	H	-	-

C-S:Client-side, S-S: Server-side, U: Up, D: Down, H: Horizontal, V: Vertical, CNI: Container Network Interface, ND: Not Defined, NC: Not Considered

service instances. Works such as [66, 74, 79] use server-side service discovery where the requests are directed to a designated entity (i.e., a load balancer, a proxy, etc.) that is responsible for directing the requests towards available service instances. "FogAtlas"[78] which is a Fog computing platform used by multiple works such as [40, 65] and other works such as [44, 79] use Kubernetes as the orchestrator along with its default proxy-based, server-side service discovery mechanisms. The work in [66] implements server-side service discovery using an API gateway as the ingress node responsible for service discovery and service composition.

2.5.2 Load Balancing

1. *Type:* Current works model load balancing using two primary approaches: *client-side load balancing* [38, 60, 69] where each client is responsible for individually executing load balancing policies, thus enabling application dependent load balancing policies to be implemented, and *server-side load balancing* [44, 79] where a dedicated load balancer sits between client and server microservices to handle load

balancing.

2. *Approach:* Integrating the effect of the load balancing mechanism and introducing novel load balancing policies to improve the performance of the services is vital in microservice application deployment. Works like [38, 46] model the end-to-end service latency based on the existing load balancing policies such as Round Robin and Weighted Round Robin to determine the number of microservice instances to deploy. Meanwhile, some works introduce custom load balancing policies: [44] proposes a load balancing policy for a multi-region Fog architecture where the requests are directed based on the residual CPU of each Fog region, [74] uses the variance of the resource occupancy rate of the edge nodes to determine where to direct the requests, [47] uses a meta-heuristic algorithm to place microservice replicas to identify flow paths by considering parameters such as service cost and service latency.

2.5.3 Networking

Distributed deployment of containerised microservices makes networking one of the integral functions of microservice compositions. This is further complicated by the federation of Fog and Cloud, which results in communication between multiple networking environments and technologies. [78] uses Kubernetes to handle the networking among microservices instances whereas [74] integrates Istio[6] service mesh framework to handle inter-service communication. [79] integrates Kubernetes with KubeEdge for the edge network. To overcome the limitation Kubernetes network model and assign subnets per host, [44] use Flannel, a Container Network Interface (CNI) and Istio on top of Kubernetes orchestration. [81] explores and compares multiple CNIs, including Flannel, Weave, Calico and OVN. [80] uses MQTT Broker, an asynchronous messaging-based communication mechanism to transmit messages over the network among decoupled microservices.

[6] https://istio.io/

2.5.4 Elasticity

Elasticity indicates the ability of the microservices to be dynamically scaled up and down dynamically in a performance-aware manner. With the use of lightweight deployment technologies such as containers, microservices can be easily auto-scaled to improve the performance of the application while ensuring optimum resource utilisation. Out of the many works that consider horizontal scalability of microservices, the majority consider this during the initial placement of the application to make use of resource-constrained Fog devices but fail to use auto-scaling/elasticity under dynamic changes in the environment (i.e., load changes, failures etc.). Dynamic placement algorithms proposed in works such as [8, 60, 74] consider elasticity. However, we can further analyse it based on the supported type of elasticity: scaling up, scaling down, and the method of scaling: horizontal, vertical. The paper in [60] considers both vertical and horizontal scaling but only considers scaling up as new user requests arrive. The work in [74] proposes a utilisation threshold-based policy to horizontally scale up/down microservices through continuous monitoring of the resources. Practical frameworks and simulators [8, 78] provide the infrastructure required to auto-scale (both up and down, horizontal and vertical) microservices through customised policy implementations.

2.5.5 Monitoring

Monitoring is the collection of application and platform metrics in such a way that they can be used to detect failures, performance degraded states, etc. and act accordingly to maintain system performance. The highly dynamic nature of containerised microservices makes monitoring a challenging task, which has resulted in the development of open-source monitoring tools that can handle the large volume of moving parts in microservices-based application deployment. Hence, [44, 74, 78] integrate Prometheus[7] to their orchestration platform to monitor multiple platform metrics (i.e., request number, response times, resource consumption, network communications, etc.). Meanwhile, works such as [79, 80] implement their own customised monitoring tools to monitor metrics across Edge-Cloud integration.

[7] https://prometheus.io/

2.5.6 Other

Other composition-related tasks include fault tolerance and security. The work in [66] proposes a conceptual framework where API Gateway handles the responsibility of fault-tolerance by data recovery, service re-composition and re-submission of a failed request. The paper in [80] introduces three main components: health check component, circuit breaker and timeout component to identify and isolate failures to avoid cascading failures under MSA. The paper in [80] implements centralised security components to manage authorisation and authentication required for microservice access.

2.5.7 Research Gaps

Based on the analysis of existing works presented in Table 2.4, we identified the following gaps:

1. Current works demonstrate less emphasis on load-balancing policies and their effect on the formulation of the placement problem.

2. There's scope for improvement in elasticity (both horizontal and vertical scaling), fault tolerance, and microservice security through proper monitoring of the deployed application at the application and platform level.

3. Effect of orchestration components (i.e., service registry, API gateways, load balancers) on the application performance needs to be analysed and demonstrated in terms of handling their possible failures, delay overheads, etc.

2.6 Performance Evaluation

Accurate policy evaluation is one of the vital steps in designing novel placement algorithms for fast-evolving fields such as IoT and Fog computing. To this end, we propose the final taxonomy by analysing the crucial aspects of the evaluation phase, as shown in Figure 2.6. Afterwards, current works are mapped to the taxonomy to identify gaps and possible improvements (see Table 2.5).

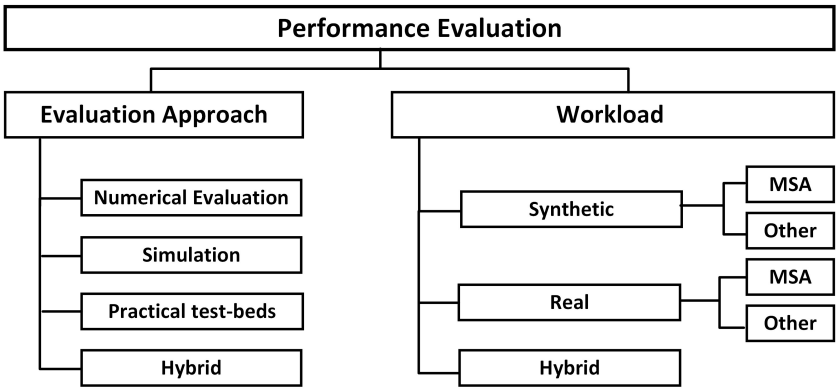


Figure 2.6: Taxonomy for performance evaluation of the placement policy

2.6.1 Evaluation Approach

Evaluation approaches can be categorised as Numerical Evaluations, Simulations, use of Practical test-beds and Hybrid approaches consisting of combinations of the above methods.

Numerical Evaluations: In numerical experiments, the algorithm is evaluated by numerically calculating specific metrics that provide insights on the fitness of the resultant placement proposed by the algorithm [40, 42, 43, 69] or/and evaluating performance metrics of the algorithm such as execution time, computation complexity and convergence [32, 69]. [40, 42] use the Gurobi mathematical optimisation solver to obtain the optimum solution to the formulated MINLP and compare the solution obtained from their proposed heuristic placement algorithm by calculating metrics such as the number of placed applications, network link usage, etc. numerically. The work in [32] evaluates their approach based on the performance metrics (i.e., the fitness of the best solution, Pareto solution spread, execution time, etc.) of their proposed multi-objective evolutionary algorithm, whereas [69] analyses the execution time of the algorithm under different experimental settings to evaluate the scalability of the algorithm, thus deriving the system size the algorithm can handle.

Simulations: For the evaluation of microservices-based application placement in Edge/-Fog computing environments, both open-source simulators (i.e., iFogSim [41, 46, 60, 63], YAFS [36], CloudSim [38, 59]) and custom simulators [44, 45, 61] are used by the current works. Simulators such as iFogSim [8] and YAFS [83] provide in-built microservice-

Table 2.5: Analysis of existing literature based on the taxonomy for performance evaluation

Work	Evaluation Approach			Workload			Work	Evaluation Approach			Workload		
	Numerical	Simulation	Practical	Synthetic	Real	Hybrid		Numerical	Simulation	Practical	Synthetic	Real	Hybrid
[36]		✓(YAFS)		✓(Other)			[40]	✓	-	✓(FogAtlas)	✓(MSA)	-	-
[60]	-	✓(iFogSim)	-	✓(MSA)	-	-	[32]	✓	-	-	-	-	✓
[42]	✓	-	-	✓(Other)	-	-	[45]	-	✓(C-MATLAB)	-	✓(MSA)	-	-
[38]	-	✓ (MATLAB + CloudSim)	-	✓(MSA)	-	-	[61]	-	✓(C-MATLAB)	-	✓(MSA)	-	-
[46]	-	✓(iFogSim)	-	✓(MSA)	-	-	[44]	-	✓(C-Python)	✓	-	-	✓
[63]	-	✓(iFogSim)	-	✓(MSA)	-	-	[62]	-	✓(ND)	-	✓(MSA)	-	-
[74]	-	-	✓	-	✓(MSA)	-	[64]	-	-	✓	-	-	✓
[67]	-	✓(C-Java)	-	-	✓(Other)	-	[47]	-	✓(ND)	-	✓(MSA)	-	-
[43]	✓	-	-	✓(MSA)	-	-	[69]	✓	-	-	✓(Other)	-	-
[66]	-	✓(C-Python)	-	-	✓(MSA)	-	[72]	-	✓(NS3)	-	✓(MSA)	-	-
[65]	-	✓(C-Python)	-	✓(Other)	-	-	[59]	-	✓(Cloudsim)	-	✓(MSA)	-	-
[73]	✓	-	-	✓(MSA)	-	-	[82]	-	✓(iFogSim)	-	-	✓(MSA)	-
[71]	-	-	✓	-	✓(MSA)	-	[79]	-	-	✓	-	✓(MSA)	-
[75]	-	-	✓(Astraea)	-	✓(MSA)	-	[41]	-	✓(iFogSim)	-	-	-	✓
[37]	-	✓(iFogSim)	-	-	-	✓	[48]	-	✓(C-C++)	-	✓(MSA)	-	-

related features such as distributed application modelling, service discovery and load balancing, thus enabling the users to simply implement their placement policy within the simulator or implement the algorithm separately (i.e., MATLAB [38], IBM CPLEX [46]) and input the resultant placement to the simulator for evaluations.

Practical: To evaluate placement algorithms using practical frameworks, [40] uses "FogAtlas", an open source framework for microservices deployment and orchestration in Fog environments, [75] implements a framework called "Astraea" for the management of GPU microservices, whereas other works such as [44, 64, 74, 79] implement customised test-beds with functionalities relevant to the placement algorithms. They use popular container-orchestration platforms such as Docker swarm [64], Kubernetes [44, 74], KubeEdge [79] in their implementations.

Hybrid: Some of the works use multiple evaluation approaches to analyse and evaluate the proposed placement policies from multiple perspectives. The work presented in [44] use both Simulations and practical test beds for evaluation. Simulations carry out large-scale experiments, whereas test beds further verify the results of the simulations by carrying out a selected set of experiments. The work in [46] use a combination of nu-

merical evaluations and simulations where numerical evaluations are used to improve and fine-tune the meta-heuristic placement algorithm, whereas the simulation evaluates the resultant placements.

2.6.2 Workload

For the analysis of the workload, we consider the nature of the application placement requests used to evaluate the placement policy. We can categorise them as Synthetic, Real, or a combination of both, denoted as Hybrid.

Synthetic: Synthetic workloads are created either by mimicking specific microservices-based applications [40, 59, 60] (categorised in the taxonomy as MSA) or using generic application models such as DGs or DAGs [36, 42, 65, 69] (categorised in the taxonomy as Other) to generate a workload consisting of multiple applications with heterogeneous resource and QoS requirements. The paper in [40] models the applications following a microservices-based IoT application for face recognition consisting of two chained microservices. The work in [60] models a smart-healthcare application and creates a synthetic workload based on the modelled application. [59] models smart city and forest surveillance applications. Meanwhile, some work like [42, 65, 69] generate random synthetic DAGs as microservices-based applications, whereas [36] uses Growing Network(GN) graph structure where graphs are created by adding nodes one at a time to existing nodes to develop microservices-based applications following Directed Graphs as the interaction pattern.

Real: Real workloads include the use of already implemented microservices-based applications (categorised in the taxonomy as MSA) or adapting performance traces of applications that follow other application models (categorised in the taxonomy as Other). [74] use Bookinfo [8], an online book store application following MSA along with the hotel reservation booking application from DeathStartBench [9] [84], which is a benchmark application suite following MSA. [71] also uses benchmark applications from DeathStartBench along with the benchmark application provided in [85]. [75] uses AI-based

[8]https://istio.io/latest/docs/examples/bookinfo/
[9]https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation

GPU microservices available in AIbench [10] to create the workload. [66] uses the curated data set available in [86] which consists of 20 open-source projects based on MSA. In contrast to the above examples, [67] uses traces from Google Cluster 2019 [87]. These traces provide data on task requests (i.e., CPU. memory, deadline, etc.), and as [67] models microservices as independent components that do not interact with other microservices, the said data set is easily adapted by this work for evaluations.

Hybrid: [32, 41] use a microservices-based e-commerce application known as Sock Shop [11] provided under Apache License 2.0, along with two other synthetic application models (an online EEG tractor beam game and intelligent surveillance application) to create the workload. [44] creates a synthetic workload for simulation-based studies and implements a microservices-based application name "Paper Miner" designed for mining research papers and deploys it on a real-world platform to evaluate the placement algorithm.

2.6.3 Research Gaps

Based on the analysis of existing works presented in Table 2.5, we identified the following gaps related to the evaluation of placement algorithms developed for microservices-based application placement within Fog environments.

1. Lack of use in practical test beds is one of the prominent drawbacks of currently used evaluation approaches. The majority of the available works use numerical evaluations or simulations to evaluate the performance of their placement policies but fail to validate them on real test beds. Thus, overheads related to orchestration tasks (i.e., service discovery, load balancing, auto-scaling etc.), failure characteristics, resource contention among microservices, etc., are not accurately captured. Moreover, the suitability of the Edge/Fog devices to act as the placement engine that runs the algorithms is not evaluated in practical settings.

2. As Edge/Fog computing paradigms are still relatively new and yet to be adopted by the service providers, simulators play a significant role in evaluating placement

[10] https://www.benchcouncil.org/aibench/index.html
[11] https://microservices-demo.github.io/

policies. However, this requires a standard open-source simulator for use among the research community and continuous improvements through collaboration. As microservices-based application placement in Fog environments is still in its infancy, we see the increased use of custom simulators due to the lack of open-source simulators that capture all related aspects of microservice orchestration.

3. Lack of real-world traces or actual implementations of applications for the deployment within test beds is another significant gap in current research. The existing benchmark applications do not include IoT applications, making it harder to capture their characteristics accurately.

2.7 Summary

This chapter focused on IoT applications designed and developed using MSA and their placement within Fog computing environments. We conducted a comprehensive background study and identified four critical aspects of microservices-based application placement, namely, modelling of MSA, placement policy creation, microservice composition and performance evaluation. We proposed taxonomies for each aspect, highlighting features related to MSA and the Fog computing paradigm. Moreover, we analysed and discussed the current literature under each taxonomy and identified research gaps. This thesis investigated some these research gaps and proposed new research directions in the last chapter.

Chapter 3

A Distributed Placement Policy for Scalable Microservice Deployment

Independent deployability and scalability, along with the lack of centralised management of microservices, demonstrate significant potential to utilise distributed, heterogeneous and resource-constrained Fog computing resources. This chapter proposes a decentralised microservices-based IoT application placement policy for heterogeneous and resource-constrained Fog environments. The proposed approach utilises microservices' independently deployable and scalable nature to place them as close as possible to the data source to minimise latency and network usage. Moreover, it aims to handle service discovery and load balancing related challenges of the microservice architecture. We implement and evaluate our policy using iFogSim simulated Fog environment. Results of the simulations show around 85% improvement in latency and network usage for the proposed microservice placement policy compared with the Cloud-only placement approach and around 40% improvement over an alternative Fog application placement method known as Edge-ward placement policy. Moreover, the decentralised placement approach proposed in this chapter demonstrates a significant reduction in microservice placement delay over centralised placement.

3.1 Introduction

IoT is spreading across a large variety of application domains such as healthcare, agriculture, defence, smart city, and IIot. Due to the socio-economic benefits generated by IoT, the number of connected devices and IoT application users keep growing rapidly.

This chapter is derived from:

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Microservices-based IoT Application Placement within Heterogeneous and Resource Constrained Fog Computing Environments", *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, Pages: 71-81, Auckland, New Zealand, December 2-5, 2019.

IoT applications demand more resources to support the ever-increasing workloads generated by IoT devices and meet the performance requirements expected by the users. While Fog computing can satisfy the stringent latency requirements of latency-critical IoT services, the resource-constrained nature of Fog devices poses a challenge in supporting increasing workloads.

Microservice architecture, which decomposes applications into fine-grained microservices, can aid in overcoming the above challenge. To this end, IoT applications in Fog environments can benefit from three main characteristics of microservices as follows:

- Independently deployable - Microservices are easily containerised due to being loosely coupled, independent and self-sustained instances. In turn, containers are suitable for Fog computing environments due to lower startup time, lower virtualisation overhead and support for scalability [22]. Moreover, this enables microservices to be deployed across Fog resources and Cloud data centres, thus allowing better utilisation of resource-constrained Fog resources for latency-critical and bandwidth-hungry microservices.

- Independently scalable - Microservices support both vertical and horizontal scalability. Vertical scalability represents the change of resource allocation as per the load, whereas horizontal scalability indicates having multiple replicas of a single microservice to support the load. As Fog environments consist of resource-constrained nodes that are heterogeneous in resource availability, the horizontal scalability of microservices can significantly impact the performance of applications deployed within Fog environments.

- Lack of centralised management - This matches with highly distributed nature of the Fog computing environments. Furthermore, this enables the federated use of Fog computing environments with resource-rich Cloud datacentres.

Thus, applications developed using microservices have the potential to be efficiently adapted to Fog environments through the dynamic deployment of scalable microservices to meet user demands. However, the introduction of microservices-based applications creates challenges in terms of dynamic service discovery, load balancing and

architectures to support decentralised management of microservices within Fog environments.

In literature, there is a notable number of works that focus on developing placement algorithms for distributed applications within Fog-Cloud environments [88], [9], [89]. However, the placement of microservices-based Fog applications has not been investigated extensively. Moreover, existing works lack a decentralised approach for placing microservices within heterogeneous and resource-constrained Fog environments, focusing on horizontal scalability and challenges such as service discovery and load balancing. In our work, we present a microservices-based IoT application placement policy addressing the abovementioned aspects.

Thus, **key contributions** of our work can be summarised as follows:

1. A decentralised placement algorithm for microservices-based IoT applications, highlighting horizontal scalability of microservices within resource-constrained and heterogeneous Fog nodes.

2. A Fog node architecture to support decentralised placement along with service discovery and load balancing.

3. An implementation of our proposed policy on the iFogSim simulation environment and comparison against different placement approaches in terms of latency, network usage and efficiency of decentralisation.

The rest of this chapter is organised as follows. In Section 2, we highlight related research. Section 3 presents the microservices-based application model, Fog architecture and Fog node architecture, along with the problem description. Our proposed solution is provided in Section 4, along with relevant algorithms. Section 5 presents steps related to the implementation of the solution using the iFogSim simulator, whereas Section 6 reflects simulation setup and performance evaluation. Finally, Section 7 concludes the chapter with future work and research directions.

Table 3.1: Summary of literature study

Work	Fog Layer Architecture		Application Model		Decentralised	Microservices-based
	Hierarchical	Clustering	DAG	BoT	Placement	Applications
Taneja et al.(2017)	✓		✓			
Gupta et al. (2017)	✓		✓			
R. Mahmud et al. (2018)	✓	✓	✓		✓	
Filip et al. (2018)				✓		✓
Faticanti et al. (2019)			✓			✓
Santoro et al (2017)				✓		✓
this work	✓	✓	✓		✓	✓

3.2 Related Work

Microservices architecture is a recent concept that has created a phenomenal impact on development of IoT applications within Cloud computing environments. Butzin et al. [26] investigate the state of the art of IoT and microservices architecture to show how their architectural goals are quite similar. Several research studies have been conducted on developing Cloud-centric IoT frameworks based on microservices architecture [90], [91], [92], [93]. However, research on adapting this concept to Fog environments is still in its early stages.

Filip et al. [59] present a centralised placement approach for Edge-Cloud scheduling of microservices using Bag of Tasks (BoT) model where each task consists of one or more microservices. In the proposed architecture, nano data centers are used as Edge resources. Scheduling engine receives jobs and assigns them to VMs in the nano data centers or Cloud. Their scheduling policy places all microservices of a certain job within the same processing element or move it towards the Cloud, based on resource availability.

Santoro et al. [94] implement a framework and a software platform for orchestration of microservices-based IoT application workloads. In the proposed architecture, applications are modeled as a collection of microservices distributed via container images. The proposed architecture consists of IoT device layer, Edge gateways, Edge cloudlets and Cloud. Microservice deployment requests are sent towards a negotiator that ac-

cepts or rejects the requests. Orchestrator calculates the most suitable device to deploy microservices of the accepted requests, based on requirements (CPU, RAM, bandwidth, etc.) defined in deployment requests.

A centralised throughput aware placement algorithm for microservices-based Fog applications is presented in [40]. The proposed system consists of Edge servers that are grouped together based on their geographical regions. In this work, application microservices are placed within the region that contains respective IoT device or within the neighboring region. A greedy algorithm is presented for mapping these microservices onto Edge servers with sufficient computational resources while ensuring that bandwidth of the involved links can satisfy the throughput requirements of the application microservices.

Moreover, there are numerous works in literature that try to solve application placement problem in Fog environments by depicting applications in a distributed manner as a collection of interdependent modules. But, the concept of microservices architecture along with its unique characteristics and challenges are not observed in these works.

Taneja et al. [88] present a resource aware module mapping algorithm for placement of distributed applications within Fog environments. This work tries to optimise resource utilisation by sorting application modules and nodes based on required resources and available capacity respectively and mapping sorted modules to resources. The proposed algorithm is compared with Cloud-only placement to depict the reduction of end-to-end latency in the Fog placement approach. This work defines a hierarchical Fog architecture where each Fog node is connected with a node in immediate upper layer of the hierarchy. Horizontal connections among Fog nodes of the same level are not defined. Moreover, placement is managed through a centralised approach.

Gupta et al. [9] propose a centralised edge-ward module placement algorithm for placing distributed applications modeled as Directed Acyclic Graphs (DAG). Their algorithm commences the placement of application modules starting from lower level Fog nodes and move upwards the hierarchy until a node with enough resources is met. But, their proposed algorithm supports only the vertical scaling of modules and does not consider horizontal connections among Fog nodes within the same Fog level.

Latency aware placement of application modules within Fog environments is pre-

sented in R. Mahmud et al. [89]. This work, proposes a decentralised module placement method that considers service access delay, service delivery time and internodal communication delay when placing application modules within Fog environments. In this approach, distributed applications consisting of interdependent modules are placed and forwarded vertically and horizontally to satisfy the latency requirements of the application while optimizing resource usage. But horizontal scaling of modules within Fog layer, microservices architecture and related challenges are not considered in this work.

A summary of the reviewed related works is presented in Table 3.1, comparing them in terms of architecture of the Fog layer, application model and application placement approach. Architecture of the Fog layer used in each work is identified as hierarchical, if the Fog tier consists of multiple Fog levels where each device is connected with a node in immediate upper layer and the latency from the IoT devices and resource availability of the nodes increase when moving towards upper levels. Clustering denotes existence of horizontal connections among Fog nodes of the same hierarchical level of the Fog architecture.

In our work, we present a placement algorithm for microservices-based IoT applications, where horizontal scalability of microservices is used within Fog node clusters that exist on the same hierarchical level of the Fog architecture. Moreover, the proposed policy also handles the challenges that come with horizontal scaling of microservices such as service discovery and load balancing. Our placement approach uses decentralised management of placement where each Fog node is responsible for placement decision making instead of having a centralised entity.

3.3 System Model and Problem Formulation

We propose a multi level, hierarchical Fog architecture where each Fog computing node is responsible for processing application placement requests.

We model IoT applications as collections of containerized microservices and place them within the Fog environment using a decentralised placement approach.

Our Fog architecture, application model, Fog node architecture and application placement problem are discussed in detail in the following subsections.

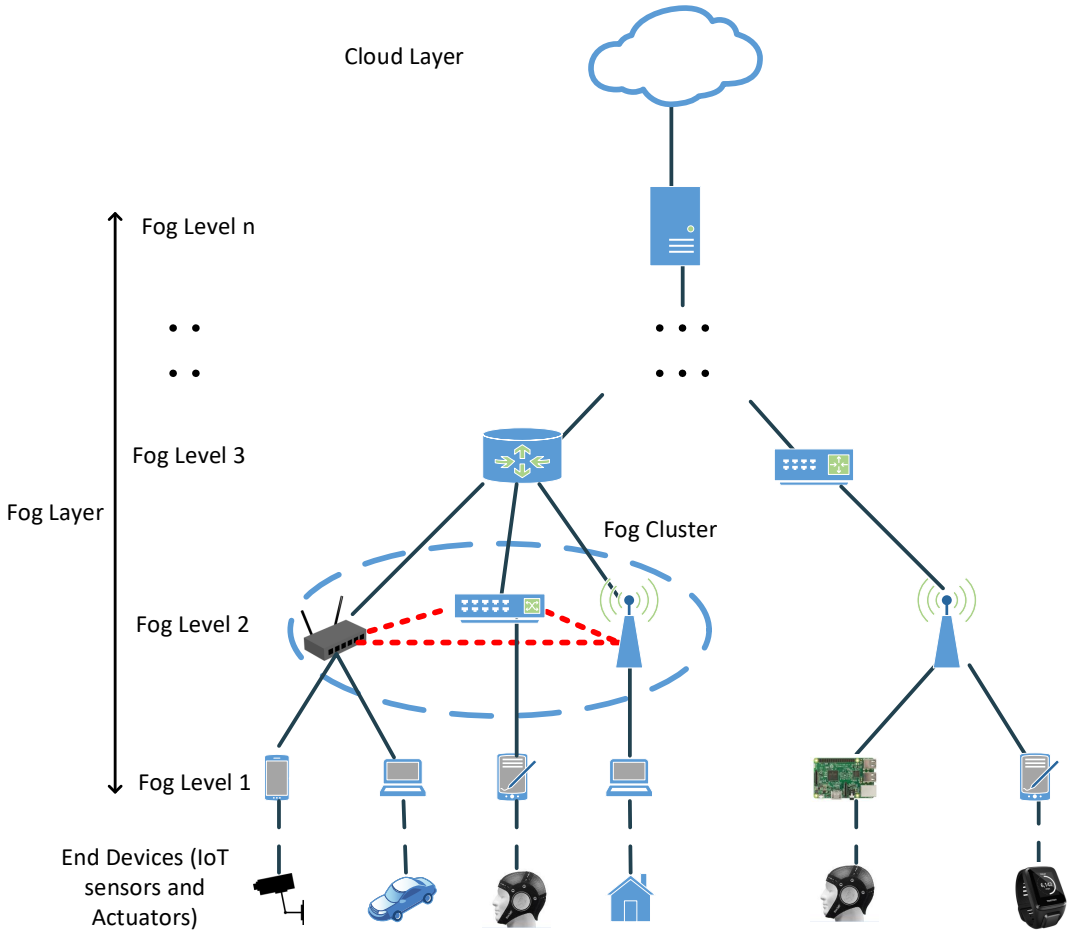


Figure 3.1: Hierarchical Fog architecture

3.3.1 Fog Architecture

Fog computing makes use of computation, networking and storage capabilities of geographically distributed, heterogeneous and resource constrained devices such as mobile phones, access points, routers, proxy servers, nano data centers that span the continuum from IoT devices to Cloud. This, in turn provides localized services to end users, thus resulting in efficient bandwidth usage and low latency.

In this work, the three-tier hierarchical Fog architecture is used, where Fog layer is placed between IoT devices and Cloud data centers [95]. In our architecture, nodes within the Fog layer are also organised hierarchically as depicted in Figure. 3.1.

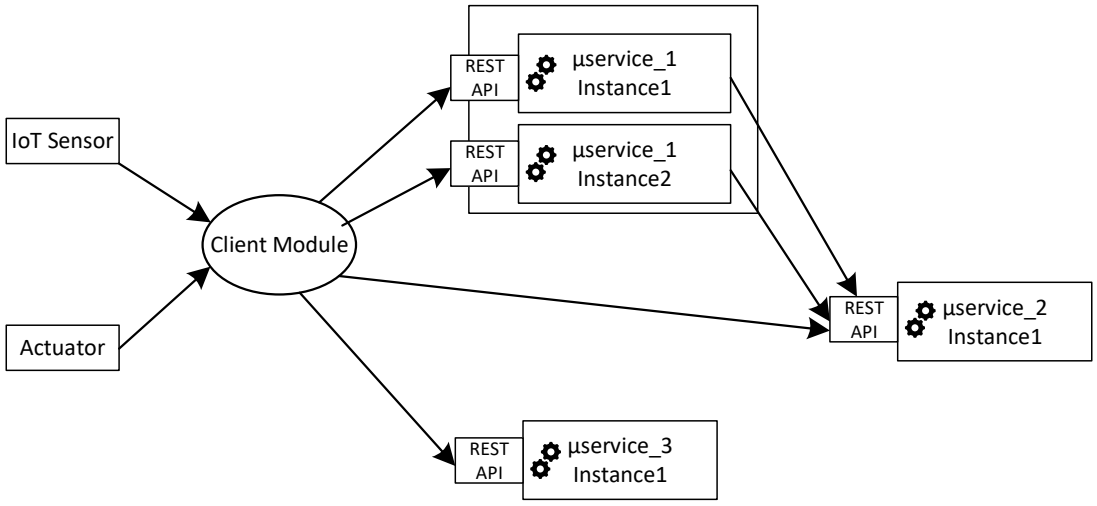


Figure 3.2: Microservices-based IoT application

Fog nodes are placed in such a way that compute, storage and networking capabilities of Fog devices vary not only among the nodes in different levels but also within the same hierarchical level of the Fog layer. Compute, storage and network capability of Fog nodes increase when moving from lower levels to higher levels inside the Fog layer. Moreover, each Fog node has a direct connection with a node in immediate upper level and also can have links with nodes in the same level forming clusters among themselves. In this work, it is assumed that a certain Fog node belongs to only one cluster at a particular time. Nodes within the same Fog cluster communicates with each other using Constrained Application Protocol (CoAP) which is a simple web transfer protocol based on REST model [89], [96]. Therefore, the communication delay among cluster nodes is extremely low.

3.3.2 Application Model

In our work, we have modeled IoT applications as a set of microservices that can be deployed, upgraded and scaled independently. Each microservice is deployed as an independent container and the resource requirement for each microservice is defined in terms of CPU, bandwidth, RAM and storage. Figure 3.2 depicts our microservices-based Fog application architecture. Each application consists of a Client module that is

deployed onto end user devices such as mobiles, tablets etc. that reside in the lowest level of the Fog layer. This module is responsible for sending data received from IoT sensors, towards relevant microservices for processing and also displaying results or sending resulting signals to the actuators. The rest of the microservices are deployed on either Fog or Cloud layer based on the placement policy. Since microservices that make up an application have data dependencies amongst them, an IoT application is depicted as a DAG [88]. In this model, each microservice is represented by vertices of the DAG, whereas edges between vertices represent data dependencies among microservices.

In microservices-based applications, microservices call each other through REST APIs to perform tasks. As microservices are horizontally scalable, a certain microservice can have multiple replicas to support the load balancing. When directing requests to microservices, two approaches are available: through server-side load balancing or client-side load balancing. In the server-side load balancing approach, a dedicated load balancer component lies between client microservices and server side microservices. This component is responsible for service discovery and directing the requests according to a load balancing policy. However, in a highly distributed environment such as Fog, using such centralised load balancing approach is not efficient. Thus, within this model, service discovery and load balancing is handled through a decentralised method by using client-side load balancing. So, in the proposed model, device that hosts the client microservice has to be aware of all microservice instances of the required services and route requests according to the load balancing logic.

3.3.3 Fog Nodes

In this chapter we present a comprehensive Fog node architecture to support decentralised placement of microservices-based applications. [89] proposes an architecture where each Fog node consists of three main components: communication component, computational component and controller component. However, their architecture does not capture the requirements of microservices architecture. So, we improve their concept to propose a Fog node architecture (see Figure 3.3) that supports placement of microservices-based applications within Fog environments.

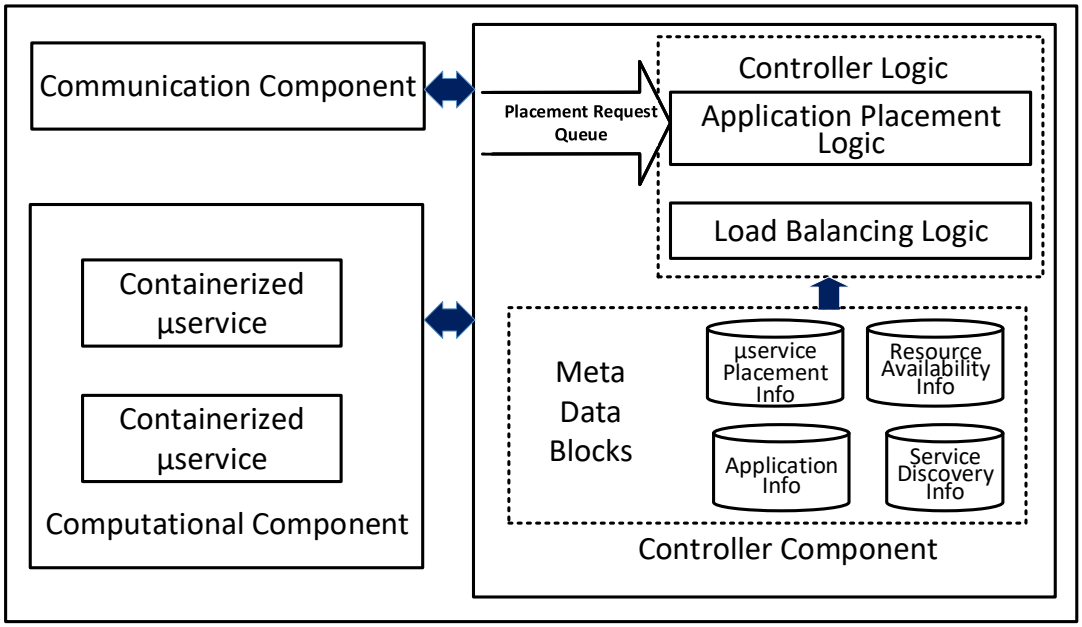


Figure 3.3: Fog node architecture

According to our model, *Placement of Microservices*, *Service Discovery* and *Load Balancing* are handled by controller component in the node. When a placement request is received by a Fog node, it is queued in *Placement Request Queue (PRQ)* in the controller component. Placement requests in the queue are processed one after the other using *Application Placement Logic*. *Service Discovery Info (SDI)* data block is a service registry that contains network locations of service instances that can be used by client microservices placed within the Fog node. Each time a client microservice makes a request, it is routed to a service instance according to the *Load Balancing Logic* using data in *SDI*.

Each node keeps track of available resources (*Resource Availability Info*) such as CPU, RAM, bandwidth and storage. This information is used when placing microservices and also when making decisions on scaling microservices across clusters of nodes that are in the same Fog level. *µservice Placement Info (µPI)* keeps track of all microservices that are placed within the Fog node along with resources allocated for each microservice. *Application Info* stores DAG representations of each IoT application available for placement within the Fog environment.

Computation component of the Fog node consists of deployed microservices. Each

Fog node deploys microservices using container images that are available in a centralised container image registry.

3.3.4 Placement Problem

Placing latency critical and bandwidth hungry microservices belonging to IoT applications within lower levels of Fog layer results in reduction of latency and network usage. But Fog nodes that reside in the lower levels of the hierarchy are more resource constrained when compared with upper level Fog devices and Cloud data centers. Even within the same level, resource availability in Fog nodes varies. Since IoT end devices are highly distributed and dynamic, load on each of these Fog nodes also varies.

Under these circumstances, lower level Fog nodes may not be able to support the service demand which results in microservices being placed at higher levels in Fog hierarchy. Moreover, due to resource heterogeneity of nodes and varying loads on each Fog node, some nodes within same Fog level can have under-utilised resources while others fail to support the service demand. This can be overcome by creating clusters among Fog nodes of the same hierarchical level and scaling application modules among the clustered devices to support the load. This has several associated challenges noted below.

1. An efficient application microservice placement algorithm is needed that can identify what application microservices to be scaled and in which device in the cluster to place them.

2. A microservice discovery method to be used by client microservices to call server microservices.

3. A Load balancing mechanism to direct requests to scaled microservice instances.

4. A decentralised approach to meet above challenges.

In this chapter, we propose a Microservice Placement Algorithm addressing aforementioned challenges.

3.4 Proposed Solution

To solve the placement problem, we propose a heuristic placement algorithm, that scales microservices across Fog device clusters to accommodate load within resource constrained and heterogeneous Fog environments. Aim of the algorithm is to place latency critical and bandwidth hungry microservices as close as possible to the data source. Closeness is defined in terms of hierarchical level of the device that hosts the microservice, as depicted in Figure 3.1. The algorithm facilitates decentralised placement of microservices, service discovery and load balancing.

3.4.1 Microservice Placement

Controller component of every Fog node contains the *Application Placement Logic* (Algorithm 1, Algorithm 2, Algorithm 3). Thus, every Fog device contributes in making placement decisions using this placement policy.

When a sensor joins a lower level Fog node, placement process is invoked by the corresponding Fog node. This Fog node which acts as the gateway to the rest of the Fog network, hosts the client module of the IoT application and rest of the module placement is carried out according to the *Application Placement Logic* starting from it. Gateway Fog node generates a *Placement Request (pr)* which consists of *Application ID, Placed microservices map, Gateway device ID* and *Placement request ID*. *Application ID* identifies each IoT application uniquely. Each Fog node has information on available IoT applications including microservices that form the applications and connections among those microservices in the form of a DAG which can be accessed using *Application ID*. *Placed microservices map* consists of already placed microservices with respect to the placement request and nodes they are placed on. *Placement request ID* is a unique ID generated per request. It can be used to uniquely identify each sensor that joins the gateway node. Gateway Fog node sends this placement request towards the parent node where it gets added to parent node's *PRQ*.

PRQ is a First in First Out (FIFO) data structure. Thus, if the queue is not empty, requests are processed starting from the first request in the queue. Each request gets processed according to the Algorithm 1. For the selected placement request, algorithm

Algorithm 1 Process Placement Request

Input: placement request pr

Output: placement request status; *Status.COMPLETED* for request processed, *Status.HALTED* for waiting for cluster placement

```
1:  procedure ProcessPlacementRequest(pr)
2:      node ← this.node
3:      a ← pr.applicationID
4:      m_p ← pr.placedMicroservices
5:      m_f ← {}                                   ▷ Placement failed μservices
6:      m_toPlace ← GetμservicesToPlace(a, m_p, m_f)
7:      while m_toPlace is not empty do
8:          if node is cloud then
9:              place all remaining microservices here
10:             send service discovery info
11:             return Status.COMPLETED
12:         else
13:             m ← m_toPlace.remove(0)
14:             placementStatus = PlaceMicroservice(m)
15:             if placementStatus = Status.PLACED then
16:                 nodes_client = GetClientNodes(m, m_p)
17:                 for every node n of nodes_client do
18:                     n.SDI.add(m, node)
19:                 m_p.add(m, node)
20:                 if m_toPlace is empty then
21:                     m_toPlace ← GetμservicesToPlace(a, m_p, m_f)
22:             else if placementStatus = Status.CLUSTER then
23:                 return Status.HALTED
24:             else if placementStatus = Status.FAILED then
25:                 m_f.add(m)
26:                 if m_toPlace is empty then
27:                     m_toPlace ← GetμservicesToPlace(a, m_p, m_f)
28:     if m_p.size() < appμserviceCount(a) then
29:         node_parent ← node.parent
30:         node_parent.PRQ.add(pr)
31:     return Status.COMPLETED
```

Algorithm 2 Place Microservice

Input: microservice to place m

Output:microservice placement status : *Status.PLACED* if placed on this node, *Status.CLUSTER* if placement on cluster nodes and *Status.FAIL* if no resources are available on this node or cluster nodes

```
1:  procedure PLACEMICROSERVICE(m)
2:      if instance of m already in node then
3:          if req(m) ≤ availCap(node) then
4:              increase resources allocated for instance of m
5:              μPI.add(m)
6:              return Status.PLACED
7:          else if node is in a cluster then
8:              send ClusterPlacementQuery to cluster nodes
9:              return Status.CLUSTER
10:     else
11:         if req(m) ≤ availCap(node) then
12:             place m on node
13:             μPI.add(m)
14:             return Status.PLACED
15:         else if node is in a cluster then
16:             send ClusterPlacementQuery to cluster nodes
17:             return Status.CLUSTER
18:     return Status.FAILED
```

determines the microservices to be placed based on the DAG representation of the application stored in *Application Info* (line 6). A microservice is selected for placement only if all the client microservices in the application that uses its service are already placed. *GetμservicesToPlace* method traverse the DAG of the application and identifies such microservices, taking already placed microservices and placement failed microservices into consideration. Once the microservices are determined, placement begins from the current node. If current node is Cloud, all the remaining microservices are placed there (line 8-11), otherwise algorithm tries to place the selected microservices on the current node by calling *PlaceMicroservice* procedure (line 14) for each microservice in the selected set of microservices, starting with the first in the set. If the placement succeeded, then the next microservice to place is found and placement process on the current node continues (line 15-21). If cluster placement is invoked (*Status.CLUSTER)*, then placement request processing is halted until cluster placement decision is made (line 22-23). If placement

Algorithm 3 Place Microservice On Cluster

Input: microservice to place m, cluster nodes C, placement request pr

```
1:  procedure PLACEONCLUSTER(m, C, pr)
2:      node ← this.node
3:      C′ ← C.requestQueueEmptyNodes
4:      for every node n of C′.nodesWithInstanceOfm do
5:          if req(m) ≤ availCap(n) then
6:              n.PRQ.add(pr)
7:              ProcessPlacementRequest(node.PRQ.dequeue())
8:              return
9:      for every node n of C′.activeNodesWithoutInstanceOfm do
10:         if req(m) ≤ availCap(n) then
11:             n.PRQ.add(pr)
12:             ProcessPlacementRequest(node.PRQ.dequeue())
13:             return
14:     for every node n of C′.inactiveNodes do
15:         if req(m) ≤ availCap(n) then
16:             n.PRQ.add(pr)
17:             ProcessPlacementRequest(node.PRQ.dequeue())
18:             return
19:     node_parent ← node.parent
20:     node_parent.PRQ.add(pr)
21:     ProcessPlacementRequest(node.PRQ.dequeue())
```

failed, then Algorithm 1 tries to place other possible microservices on the current node (line 24-27). After placing all possible microservices on the current node, pr is sent towards the parent node to place rest of the microservices of the application or pr processing is finished if all microservices of the application are placed (line 28-31). If Algorithm 1 returns $Status.COMPLETED$, next request in PRQ is selected for processing.

Microservices placement on each Fog device is carried out according to the Algorithm 2. If current node already contains an instance of the microservice, placement policy tries to scale the microservice. At this point microservice is either scaled vertically or horizontally. If considered node has requested amount of resources, allocated resources for the microservice are increased (line 3-5) whereas if not, microservice is scaled across the cluster to accommodate the load (line 7-8). If the node does not already contain an instance of the microservice, algorithm tries to place microservice on current node or on any of the nodes within Fog node cluster (line 11-17). If horizontal placement within

a particular Fog level is not possible, procedure returns *Status.FAILED*, so that the *pr* is sent to the next level towards the parent node of the current Fog device.

When a Fog node does not have enough resources to support placement of a microservice, our proposed placement policy checks whether this node is in a cluster and if so tries to place the microservice within cluster nodes. To achieve this, a *Cluster Placement Query* is sent to all nodes in the cluster, to which cluster nodes reply with information on available resources (from *Resource Availability Info*), microservices already deployed on the node (from *µservice Placement Info*) and current *PRQ* size. Once replies from all the cluster nodes are received, Algorithm 3 is used to determine the suitable Fog node to place the microservice. For placement, cluster nodes with *PRQ* size of zero is considered. Here priority is given to nodes that already have required microservice placed on the device (line 4-8). If it failed, other active nodes in the cluster are considered (line 9-13). Inactive nodes are considered if this failed (line 14-18). Here inactive Fog nodes are the devices that does not have any microservices deployed and has no placement requests in *PRQ*. Once the cluster node selection is completed, current *pr* is sent either towards the selected cluster node or towards the parent node in case no suitable cluster nodes are found. Then the next placement request in the queue is taken for processing by the current node.

The proposed placement policy propagates *pr* among Fog nodes until all microservices in the application are placed or scaled to support processing of the data generated by newly joined sensor. Each Fog node that receives the *pr*, processes it using *Application Placement Logic* and determine whether to deploy microservices on the node, send *pr* for placement within a cluster node or send towards the parent node. Once all microservices are placed, placement completion is informed to the gateway node along with *placement request ID* of the request. Afterwards, gateway node starts accepting data from the associated sensor, identified based on the *placement request ID*.

3.4.2 Service Discovery

Microservices architecture has two approaches to handle service dicovery; server-side service discovery where all API requests are sent towards a centralised load balancer

that directs them using information stored in a service registry, client-side service discovery where client retrieves service locations by directly contacting the service registry and uses its own load balancing logic to direct them. Due to the highly distributed and hierarchical nature of the Fog architecture server-side service discovery is not suitable. In both approaches having a separate centralised service registry adds an extra overhead to the load balancing and routing process. Moreover, in the above described client-side approach, client would have to communicate with service registry before every API call which results in a large number of messages flowing among them.

As a solution to these challenges, in our work we propose a decentralised client-side service discovery approach. Every time a microservice is placed or scaled, *Placed microservices map* in the placement request can be used to find nodes that host the client microservices. Afterwards, each of these nodes are notified of the service placement. This information is stored within the *SDI* data structure of the recipient nodes. Thus, each Fog device maintains a service registry that contains location details of only the services that are accessed by that device instead of accessing a separate service registry. Moreover, service discovery related messages are transmitted only when a microservice is placed or scaled and the messages are exchanged only among the service Fog node and client Fog nodes related to the *pr*, which limits the service discovery related message flow.

As service discovery messages are sent after placement and scaling of microservices, a client can receive multiple service discovery messages with reference to a service deployed on a certain node. The number of such messages received by a Fog device acts as an indication of the amount of resources allocated for a service instance deployed on a certain node to handle the client requests. Hence, it is also stored within *SDI* and used later as the weighting factor for load balancing.

3.4.3 Load balancing

Information stored in SDI of each Fog node is used for load balancing. When making calls to services, this data is used, and API call is directed based on *Load Balancing Logic*. In this work, we've used a Weighted Round Robin method for load balancing where

weighting is done based on the amount of resources allocated for each available service instance, which is stored within *SDI*.

3.4.4 Time Complexity Analysis

As our microservice placement approach is executed in a distributed manner, the rate of processing PRs received by each Fog node depends on the time it takes for two placement-related stages: Stage 1) time for the Fog node to check if the eligible microservices of the *pr* can be placed within the node, which happens through the execution of Algorithm 1 and Algorithm 2, Stage 2) time to select a cluster node for forwarding incompleted PRs based on cluster feedback.

In Algorithm 1, *GetμservicesToPlace* procedure removes placed microservices (m_p) from the application DAG and traverse the resultant DAG to find vertices without any incoming edges, while taking placement failed microservices of the current node (m_f) into consideration. If the application consists of M microservices that represent vertices of the DAG and E connections among them that represent edges of the DAG, above function has time complexity of $O(|M| + |E|)$. *PlaceMicroservice* function in Algorithm 2 check if selected microservices can be placed within the current Fog node, which is completed in constant time with complexity of $O(1)$. If it's not possible to place within the current node, a query is broadcast to all cluster nodes, which takes $O(|C|)$ for C number of nodes and the process exits Algorithm 2. The best time complexity for Stage 1 occurs if no microservices are placed within the current node, and the *pr* gets forwarded directly to the parent node as the current node is not part of a cluster. This results in a time complexity of $O((|M| + |E|))$. Worst time complexity of $O([|M| * (|M| + |E|)] + |C|)$ occurs when all except the final microservice of the DAG is placed, and a query is sent for cluster nodes for *pr* placement completion.

For the Stage 2, for a Fog node cluster with C number of nodes, worst case time complexity of Algorithm 3 is of linear time where all nodes are checked for resource availability to place the selected microservice. Thus, the time complexity of selecting a cluster node for the placement of a microservice is $O(|C|)$.

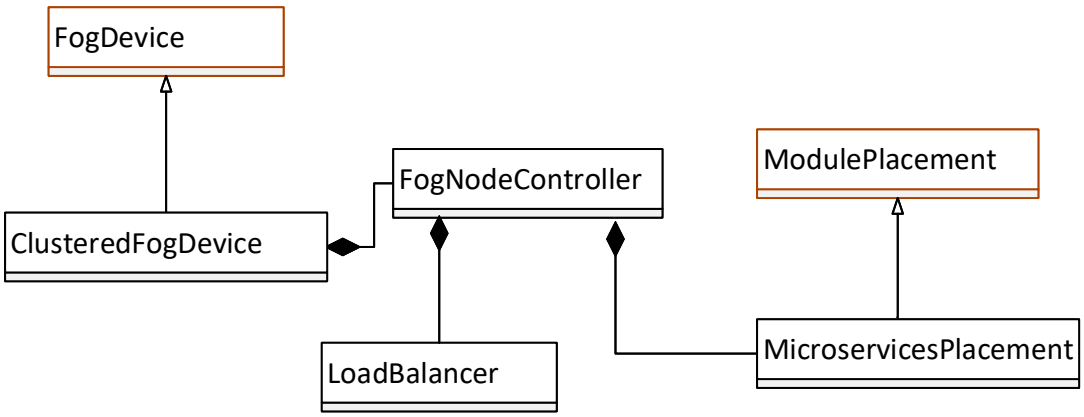


Figure 3.4: Class diagram of extensions made to iFogSim simulator (existing classes: FogDevice.java and ModulePlacement.java)

3.5 Design and Implementation

To evaluate the performance of the proposed policy, we implemented and simulated a Fog computing environment using iFogSim Simulator [9]. iFogSim is a simulation toolkit developed for the simulation of Fog environments. It is built based on CloudSim simulator [97] which is widely used for evaluating resource-management and scheduling policies for Cloud computing environments. iFogSim supports creation of hierarchical Fog architectures, modelling of distributed applications and evaluation of scheduling policies based on performance metrics such as latency, network usage and power consumption. Since these features are significant in modelling the proposed system, iFogSim was chosen for simulations. Moreover, several features were added to iFogSim simulator to support modelling of the proposed system. Figure 3.4 represents new classes implemented within iFogSim simulator to support our placement policy.

iFogSim supports a centralised approach for application module placement where module placement is handled by a broker that has knowledge of overall Fog architecture and resource availability of each Fog node. Since our placement approach is decentralised, simulator was extended to support this. Instead of using the existing broker class (FogBroker.java), Fog nodes were implemented according to the Fog node architecture introduced in Figure 3.3. Thus, in our implementation, each Fog node has a Fog node controller (FogNodeController.java) that handles microservices placement

(MicroservicesPlacement.java) and load balancing (LoadBalancer.java).

iFogSim provides capabilities to create hierarchical Fog architectures with multiple Fog levels. But connections are made only vertically. Horizontal connections within the same Fog level are not available. Thus, clustering of Fog nodes within the same level cannot be simulated using current iFogSim version. So, the simulator was extended to support creation of clusters by forming connections among nodes of the same Fog level.

In iFogSim, data streams are realised using an object that is characterised by source and destination application modules. As a result, when a workload is simulated, data streams are always sent up the hierarchy till a Fog node that hosts the destination module is met. This implementation is not compatible with the proposed solution due to horizontal scaling and load balancing features introduced in our policy. This requires the simulator to direct data streams based on destination device instead of destination module. Moreover, due to clustering, data streams need to be routed to clustered nodes through horizontal links as well. These features were also added to the simulator to simulate the proposed placement policy.

In the proposed model, applications are developed as a collection of microservices where each microservice is deployed on a separate container using operating system level virtualization. In iFogSim, distributed applications are modeled as a collection of modules (AppModule.java), where resource requirement of each module can be defined. Even though AppModule class is implemented as an extension of VM class in CloudSim, it can be realised as OS level virtualization of containers by defining resource requirements accordingly and changing startup delay to match that of containers.

3.6 Performance Evaluation

We evaluated our Microservice placement policy through simulation of a smart healthcare application and compared it with two existing application placement algorithms, in terms of latency and network usage of the application after placement. Moreover, we compared our distributed placement approach against centralised placement to evaluate it based on placement time of microservices within the Fog layer.

3.6.1 Experimental Configurations

To evaluate the performance of the proposed placement algorithm, we have used a synthetic workload generated by modelling a smart healthcare application on "*ECG Monitoring*" [98], [99]. Application is modeled according to the Microservices-based IoT application architecture mentioned earlier in Figure 3.2. This application uses a wearable ECG sensor that transmits data towards Level 1 Fog nodes using Bluetooth technology. Application consists of two microservices, *ECG Feature Extractor Microservice* which extracts features from ECG to detect and notify about any existing abnormal situations and *ECG Analyser Microservice* which carries out further analysis on ECG data collected and stored for a longer duration of time. *ECG Feature Extractor Microservice* provides a latency critical service and is placed on either Fog layer or Cloud according to the placement policy. *ECG Analyser Microservice* receives results from *ECG Feature Extractor Microservice* where it further processes the extracted data, so that they can be used by remote health monitoring purposes of hospitals. Service provided by this microservice is neither latency critical nor bandwidth consuming. Moreover, it requires a large amount of storage, as it stores and processes data received by *ECG Feature Extractor Microservice* to provide long term analysis. Thus, this microservice is always placed on Cloud. Data flow among different microservices of the application along with resource requirements of each microservice is depicted in Figure 3.5.

The fog environment modelled for simulations follows the system model presented in Section 3.3.1 and consists of four Fog layers with devices that are heterogeneous to each other in terms of resource availability. Clusters are formed between Fog devices in Fog Level 2 that are connected to the same Fog Level 3 device. Table 3.2 and Table 3.3 depict the parameters used in creating the Fog environment. Parameters of the simulation environment are determined based on the previous studies that model heterogeneous and hierarchical fog computing environments presented in works [9, 88, 100, 101].

3.6.2 Results and Analysis

Performance of the proposed placement policy is evaluated based on three performance metrics: latency of the latency critical path of the modeled application, network usage

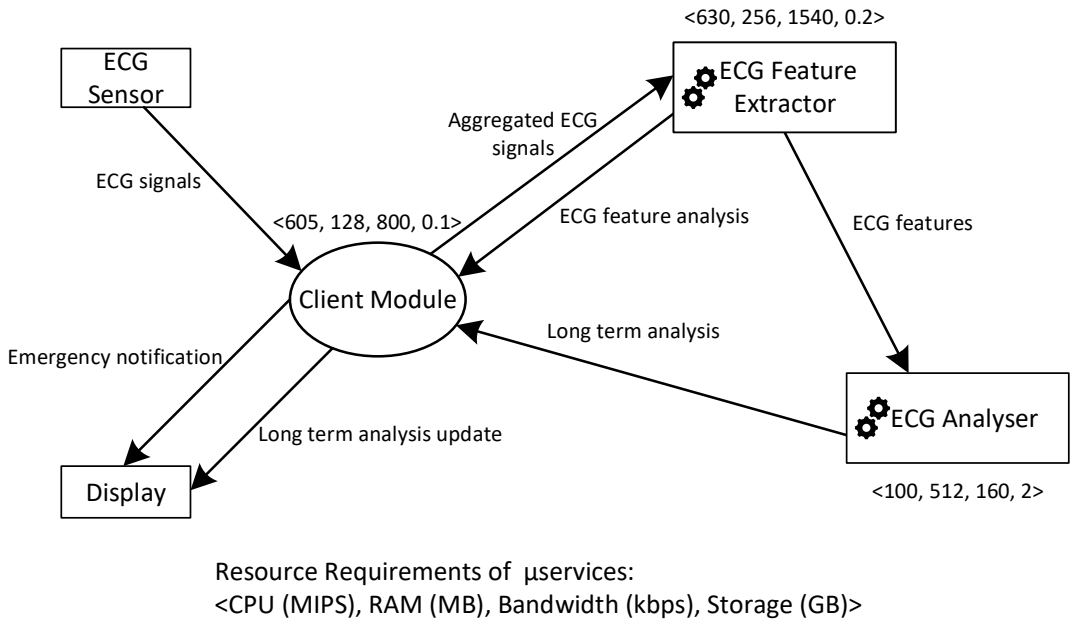


Figure 3.5: ECG monitoring application data flow and resource requirements

after placing the application and required time for application microservice placement. To evaluate performance based on latency and network usage, the proposed microservice placement policy is compared with two other placement approaches.

1. Cloud-only placement - All microservices of the application are placed within Cloud layer.

2. Edge-ward placement proposed in [9] - In this algorithm horizontal placement of the microservices across Fog node clusters is not considered. If a microservice placed on a certain Fog device does not have enough resources to handle the load, that microservice gets moved up the Fog hierarchy until a device that can handle the load is met.

The proposed microservice placement policy targets to optimise placement within Fog environments that consist of heterogeneous and resource constrained Fog nodes. Thus, three scenarios that capture the above mentioned aspects, were used for the evaluation

Table 3.2: Evaluation parameters

Parameter	Value
Latency values:	
IoT device to Fog Level 1	5ms
Fog Level 1 to Fog Level 2	20ms
Fog Level 2 to Fog Level 3	30ms
Fog Level 3 to Fog Level 4	50ms
Fog Level 4 to Cloud	150ms
Among cluster nodes	2ms
ECG sensor data transmission interval	5ms
Device count for Scenario 1-3	
ECG sensors	60
Fog level 1 nodes	60
Fog level 2 nodes	16
Fog level 3 nodes	8
Fog level 4 nodes	1
Container startup time	300ms
Placement Calculation time of a microservice	2ms
Simulation Time	120s

Table 3.3: Configuration of Fog devices

Device Type	Cloud	Fog Level 4	Fog Level 3	Fog Level 2	Fog Level 1
CPU(MIPS)	80000	10000	8000	2800-6000	1000
RAM(GB)	48	8	4	2-4	2
BW to ↑ level (Gbps)	-	100	10	10	0.15
BW to ↓ level (Gbps)	100	10	10	0.15	0.002
Cluster link BW (Gbps)	-	-	-	0.15	-
Storage(GB)	1000	256	256	128	32

of the proposed placement policy.

1. Scenario 1 - Nodes on Fog level 2 have same resource capacities. But the number of Fog level 1 nodes per each Fog Level 2 node differs.

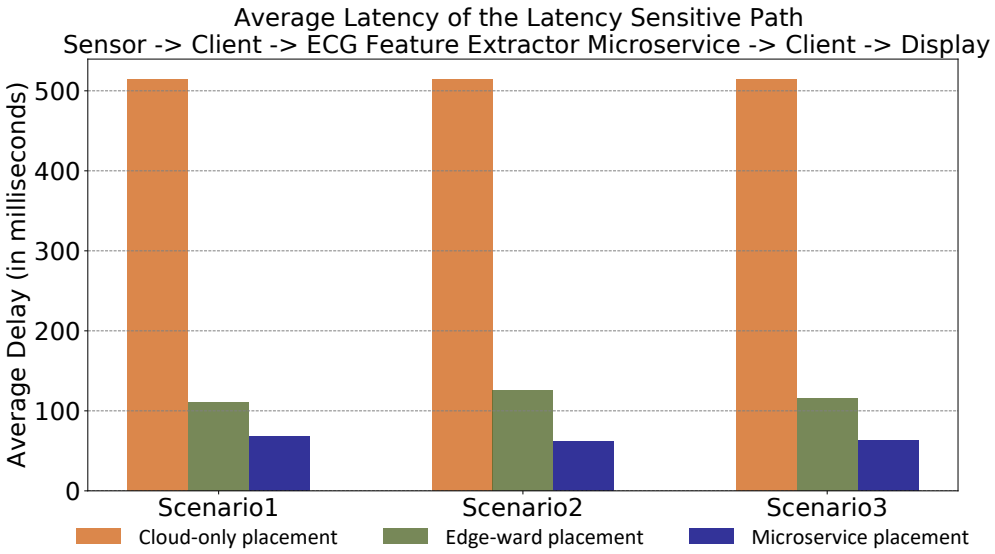


Figure 3.6: Average delay for latency sensitive path

2. Scenario 2 - Nodes on Fog level 2 have same number of Fog Level 1 nodes connected. But resource capacities among Fog Level 2 devices differ.

3. Scenario 3 - Both resource capacity and number of connected Fog Level 1 nodes differ among Fog Level 2 nodes.

All three scenarios depict heterogeneous and resource constrained Fog environments where some of the nodes get overloaded whereas others are under-utilised.

For these three scenarios, average latency of the latency sensitive loop (Figure 3.6) and average network usage (Figure 3.7) were measured after simulations.

In all three scenarios, Cloud-only placement shows a significant increase in both latency and network usage when compared to Edge-ward placement and Microservice placement approaches. Moreover, Microservice placement approach proposed in this chapter outperforms both Cloud-only placement and Edge-ward placement approaches in terms of both latency and network efficiency.

In Cloud-only placement, as all microservices are placed within Cloud layer, data generated by geo-distributed sensors have to be sent towards centralised Cloud which is multiple hops away from the edge of the network. Since all the generated data are sent towards Cloud, amount of data flowing through the core network increases, resulting

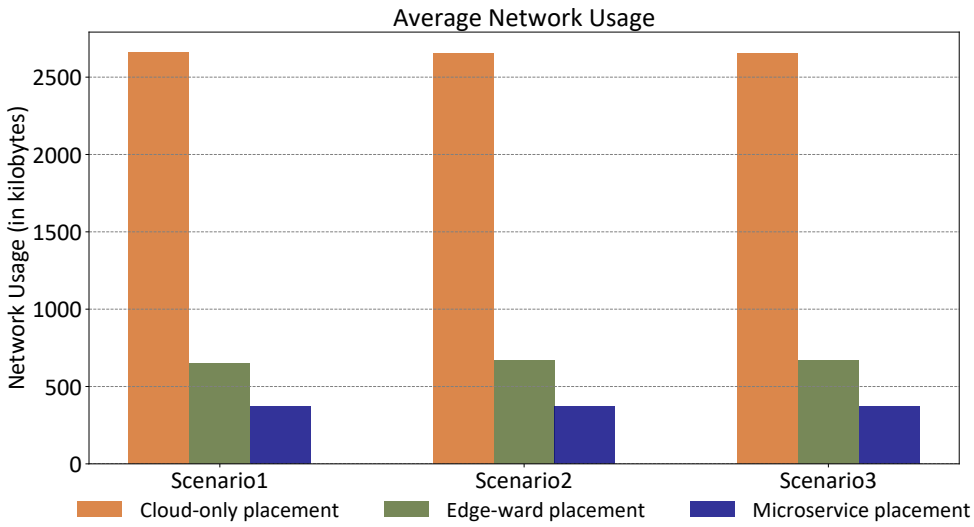


Figure 3.7: Average network usage

in network congestion. Due to these two reasons, latency of the services deployed on Cloud is significantly higher than other two scenarios where latency critical microservice is placed within Fog layer closer to the data source.

Raw data transmitted from IoT sensors requires a large amount of bandwidth. In the modeled application, *ECG Feature Extractor Microservice* analyses raw ECG data and produce results. As a result, large volumes of sensor data get reduced into meaningful information and these information gets sent towards the *ECG Analyser Microservice* and the display. Thus, if the *ECG Feature Extractor Microservice* is placed at the Fog layer, volume of data transmitted through the core network reduces dramatically. This results in efficient utilisation of bandwidth in Fog placement approaches when compared with Cloud-only placement.

In Edge-ward placement, horizontal scaling of microservices is not considered. So, if a certain instance of a microservice deployed on a Fog node does not have enough resources available to handle received workload, that particular microservice is moved up the Fog hierarchy to a node with higher resource capacity. In contrast to this, the Microservice placement approach utilises horizontal scaling and load balancing. If a certain microservice instance does not have enough resources to support the workload, microservice is horizontally scaled across nodes within clusters. These nodes are in the

same Fog level and latency among nodes within the same cluster is extremely low due to the use of light weight web transfer protocols such as CoAP. Thus, our proposed approach utilises microservices architecture to place latency critical services within lower Fog levels closer to the data source.

In all three scenarios, due to heterogeneity among resource constrained Fog nodes and difference of load on nodes, some Fog nodes gets over-utilised while others are under-utilised. Under such circumstances, proposed approach ensures that microservices are deployed in such a way so that available resources within Fog node clusters are utilised before moving towards higher level Fog nodes with higher resource availability. As a result, Microservice placement approach places modules in lower Fog levels when compared to Edge-ward placement. This results in further reduction of latency and network usage when using the proposed placement policy.

Based on the generated results, our proposed microservice placement policy demonstrates around 85% improvement over Cloud-only placement and around 40% improvement over Edge-ward placement policy, in terms of both latency and network usage.

To evaluate the efficiency of using a decentralised placement approach, we evaluated the proposed method based on total time taken to place *ECG Feature Extractor* microservice within Fog layer. We implemented the same placement algorithm using a centralised placement method and compared the placement delay with the proposed decentralised approach.

In modelling the centralised placement method, a separate Fog node on Fog Level 4 was chosen as the centralised application scheduler. A node in this level was chosen because it resides in the highest Fog level, which enables it to have a full view of all the Fog levels. Once an ECG sensor joins the network, a placement request is sent by Fog Level 1 node towards this scheduler node. It maintains a complete view of the Fog hierarchy below Fog Level 4 and calculates the nodes to place the requested microservices. This decision is sent towards the selected nodes in order to deploy an instance of a microservice on respective Fog nodes.

Experiments were carried out changing the number of sensors connected to the Fog environment. Number of sensors were increased by increasing number of Fog Level 1 devices connected to each Fog Level 2 device where each Fog Level 1 device has one

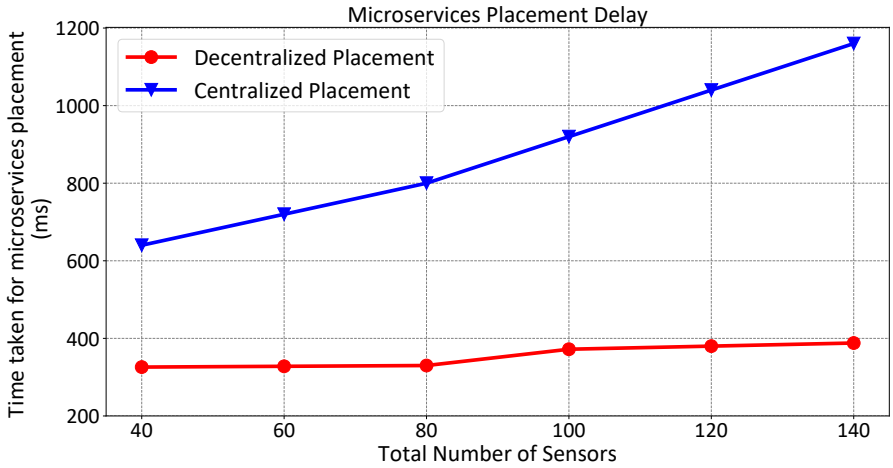


Figure 3.8: Total time taken for deployment of microservices within Fog layer

ECG sensor connected to it.

As per Figure 3.8, placement delay of the centralised method is significantly higher than the decentralised placement. In centralised approach, all placement requests have to be sent towards Fog Level 4 scheduler node which is multiple hops away. After processing the request and selecting a Fog node to place the microservice, scheduler node has to inform this to the selected node placed within lower Fog levels. This induces a communication delay on all placement requests. But, in decentralised approach, placement request processing starts from Fog Level 1 node and the placement requests propagate up the hierarchy until suitable nodes are met for the deployment of microservices. This results in significantly lower placement delay in decentralised placement.

Moreover, in centralised management all requests are sent towards a central scheduler node. Thus, as the number of sensors increases, the number of placement requests that needs to be processed by the centralised scheduler is higher than that of each Fog node in decentralised case. As a result, there's a rapid increase in placement delay for centralised management whereas increase of delay is quite small in decentralised approach as the number of sensors increases.

As the number of placement request increases, placement of microservices is moved towards upper level Fog nodes with higher resource availability. Slight increase of placement delay in decentralised placement is caused due to this. Our results show that due

to highly distributed nature of the Fog nodes it is much efficient and scalable to use decentralised placement and service discovery methods.

3.7 Summary

In this chapter, we explored the potential of microservice architecture in improving the performance of IoT applications within Fog environments by considering three main characteristics of microservices: independent deployability, independent scalability, and decentralised management. To this end, we proposed a decentralised placement algorithm for microservices-based IoT applications, highlighting the horizontal scalability of microservices within resource-constrained and heterogeneous Fog nodes. Moreover, we proposed a Fog node architecture to support decentralised placement through distributed placement algorithm execution, dynamic service discovery and load balancing.

We conducted simulation-based experiments using iFogSim simulated Fog environment to demonstrate the performance of the proposed solution. Based on the simulation results, the proposed placement policy significantly reduced latency and network usage within heterogeneous and resource-constrained Fog environments. We also compared our approach with a centralised placement approach, highlighting the suitability of the decentralised management within Fog environments in terms of application placement delay and scalability of placement.

This chapter presented a placement algorithm that sequentially processes application placement requests in a first come, first serve manner to reduce latency and network usage by placing microservices as close as possible to the network edge. With the diversity in performance requirements of IoT application services (i.e., latency-critical, latency-tolerant, bandwidth-hungry, computation-intensive, etc.) and conflicting QoS requirements (i.e., latency vs budget), placement algorithms can be improved by integrating QoS-aware dynamic prioritising and multi-objective optimisation to meet multiple QoS parameters. Thus, in the next chapter, we study QoS-aware batch placement of microservices-based IoT applications to improve performance satisfaction in terms of throughput, makespan and budget while dynamically utilising Fog and Cloud resources.

Chapter 4

QoS-aware Batch Placement Approach for Heterogeneous IoT Applications

Microservice architecture improves the granularity of application decomposition, thus providing scope for improvement in QoS-aware placement of diverse IoT applications within distributed, heterogeneous and resource-constrained Fog environments. In this chapter, we harvest the characteristics of microservice architecture to propose a scalable QoS-aware application placement policy for batch placement of microservices-based IoT applications within Fog environments. Our proposed policy, QoS-aware Multi-objective Set-based Particle Swarm Optimisation (QMPSO), aims at maximising the satisfaction of multiple QoS parameters (makespan, budget and throughput) while focusing on the balanced utilisation of Fog and Cloud resources. Besides, QMPSO adapts and improves the Set-based Comprehensive Learning Particle Swarm Optimisation (S-CLPSO) algorithm to achieve better convergence in the Fog application placement problem. We evaluate our policy in a simulated Fog environment. The results show that compared to the state-of-the-art solutions, our placement algorithm significantly improves QoS in terms of makespan satisfaction (up to 35% improvement) and budget satisfaction (up to 70% improvement) and ensures optimum usage of computing and network resources, thus providing a robust approach for QoS-aware placement of microservices-based heterogeneous applications within Fog environments.

This chapter is derived from:

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "QoS-aware placement of microservices-based IoT applications in Fog computing environments", *Future Generation Computer Systems (FGCS)*, Volume 131, Pages: 121-136, ISSN: 0167-739X, June 2022.

4.1 Introduction

The growing popularity of the IoT paradigm has resulted in a rapid increase in the number of smart devices generating data and IoT applications processing the generated data. With the rise in the number and the diversity of IoT services provided by these applications, the optimal usage of limited Fog resources becomes a significant challenge due to the competing QoS requirements among services. To overcome this challenge, intelligent placement algorithms must be developed to dynamically place applications across Fog and Cloud resources in a QoS-aware manner.

Microservices architecture decomposes applications into fine-grained components, thus giving rise to composite IoT services, where end-to-end services accessed by the users can consist of multiple inter-connected microservices. A microservices-based IoT application would contain multiple such services with heterogeneous quality requirements and characteristics (i.e., latency-critical, latency tolerant, high bandwidth, etc.). Moreover, due to the fine granularity of the architecture, microservices within composite services can have complex data dependencies where some microservices can also be part of more than one service [60]. Such application design enables the definition of per-service quality requirements in terms of parameters like makespan, budget, and throughput. By properly utilising this information and handling the complexities introduced by the granularity of inter-connected microservices, application placement policies can optimally harness both Fog and Cloud resources to maximise QoS satisfaction.

Due to latency and bandwidth improvements at the edge of the network, resource providers can charge higher prices for Fog resources compared to the Cloud resources [102, 103], which has led many existing placement policies to consider minimising total latency and cost to reach a trade-off between the two [33, 104, 105]. But due to limited resources, makespan and budget aware prioritising is crucial to distribute Fog resources among competing applications or services. Per service makespan and budget expectations defined for microservice applications along with batch placement can enable this. Moreover, throughput expectations of the services can be used to reap the benefits of vertical and horizontal scalability of the microservices to make maximum use of heterogeneous Fog resources to minimise the effect of resource limitations on application

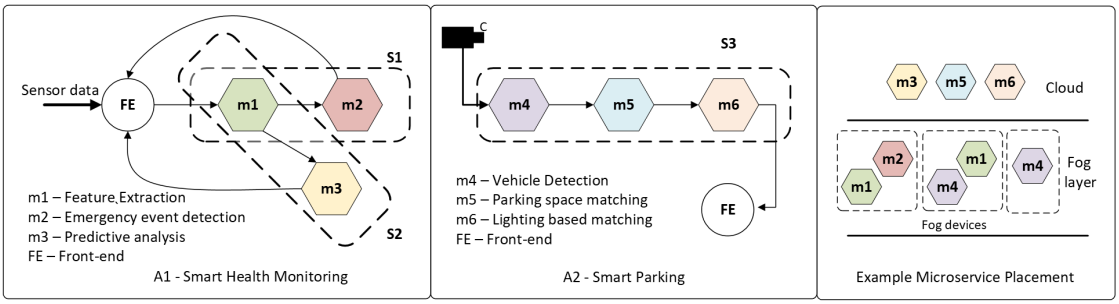


Figure 4.1: Example scenarios for IoT application placement

performance.

As a motivating scenario, Figure 4.1 presents a smart health care application for patient monitoring ($A1$) [106] and a smart city application for parking occupancy detection ($A2$) [107]. $A1$ consists of three microservices that communicate together to provide two services to the user. m_1, m_2 form a latency-critical emergency alert service, whereas $m1$, $m3$ form a latency tolerant service that generates long term analysis reports for the user. $A2$ consists of three microservices forming a single service that detects parking spot occupancy in real-time. Due to the microservice-based decomposition of the applications, QoS requirements (i.e, makespan, budget, throughput etc.) can be defined separately for each service. Using this knowledge, placement decisions can be made to satisfy the QoS requirements of each service by utilising both Fog and Cloud resources (i.e, $m1$ and $m2$ are mapped to Fog layer devices to satisfy stringent latency requirements; $m4$ is mapped to Fog layer to reduce the amount of data sent towards the Cloud; $m3$, $m5$ and $m6$ are mapped to the Cloud to satisfy their high computation resource requirements). Furthermore, microservices placed in Fog can be horizontally scaled to satisfy the throughput requirements of the services under resource limitations of Fog devices (i.e, two instances of $m1$ placed on two separate Fog devices when a single device doesn't have enough processing power to support the request rate). $A1$ and $A2$ represent two heterogeneous applications trying to utilise Fog resources. In the example scenario, the service S_1 in $A1$ is more latency-critical compared to the service $S3$ in $A2$ and due to the resource-constrained and heterogeneous nature of the Fog devices, batch placement of applications has the potential to prioritise $S1$ over $S3$ to ensure QoS satisfaction.

Although Fog application placement has been studied extensively, microservices ar-

chitecture provides a novel perspective, where per service quality requirements and independently scalable nature of the microservices can enable harnessing the power of both the Fog devices and Cloud data centres to improve application performance through batch placement. Research on this is still at its early stages and has much room for improvement. Therefore, in this work, we propose a QoS-aware placement algorithm that improves the total QoS satisfaction considering multiple QoS parameters (makespan, cost, and throughput) and at the same time, ensures optimum resource usage through collaboration among Fog and Cloud resources. **The key contributions** of our work are as follows:

1. We formulate the Fog application placement problem as a Lexicographic Combinatorial Optimisation Problem considering QoS satisfaction (in terms of makespan, budget, and throughput) as the primary objective and optimum resource usage as the secondary objective.

2. We propose an IoT application batch placement technique based on Set-based Comprehensive Learning Particle Swarm Optimisation (S-CLPSO). To improve the convergence rate, we introduce a heuristic-driven swarm initialisation and fitness parameter normalisation method and further incorporate a priority-based particle construction technique to overcome premature convergence due to the resource constraints of the Fog devices.

3. We implement our policy using iFogSim2 [8] simulated Fog environment and compare it against existing scheduling approaches based on their resultant QoS satisfaction and balanced Fog and Cloud resource usage.

The rest of the chapter is organised as follows. Section 4.2 highlights related research followed by Section 5.3, which presents system architecture. In Section 4.4, the Fog application placement problem and our proposed solution is detailed. Section 4.5 presents performance evaluation and Section 4.6 concludes the chapter and draws future work and research directions.

4.2 Related Work

In this section, we summarise existing work on Fog application placement, compare them based on their key features and also provide a detailed background on Particle Swarm Optimisation (PSO) algorithm and its derivatives used in designing our placement policy.

4.2.1 Application Placement in Fog Environments

Existing research propose numerous algorithms to schedule applications within Fog environments. They mainly fall under two categories: application offloading and application service placement, where offloading deploys application modules from client devices to the Fog to be used by each client separately while service placement refers to the deployment of application services in the Fog so that multiple clients can use them [32]. Since our work focuses on the latter, in this section we summarise research related to the application service placement in Fog.

Brogi et al. [108] present a placement policy to place multi-component applications within a Fog environment when inter-component link bandwidth and latency requirements are defined. They propose a heuristic placement algorithm consisting of two steps, where it first searches for all eligible nodes to host each application component based on its software and hardware requirements and then employs a greedy backtracking algorithm to place each component considering inter-component latency and bandwidth requirements. Yousefpour et al. [105] implement a framework that supports dynamic deployment and release of IoT services. Their work presents two separate greedy algorithms for minimising delay violation and minimising total cost for IoT services with delay constraints. In their work, each service is an independent task with an expected deadline for its completion and applications are built as a collection of such independent services. Skarlat et al. [33] propose a deadline-aware policy using Integer Linear Programming (ILP) to place applications within micro data centres know as Fog colonies. They model applications as a set of independent tasks and define a deadline for the entire application. Their placement policy prioritises applications based on the deadline and tries to maximise the placement of applications within the Fog layer such

that for each application, the total deployment and execution time of the tasks does not exceed the application deadline. Skarlat et al. [109] extend the work proposed in [33] and solve the proposed optimisation problem using GA. GA based approach is evaluated against a mathematical programming based optimisation method. Results indicate that the GA algorithm can reduce deployment delays. Xie et al. [104] present a workflow application scheduling algorithm based on Particle Swarm Optimisation aiming to minimise the weighted sum of total latency and cost. They propose a non-local convergent PSO algorithm introducing a non-linear inertia weight calculation method along with a directional search process.

Deng et al. [38] form the microservices-based application scheduling problem in Fog to minimise the cost of application deployment adhering to resource constraints and expected response time of the mobile services. The placement problem is solved through ILP. Their placement algorithm handles only the placement of a single application each turn. Thus, prioritising applications based on their latency requirements is not captured in their work. Guerrero et al. [32] compare three evolutionary algorithms; Weighted Sum Genetic Algorithm (WSGA), non-dominated sorting genetic algorithm (NSGA-II) and multi-objective evolutionary algorithm based on decomposition (MOEA/D), for solving Fog service placement focusing on optimising latency, service spread and use of resources.

Chen et al. [110] apply Set-based Comprehensive Learning Particle Swarm Optimisation (S-CLPSO) algorithm for workflow application scheduling in Cloud environments to satisfy user-defined QoS constraints in terms of deadline, budget and reliability. The proposed algorithm allows the user to select one of the QoS parameters as the optimisation objective while keeping the other two as constraints. Meta-heuristic algorithm combined with multiple heuristics to speed up the search process provides better results in satisfying QoS requirements. Verma et al. [111] propose a non-dominated sorting based PSO for minimisation of execution time, cost and energy consumption for workflow scheduling in Cloud environments.

Table 4.1 compares features of the related works with our work in terms of three main categories; application model, placement properties and algorithm type. Features analysed under the application model aim to capture the granularity of the components

Table 4.1: Comparison of existing application placement policies

Work	Environment		Application Model							Placement Properties								Algorithm
	Cloud	Fog/ Edge	μservice architecture	QoS granularity	Application Composition				Batch placement	Decision Parameters							Scalability	type
					services	service composition				QoS-aware			QoS-unaware		Resource			
					per app	single	chained	aggregator		makespan	budget	throughput	total makespan	total budget	usage			
[108]		✓		per link	multiple	✓	✓			✓		✓						heuristic
[105]		✓		per service	multiple	✓				✓		✓		✓				heuristic
[33]		✓		per app	single	✓	✓		✓	✓								ILP solver
[109]		✓		per app	single	✓	✓		✓	✓								meta-heuristic
[104]		✓		per app	single	✓	✓	✓					✓	✓				meta-heuristic
[38]		✓	✓	per app	single	✓	✓			✓		✓		✓			✓	ILP solver
[32]		✓	✓	per app	multiple	✓	✓						✓		✓		✓	meta-heuristic
[110]	✓			per app	single	✓	✓	✓		✓	✓							meta-heuristic
[111]	✓			per app	single	✓	✓	✓		✓	✓							meta-heuristic
our work		✓	✓	per service	multiple	✓	✓	✓	✓	✓	✓	✓			✓		✓	meta-heuristic

(i.e., microservices, modules) of the application and their data dependencies. Application composition analyses each application model based on the number of services and service composition based on the collaboration pattern of the components that work together to perform a service. In literature, the term ”service” is used for modules, microservices, components, processes etc. In our work, we define service from a user perspective where it represents a business functionality accessed by the user. Due to the fine-grained design of microservices, a service can consist of multiple microservices communicating together to perform a service. As existing works do not capture these features properly, we model our application placement problem to represents the granularity of the microservice design.

Placement properties characterise the works based on their ability to perform batch placement, decision parameters, Fog/Cloud balanced resource utilisation and scalability of application components. Decision parameters are analysed under two categories: QoS-aware parameters that represent quality expectations of the services in terms of makespan, budget and throughput, QoS-unaware parameters which focus on total makespan and budget of the placement irrespective of their QoS expectations. Due to resource constraints within Fog environments, Fog application placement can benefit from batch placement, QoS-awareness and optimum use of Fog resources, which allow prioritising of services with stringent QoS requirements to achieve a balance between Fog and Cloud resource usage. Moreover, developing applications as microservices enables each microservice to scale independently, so that each microservice can be

vertically or horizontally scaled based on the resource availability of heterogeneous Fog devices.

Considered related works use three main types of algorithms to solve the application placement problem; heuristic, meta-heuristic and mathematical programming. As heuristic and greedy algorithms are unable to handle multiple objectives, [105, 108] propose multiple heuristic algorithms where each focuses on one of the decision parameters. Mathematical programming-based approaches can only obtain the optimum solution when search space is small and not suitable for batch placement problems. Thus, due to the ability to handle multiple objectives and also to reach near-optimum solutions faster, meta-heuristic algorithms have become a popular approach in solving multi-objective optimisation problems. Meta-heuristics such as Genetic Algorithm (GA), Ant Colony Optimisation (ACO), Particle Swarm Optimisation (PSO) are popularly used in solving scheduling problems in both single objective and multi-objective scenarios. However, one of the main challenges when adapting meta-heuristics to the Fog application placement is handling Fog resource constraints without trapping the algorithm to a local optimum. This issue is not properly addressed in existing works in both Fog and Cloud placements. Proper use of heuristics to populate the initial solution and efficiently normalise weighted parameters is another area with scope for improvement. So in our work, we propose a placement technique based on Set-based Comprehensive Learning Particle Swarm Optimisation (S-CLPSO) and improve it to reach a near-optimum solution for batch placement of microservices-based IoT applications.

4.2.2 Particle Swarm Optimisation

Particle swarm optimisation (PSO) is a population-based meta-heuristic algorithm [112], which was originally introduced for the optimisation of problems defined in continuous solution space. In PSO, a set of solutions identified as a swarm of particles moves within the solution space using not only its own experience but also the experiences of other particles. Each particle is characterised based on two factors; its position and velocity. In each iteration, particles update their velocity taking their own best position (*pbest*) and best position of the swarm (*gbest*) into consideration and modify their position to

move towards a better solution within the solution space. PSO is simple in concept, computationally inexpensive and has a higher convergence rate due to social sharing of information among particles in the swarm and use of previous experiences of particles for the decision making process.

As the traditional PSO is not designed for solving discrete optimisation problems, multiple approaches have been introduced to adapt PSO for discrete cases, including Binary PSO (BPSO) and Discrete PSO (DPSO). Among these, Chen et al.[113] propose a Set-based PSO (S-PSO) that can be used for solving Combinatorial Optimisation Problems (COPs) in the discrete space and demonstrate that this approach can efficiently navigate within discrete solution space and successfully solve COPs.

S-PSO employs a set-based representation of particles where particle position is depicted as a crisp set, whereas the velocity is a set with possibilities. In [113], COP is formulated as "finding a set of candidate solutions X which is a subset of the universal set of elements E, such that X satisfies some pre-defined constraints Ω and optimises the objective function f".

For a universal set E divided into N dimensions, velocity and position updating functions for n^{th} dimension of k^{th} particle are defined as,

$$V_k^n = \omega V_k^n + c_1.r_1^n.(pbest_k^n - X_k^n) + c_2.r_2^n.(gbest_k^n - X_k^n) \tag{4.1}$$

$$X_k^n = X_k^n + V_k^n \tag{4.2}$$

where, ω is the inertia weight that controls the momentum of the particle, c_1 and c_2 are learning factors related to particle's own experience and swarm's experience respectively and r_1, r_2 are random values in the range [0,1]. Equation 4.1 depicts the velocity updating rule proposed by the original PSO algorithm. This tends to get trapped in local optimum solutions, specially when applied for discrete optimisation problems. Thus, [114], shows that for their proposed S-PSO for discrete space, Comprehensive Learning Particle Swarm Optimisation(CLPSO) algorithm [115], which is a variant of PSO, gives better performance.

CLPSO uses the following equation for velocity updating:

$$V_k^n = \omega V_k^n + c.r_1^n.(pbest_{f_k(n)}^n - X_k^n) \tag{4.3}$$

where $f_k(n)$ depicts the particle whose *pbest* is used by k^{th} particle for updating its n^{th} dimension. In this approach, instead of using *gbest* of the swarm, *pbest* of any particle including its own can be used. CLPSO uses tournament selection to select $f_k(n)$ depending on a probability (P_c) known as learning probability or uses its own *pbest*. To ensure that particles don't move towards poor directions, these exemplars are updated after a certain number of iterations (refreshing gap m), if the particle fitness fails to improve.

For the calculation of the new velocity, the following set-based operations are defined,

- Coefficient $\times$ Set with possibilities

  For a coefficient $c \geqslant 0$ and a set with possibilities defined on universal set E, depicted as $V = \{e/p(e)|e \in E\}$, product of the two is a set of possibilities $cV = \{e/p'(e)|e \in E\}$ calculated as,

$$p'(e) = \begin{cases} 1 & \text{if c} \times \text{p(e)} > 1 \\ c \times p(e) & \text{otherwise} \end{cases} \tag{4.4}$$

- Crisp Set - Crisp Set

  For two crisp sets X_1 and X_2, minus operator between the two ($X_1 - X_2$) is defined as the crisp set of elements that are available in X_1, but not in X_2.

- Coefficient $\times$ Crisp Set

  For a coefficient $c \geqslant 0$ and a crisp set $X \in E$, product of the two results in a set of

possibilities, $cX = \{e/p'(e)|e \in E\}$ calculated as,

$$p'(e) = \begin{cases} 1 & \text{if e} \in \text{X and c}>1 \\ c & \text{if e} \in \text{X and c}\leq 1 \\ 0 & \text{if e} \notin \text{X} \end{cases} \tag{4.5}$$

- Set with Possibilities + Set with Possibilities

  Plus operator between two sets with possibilities generates a set with possibilities containing larger possibility for each element.

Updated velocity is used to adjust the position. Since solutions in discrete space should meet a pre-defined set of constraints, feasible positions are obtained using two main strategies; step-by-step construction, build and repair [114]. ω is used to achieve exploitation and exploration to overcome local optimums and move towards the global optimum of the problem. As larger values of ω supports global search whereas local search is supported by small ω values, changing ω from larger values to smaller values through iterations enables the algorithm to converge into the global optimum value. The most common method of achieving this is by linearly changing ω over the iterations. But, [104] presents a non-linear function for varying ω that results in improved convergence.

Set-based CLPSO (S-CLPSO) is a successful method for solving COPs in discrete space with a higher convergence rate and simple implementation. In our work, we base our placement policy on S-CLPSO, integrate a lexicographic fitness function and further improve its performance by integrating multiple heuristics and propose a prioritised particle construction method to handle Fog resource constraints to mitigate the issue of converging to a local optimum solution.

Table 4.2: Notations

Symbol	Definition	Symbol	Definition
F	Set of all devices available in Fog layer.	C	Cloud.
E	Set of all client devices that connects to Fog Gateways.	D	All available devices. ($F \cup C \cup E$)
A	Set of all requested applications for placement.	M_a	Set of all microservices of application $a \in A$.
S_a	Set of all services defined for application $a \in A$.	M_a^s	Set of microservices of service $s \in S_a$.
P_a^s	Set of data paths in service $s \in S_a$.	df_p^s	Set of all data flows in path $p \in P_a^s$.
$\Delta_{mm'}$	Size of data transmitted from m to m'.	$i_{mm'}$	Number of instructions to process data sent from m to m'
$R_{mm'}$	Access rate among microservices m & m'.	d_{ij}^p	Network propagation delay among device $i, j \in D$
l_s	makespan requirement of service $s \in S_a$.	b_s	Budget requirement of service $s \in S_a$.
r_s	Throughput requirement of service $s \in S_a$.	ϕ_d	Processing capacity of device $d \in D$
ω_d	RAM of device $d \in D$.	γ_d	Storage capacity of device $d \in D$.
Φ_m	Processing capacity required by microservice $m \in a$.	Ω_m	RAM required by microservice $m \in a$.
Γ_m	Storage capacity required by microservice $m \in a$.	v_s^l	makespan violation of service $s \in S_a$.
v_s^b	Budget violation of service $s \in S_a$.	v^l	Total makespan violation.
v^b	Total budget violation.	$\eta(v^l)$	Normalised makespan.
$\eta(v^b)$	Normalised budget.	τ_{nw}	Total network usage due to placement.
τ_r	Total active devices due to placement.	Y	Set of devices $Y \subset D$, that are not eligible for placement of any microservice
$x_{m_i}^d \in \{0,1\}$	Equals to 1 if i^{th} instance of microservice m is mapped to $d \in D$, 0 otherwise.	$act_f \in \{0,1\}$	Equals to 1 if at least one microservice is placed on $f \in F$, 0 otherwise.

4.3 System Model and Architecture

This section details the microservices-based application model, Fog architecture along with the pricing model used in this work. Table 4.2 summarises all the notations used in this chapter.

4.3.1 Application Model

To support the rapid modifications and agile development of IoT applications, microservices architecture is used to design and develop these applications. As microservices are

designed as independently deployable modules adhering to a single business capability, the number of components that builds a single application increases. Due to the higher level of granularity presented by microservices architecture, a single service can consist of multiple microservices that collaborate to complete end-user requests. Moreover, a single microservice can be used by multiple services as well. So, higher flexibility and agility can be achieved by defining QoS parameters at these composite service level, instead of at the microservice or application levels.

Figure 4.2.a shows the general representation of a microservices-based IoT application. Application is depicted using a DAG where vertices represent microservices and edges denote the data dependencies among microservices. The starting point of the arrow indicates the client microservice and the arrowhead indicates the microservice invoked by the client microservice. Each application consists of a front-end denoted as *Client Module* that is always placed within client devices such as mobile phones, tablets, laptops that connect directly with IoT devices. The rest of the application consists of microservices that are placed either on Fog or Cloud resources based on the placement policy. Each application, $a \in A$ can be depicted as a tuple containing a set of microservices, data flows among them and a set of services where microservices collaborate to perform a function useful to the end-user; $< M_a, df^a, S_a >$. Each microservice is characterised by its resource requirements; $< \Phi_m, \Omega_m, \Gamma_m, r_m >$ indicating CPU, RAM and storage requirement of microservice $m \in M_a$ to support the request rate of r_m. This resource definition is used as the basic deployment unit of the microservice container and it is scaled horizontally or vertically based on the expected rate of the requests received by the microservice.

Based on the collaboration patterns among microservices, we have identified 3 types of service representations; *Chained*, *Aggregator* and *Hybrid* representation (Figure 4.2.b). In a chained pattern, data flow within the service can be represented as a single chain whereas in an aggregator pattern, multiple data paths are invoked and the aggregator microservice waits for the processed data from those paths and aggregates them to return a single response. Aggregator microservice can invoke chains of microservices as well, which results in a hybrid representation. Thus, the completion time of each service differs based on the collaboration pattern of the microservices in the service. Microser-

vices use the Asynchronous Request-Reply pattern, so once a request is made client microservice proceed to process other incoming requests until the response arrives. Each service $s \in S_a$ is denoted by a tuple containing a set of microservices creating the service and all possible data paths of the service; $< M_a^s, P_a^s >$. The number of data paths in each service depends on the collaboration pattern of the microservices in the service.

The QoS profile of each application consists of QoS parameters that are defined separately per each service within the application. Our work considers makespan: end-to-end completion time of the service, budget: the amount the user expects to pay for the service and throughput: supported request rate by the service, as QoS requirements.

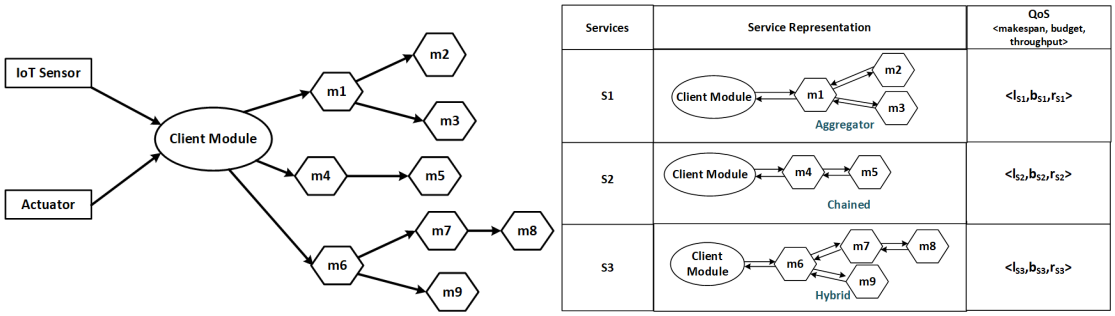


Figure 4.2: Microservices-based IoT application architecture (a) DAG representation, (b) Service composition patterns

4.3.2 Fog Architecture

Fog computing environment is a multi hierarchical environment consisting of IoT/client devices, Fog layer and Cloud layer. The Fog layer is an intermediate layer that resides between the IoT and Cloud layer, thus providing computational, networking and storage capabilities closer to the edge of the network. Figure 4.3 depicts the architecture followed in this work. The Fog layer consists of clusters of Fog nodes deployed by multiple service providers. IoT sensors and actuators that connect to client devices (i.e., mobile phone, tablets) access Fog clusters through *Fog Gateway Devices* (i.e., wireless access points, base transmission systems) and further connection to the Cloud is maintained through a *Fog Cloud Gateway* node. We refer to this node as the *Fog Orchestration Node (FON)*, as it's responsible for monitoring Fog nodes in the cluster and scheduling

applications within them.

The Fog layer consists of heterogeneous devices in terms of resource availability and access technologies. Each Fog device ($f \in F$) is characterised by its resources in terms of CPU (ϕ_f), RAM (ω_f) and storage (γ_f). Fog nodes within the same cluster communicate with each other using Local Area Network (LAN) links which have considerably high bandwidths when compared with the Wide Area Network (WAN) links that connect Fog clusters to the Cloud. Multiple IoT and client devices use Wireless Local Area Network (WLAN) to connect to *Fog Gateway Devices*.

4.3.3 Pricing Model

Due to distributed, scalable and independently deployable nature of the microservices, container technology has become the best-suited method of packaging and deploying microservice applications. Cloud service providers have server-less compute engines to support easy container deployment, by relieving the users of the responsibility to provision and manage servers. Such server-less platforms provide flexible pricing where users pay only for the amount of resources used by the containers. AWS Fargate [116] and Azure Container Instances [117] determine the pricing based on requested vCPUs, memory and storage amount where all three can be configured independently. In our work, we use the on-demand pricing models used by the above server-less platforms where the price of each Fog/Cloud device is defined as the total price for vCPUs, memory and storage.

4.4 QoS-aware Application Placement

We formulate the microservices-based application placement in Fog environments as a *"Lexicographic Multi-objective Combinatorial Optimisation Problem"*, which aims at minimising QoS violation of services and ensures optimum use of Fog and Cloud resources while adhering to the resource requirements of the microservices. The proposed policy explores batch placement of services and also incorporate independently scalable nature of the microservices to obtain a more efficient placement.

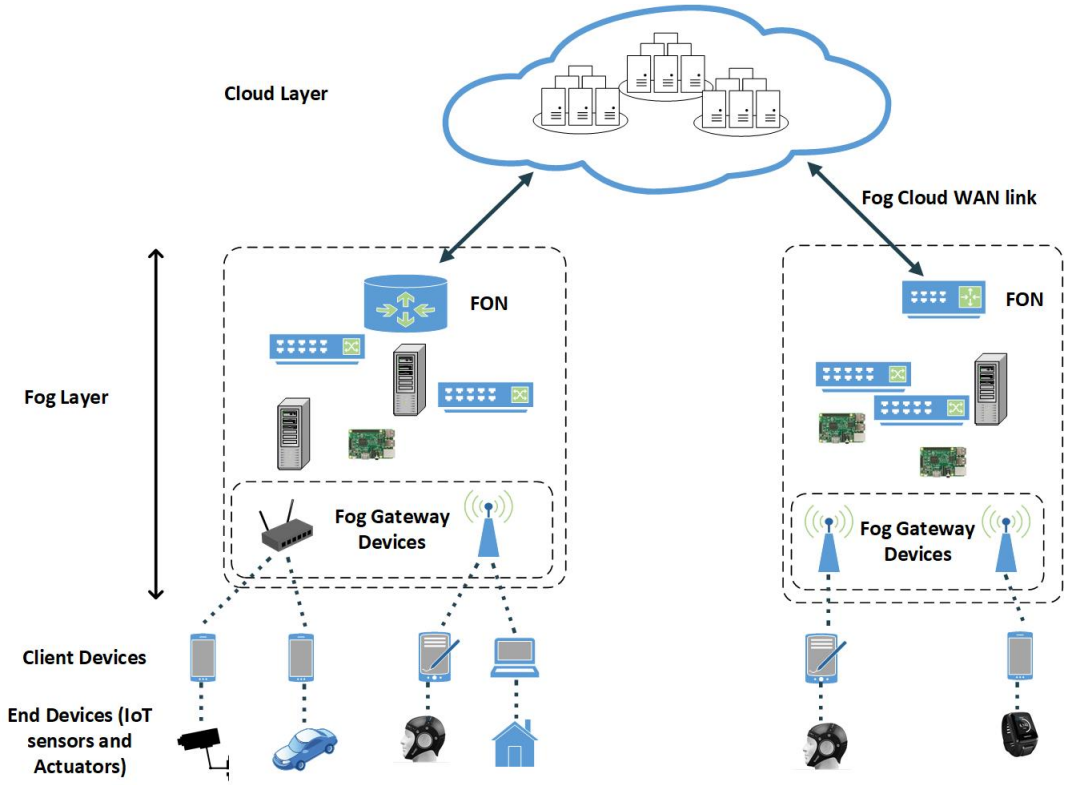


Figure 4.3: An overview of the Fog architecture

4.4.1 Problem Formulation

Placement approaches within Fog environments should aim to maximise the QoS satisfaction of the application services while utilising the limited amount of Fog computing resources. As depicted in Section 4.1 through a motivation scenario, such placement approaches require the incorporation of knowledge related to heterogeneous QoS requirements at the service level (i.e., capturing requirements such as latency, budget and throughput at composite service level) to meet QoS requirements and further knowledge on microservice heterogeneity (i.e., capturing computation and bandwidth requirements of the microservices) in terms of their resource consumption. To this end, Fog application placement problem needs to focus on both application and Fog resource heterogeneity and prioritise microservices for limited Fog resources and rest for resource-rich Cloud. Thus, we formulate the placement problem to support the placement of a batch of applications (A) onto a set of devices (D) within the Fog environment. As

microservices are independently scalable, multiple instances of each microservice can be deployed. For each microservice m, number of instances is denoted by ins_m where m_i represents the i^{th} instance. Resultant placement is expressed by $x^d_{m_i}$ where $x^d_{m_i} = 1$ depicts that the i^{th} instance of microservice m is mapped to $d \in D$.

The main goal of the placement is to achieve maximum possible QoS satisfaction considering makespan, budget and throughput requirements. The throughput requirement of each service is satisfied by scaling the microservices. Throughput aware instance calculation for the microservices is discussed in detail in Section 4.4.1. Makespan and budget satisfaction are achieved by the first objective of the optimisation problem (equation 4.6). Due to resource limits in the Fog layer, it is not guaranteed that makespan and budget requirements of all services can be satisfied. Thus, we formulate equation 4.6 to minimise the weighted sum of normalised makespan violation ($\eta(v^l)$) and budget violation ($\eta(v^b)$) so that services with stringent QoS requirements are prioritised.

The purpose of the Fog layer is twofold: to support latency-critical services and to reduce network usage by supporting bandwidth-hungry services. While the first objective moves latency-critical services towards the edge of the network, it does not focus on bandwidth-hungry microservices that are part of latency-tolerant services. Thus, the second objective is introduced with a sub-objective to ensure that such microservices are moved towards the edge of the network. At the same time, due to the resource-constrained nature of Fog devices, it is crucial to strike a balance between Fog and Cloud resource usage to avoid over-use of limited Fog resources. To this end, we formulate the second objective (equation 4.7) to achieve a trade-off between total network resource usage (τ_{nw}) and total Fog resource usage (τ_r). Hence, the second objective aims to improve the QoS-aware placement achieved by the first objective, by enhancing resource utilisation through the dynamic and balanced use of Fog and Cloud resources.

As QoS satisfaction is the primary objective of the placement, the second objective can be considered as a further improvement on the schedule proposed by the first objective. To ensure this, the placement problem is solved as a lexicographic optimisation where equation 4.6 is the primary objective and equation 4.7 is the secondary objective.

$$\text{minimise } [\omega_l \ \eta(v^l) + \omega_b \ \eta(v^b)] \tag{4.6}$$

$$\text{minimise } [\omega_{nw}\ \eta(\tau_{nw}) + \omega_r\ \eta(\tau_r)] \tag{4.7}$$

subject to,

$$\sum_{\forall d \in D} x^d_{m_i} = 1; \forall i \in [1, ins_m], \forall m \in M_a, \forall a \in A \tag{4.8}$$

$$\sum_{\forall d \in Y} \sum_{i=1}^{ins_m} x^d_{m_i} = 0; \forall m \in M_a, \forall a \in A \tag{4.9}$$

$$\sum_{\substack{\forall a \in A \\ \forall m \in M_a}} \sum_{i=1}^{ins_m} x^d_{m_i} \Phi_m \leq \phi_d; \forall d \in D \tag{4.10}$$

$$\sum_{\substack{\forall a \in A \\ \forall m_a \in M_a}} \sum_{i=1}^{ins_m} x^d_{m_i} \Omega_m \leq \omega_d; \forall d \in D \tag{4.11}$$

$$\sum_{\substack{\forall a \in A \\ \forall m \in M_a}} \sum_{i=1}^{ins_m} x^d_{m_i} \Gamma_m \leq \gamma_d; \forall d \in D \tag{4.12}$$

Both objectives are optimised under multiple constraints; placement constraints where each microservice instance is mapped to a single device (equation 4.8) and each microservice is mapped only to eligible devices (equation 4.9), resource constraints of all Fog devices in terms of CPU, RAM and storage (equation 4.10, 4.11, 4.12 respectively).

Throughput aware instance count calculation

In our proposed system model, resource requirements of each microservice are defined to support a certain request rate (r_m). We take this as the base instance and scale each microservice vertically or horizontally according to the expected service throughput (r_s per service s) in the application's QoS profile. Using the DAG representation of the microservice application, the expected request rate of each microservice (r'_m) can be cal-

culated using the following equations:

$$r'_m = \sum_{\forall m' \in CM(m)} R_{m'm} \tag{4.13}$$

$$R_{m'm} = \begin{cases} r_s & m' \text{ is Client Module} \\ \alpha . r'_{m'} & \text{otherwise} \end{cases} \tag{4.14}$$

Function $CM(m)$ calculates all the client microservices of microservice m, that sends requests towards m. Thereby, equation 4.13 calculates access rate of the microservice m by taking summation of the request rates of all incoming edges of m. $\alpha \in [0, 1]$ represents the relationship between incoming and outgoing request rates of m'.

Accordingly, instance count for the microservice m is calculated as:

$$ins_m = \frac{r'_m}{r_m} \tag{4.15}$$

Primary Objective - QoS Violation

The first objective of the multi-objective Fog placement is depicted in equation 4.6. Here the aim is to minimise the total violation of QoS in terms of makespan and budget requirements for a batch of services. Since QoS parameters are of different units, normalised values of each QoS parameter violation are used. The weighted sum of normalised sub-objectives forms the objective function. Weights are chosen to prioritise among the two parameters, maintaining $\omega_l + \omega_b = 1$.

1. Makespan violation -

Calculation of the makespan of a service depends on the data flow pattern of the service. For a service representing chained microservices, makespan can be calculated as the total processing time of each microservice and the data communication delay among microservices. But for aggregator and hybrid service patterns, aggregator microservice can't complete the processing until results from all the data paths invoked by aggregator microservice are completed. Thus, the makespan of such services depends on the data path that takes the longest to complete.

Accordingly, makespan violation of service $s \in S_a$ where S_a indicates the set of services of application $a \in A$, can be calculated as,

$$v_s^l = max\{L(df_p^s); \forall p \in P_a^s\} - l_s \tag{4.16}$$

Equation 4.16 calculates the difference between makespan defined in the QoS profile of the service (l_s) and makespan due to proposed placement. $L(df_p^{sa})$ is the function used to calculate the makespan of the datapath p of service s due to proposed placement. Data path with maximum makespan is equal to the makespan of the service irrespective of the data dependency pattern of the service.

This calculation considers both processing latency and network latency. Network latency includes transmission latency as well as propagation latency among different Fog/ Cloud nodes where microservices are placed. Since each microservice m has ins_m instances, we consider that for the dataflow among m and m', requests generated from ins_m are equally load balanced among ins'_m microservices. So, equation 4.17 aims to find the highest latency of the path considering all instances of a microservice.

$$L(df_p^s) = L_{nw}(df_p^s) + L_{proc}(df_p^s) \tag{4.17}$$

$$L_{nw}(df_p^s) = \sum_{\forall mm' \in df_p^s} max\left[\sum_{\forall dd' \in D} x_{m_i}^d x_{m'_j}^{d'} (d_{dd'}^p + L_{tr}); \forall i, j\right] \tag{4.18}$$

where $1 \leq i \leq ins_m$ and $1 \leq j \leq ins_{m'}$.

$$L_{tr}(d, d') = \Delta_{mm'}[\frac{\rho}{bw_{WLAN}} + \frac{\sigma}{bw_{LAN}} + \frac{\psi}{bw_{WAN}}] \tag{4.19}$$

ρ, σ and ψ contains binary values (0 or 1) depending on the d and d' device types.

$$\rho = \begin{cases} 1 & if\ (d \in E) \oplus (d' \in E) \\ 0 & otherwise \end{cases} \tag{4.20}$$

$$\sigma = \begin{cases} 1 & if\ (d \in F) \wedge (d' \in F) \wedge d \neq d' \\ 0 & otherwise \end{cases} \tag{4.21}$$

$$\psi = \begin{cases} 1 & if\ (d \in C) \oplus (d' \in C) \\ 0 & otherwise \end{cases} \tag{4.22}$$

$$L_{proc}(df_p^s) = \sum_{\forall (m,m') \in df_p^s} \frac{i_{mm'}}{\Phi_{m'}} \tag{4.23}$$

Total makespan violation of the placement is calculated taking the sum of violations as follows,

$$v^l = \sum_{\forall a \in A} \sum_{\forall s \in S_a} max\left[v_s^l, 0\right] \tag{4.24}$$

2. Budget violation -

Budget violation of service $s \in S_a$, where S_a indicates the set of services of application $a \in A$, can be calculated as,

$$v_s^b = \left(\sum_{\substack{\forall m \in M_a^s \\ \forall d \in D}} \sum_{i=0}^{ins_m} x_{m_i}^d C_m^d \right) - b_s \tag{4.25}$$

Cost of executing microservice m on device d, C_m^d is calculated based on the pricing model presented in Section 4.3.3. Total budget violation of the placement is calculated taking the sum of violations as follows,

$$v^b = \sum_{\forall a \in A} \sum_{\forall s \in S_a} max\left[v_s^b, 0\right] \tag{4.26}$$

Secondary Objective - Resource utilisation

The second objective function (equation 4.7) handles resource utilisation under two sub-objectives, computation resources and network resources. Similar to the primary objective, this also calculates the weighted sum of the two normalised sub-objectives.

1. Computation resource usage:

The Fog layer consists of resource-constrained devices that are heterogeneous in their resource capacities. Thus within Fog environments, it is important to place applications in such a way that limited computation power is utilised by using a minimum number of Fog nodes so that only the services with stringent QoS requirements use limited Fog resources. This provides Fog service providers with the ability to host more applications within their Fog infrastructure. Besides, this encourages a balance between horizontal and vertical scaling, thus reducing the carbon footprint as well.

$$\tau_r = \sum_{\forall f \in F} act_f \tag{4.27}$$

2. Network resource usage:

Fog resources can be used to reduce the amount of data sent towards Cloud data centres by hosting bandwidth-hungry microservices. To this end, our placement policy introduces this sub-objective to reduce network usage, thereby increasing the placement of bandwidth-hungry microservices within the Fog layer.

$$\tau^a_{nw} = \sum_{\substack{\forall mm' \in df^a \\ \forall dd' \in D}} \sum_{i=1}^{ins_m} \sum_{j=1}^{ins_{m'}} x^d_{m_i} x^{d'}_{m'_j} \frac{d^p_{dd'} \, \Delta_{mm'} \, R_{mm'}}{ins_m \, ins_{m'}} \tag{4.28}$$

$$\tau_{nw} = \sum_{\forall a \in A} \tau^a_{nw} \tag{4.29}$$

4.4.2 QoS-aware Multi-objective S-CLPSO (QMPSO)

To solve the Fog application placement problem, we propose a placement policy based on the S-CLPSO algorithm described in Section 4.2.2 and integrate multiple heuristics to improve the convergence rate by proposing novel approaches for multi-objective normalisation and particle construction. Algorithm 4 presents the overview of our proposed **QoS-aware Multi-Objective S-CLPSO (QMPSO)** placement policy.

QMPSO algorithm first derives the number of instances per each microservice (line 1) using equation 4.15, which calculates the instance count based on the throughput requirement of each service (r_s). Then the algorithm initialises the minimum and max-

Algorithm 4 QMPSO Algorithm

Input: Placement Requests and Meta-data
Output: Microservices to devices mapping

1: Calculate the number of instances per microservice
2: Initialise Min/Max sub-objective values using heuristics
3: Set iteration count $i \leftarrow 1$
4: Initialise population of N particles using **SWARM_INIT**
5: **while** $i \leq Iterations$ **do**
6: Calculate fitness values of all sub-objectives for each particle
7: Update Min/Max of sub-objectives
8: Obtain the normalised fitness values for each sub-objective
9: Calculate fitness values of the two main objectives for each particle using normalised values
10: Update $pBest$ position of each particle
11: Update $gBest$ position of the swarm
12: Select exemplar dimensions for each particle
13: Update velocity of each particle
14: Update position for each particle using **CPPC_VA**
15: Set $i \leftarrow i + 1$
return $gBest$ of the swarm

imum possible values for each sub-objective of the multi-objective fitness function formulated in Section 4.4.1 (line 2). QMPSO uses multiple heuristics to obtain estimates for these values. In our multi-objective optimisation problem, these values are used to calculate the normalised sub-objectives. Then an initial population is created using both heuristic-based and random placements (line 4), which is explained in detail in the Algorithm 5. After creating the initial population, the algorithm calculates fitness values for each sub-objective for all particles (line 6) and accordingly update the minimum and maximum values of each sub-objective (line 7). This enables the normalisation calculations to become more accurate as the swarm progresses through solution space in each iteration. Afterwards, normalised fitness values are calculated for each particle (line 8). Based on the values obtained for the two main objectives (line 9), the personal best ($pBest$) of each particle and the global best ($gBest$) of the swarm are updated using lexicographic comparison (lines 10-11). Then, the velocity matrix is updated using exemplar dimensions according to the S-CLPSO algorithm (lines 12-13). Finally, the new position of each particle is updated using the velocity matrix of the particle (line 14). Each created particle should adhere to the resource constraints of the Fog devices. To satisfy this

constraint, QMPSO proposes a novel particle construction process as **Constraint-aware Prioritised Particle Construction (CPPC)**. This contains two algorithms; **CPPC_INIT** for random construction of particles during swarm initialisation and **CPPC_VA** for velocity aware construction of particles after updating the velocity of the particle during each iteration. After executing these steps for a pre-defined number of iterations, the algorithm returns the global best position of the swarm as the placement mapping.

Integral steps of the QMPSO algorithm, which includes problem mapping, initial swarm, fitness calculation and particle position update are described in detail in the following sub-sections.

Mapping microservice application placement problem to S-CLPSO

As per the S-CLPSO, each particle representing a possible solution to the problem is characterised by a position vector and velocity matrix (Figure 4.4). For the considered microservice placement problem, the position vector is a crisp set that maps each microservice instance to a device. Number of microservice instances are calculated at the start of the Algorithm 4 according to equation 4.15. The dimension of the position vector is equal to the total number of microservice instances that are to be placed. Velocity matrix is a set of possibilities that contains the possibility of each microservice instance being placed on each device. All position and velocity related basic calculations follow the concepts introduced in Section 4.2.2.

Position Vector

Microservice	$m1_1$	$m1_2$	$m2_1$	$m2_2$	$m3_1$	$m4_1$
Device	d1	d4	d1	d3	d2	d4

Velocity Matrix

	$m1_1$	$m1_2$	$m2_1$	$m2_2$	$m3_1$	$m4_1$
d1	0.5	0.57	0.8	0.12	0.42	0.63
d2	0.25	0.4	0.73	0.44	0.91	0.5
d3	0.32	0.1	0.43	0.46	0.77	0.04
d4	0.12	0.74	0.68	0.33	0.8	0.65

Figure 4.4: QMPSO particle representation

Algorithm 5 SWARM_INIT

Input: *numParticles*
Output: *Swarm*

1: $Swarm \leftarrow \{\}$
2: $pv \leftarrow$ create position vector from **OHPP** placement
3: $vm \leftarrow$ initialise velocity matrix
4: $particle \leftarrow createParticle(pv, vm)$
5: $Swarm.add(particle)$
6: $particleCount \leftarrow 1$
7: **while** $particleCount < numParticles$ **do**
8: $pv \leftarrow$ random construct using **CPPC_INIT**
9: $vm \leftarrow$ initialise velocity matrix
10: $particle \leftarrow createParticle(pv, vm)$
11: $Swarm.add(particle)$
12: $particleCount \leftarrow particleCount + 1$
return *Swarm*

Initial Swarm

Initial position vectors of the particles are generated using two methods: heuristic-based particle generation and random particle construction (Algorithm 5). We propose a novel makespan and budget aware heuristic named, **Osmotic Heuristic Placement Policy (OHPP)** to seed the initial population (lines 2-4). The rest of the particles are generated using the random particle construction path of the CPPC process (lines 7-12).

OHPP: For the swarm to reach global optimum position faster, we introduce **Osmotic Heuristic Placement Policy (OHPP)**. This policy follows the concept of Osmotic computing and tries to move latency-sensitive services to the Fog layer (Algorithm 6). The algorithm starts by placing all microservices on the Cloud and afterwards calculates the latency violation of each service using equation 4.16 (line 1-2). Makespan violated services are sorted from minimum to maximum budget requirement (line 3-4) and matched on to Fog devices sorted from minimum to maximum pricing (line 5-22). Other than providing a feasible placement, OHPP is also used to prioritise microservices for placement within the Fog layer (lines 23-24). OHPP algorithm derives all microservices of the *FogServices* and outputs them as prioritised microservices to be placed on Fog (*ToFogM*), while the rest of the microservices are added to a separate list (*ToCloudM*). This prioritisation plays an important role in the particle construction process of the CPPC

(in both CPPC_INIT and CPPC_VA).

Algorithm 6 OHPP

Input: Placement Requests and Meta-data
Output: Microservices to devices mapping

```
1: Place all microservice instances in Cloud
2: Calculate deadline violation using equation 4.16
3: FogServices ← get all deadline violated services
4: sorted ← sort FogServices from min to max budget requirement
5: FogDevices ←sort from min to max pricing
6: for each service s in sorted do
7:     M_inst ← topologically sorted μservice instances of s from meta-data
8:     for each m in M_inst do
9:         if m.predecessor in Cloud then
10:            d ← Cloud
11:        else
12:            d ← FogDevices.first
13:        while m is not placed do
14:            if d.availResources ≥ m.resources then
15:                place m in d
16:                update availResources of d
17:            else
18:                if d = FogDevices.last then
19:                    d ← Cloud
20:                else
21:                    d ← FogDevices.next
22: Place the rest on Cloud
23: ToFogM ← microservices of FogServices
24: ToCloudM ← microservices not included in FogServices
25: return microservice placement, ToFogM, ToCloudM
```

CPPC_INIT: For the creation of the rest of the particles, the random construction path of the CPPC Algorithm is used (Algorithm 7). CPPA_INIT uses microservice prioritisation for the construction of placements under resource constraints. The algorithm prioritises latency-critical microservices in *ToFogM* for the placement within resource-constrained Fog devices (lines 2-9). Afterwards, the algorithm proceeds with mapping *ToCloudM* microservices (line 10-16).

The above methods together populate the initial swarm with diverse and feasible solutions, thus improving the ability of the swarm to reach its global optimum solution

within less amount of time. For the initialisation of the velocity matrix, a value in the range [0,1] is assigned for the mapped device of each microservice instance and the rest of the devices are assigned 0 for the said microservice.

Algorithm 7 CPPC_INIT Algorithm

Input: D devices, $ToFogM$, $ToCloudM$
Output: $PositionVector$

```
1:  PositionVector ← {};
2:  devices ← D.getFogDevices();
3:  devices.add(D.getCloudDevices());
4:  for each microservice m in ToFogM do
5:      for each device d in devices do
6:          if d.availResources ≥ m.resources then
7:              PositionVector.add(m, d)
8:              update availResources of d
9:              break;
10: devices.shuffle();
11: for each microservice m in ToCloudM do
12:     for each device d in devices do
13:         if d.availResources ≥ m.resources then
14:             PositionVector.add(m, d)
15:             update availResources of d
16:             break;
    return PositionVector
```

Normalised Fitness Calculation

The placement problem is modelled with two main objectives where each is calculated as the weighted sum of its two sub-objectives (Section 4.4.1). As each sub-objective value has different units and has different ranges of values, a normalised weighted sum is required to reach a proper trade-off between the sub-objectives. The best approach for this would be to minimise and maximise each sub-objective separately to obtain the range of values for each. But due to the higher time consumption of this method, we propose a heuristic driven normalisation approach to initialise minimums and maximums for each sub-objective.

We use the following heuristics to initialise maximums and minimums for each sub-objective with close enough estimates:

Deadline-aware heuristic placement: Services are sorted from minimum to maximum makespan requirement and placed starting from Fog layer and move to Cloud-only if non of the Fog devices have enough resources to host the microservice. This placement provides an estimate for minimum latency violation and minimum network usage.

Budget maximisation placement: To find an estimate for maximum cost violation, we propose a heuristic where devices are sorted from maximum to minimum unit price, services are sorted from minimum budget to maximum budget requirement and microservices from ordered services are matched with ordered devices.

Cloud only placement: All microservices are placed in the Cloud providing an estimate for maximum latency violation, maximum network usage and minimum budget violation.

Moreover, for Fog resource usage, the minimum is set to 0 and maximum is determined as *min(Fog device count, microservice instance count)*.

During each iteration of the QMPSO algorithm, minimum and maximum values are updated based on the fitness values of the particles in the swarm. Updated minimum and maximum values are used to calculate Min-Max Normalisation, which scales the value of each objective to the range 0-1. For an objective i (obj_i), with current value of x, normalised value using Min-Max Normalisation is calculated as follows:

$$\eta(x) = \frac{x - min(obj_i)}{max(obj_i) - min(obj_i)} \tag{4.30}$$

This approach enables the QMPSO algorithm to avoid premature convergence and reach a fair trade-off between the weighted sub-objectives.

Position update - Constraint-aware Prioritised Particle Construction (CPPC)

Position updating undergoes three main steps; 1) selection of exemplar dimensions for the particle 2) update particle velocity 3) construct new particle position.

Selection of exemplar dimensions is the process of selecting which particle's $pBest$ should be followed by each dimension for velocity updating as depicted in equation 4.3. Particle position update is carried out using the updated velocity matrix. Due to resource constraints of the Fog resources, updated particle positions have to adhere to

resource constraints, which is one of the main challenges of Fog service placement, as mending particles based on resource constraints can considerably hinder the convergence to the global optimum position.

To overcome this issue, we propose CPPC_VA, which is the velocity aware path of our proposed CPPC process. It is a particle construction algorithm that checks resource constraints at the time of particle position update based on the velocity matrix. CPPC_VA uses two main features to improve convergence (Algorithm 8):

Use of prioritised microservices: Similar to CPPC_INIT, this algorithm also uses microservice prioritisation into two groups as *toFogM* and *toCloudM*. The idea behind this is to enable *toFogM* microservices a higher chance of being assigned to the devices indicated by its velocity matrix (lines 1-2).

Velocity aware device selection: For each microservice, the algorithm tries to determine the new Fog device in a velocity conscious manner. First, all devices with higher or equal velocity values are identified (lines 5-7) and each selected device is considered for placement until a device with the required resource amount is met (lines 8-12). If all selected devices are infeasible for placement, microservice is added to a separate list (*notMapped*) for placement later (lines 13-14). After iterating through all microservices, each microservice in *notMapped* list are mapped to feasible devices randomly (lines 15-20). Here all devices are considered irrespective of the velocity value. This method provides a proper balance between exploitation and exploration, thus generating a diverse solution set and improving algorithm convergence.

4.5 Performance Evaluation

We evaluated the performance of our QMPSO algorithm through simulation of synthetic workloads of microservices-based IoT applications, that have heterogeneous QoS requirements in terms of makespan, budget and throughput. Evaluations are completed under two main categories:

QMPSO Performance Evaluation: Section 4.5.3 evaluates the performance of the QMPSO algorithm in terms of convergence improvement against two adaptations of the S-CLPSO algorithm for Fog application placement problem.

Algorithm 8 CPPC_VA Algorithm

Input: D devices, particle P, $ToFogM$, $ToCloudM$
Output: updated particle P

```
1: microservices ← ToFogM
2: microservices.add(ToCloudM)
3: notMapped ← {}
4: for each microservice m in microservices do
5:     currDevice ← P.positionVector.get(m);
6:     currVelocity ← P.velocityMatrix.get(currDevice)
7:     D' ← get devices with velocities ≥ currVelocity
8:     for each device d in D' do
9:         if d.availResources ≥ m.resources then
10:            P.PositionVector.add(m, d)
11:            update availResources of d
12:            break;
13:    if m is not mapped to a device then
14:        notMapped.add(m)
15: for each microservice m in notMapped do
16:    D' ← get all possible devices from D
17:    r = random(1, D'.size)
18:    d ← D'.get(r);
19:    P.PositionVector.add(m, d)
20:    update Avail_Resources
return updated particle P
```

1. **No-Heuristics:** Directly adapts the S-CLPSO algorithm to Fog application placement problem without incorporating any heuristics.

2. **No-Prioritised-Construct:** Heuristics are used in this approach for swarm initialisation and fitness normalisation. But particle construction does not prioritise microservices during the construction process, but randomly select microservices.

QMPSO Placement Evaluation: Section 4.5.3 compares QMPSO with four other Fog application placement approaches in terms of QoS satisfaction and optimum Fog-Cloud resource usage. These approaches are selected to cover different types of algorithms including optimisation-based, meta-heuristic and heuristic approaches to solve application placement problem within Fog environments.

1. **Constraint Programming based Placement Algorithm (CPPA)** - Placement prob-

lem introduced in Section 4.4.1 is solved using a Constraint Programming solver.

2. **OHPP** - Algorithm which is used in generating our initial swarm of particles. We use this to demonstrate how the incorporation of improved S-CLPSO results in better placement decisions.

3. **FSPP** - Fog service placement approach proposed in [32], where service spread (scale microservices to evenly spread them within Fog environment), latency (minimise communication delay among microservices) and resource usage (maximise Fog device usage) are the focus of placement decision making.

4. **DNCPSO** - Algorithm proposed in [104] for workload scheduling in cloud-edge environments to minimise the total latency and cost of the placement.

Out of the existing works analysed in Section 4.2.1, FSPP and DNCPSO are the only works that can be adapted and applied to the batch placement of microservices-based applications addressed in our work. So, they are chosen for the performance comparison.

4.5.1 Implementation of the Algorithms

For the performance evaluation, all placement algorithms were implemented using iFogSim2 [8] simulator, which is a toolkit for the simulation of Fog computing environments. iFogSim2 extends iFogSim [9] simulator and provides support for simulation of microservice application placement through its advanced features such as service discovery and load balancing.

QMPSO Implementation

To support the simulation of the proposed system architecture along with the QMPSO algorithm, we extended the iFogSim2 simulator by integrating features to support multiple service composition patterns and QoS profiles containing per service QoS definitions. Afterwards, the QMPSO algorithm was developed and simulated on top of that.

CPPA Implementation

An optimised solution for the placement problem can be obtained by solving the problem modelled in Section 4.4.1 using a solver. In this work, we have used the Constraint Programming(CP) engine of IBM ILOG CPLEX 12.10.0 solver [118] for obtaining the optimum solution for Fog application placement. iFogSim2 is used for the implementation of the placement policy by using Java API available in the solver.

CPPA approach uses lexicographic optimisation with objectives ordered as, minimising QoS violation as first objective (equation 4.6) and resource utilisation as second objective (equation 4.7). Normalised values of each objective ($\eta(v^l), \eta(v^b), \eta(\tau_{nw}), \eta(\tau_r)$) is calculated by taking "Nadir point" as minimum value and "Utopia point" as maximum for each objective [119] . These two points are calculated by optimising each objective separately. Afterwards, the placement problem is solved using multi-objective optimisation. To obtain the results within a reasonable time limit, the failure limit parameter is set to 10^7 failures during the search process.

DNCPSO and FSPP

DNCPSO and FSPP were implemented and simulated in iFogSim2 based on the algorithms described in [104] and [32] respectively. Necessary adaptations were made to the algorithm to adapt it to our proposed system model and IoT application batch placement scenario while maintaining core principles and fitness functions as proposed in the said works.

4.5.2 Experimental Configurations

Simulation Environment

To evaluate the performance of the algorithms, we created synthetic workloads based on the microservices-based application model proposed in Section 4.3.1. Each workload consists of multiple applications, including Smart health monitoring and Smart Parking application presented in the motivation scenario along with synthetic DAG-based applications created to represent all service composition patterns introduced in this work.

Table 4.3: Evaluation parameters

	Parameter	Value
Communication links	LAN	0.5ms, 1 Gbps
(latency, bandwidth)	WAN	30ms, 100 Mbps
[105, 120–122]	WLAN	2ms, 150 Mbps
Fog device resources	CPU (MIPS)	1500-3000
[123, 124]	RAM (GB)	2-8
	Storage (GB)	32-256
Cost Model	CPU (Cloud)	\$0.040480 per 150 MIPS per hour
parameters	RAM (Cloud)	\$0.004445 per GB per hour
[116]	Storage (Cloud)	\$0.000111 per GB per hour
	Increase factor for fog	1.2-1.5
QoS parameters	Makespan (l_s)	20-150ms
[125, 126]	Budget (b_s)	\$0.25-1.50 per hour
	Throughput (r_s)	200-800 requests/s

Table 4.4: Parameters for placement algorithms

Parameter	QMPSO	DNCPSO	FSPP
No. of particles in swarm	50	50	100
No. of iterations	300	300	400
Mutation rate	-	0.25	0.25
ω_{min} - ω_{max}	0.4 - 0.9	0.4-0.9	-
c_1, c_2	-	2	-
c	1.49445	-	-
m (refresh gap)	0	-	-
ω_l, ω_b, ω_{nw}, ω_r	0.5	-	-

Heterogeneity within the workloads is further ensured by modelling the diversity of microservices in terms of computation cost of the microservices (300-900 MIPS) and bandwidth usage among microservices (200-1500 bytes/packet). Moreover, when defining resource requirements of each microservice, the request rate supported by the basic deployment unit (r_m) is chosen between 100-200 requests/s. All the above parameter values are determined based on the IoT simulation benchmarks presented in previous simulation studies [8, 100]. Diversity among services is maintained in terms of QoS by varying makespan, budget, and throughput requirements.

The Fog environment is constructed according to the architecture proposed in Section 4.3.2. Table 4.3 lists the configurations used in constructing the simulated Fog envi-

ronment. Parameters of the Fog network such as communication link latency and bandwidth represent novel network technologies like Wi-Fi 6 [120], 5G [121] for WLAN, and gigabit Ethernet [105] for LAN connections, acquired from edge network performance studies. Fog resources are modelled as a pool of heterogeneous devices with varying resource capacities similar to [123, 124] which include heterogeneous Fog devices such as Raspberrypi 4B, Jetson Nano, Dell PowerEdge, etc. Cost of execution of the microservices is modelled according to AWS Fargate pricing [116] defined for CPU, RAM, and storage separately. Due to service level improvements provided by the Fog environment, Fog resource prices are determined by multiplying on-demand prices of Cloud resources by an increase factor between 1.2-1.5 according to [102], which models on-demand pricing within Fog environments. vCPU to MIPS mapping for the simulation is obtained based on Microsoft Azure industrial benchmark where 150MIPS estimates to 1vCPU [127].

QoS parameters are varied to ensure makespan and budget limits of the services span from the edge of the network to the Cloud. Makespan requirement is varied within 20-150ms, following the IoT application latency requirements discussed in the previous studies [125, 126]. The budget requirement is set based on the resource requirements of each microservice in the synthetic workload and cost parameters of the environment in such a way that the values span both Cloud and Fog deployment. Moreover, the budget parameter is adjusted so that latency-critical and bandwidth-hungry services have higher budget limits to enable their placement within the Fog layer. For throughput requirement of services (r_s), we have considered a wide range of values (200-800 requests/s) compared to r_m of each microservice, to evaluate how the placement algorithm handle the scalability of microservices.

Algorithm Parameter Tuning

Table 4.4 lists parameters and their values for QMPSO, DNCPSO and FSPP algorithms. For QMPSO algorithm preliminary experiments were carried out to observe the fitness value achieved by the algorithm for different values of swarm size, iteration count and refreshing gap. Based on the observations, we set particle count to 50, iteration count to

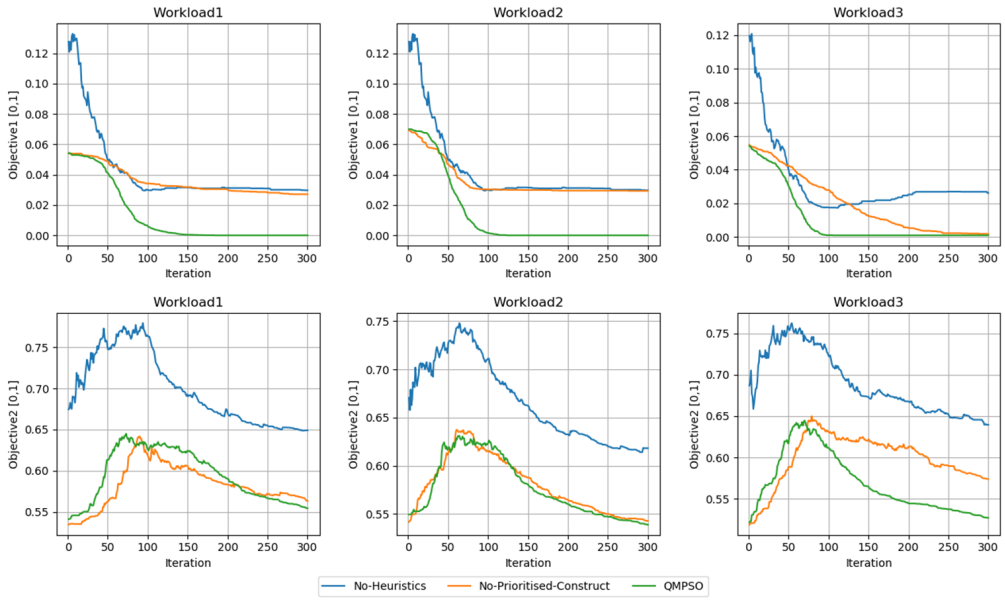


Figure 4.5: Variation of fitness values for different adaptations of S-CLPSO

300 and refreshing gap to 0. Further improvements to the fitness values can be obtained by increasing particle and iteration counts at the cost of increased algorithm execution time. Values for inertia weight ω and coefficient c are chosen based on previous studies on PSO algorithm [104, 115] conducted to determine the optimum values for these parameters. ω is changed from ω_{max} (0.9) to ω_{min} (0.4) over the iterations, according to the non-linear equation proposed in [104]. Coefficient c is set to 1.49445 as per [115]. For the performance evaluation, we consider all sub-objectives are equally important. Due to objective normalisation used in the QMPSO algorithm, this is achieved by setting all weights to 0.5.

For DNCPSO and FSPP algorithms, parameters are set according to [104] and [32] respectively.

Table 4.5: Mean fitness values and standard error of the objectives for different adaptations of S-CLPSO to Fog placement problem

	QMPSO		**No-Prioritised-Construct**		**No-Heuristics**	
	Obj1	**Obj2**	**Obj1**	**Obj2**	**Obj1**	**Obj2**
Workload1	0	0.5544± 0.0025	0.0271± 0.0020	0.5633± 0.0068	0.0297± 0.0012	0.649± 0.0074
Workload2	0	0.5389± 0.0017	0.0293± 0.0007	0.5429± 0.0040	0.0262± 0.0013	0.6183± 0.0072
Workload3	0.0010± 6.145E-06	0.5271± 0.0018	0.0019± 0.0007	0.5738± 0.0049	0.005± 0.0015	0.6395± 0.0067

4.5.3 Results and Analysis

QMPSO Performance Evaluation

This section evaluates the performance of QMPSO by analysing how the values for the two main objectives gradually evolve with iterations. For the evaluation, three synthetic workloads are created according to the specifications detailed in Section 4.5.2, where Workload1, Workload2 and Workload3 consist of 5, 7 and 10, microservice-based applications respectively. For placement of the workloads, a Fog environment with 17 Fog devices is considered. For each workload, placement is generated using No-Heuristics, No-Prioritised-Construct and QMPSO algorithms and the fitness values for Objective1 (QoS violation) and Objective2 (Resource usage) are recorded over 300 iterations. Each algorithm is repeated 100 times and the average fitness values are obtained.

Figure 4.5 depicts the variations of fitness values while Table 4.5 lists the average fitness value of each algorithm after 300 iterations. Results show that the QMPSO algorithm outperforms the other two approaches in reaching the global optimum solution within a lesser number of iterations. For both objectives, No-Heuristics demonstrates a higher fluctuation in fitness value during early iterations. In No-Prioritised-Construct and QMPSO algorithms, this behaviour is not present for Objective1 due to heuristics based minimum and maximum initialisation. No-Heuristics algorithm updates mini-

mum and maximum values for each sub-objective only based on the particles available in the swarm. So it takes the algorithm a larger number of iterations to obtain accurate values, which results in the fluctuations. Besides, the use of OHPP in the initial swarm provides No-Prioritised-Construct and QMPSO with a better starting point. Moreover, both No-Heuristics and No-Prioritised-Construct tend to converge to local-optimum positions. QMPSO has overcome this with the proposed particle construction algorithm, CCPC. The use of prioritised microservices in CCPC ensures a proper balance between exploitation and exploration to make sure that the algorithm moves towards the global-optimum solution for Objective1. As solution space is a discrete space limited by resource constraints, there's a higher chance of algorithms converging to a local optimum solution. But prioritised particle construction in QMPSO helps the algorithm to traverse the discrete solution space successfully without getting stuck in local optimums.

As the placement problem is modelled as a lexicographic optimisation, fluctuations are expected to occur in the Objective2 value until Objective1 converges. This explains the increase in Objective2 during early iterations in all three approaches. In No-Heuristics, the increase and fluctuations in the value are considerably higher because it takes more time for this approach to obtain the accurate minimum and maximum values for the sub-objectives without the use of heuristics. Thus, faster convergence in Objective1 results in better results of Objective2 as well. This is evident in the behaviour of the QMPSO algorithm. Objective values denoted in Table 4.2.2 shows that QMPSO reaches lower objective values for both objectives. Besides, the standard error of the achieved values is also lower in QMPSO when compared with other approaches. This indicates that the performance of QMPSO is consistent over multiple runs.

The above results demonstrate that the proposed QMPSO algorithm can reach better performance due to multiple features we've incorporated with the algorithm, including OHPP-based swarm initialisation (SWARM_INIT), heuristic-driven fitness value normalisation and prioritised particle construction (CCPC).

QMPSO Placement Evaluation

In this section, we evaluate the placement generated by our algorithm using several performance metrics: makespan satisfaction percentage and budget satisfaction percentage are used to evaluate the QoS satisfaction of the placement, network usage and the total number of active Fog devices to evaluate the Fog resource usage.

Makespan Satisfaction: This metric is calculated as the number of service requests that meets makespan requirements of the said service, as a percentage of all the service requests received by the Fog environment.

Budget Satisfaction: A metric reflecting budget satisfaction percentage of the Fog environment. This metric is calculated as the difference between the cost violation after placement and the maximum possible cost violation of the environment for the requested placement, as a percentage of the maximum possible violation.

Network Usage: Indicates network occupancy as a measurement of *packet size (kilobytes) x link delay (ms)* within the duration of the simulation for all packets sent through the Fog environment.

Active Fog Devices: Depicts the number of devices with at least one microservice deployed onto the device. Optimum usage of Fog computing resources can be evaluated based on two main aspects; balanced use of Fog and Cloud where Fog resources are used only for latency-critical and bandwidth-hungry services which mitigates overuse of limited Fog resources, and the ability to avoid unnecessary dispersion of microservices within highly distributed Fog environments. Active Fog device count is a quantitative metric that can provide accurate insight on both of these aspects.

Solution Space Analysis: Experiments are conducted to observe the performance of each algorithm as the solution space grows. The size of the solution space depends on two parameters; the number of microservice instances to be placed and the number of devices considered for placement of the microservices. Figure 4.6 depicts the performance for different device counts (15, 20 and 25 Fog devices) keeping microservice count a constant (8 Applications, 30 microservices) whereas Figure 4.7 is for the scenarios where the microservice count is changed keeping device count a constant (20 Fog devices). Microservice count is increased by increasing the number of applications considered for placement (5, 7 and 10 applications). Moreover, it varies the degree of

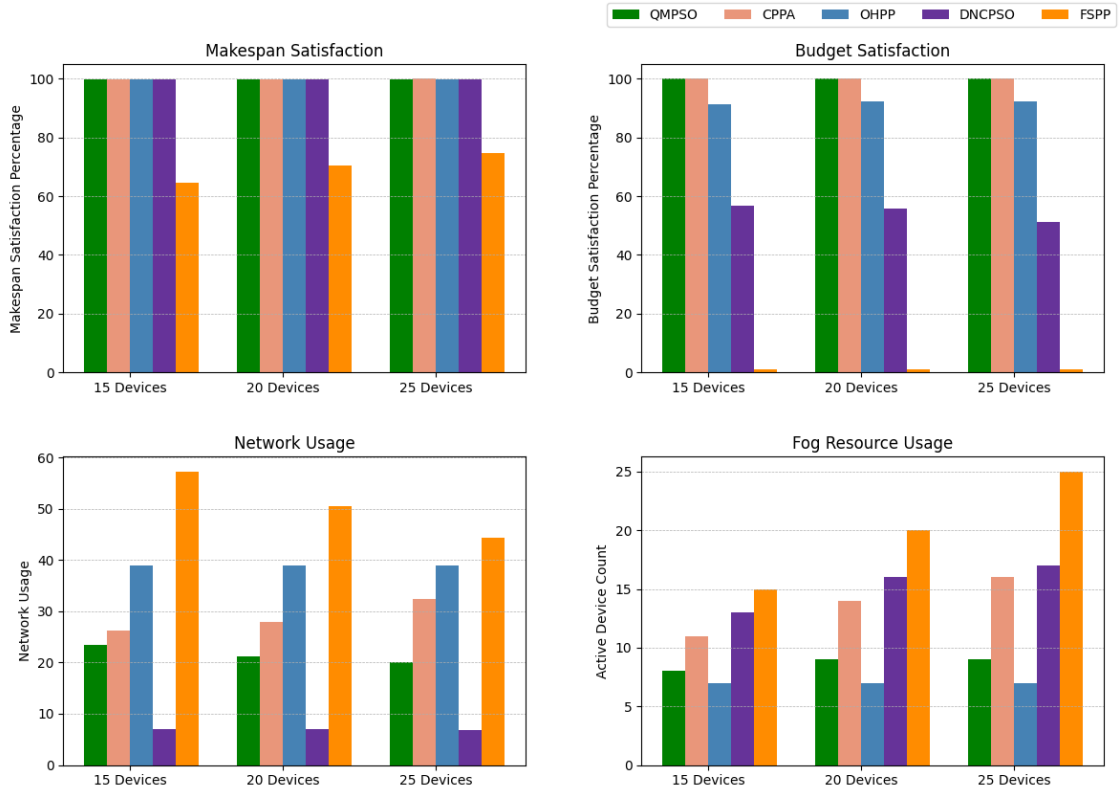


Figure 4.6: Performance for different device counts

heterogeneity within the batch of services available for placement.

Based on the results depicted in Figure 4.6 and Figure 4.7, for QoS satisfaction, QMPSO and CPPA achieve the highest satisfaction percentage in both makespan and budget for all scenarios. But for network usage and Fog resource usage which indicate the ability of the algorithm to obtain a proper balance between Fog and Cloud usage, QMPSO outperforms CPPA. As the solution space grows, network usage and active device count for CPPA placement increase. Due to the NP-complete nature of the Fog application placement problem, CPPA is limited by a failure limit of 10^7 to obtain a solution within reasonable time limits. Thus, QMPSO with its meta-heuristic approach outperforms CPPA. Figure 4.8 compares the execution time of QMPSO and CPPA algorithm as solution space grows. Both the execution time and increase in execution time with solution space growth is considerably higher in CPPA.

According to Figures 4.6 and 4.7, OHPP, which is our proposed heuristic for QMPSO

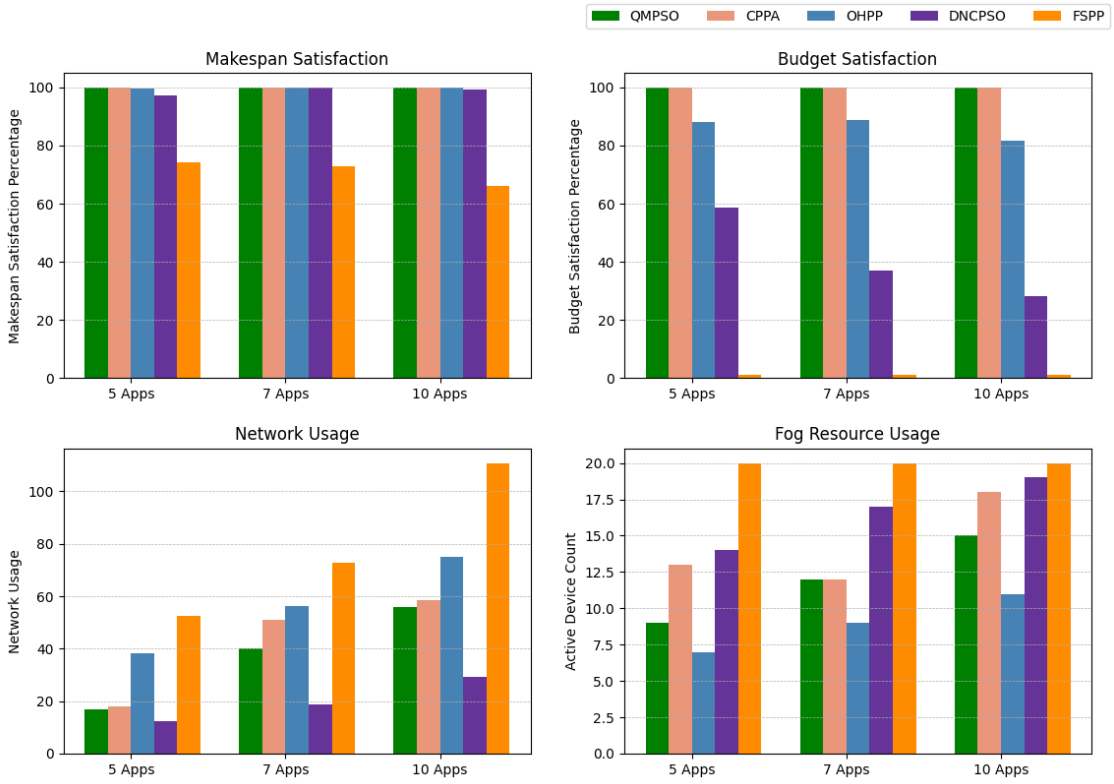


Figure 4.7: Performance for different application/microservice counts

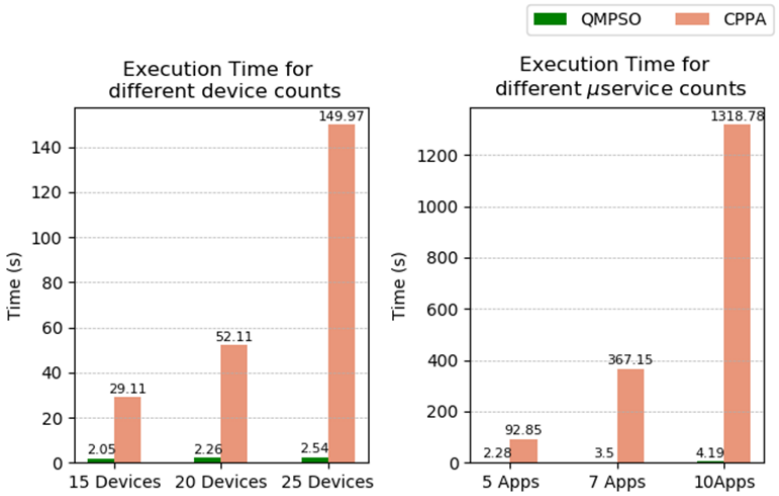


Figure 4.8: Execution time of the QMPSO and CPPA algorithms

initialisation, can achieve high makespan satisfaction but lacks budget satisfaction. OHPP prioritises services based on stringent makespan and budget requirements but fails to handle the complexity introduced due to data dependencies among microservices. As a result, OHPP shows a decrease in budget satisfaction as the number of applications increases. Moreover, OHPP only focuses on moving latency-critical service to the Fog and place the rest of the service in the Cloud. As a result, bandwidth-hungry microservice of latency tolerant services are placed in the Cloud, which under-utilises Fog resources and increases network usage.

The fitness function of the DNCPSO algorithm is designed to minimise the weighted sum of total latency and cost of the placement. Moreover, DNCPSO aims to re-arrange particles to place all latency-critical microservices in the Fog layer which results in high makespan satisfaction. However, the budget satisfaction of the algorithm drops significantly (up to 70%), as the fitness calculation does not contain budget awareness, but try to minimise the total cost. This approach lacks prioritisation of services with stringent budget requirements which moves more services to the Fog to reduce total latency. This results in lower network usage, but over utilises Fog resources and reduces budget satisfaction significantly.

Similarly, FSPP tries to minimise the latency of the services without taking their makespan requirements into consideration. As this approach does not prioritise latency-critical microservices, makespan satisfaction reduces up to 35% within a resource-constrained Fog environment. As the number of devices increase, more latency-critical microservices are placed inside the Fog layer, which results in a slight increase in makespan satisfaction. FSPP aims to increase the Fog resource usage by placing replicas of the microservices without imposing a budget constraint, which results in closer to zero budget satisfaction. FSPP tries to maximise Fog resource usage irrespective of the throughput requirements of the services. As a result, all Fog devices are active in all placement scenarios. As FSPP scale microservices randomly in the Fog layer, the number of microservices pushed to the Cloud increases due to the resource constraints of Fog devices. This results in a significant increase in network usage as well.

Scalability Analysis: Experiments are conducted to analyse the performance of the placement as throughput requirements of the services change. To this end, a workload

of 5 applications is considered for two scenarios where throughput requirement in QoS profile of each application is doubled in scenario2 when compared with scenario1 (Figure 4.9). For both scenarios, 25 Fog devices are considered for placement.

As throughput requirement increases, microservices are horizontally scaled in QMPSO, CPPA and OHPP placement policies due to the throughput aware instance count calculation proposed in Section 4.4.1. This increases the number of microservice instances to be placed, thus expanding the solution space. As a result, the performance of the CCPA reduces with increased throughput (scenario2). In scenario1, CPPA is able to reach similar QoS satisfaction values as our QMPSO algorithm. But in scenario2, CPPA is unable to reach an optimum solution within the specified failure limit of the algorithm, which results in the reduction of makespan satisfaction. Although OHPP is able to achieve full makespan satisfaction, budget satisfaction drops significantly (up to 20% reduction) as throughput increases. Thus, as the number of microservice instances increase heuristic approach fails to provide satisfactory results. DNCPSO does not consider the scalability of microservices. So, as throughput increases, resource requirements of each microservice increase and resource-constrained Fog devices are unable to handle them. As a result, DNCPSO moves these microservices towards the Cloud, which results in the increase of latency violation and network usage. Without using horizontal scalability, DNCPSO is unable to fully utilise Fog devices with limited resources. FSPP scales microservices to spread them evenly across the Fog environment. So, FSPP does not demonstrate a significant difference in makespan satisfaction as sufficient microservice instances are available in both scenarios. But, FSPP randomly scales microservices without supporting throughput aware scalability which result in the overuse of Fog resources.

Compared to other approaches, QMPSO achieves improved performance in all considered metrics for both scenarios. Our placement vertically and horizontally scales microservices based on their throughput requirements, which results in proper utilisation of resource-constrained Fog devices to maximise makespan and budget satisfaction. This also indicates the ability of the QMPSO algorithm to successfully navigate larger solution spaces, unlike CPPA and OHPP algorithms.

Based on the solution space analysis and scalability analysis, it is evident that QMPSO

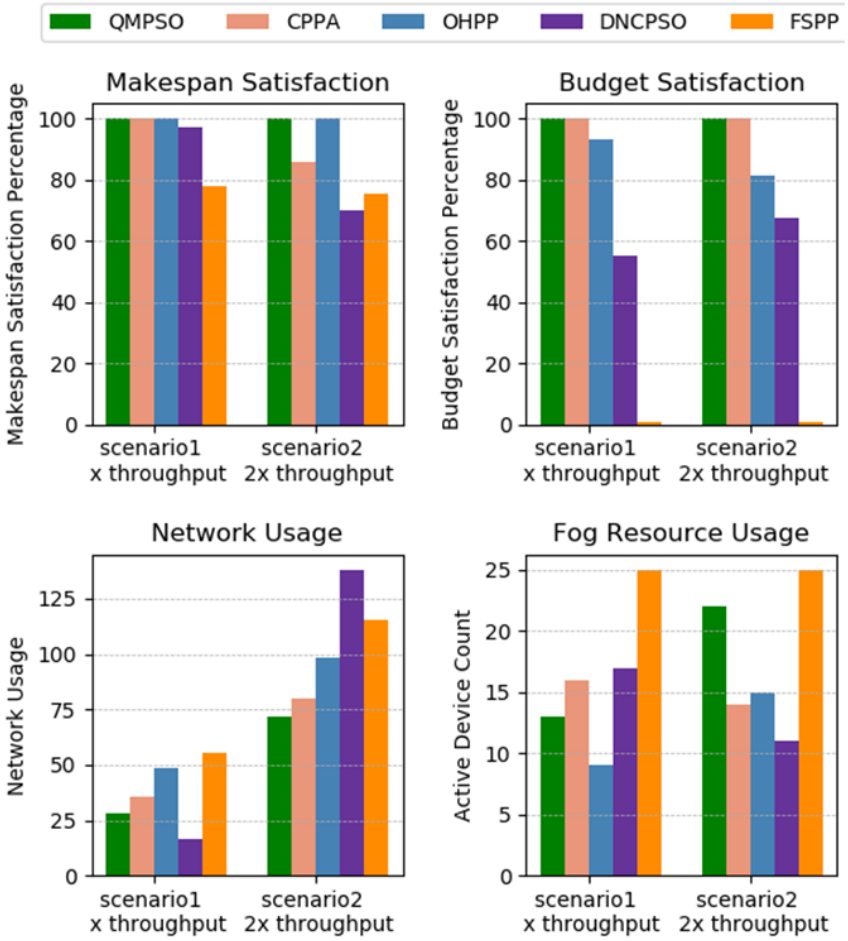


Figure 4.9: Performance for different throughput requirements

significantly improves QoS satisfaction along with resource utilisation. For the considered scenarios, QMPSO records up to 35% improvement in makespan satisfaction and up to 70% improvement in budget satisfaction. These results indicate the ability of the QMPSO algorithm to navigate large solution spaces successfully to reach optimum QoS satisfaction. Moreover, results depict that QoS-awareness in the fitness function of QMPSO enables it to successfully utilise both Fog and Cloud resources to handle heterogeneous QoS requirements. Thus, QMPSO provides a robust algorithm capable of harvesting Fog and Cloud resources to obtain an efficient placement schedule for heterogeneous microservice-based IoT applications.

Table 4.6: Complexity analysis

	QMPSO	**DNCPSO**	**FSPP**
Initialisation	$\mathcal{O}(Slog(S) + \|D'\|log(\|D'\|) + S.M.I.\|D'\|)$	$\mathcal{O}(S.M.\|D'\|)$	$\mathcal{O}(S.M.\|D'\|)$
Evolution	$\mathcal{O}(S.M.I.\|D'\|)$	$\mathcal{O}(S.M.\|D'\|)$	$\mathcal{O}(S.M.\|D'\|)$

Algorithm Complexity Analysis

We have introduced multiple approaches/algorithms to improve the performance of our QMPSO placement algorithm. In this section, we evaluate the time complexity introduced by these novel approaches and compare them with approaches used in DNCPSO and FSPP algorithms which use PSO and NSGA-II respectively. All three evolutionary algorithms have two main phases that affect the overall complexity of the algorithms and Table 4.6 presents their complexities. We consider the number of services for placement as S with each having a maximum of M microservices along with I instances per microservice for the placement within $|D'|$ devices where $D' = F \cup C$. The effect of population size and iteration count is ignored as they are constants in all three algorithms.

Initialisation : For QMPSO, initialisation includes the creation of initial solution space (**SWARM_INIT**) and heuristic based initialisation of minimum and maximum values required for normalisation of the sub-objectives. SWARM_INIT consists of **OHPP** and **CPPC_INIT** algorithms. OHPP contains two main steps; sorting of services and devices which is completed with linearithmic time complexity of $\mathcal{O}(Slog(S))$ and $\mathcal{O}(|D'|log(|D'|))$ respectively, and mapping of each microservice instance of the service to a device ($\mathcal{O}(M.I.|D'|)$), which results in time complexity of $\mathcal{O}(S.M.I.|D'|)$ for all services. CPPC_INIT iterates through prioritised microservice instances to randomly find the eligible device. As prioritising is already completed by OHPP, the algorithm can be completed with worst case time complexity of $\mathcal{O}(S.M.I.|D'|)$. The total time complexity of SWARM_INIT is $\mathcal{O}(Slog(S) + |D'|log(|D'|) + S.M.I.|D'|)$. Heuristic approaches used for normalisation (Deadline-aware heuristic placement and Budget maximisation placement) follow the same placement approach as OHPP with different sorting orders for services and devices, thus resulting in polynomial time complexity of $\mathcal{O}(Slog(S) + |D'|log(|D'|) + S.M.I.|D'|)$. Thus, for the Initialisation phase worst-case time complexity of the QMPSO resolves to $\mathcal{O}(Slog(S) + |D'|log(|D'|) + S.M.I.|D'|)$.

Both DNCPSO and FSPP, do not use heuristics when creating initial population nor use heuristic-based normalisation. Furthermore, these algorithms do not support throughput aware scalability of microservices. Thus, random initialisation of eligible solutions within resource-constrained devices results in time complexity of $\mathcal{O}(S.M.|D'|)$ for DNCPSO and FSPP.

Evolution: For this phase time complexity of the algorithm is dominated by the construction of the next solution. For QMPSO, this consists of velocity update and position update. The time complexity of velocity update is equal to the number of elements in the velocity matrix, which is $\mathcal{O}(S.M.I.|D'|)$. QMPSO uses CPPC_VA algorithm to update the particle positions. Similar to CPPC_INIT, CPPC_VA also acquires prioritised microservice instances generated from OHPP, which does not add extra computations to the algorithm. To make velocity-aware updates, algorithm iterates through devices for each prioritised microservice instance which results in $\mathcal{O}(S.M.I.|D'|)$ iterations during the worst case. This results in time complexity of $\mathcal{O}(S.M.I.|D'|)$ for the Evolution phase. The time complexity of DNCPSO and FSPP for this phase becomes, $\mathcal{O}(S.M.|D'|)$ due to the lack of throughput aware scalability of microservices.

Although the novel approaches introduced in QMPSO add extra complexity to the algorithm, lack of these features results in a slower convergence rate, convergence to local optimums, lower QoS satisfaction, and lower resource utilisation as demonstrated by the results in Sections 4.5.3 and 4.5.3. Thus, this trade-off between accuracy and extra computation time is vital in solving the microservices-based application placement in Fog. Moreover, the added time complexity due to these improvements is limited to linearithmic increase for sorting operations and an increase by a factor of I for throughput aware scaling of microservices. Thus, QMPSO reaches a fair trade-off between performance of the placement and extra time-complexity by maximising QoS satisfaction and resource usage of the placement while avoiding a drastic increase in time complexity.

4.6 Summary

Rapid growth in IoT has resulted in the emergence of diverse and complex applications developed using the microservices architecture. To fully leverage the capabilities

of Fog devices to support multiple heterogeneous applications, we exploited the granularity and scalability of microservice architecture and formulated the Fog application placement problem as a Lexicographic Combinatorial Optimisation Problem for batch placement of IoT applications, where QoS satisfaction and optimum resource usage are the primary and secondary objectives respectively. To solve the placement problem, we proposed an algorithm by adapting and improving the S-CLPSO technique. Extensive experiments are carried out to evaluate the effectiveness of the proposed technique under two aspects; convergence improvement against other adaptations of the S-CLPSO and efficiency of the resultant placement against state-of-the-art techniques. Obtained results depict that our approach successfully navigates large solution spaces and generates placements with higher QoS satisfaction (35% and 70% improvement in makespan and budget satisfaction, respectively) while ensuring optimum Fog and Cloud resource usage.

The placement approach proposed in this chapter considered throughput, makespan and budget as QoS parameters but did not account for the uncertainties caused by the failures within Fog environments, which is a vital requirement for mission-critical applications. In the next chapter, we focus on mission-critical IoT applications and study the proactive redundant placement of microservices to satisfy their stringent reliability requirements.

Chapter 5

Reliability-aware Proactive Placement of Mission-critical IoT Applications

Reliability remains one of the most critical QoS requirements for mission-critical IoT services deployed within Fog environments due to the lower dependability of Fog resources compared to the Cloud. Granular microservices with independent deployment and scaling exhibit great potential in utilising resource-constrained Fog resources to improve reliability through redundant placement. However, current research on service placement lacks reliability-aware holistic approaches that combine the MSA features and failure characteristics of Fog resources under independent and correlated failures. Moreover, proactive redundant placement approaches must consider constraints due to resource limitations within Fog environments and the increase in the cost of deploying redundant microservice instances. Hence, we analyse MSA and formulate the reliability-aware placement problem by modelling composite services as k-out-of-n serial-parallel systems in a throughput-aware manner for placement under Fog resource failures. Our proposed Reliability-aware Placement Method (RPM) is a hierarchical policy combining improved PSO and NSGA-II algorithms. We integrate it with Monte Carlo reliability calculations to produce redundant placements to maximise reliability satisfaction. Moreover, the proposed approach aims to reduce the cost of deployment as a secondary objective. The performance results reveal that compared to the benchmarks, our algorithm shows significant improvements in reliability satisfaction (up to 25%) and time to first failure (up to 40%), thus providing a robust placement method.

This chapter is derived from:

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "Reliability-aware Proactive Placement of Microservices-based IoT Applications in Fog Computing Environments", *IEEE Transactions on Mobile Computing (TMC)*, (revision, August 2023).

5.1 Introduction

IoT applications include highly safety-critical and mission-critical services such as smart healthcare, intelligent transportation, and Industrial Internet of Things (IIoT). For such services, high reliability is a crucial requirement [128]. Moreover, heterogeneity of Fog resources and their resource-constrained and geo-distributed nature results in lower dependability compared to the powerful, robust and centralised Cloud servers [129, 130]. Thus, application deployment within Fog computing environments should incorporate reliability awareness to minimise the application unavailability caused by Fog device failures (i.e., hardware, software, power, network, etc.) while satisfying multiple other QoS requirements such as deadline and throughput.

Over the years, two main approaches have been introduced to maintain application reliability: proactive failure avoidance and reactive failure recovery techniques. For IoT services with stringent latency requirements, reactive algorithms that focus on healing after faulty events are insufficient to ensure the higher level of availability required to meet low and ultra-low latency expectations, which fall within millisecond deadline limits [45, 131]. Hence, proactive methods driven by redundant placement are identified as viable solutions. In Cloud environments, redundant placement is limited by the high costs incurred by deploying multiple copies of the application. In Fog environments, this is further restrained by the limited availability of computing resources.

Under such challenges, the shift in IoT application development from monoliths to microservices has the potential to improve the proactive redundant placement within Fog environments due to their fine-grained design. Ability of MSA to support independent scalability of microservices, including both vertical and horizontal scalability, enhances the chances of throughput and reliability-aware redundant placement within resource-limited Fog devices (i.e., Raspberry Pis, small-cell base stations, nano data centres, edge servers etc.) with heterogeneous failure characteristics. Moreover, the granularity of microservices with well-defined business boundaries results in complex interactions among microservices to create composite services. Furthermore, this results in each microservice-based application being a composition of multiple services with heterogeneous QoS requirements (i.e., latency-critical, latency tolerant, high bandwidth

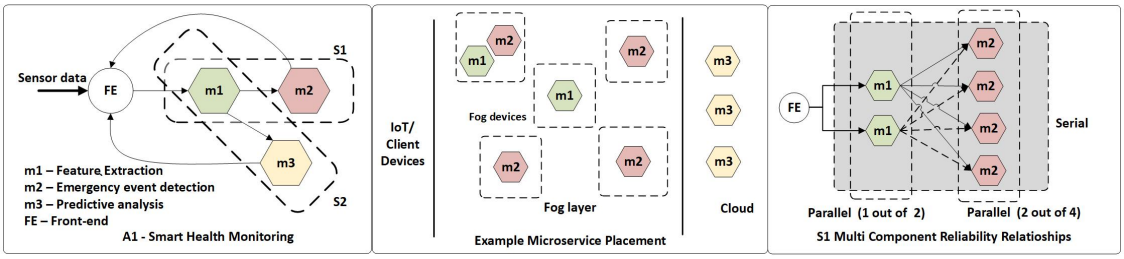


Figure 5.1: A scenario of usecase in the context of smart heath monitoring

consuming etc.) where some microservices are shared among various services. This enables per-service QoS definitions which can be used with batch placement to utilise Fog and Cloud resources in a balanced manner [46]. While MSA presents potential improvements to the reliability-aware proactive placement of IoT applications, they also introduce critical challenges. Complex interaction patterns among scalable microservices results in cascading and correlated failures which increases the complexity of the reliability model of the microservice-based applications.

Microservices-based application placement falls under Fog Service Placement Problem (FSPP) [32], [46], where each application service is deployed to provide shared access to a large number of users. Thus, concepts such as throughput-aware service scalability and load sharing are important aspects of FSPP, which sets it apart from DAG-based workflow scheduling and task offloading problems studied in the existing literature [32]. Existing research on FSPP mainly focuses on QoS parameters such as latency, cost and throughput. Thus, reliability-aware placement has a lot of room for improvement, especially for IoT applications developed using MSA. Existing works lack proper analysis of the potential of microservices-based IoT application architecture to introduce novel placement algorithms that enable the proactive redundant placement to improve the reliability of the services under both independent and correlated failures of the Fog resources. To further highlight this idea, we present an IoT use case modelled using MSA and examine its reliability-aware placement.

5.1.1 Motivational Scenario

We consider a use case of a smart health monitoring application (see Figure 5.1) to demonstrate how MSA features can be utilised in achieving high reliability in Fog applications.

Due to the granularity of microservices, QoS requirements can be defined at the composite service level. Thus, $A1$ can be represented as a composition of two composite services: a mission-critical emergency event detection service (service *S1* consisting of microservices, *m1* and *m2*) and a latency tolerant, computationally intensive analysis service (service *S2* consisting of microservices, *m1* and *m3*) [46]. The loosely coupled nature of the microservices enables dynamic deployment of microservices across Fog layer resources and Cloud resources in a QoS-aware manner. In our example scenario, $m1$ and $m2$ are deployed in the Fog layer to accommodate the low latency requirement of $S1$, whereas $m3$, which only contributes to the latency tolerant service $S2$, is placed within Cloud data centres. It improves Fog resource utilisation, thus allowing more Fog resources to be allocated for services with stringent latency requirements.

Being a mission-critical service, *S1* has high-reliability expectations so that in case of an emergency, the application can react within the stringent latency expectations of the service. As services like *S1* have latency requirements in the millisecond range, in case of Fog resource failures, the effect on the service would be adverse if only reactive fault-tolerance methods were employed. Thus, such application services can benefit from proactive reliability ensured by redundant placements [45]. However, this is limited by the heterogeneity and resource-constrained nature of the Fog devices. The independently deployable and scalable nature of the microservices can be utilised to overcome this challenge. To this end, microservice instances packaged as lightweight Docker containers can be scaled horizontally or/and vertically in throughput and reliability-aware manner. Example use case indicates that to support user requests, at least one instance of $m1$ and two instances of $m2$ are required. Failure characteristics of the Fog devices can be used to improve this placement further so that redundant microservice instances are deployed to improve the service reliability. For example, the number of redundant placements can be increased if their deployed devices have low reliability (four instances of $m2$ and two instances of $m1$ depending on the failure characteristics of the Fog devices

they are deployed on). Hence, with MSA, each composite service is represented as a serial-parallel hybrid system, with each horizontally scaled microservice being a k out of n load-balanced sub-system of the end-user service. Here, k is the minimum number of microservice instances that can cater for the incoming user request volume, determined in a throughput-aware manner, whereas n is determined by integrating knowledge of the failure characteristics (i.e., independent and correlated failures) of Fog devices to ensure availability of at least k instances during application run time.

Thus, it is evident that MSA can provide the flexibility required to utilise resource-constrained Fog resources to improve the reliability of the deployed applications by introducing robust placement policies that combine MSA features with the failure characteristics of Fog resources.

5.1.2 Proposed Approach and Contributions

The above use case demonstrates that proper utilisation of MSA characteristics can potentially improve the reliability of mission-critical IoT services through proactive and dynamic redundant placement of microservices in a "reliability and throughput aware" manner. Research that emphasises the said characteristics is still in its early stages and has much room for improvement. Existing research lacks in multiple areas, such as utilising microservice features (i.e., granular design, independent deployment and scalability, balanced deployment between fog and cloud, per-service QoS-awareness), overcoming challenges of the MSA (i.e., complex interaction patterns among microservices), application batch placement to prioritise mission-critical services, consideration of multiple failure types (i.e., independent failures, correlated failures) and dynamic redundant placement of microservice. In this work, we aim to address these shortcomings by proposing a holistic placement approach that improves the reliability of the services under multiple reliability-related metrics, such as availability and time to first failure. The **key contributions** of our work are:

1. In order to capture MSA characteristics, we model the microservices-based application services as *k out of n* serial-parallel systems and formulate the placement problem to capture reliability, throughput awareness, and cost at the composite

service level. The problem formulation captures both independent and correlated failures within repairable fog environments and, dynamically calculate and place redundant microservice instances proactively.

2. Based on the problem formulation, we propose a hierarchical placement algorithm to place microservice replicas within fog environments proactively. Our proposed algorithm operates at two levels; Particle Swarm Optimisation based Throughput-aware Scalable Placement (TSP), Genetic Algorithm based Reliability-aware Redundant Placement (RRP), which together provide a robust placement method under failures in fog resources. Furthermore, a Monte Carlo-based approach is incorporated to calculate reliability-related parameters.

3. We improve the performance of the algorithm by introducing multiple novel processes: an availability-aware fitness function for TSP, an availability-aware heuristic redundancy placement for the initialisation of RRP and a reliability-aware dominant selection method for RRP.

4. We implement our policy using iFogSim2 [8] simulated fog environment and evaluate against multiple benchmarks based on reliability satisfaction, time to first failure and deployment cost.

5.2 Related Work

In this section, we summarise current works in Cloud and Fog environments (see Table 5.1) related to reliability-aware placement and proactive redundant placement, considering multiple placement problems such as FSPP, DAG workflow scheduling and task offloading. We also make a qualitative comparison between existing approaches and our work.

Multiple works consider reliability in Cloud environments for the deployment of workflows, where the majority focus on scientific workflows. Rehani et al. [132] propose a DAG workflow scheduling algorithm that considers the reliability of repairable Cloud resources for assigning tasks to VMs. They model the Cloud failures and repairs

using Weibull distribution and use Monte Carlo Failure Estimation to accurately calculate the time to failure and time to repair for each Cloud resource. Tang et al. [133] consider a multi-cloud scenario to improve the reliability of the DAG-based scientific workflows to reach a trade-off between cost and reliability using the hazard rates of VMs and their connected links. Zhu et al. [134] also present a fault-tolerant DAG placement by proposing primary-backup copy placement (PB) with one replica per task deployed as a backup. Their work assumes no simultaneous failures among devices and considers only one host fails at a time. The work in [135] extends this to consider network failures that can result in simultaneous failures of the hosts and propose a placement algorithm to place primary and its backup copy in different subnets to overcome such failures.

Works such as Yao et al. [136], Liu et al. [137] and Aral et al. [138] focus on reliability-aware scheduling within Edge computing environments. The works proposed in [136, 137] consider task-offloading problem considering failures of the edge VMs. Yao et al. [136] consider independent tasks whereas Liu et al. [137] model the application dataflows as DAGs. Both of these works assume the VM failures to be repairable and independent of each other. Yao et al. [136] try to achieve a trade-off between cost and reliability, whereas Liu et al. [137] aim to balance reliability and network usage. Aral et al. [138] introduce a Bayesian Network-based approach to model and detect correlated failures among edge nodes and combine it with link failure probabilities to calculate the joint failure probability of edge devices. When the minimum required replica count for each single-component service is provided as input, the approach presented in [138] outputs a redundant placement to minimise the joint failure probability of the replicas. The works presented in [32, 46, 139, 140], and [45] explore the effect of replica placement to improve the performance of the Fog application services. The works in [139, 140] consider monolith applications, whereas the works in [32, 45, 46] model the applications following MSA. The approaches proposed in [139, 140] and [46] place the minimum number of required microservice replicas to satisfy the throughput requirements of the services but do not consider redundant placements to handle uncertainty. The work in [45] tries to overcome the throughput uncertainty of the services where some of the microservices have multiple candidates, whereas the approach presented in [32] proposes a method to evenly distribute microservices across the Fog resources to

Table 5.1: Comparison of existing research

Work	Research Problem	Environment	Application Model	QoS			Failure Characteristics		Scalability			Batch Placement
				Reliability	Throughput	Other	Type	Repairable	Redundancy	Replica Calc.	Load Balance	
[132]	Workflow Scheduling	Cloud	DAG Workflows	✓	-	Latency	Independent	✓	-	-	-	-
[133]				✓	-	Latency, Cost	Ind., Corr.(Network)	-	✓	Static - (PB)	-	-
[134]				✓	-	Latency	Independent	✓	✓	Static - (PB)	-	-
[135]				✓	-	Latency	Ind., Corr. (Network)	-	✓	Static - (PB)	-	-
[137]	Task Offloading	Edge-Only	Independent Tasks	✓	-	Latency, Bandwidth	Independent	✓	-	-	-	✓
[136]				✓	-	Latency, Cost	Independent	✓	-	-	-	✓
[138]				✓	-	Latency, Cost	Correlated	-	✓	Dynamic	✓	-
[32]	FSPP	Fog	MSA	-	-	Latency	-	-	✓	Dynamic	✓	✓
[45]				-	✓	Latency	-	-	✓	Dynamic	✓	-
[46]				-	✓	Latency, Cost	-	-	-	Dynamic	✓	✓
[139]		(Edge-Cloud)	Monolith	-	✓	Latency	-	-	-	Dynamic	✓	✓
[140]				-	✓	Latency	-	-	-	Dynamic	✓	✓
Our	FSPP	Fog	MSA	✓	✓	Latency, Cost	Ind., Corr.	✓	✓	Dynamic	✓	✓

improve service availability.

Qualitative Comparison: DAG workflow scheduling in the Cloud [132–135] and IoT application offloading in the edge [136, 137], both consider workloads with ephemeral life cycles where the problem is addressed from the user perspective such that the DAGs/-tasks are deployed to be used by a particular user, and after the execution, each task is removed from the environment giving way to the following tasks in the queue. In contrast to this, our work considers Fog Service Placement Problem (FSPP) described in many previous works such as [32, 45, 46], where the placement is addressed from the application provider's perspective where applications are used by a large number of users and process continuous requests, making their life cycle perpetual. This makes it infeasible to adapt former approaches to reliability-aware FSSP. Furthermore, throughput-awareness, horizontal/vertical scaling, and load balancing become essential aspects of the FSPP, which are not considered in [132, 136, 137], etc. Moreover, MSA creates composite services with complex interaction patterns among microservices. Existing works like [136, 138–140] consider independent tasks or single component services, thus failing to capture the effect of such dependencies in modelling system reliability. Works such as [32, 45] consider complex interactions among microservices along with redundant placement of microservices but do not consider failure characteristics of the Edge/Fog nodes to improve the reliability of the placement. Among the works that consider failures within Fog environments, some consider independent failures [132, 134], whereas

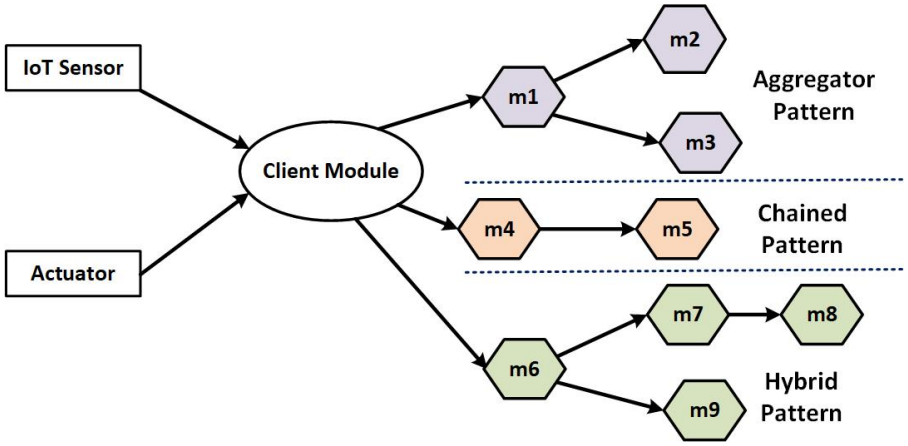


Figure 5.2: Microservices-based application model

works like [138] consider correlated failures. The work in [135] considers both independent and correlated failures but limits it to network failures that can be isolated at the subnet level.

Based on the above analysis, existing works lack a holistic approach that captures all the above characteristics. To this end, in our work, we consider MSA characteristics (i.e., composite services, microservice interaction patterns, independent scalability, load balancing, etc.) and propose a reliability-aware redundant placement approach for application batch placement under Fog resource failures (both independent and correlated failures). We further improve the robustness of the algorithm by dynamically calculating the number of microservice replicas in a "throughput and reliability-aware" manner while reaching a trade-off between reliability and cost.

5.3 System Model and Problem Formulation

5.3.1 Microservices-based Application Model

Microservices-based applications can be modelled using a DAG [46] where vertices denote microservices and edges represent the interactions among microservices with direction from client microservice towards the invoked microservice (Figure 5.2). Each application, $a \in A$, is depicted as a collection of microservices, data flows among them,

and a set of composite services providing end-user requested functionalities denoted as $< M_a, df^a, S_a >$. Each microservice is defined based on its resource requirements; $< \Gamma_m, r_m >$ where Γ_m can be a combination of multiple resources such as CPU, RAM and storage requirements of microservice $m \in M_a$ to support the request rate of r_m. This acts as the basic deployment unit of each microservice, which can be independently scaled (horizontally and vertically).

The granularity of MSA supports complex interactions, thus creating various composite service patterns (i.e., *Chained*, *Aggregator* and *Hybrid*) with diverse data flow representations (i.e., chained pattern as a single chain, aggregator pattern where multiple data paths are invoked and results are aggregated to return a single response, etc.). These data flow characteristics affect the end-to-end latency of the composite services. Thus, we represent each service $S \in S_a$ by a tuple containing the set of all microservices of the service and all possible data paths within the service: $< M_a^s, P_a^s >$.

5.3.2 Fog Computing Environment Model

The Fog environment is represented by a hierarchical architecture consisting of three main layers: IoT/client devices, Fog layer and Cloud layer. The Fog layer, which resides between end devices and the Cloud, contains heterogeneous, resource-constrained, distributed devices that provide computational, networking and storage closer to the edge of the network. We model the Fog layer as clusters of such Fog nodes managed by multiple service providers. Client devices access Fog resources through gateway devices such as wireless access points and base transmission systems using WLAN technologies. These Fog clusters maintain seamless connectivity with the Cloud with WAN links through fog-cloud gateways. Intra-cluster communication is established using high bandwidth LAN to achieve high throughput and low latency within the Fog clusters. As Fog devices are heterogeneous in resource availability, we characterise each device ($d \in D$) based on its resources (γ_d). γ_d can be a combination of resources including, but not limited to, CPU, RAM and storage. Moreover, in this work, we also consider the failure characteristics of the Fog devices, detailed in the following sections.

5.3.3 System and Failure Characteristics

In this section, we analyse microservices-based application architecture and Fog environments to create a reliability model.

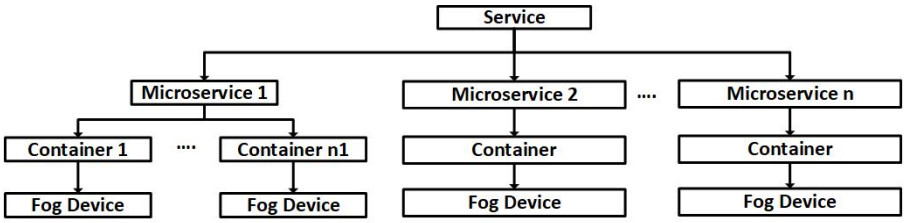
Reliability Analysis of Microservice Applications

A failure is an event that causes a system to become unable to perform its intended task reliably [141]. A system can consist of one or more components, where system reliability depends on the failure and repair characteristics of these components. Thus, for the microservices-based application placement, we identify the system boundaries, decompose the system to identify its components and their failure characteristics, and afterwards model their effect on system performance. Figure5.3a depicts the multi-level representation of the system.

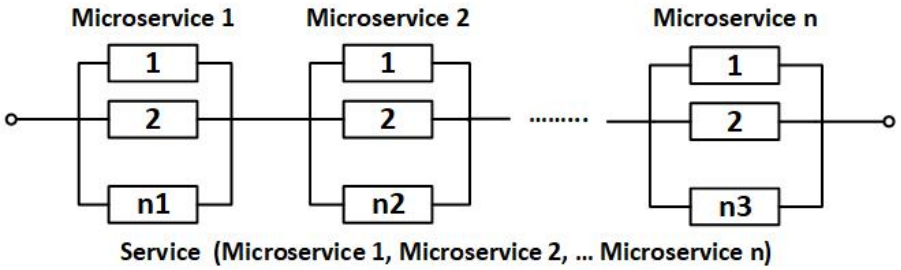
For the reliability modelling of a microservices-based Fog applications, we consider each end-user service as a separate system with reliability requirements realised at the service level. Each service consists of one or more independently deployable and scalable microservices with data dependencies among them. Accordingly, we formulate the block representation of the system (Figure 5.3b) to analyse the effect of component failures on the system performance. For a service S with $\overline{M}^s$ critical microservices ($\overline{M}^s \subset M^s$), each microservice $m \in \overline{M}^s$ can be horizontally and vertically scaled to meet the user demand by utilising resource-constrained Fog resources. Service failure occurs when the service is unable to maintain the expected level of QoS (i.e., deadline and throughput) due to the failure of one or more microservice instances belonging to the service.

If microservice m requires a minimum of k instances to support the expected throughput demand, m is considered to be operating as expected if a minimum of k instances out of the deployed n are running without failures. Furthermore, to maintain service availability, all critical microservices of the service should be running without failures. This results in a serial relationship among microservices of the service where the failure of one or more microservices results in degrading the service performance or making the service unavailable until the system is restored.

Hence, for a microservices-based IoT application, reliability can be analysed per each composite service by modelling the service as a ***serial-parallel hybrid system*** of its critical microservices and their replicas. Following this model, we analyse the effect of underlying Fog resource reliability on the availability of the service under two main resource failure types: independent and correlated.



(a) Multi-level representation



(b) Block diagram representation

Figure 5.3: Multi-component system reliability model

Independent Failures

Independent failures in distributed computing environments include failures of servers/nodes due to factors such as hardware failures (i.e., disk failures) and software/OS failures (i.e., kernel failures, firmware failures etc.) that occur individually and independently among nodes. In literature, such failures are analysed using failure probability density functions (i.e., Weibull, Lognormal, Poisson etc.) of each node defined independently [132, 137]. Using this information, the reliability of multi-component systems can be analysed based on metrics such as Time To Failure (TTF) and availability [138].

Within Fog and Cloud environments, computation nodes can be repaired after failures or deployed containers can be redeployed or migrated to working nodes upon the

failure of the current nodes. As a result, in analysing the reliability of such systems, TTF can be identified as an essential metric. By maximising the TTF of services, we can minimise the number of times the microservice instances have to be redeployed or migrated to maintain service QoS, thus improving service reliability in mission-critical scenarios. At the same time, Service Level Agreements (SLAs) of the services include the reliability of the service in terms of expected average uptime availability. For microservices-based IoT applications, this can be defined at the composite service level. Hence, in this work, we create the reliability model considering both TTF and availability.

1. TTF Calculation

Based on the proposed serial-parallel hybrid reliability model of a service, the TTF of service S can be defined as,

$$TTF(S) = min[TTF(m); \forall m \in \overline{M}^s] \tag{5.1a}$$

For each microservice, the TTF is determined considering the ***k-out-of-n load balancing system*** represented by its instances. For microservice $m \in \overline{M}^s$, if I_m is the set of $|I_m| = n_m$ instances, the TTF of m is defined as,

$$TTF(m) = min[TTF(I'_m); \forall I'_m \subset I_m] \tag{5.1b}$$

where $|I'_m| \geq (n_m - k_m + 1)$.

As we consider failure of each microservice instance due to the underlying host failures, failure of m occurs when the Fog devices that host $n_m - k_m + 1$ instances or more of the microservice fail. If $f[d_{m_i}]$ indicates the failure of Fog device hosting instance m_i of microservice m, $TTF(m)$ can be reduced to the minimum time to joint failure of the devices as follows,

$$TTF(I'_m) = min[T(\bigcap_{m_i \in I'_m} f[d_{m_i}])] \tag{5.1c}$$

2. Availability Calculation

Based on the proposed reliability model, the availability of service S can be defined as,

$$\mathrm{AV}(S)_{t1,t2} = \frac{1}{(t2-t1)} \int_{t1}^{t2} Av_S(t)dt \tag{5.2a}$$

$$Av_S(t) = \begin{cases} 1 & \mathrm{Up}(I_{m,t}) \geq k_m; \forall m \in \overline{M}^s \\ 0 & \text{otherwise} \end{cases} \tag{5.2b}$$

Eq. 5.2a defines mean availability of the service S within [t1,t2] time period in terms of service uptime. Function $Av_S(t)$ denotes if the service is in up or failed status at time t. In Eq. 5.2b, function $\mathrm{Up}(I_m, t)$ calculates the number of running instances of microservice m at time t. Above two equations together calculate the average uptime availability of the service S following *k out of n* load balancing model.

Correlated Failures

Correlated or dependent failures, also known as Common Cause Failures (CCF), indicate one or more components of the system failing simultaneously due to a common cause. Within distributed computing environments, this can be due to failures of shared power supplies, virtual networks, network component failures, software updates, etc. [135, 138]. Such failures affect the redundant placement decisions as deploying redundant instances within a group of servers that belong to the same Common Cause Failure Group (CCFG) reduces its effectiveness. Considering this, we propose a Discorrelation Index (DI) for each microservice as follows:

$$DI(m) = \frac{\sum_{\forall g \in G} min[\frac{|I_m \setminus F_G(g, I_m)|}{k_m}, 1]}{|G|} \tag{5.3a}$$

Eq. 5.3a considers each sub system (microservice) having a parallel relationship among its components (microservice instances). Here, $F_G(g, I_m)$ returns the instances that belong to the same CCFG ($g \in G$) and calculates the *k out of n* instance satisfaction under CCF. Based on this, calculations for each service S can be represented as follows:

$$DI(S) = \frac{\sum_{\forall m \in \overline{M}^s} DI(m)}{|\overline{M}^s|} \tag{5.3b}$$

5.3.4 Throughput-aware Minimum Instance Calculation

In the k out of n parallel model derived for each microservice, k can be determined in a throughput-aware manner where the throughput requirement is defined per service (r_s for service S). We take the microservice definition proposed in our application model (Section 5.3.1), where the resource requirement for the microservice is defined to support a certain request rate. We consider this as the base microservice instance to be deployed as a Docker container and calculate the number of instances required to support the incoming request volume. For each microservice in the DAG representation, its expected incoming request rate (r'_m) is calculated using the following equations:

$$r'_m = \sum_{\forall m' \in CM(m)} R_{m'm} \tag{5.4a}$$

$$R_{m'm} = \begin{cases} r_s & m' \text{ is Client Module} \\ \alpha . r'_{m'} & \text{otherwise} \end{cases} \tag{5.4b}$$

The access rate of the microservice m is calculated by identifying all incoming edges of m and adding their request rates (Eq. 5.4a). To achieve this, the function $CM(m)$ outputs the client microservices of m based on the DAG representation of the application. $\alpha \in [0, 1]$ indicates the difference in rates between incoming and outgoing requests of m'. Afterwards, the minimum instance count for the microservice m is calculated as,

$$k_m = \frac{r'_m}{r_m} \tag{5.4c}$$

5.3.5 Service Latency Model

Due to the granularity of the MSA, deadlines can be defined at the composite service level, where the latency of each service depends on the data flow pattern of the service. Considering multiple service composition patterns, the deadline violation of service S with a deadline of l_S can be calculated based on the latency of the longest data path of the service. Considering each data path within the service ($p \in P^S$), function $L(df_p^S)$ calculates the total latency of the datapath p of service S for the proposed placement.

Due to distributed nature of the fog resources, the total latency consists of network latency ($L_{nw}(df_p^S)$) and processing latency ($L_{proc}(df_p^S)$), where network latency is a combination of transmission latency and propagation latency among different fog/cloud nodes where the microservices are deployed.

$$v_S^l = max\{L(df_p^S); \forall p \in P^S\} - l_S \tag{5.5a}$$

$$L(df_p^S) = L_{nw}(df_p^S) + L_{proc}(df_p^S) \tag{5.5b}$$

5.3.6 Pricing Model

Cloud service providers support container deployment through serverless compute engines (i.e., AWS Fargate, Azure Container Instances etc.) where pricing is calculated based on the requested vCPUs, memory and storage and flexibility is provided to configure each separately. In our work, we use the above on-demand pricing model to determine the price of deploying microservices within Fog and Cloud servers using container technology. For a service S having a set of M^s microservices, the total cost can be calculated as,

$$C(S) = \sum_{\substack{\forall m \in M^s \\ \forall d \in D}} \sum_{i=1}^{n_m} x_{m_i}^d C_m^d \tag{5.6}$$

C_m^d indicates the total cost of deploying microservice m on device d. $x_{m_i}^d \in \{0,1\}$ is a binary variable which is set to 1 if the i^{th} instance of the microservice m is deployed on device d. According to the above equation total cost for service S is calculated as the total cost for deployment of all microservices instances.

5.3.7 Problem Formulation

Based on the system model, we formulate the reliability-aware placement problem as a multi-objective optimisation. As proactive redundant placement of microservices is limited by the cost of resource allocation and resource availability in Fog environments,

the placement problem aims to reach a trade-off between maximising reliability (Eq. 5.7) and minimising the cost (Eq. 5.8). Based on the proposed reliability model, the reliability of the services is represented as a composite of three metrics: TTF, availability and DI. Furthermore, the placement aims to satisfy three constraints: resource constraints (Eq. 5.9a), service deadline (Eq. 5.9b) and throughput requirements of the services (Eq. 5.9c).

$$\max \mathrm{P}(A_s) = \sum_{\forall S \in A_s} \Big[TTF(S), AV(S), DI(S) \Big] \tag{5.7}$$

$$\min C = \sum_{\forall S \in A_s} C(S) \tag{5.8}$$

Subject to,

$$\sum_{\substack{\forall a \in A \\ \forall m \in M_a}} \sum_{\forall m_i \in I_m} x^d_{m_i} \Gamma_m \leq \gamma_d; \forall d \in D \tag{5.9a}$$

$$V^l_S = 0; \forall S \in A_s \tag{5.9b}$$

$$n_m \geq k_m; \forall m \in M_a; \forall a \in A \tag{5.9c}$$

As application placement within Fog environments has to utilise the limited Fog resources and achieve a proper balance between Fog layer resource usage and Cloud usage, the batch placement of applications contributes to prioritising services based on heterogeneous QoS requirements. Thus, we formulate our placement problem to support the placement of a set of applications A, where all the available services are depicted by A_s.

5.4 Reliability-aware Placement Method (RPM)

5.4.1 Overview

Based on the problem formulation, we propose a ***Reliability-aware Placement Method (RPM)*** for the proactive redundant placement of microservices-based IoT applications. Figure 5.4 presents a high-level representation of the method. Our approach consists of four main processes:

- Monte Carlo Simulation-based Service Reliability calculation process: It uses empirical data derived from past failures of the devices to calculate time to failure $(TTF(S))$ and availability $(AV(S))$ metrics based on independent failures.
- DI calculation process: It calculates DI using data on CCFGs derived from common course failure data.
- Throughput-aware Scalable Placement **(TSP)** - It generates initial microservice placement with the minimum number of microservice instances to satisfy the throughput demand.
- Reliability-aware Redundant Placement **(RRP)** - It extends TSP to accommodate the redundant deployment of microservices to improve reliability in a cost-aware manner.

Monte Carlo reliability calculation and DI calculation provide service reliability-related metrics considering independent and correlated failures of the fog devices (i.e. TTF, Availability, *DI*). These metrics are used by TSP and RRP, which create a hierarchical approach for throughput, reliability and cost-aware redundant placement of a batch of IoT applications.

Our proposed approach assumes the availability of previous failure data of the fog resources and meta-data derived from them. This includes data related to both independent failures and correlated failures. Previous works such as [142, 143] use publicly available failure and repair data of cloud data centres to derive statistical parameters for failure and repair distributions using empirical analysis. In our approach, such parameters derived for each fog node are provided as metadata to Monte Carlo-based reliability calculation process to derive reliability metrics based on independent failures of fog devices. To identify the possibility of correlated failures among devices, the *CCF* analysis also can be conducted using past failure data to identify spatial and temporal dependencies among fog nodes. [138] proposes a method based on a Dynamic Bayesian Network to identify fog nodes that can fail together. Using such approaches, fog devices that belong to the same *CCFG* can be determined to be used as input for calculating DI by the DI calculator process.

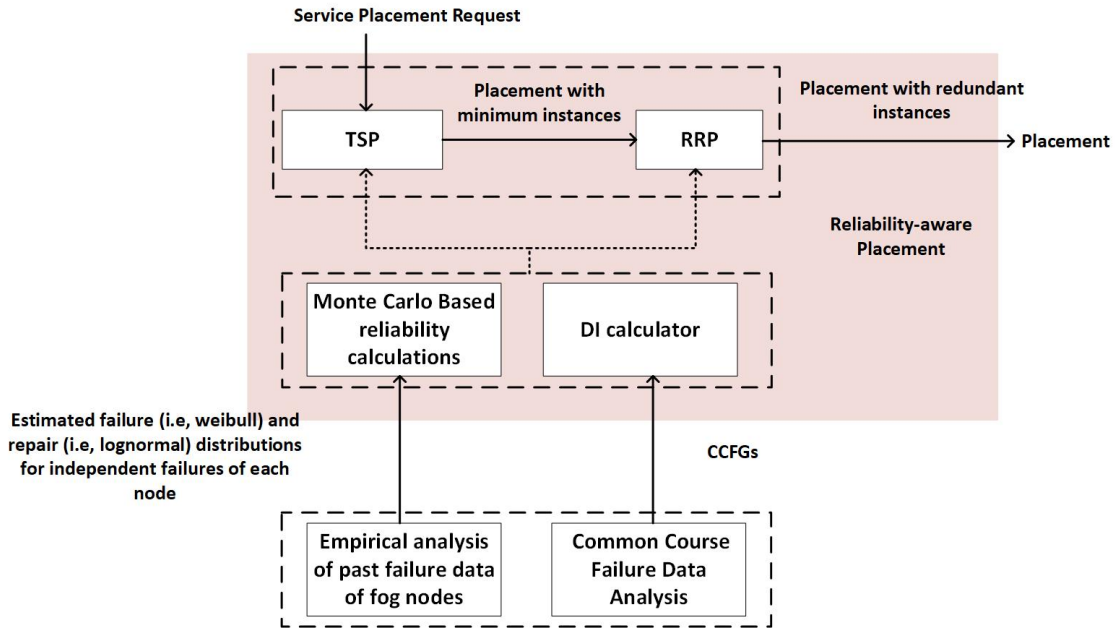


Figure 5.4: Reliability-aware placement process

Our placement method (RPM denoted in Figure 5.4) uses these data to propose a redundant placement method following the reliability model proposed specifically for microservices-based IoT applications. Thus, the process of deriving statistical parameters and dependency information from past failure data is out of the scope of this work. We base our policy on the derived metadata with the flexibility of updating the methods used to extract the metadata.

5.4.2 Monte Carlo Simulation-based Service Reliability

Due to non-constant failure/repair rates of the components, the use of Markov chains and Bayesian Networks for reliability analysis becomes impractical [144]. For such repairable systems, Monte Carlo Simulation is better suited. Monte Carlo Simulation performs a virtual experiment that simulates random walks within the stochastic environment using random number generation from known probability distributions [144]. When the parameters for the failure and repair distributions of each Fog node are estimated from past failure data, the Monte Carlo method uses values drawn from a uniform random variable $U(0,1)$ together with the Inverse Cumulative Distribution Func-

tion (ICDF) of the distribution to generate failure and repair times repeatedly to create histories of the system that are used to derive failure and repair times within a considered time duration.

Data centre failure and repair data analysis presented in [142, 143] shows that server failures best fit the Weibull distribution while repair times can be best modelled using Lognormal distributions. Thus, in our work, we consider these distributions to model failure and repair times of the Fog nodes. However, the use of Monte Carlo Simulations to determine reliability metrics makes the approach easily adaptable to any distribution due to its use of the inverse transform method.

As most of the failures in Fog resources are repairable, the effect of the repair/maintenance actions on the status of the Fog nodes needs to be considered. Kijima [145] analyses such systems and proposes a model based on the system repair condition known as *general renewal process* which models general or imperfect repair of the components where the failed system is returned to a state between new and prior to the most recent failure by introducing a virtual age to the component. For a component having virtual age $V_{i-1} = v$ after the $(i-1)^{th}$ repair, the CDF for the time to i^{th} failure T becomes,

$$F(T|V_{i-1} = v) = \frac{F(T+v) - F(v)}{1 - F(v)} \tag{5.10}$$

For failures following the weibull distribution this results in the ICDF,

$$t_1 = \eta \sqrt[\beta]{-ln(1-U)} - t' \tag{5.11}$$

where t' is the time elapsed since last failure of the component from the historical data. For $i \geq 2$, ICDF is calculated as,

$$t_i = \left[\eta \sqrt[\beta]{(\frac{v_{i-1}}{\eta})^{\beta} - ln(1-U)}\right] - v_{i-1}; i \geq 2 \tag{5.12}$$

where virtual age is calculated using repair degree q [146, 147] as follows:

$$v_{i-1} = q(t_1 + t' + t_2 + ... + t_{i-1}); 0 \leq q \leq 1 \tag{5.13}$$

Parameter q enables the system reliability measurements to be adjusted based on

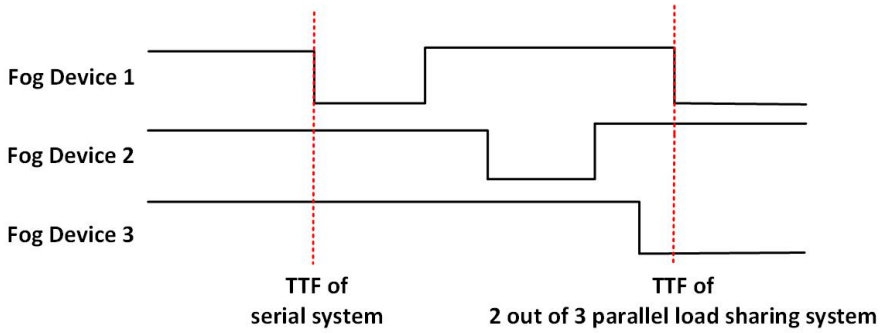


Figure 5.5: Monte Carlo based TTF calculation

repair characteristics. q indicates the remaining damage after the repair, where $q = 0$ and $q = 1$ represent the two extreme cases of perfect repair and minimal repair, respectively [148].

Accordingly, we propose Algorithm 9 to calculate the expected $TTF(S)$ and $AV(S)$ of each service using Monte Carlo simulations. Figure 5.5 shows a visual representation of how the algorithm calculates TTF for a service. For clarity, Algorithm 9 is presented as a combination of conducting Monte Carlo simulations (lines 3-20) and calculating relevant metrics using resultant events (lines 21-26). However, it's important to note that Monte Carlo simulation is a less frequently process that needs to be done as new empirical data become available or periodically. Calculated events are stored and used by placement policy which is a more frequent process. Due to this approach, Monte Carlo simulations can be carried out in Cloud servers, thus overcoming the computation complexity of the process and mitigating its effect on the placement algorithm.

5.4.3 Stage 1 - Throughput-aware Scalable Placement

Throughput-aware Scalable Placement (TSP) is the first stage of our hierarchical placement policy (see Algorithm 10). TSP outputs a reliability-aware, scalable placement based on the throughput requirements of the services but does not focus on the deployment of redundant microservice instances.

At this stage, since the number of exact instances per each microservice is calculated using Eqs. 5.4a-5.4c, we use a Particle Swarm Optimisation (PSO) based meta-heuristic

Algorithm 9 Monte Carlo based Service Reliability

Input: Placement P for service S, Estimated failure and Repair distributions for each Fog device

Output: $TTF(S)$, $AV(S)$, $Events$

$\overline{M}^s \leftarrow S$.getMicroservices(); $D \leftarrow P$.getAllMappedDevices()
$Events \leftarrow \{\}$
for d in D **do**
    $i \leftarrow 0$;
    for $i \leq simTimes$ **do**
        set $t \leftarrow 0.0$; $status \leftarrow UP$; $currentEvent \leftarrow$ first event
        while $t \leq T_{max}$ **do**
            $u \leftarrow sample(U(0,1))$
            if $status = UP$ **then**
                $\Delta t \leftarrow$timeToNextFailure(d,u,$Events.get(d)$) ▷ This is calculated using Eqs. 5.11, 5.12
            else
                $\Delta t \leftarrow$timeToRepair(d,u); ▷ This is calculated using the ICDF of Lognormal distribution
            $Events$.updateAverage($d, currentEvent, \Delta t$)
            $t \leftarrow t + \Delta t$; $currentEvent \leftarrow$ next event
            $status \leftarrow (status = UP)?DOWN : UP$
        $i \leftarrow i + 1$
for m in $\overline{M}^s$ **do**
    $D_m \leftarrow P$.getMappedDevices(m); $n_m \leftarrow$ no of min instances
    $ttf_m \leftarrow$ calculate time to $(n_m - k_m + 1)$ or more simultaneous failures based on $Events$ related to D_m
$TTF(S) \leftarrow minimum(ttf_m; \forall m \in \overline{M}^s)$
$AV(S)_{0,t} \leftarrow$ calculateAvailability($Events$) ▷ AV(S) is calulated applying Eqs. 5.2 to the calculated $Events$
return $TTF(S)$, $AV(S)$, $Events$

to achieve throughput-reliability aware placement under resource and deadline constraints. In our previous work [46], we examined the adaptability of Set-based Comprehensive Learning Particle Swarm Optimisation (S-CLPSO) for microservices-based application placement to satisfy throughput, latency and cost requirements and introduced multiple approaches to improve its ability to achieve quicker convergence and reach the global optimum. Thus, in this stage, we adapt the improved S-CLSPO algorithm but extend and further improve it to solve the reliability-aware placement problem as follows:

Algorithm 10 TSP Algorithm

Input: Placement Requests and Meta-data
Output: Microservices to devices mapping

Calculate the number of instances per microservice (Eqs.5.4)
Set iteration count $i \leftarrow 1$
▷ Prioritise microservices based on deadline of the composite services they belong to
Place all in Cloud and calculate deadline violation (Eq. 5.5a)
$ToFogM \leftarrow$ deadline violated; $ToCloudM \leftarrow$ deadline satisfied
▷ Construct a random swarm of N particles under deadline and resource constraints
$Particles \leftarrow$ initialise($N, ToFogM, ToCloudM$)
while $i \leq Iterations$ **do**
    Calculate fitness of each particle using **AFF**;
    Update $pBest$ and $gBest$
    Select exemplar dimensions for each particle
    Update velocity of each particle
    ▷ Update position using deadline-resource constrained prioritised construction
    for $p \in Particles$ **do**
        for $m \in ToFogM$ **do**
            $D' \leftarrow$ eligibleFogDevices(m,p.velocityMatrix)
            Try to place m in a $d \in D'$ s.t resource constraints satisfied
            if not placed **then** $notPlaced.add(m)$
        for $m \in notPlaced$ **do**
            Try to place m in a $f \in fogDevices$ s.t resource constraints satisfied
            if not still placed **then** Place in $Cloud$
        Place $ToCloudM$ in $Cloud$
    Set $i \leftarrow i + 1$
return $gBest$ of the swarm

1. Availability-aware fitness function (**AFF**): Being the first stage of the policy, the aim of TSP is to provide an output that has the potential to be further improved with redundant placements in the next stage. To this end, we introduce a novel fitness function (Eq. 5.14a) with 3 metrics: 1) TTF of each service, 2) a novel Availability Score for each microservice (Eqs. 5.14b, 5.14c) which is introduced by modifying Eqs. 5.2 to calculate the mean number of active instances during service failure, thus aiming to minimise the simultaneous failures among its instances and improve the possibility of finding redundant placements during Stage 2 of the algorithm, and 3) DI of the placement which is also used to minimise simultaneous failures. For each particle, fitness is calculated as

the summation of reliability, $\rho(S)$ of all the services considered for placement.

$$\max \rho(S)_{t1,t2} = \Big[\frac{TTF(S)}{t2-t1} + \sum_{\forall m \in \overline{M}^s} AS(m).DI(m)\Big] \tag{5.14a}$$

$$\mathrm{AS}(m)_{t1,t2} = \frac{1}{(t_{fail})} \int_{t1}^{t2} As_m(t)dt \tag{5.14b}$$

$$As_m(t) = \begin{cases} \frac{\mathrm{Up}(I_{m,t})}{k_m} & \mathrm{Up}(I_{m,t}) < k_m \\ 0 & \text{otherwise} \end{cases} \tag{5.14c}$$

2. Multiple constraint handling: Each particle has to satisfy three main constraints to be considered a valid placement: throughput requirement of the service, resource constraints of Fog devices and deadline of the services. The throughput requirement is handled at the start of the algorithm (line 1) by calculating the minimum number of instances (k_m) required. Other constraints are handled at the particle construction during the initial swarm creation (line 5) and the position updates conducted in each iteration (lines 11-22). To achieve deadline satisfaction, first, the deadline stringent microservices are identified (lines 3-4) and prioritised for placement within Fog under resource constraints. For initialisation (line 5), the algorithm constructs particles through random assignment of microservice to devices such that the constraints are satisfied. To further improve the convergence, we seed the initial swarm with a reliability-aware heuristic placement that sorts Fog devices based on their time to first failure and map the *ToFogM* to devices with the highest time to failures. For the particle position update process, a velocity-aware position update method is implemented with deadline-resource constrained construction of particles to ensure the satisfaction of the constraints. Position update is conducted in a prioritised manner, starting with latency-sensitive microservices (lines 12-20). *eligibileFogDevices()* (line 13) method finds eligible devices in a velocity-aware manner where devices with equal or higher velocity compared to the current placed device are selected as eligible devices for the subsequent placement. This prioritises latency-critical microservices for placement within the Fog, thus maximising the deadline satisfaction of the placement.

3. Updating *pBest* and *gBest*: Due to resource constraints, Fog may not be able to ac-

commodate all latency-critical services in some particles. Hence, constructed particles, while satisfying resource constraints, may not be able to satisfy the deadline requirements after position updates. To mitigate the effect of such scenarios, *pBest* and *gBest* selection consider deadline satisfaction of the placement before comparing the fitness values.

5.4.4 Stage 2 - Reliability-aware Redundant Placement

Algorithm 11 RRP Algorithm

Input: TSP, $ToFog$, $ToCloud$ and Meta-data
Output: Microservices to devices mapping

Initialise population of N chromosomes using **AHI**
Calculate fitness using Eqs 5.16
calculateDominants($population$) using **RDS**
$fronts \leftarrow$ calculateFronts($population$)
$crowdingDist \leftarrow$ calculateCrowdingDistance($population, fronts$)
while $i \leq Iterations$ **do**
    $childChromosomes \leftarrow \{\}$ ▷ $2N$ chromosomes
    while $childChromosomes \leq N$ **do**
        $orderedParents \leftarrow$ order($populations, fronts, crowdingDist$)
        $parents \leftarrow$ tournamentSelect($orderedParents$)
        $children \leftarrow$crossover($parents$)
        $childChromosomes$.add($children$)
    mutate($childChromosomes$)
    Calculate fitness using Eqs 5.16
    $population \leftarrow population \cup childChromosomes$
    calculateDominants($population$) using **RDS**
    $fronts \leftarrow$ calculateFronts($population$)
    $crowdingDist \leftarrow$ calculateCrowdingDistance($population, fronts$)
    $ordered \leftarrow$ order($populations, fronts, crowdingDist$)
    $population \leftarrow$ get 1^{st} N chromosomes
    calculateDominants($population$) using **RDS**
    $fronts \leftarrow$ calculateFronts($population$)
    $crowdingDist \leftarrow$ calculateCrowdingDistance($population, fronts$)
return $population.best + TSP$

During this stage, the placement generated from TSP is used as the input to the Reliability-aware Redundant Placement (RRP) algorithm (see Algorithm 11) to create redundant microservice deployments to improve the reliability further. As the number

of redundant instances is not known prior to algorithm execution but decided based on the optimisation objectives, we propose an algorithm by improving NSGA-II. NSGA-II is a genetic algorithm for multi-objective optimisation where each placement can be depicted as a 2D chromosome. This representation enables the count of instances to be adjusted flexibly to reach a trade-off between reliability and cost. We make multiple improvements to adapt the NSGA-II algorithm to our specific placement problem as follows:

1. Availability-aware Heuristic Initialisation (**AHI**): This heuristic is used to populate the initial population in a reliability-aware manner (Algorithm 12) to achieve faster convergence by having a strong population as the starting point of the algorithm. To achieve this, AHI first calculates alternative Fog devices for each device based on how they complement each other from a reliability perspective (lines 3-9). We introduce a alternative device score ($Alt\_Score_{d_1,d_2}$) based on TTF improvement ($ttf_{ext}^{d_1,d_2}$) and availability improvement ($av_{ext}^{d_1,d_2}$) as follows:

$$Alt\_Score_{d_1,d_2} = ttf_{ext}^{d_1,d_2} + av_{ext}^{d_1,d_2} \tag{5.15a}$$

$$ttf_{ext}^{d_1,d_2} = \begin{cases} \frac{ttf_{d_1 \cup d_2} - ttf_{d_1}}{t2-t1} & \{d_1,d_2\} \not\subset g; \forall g \in G \\ 0 & \text{otherwise} \end{cases} \tag{5.15b}$$

$$av_{ext}^{d_1,d_2} = \begin{cases} \frac{\left[\int_{t1}^{t2} Av_{d_1 \cup d_2}(t)dt - \int_{t1}^{t2} Av_{d_1}(t)dt\right]}{t_{f,d_1}} & \{d_1,d_2\} \not\subset g; \forall g \in G \\ 0 & \text{otherwise} \end{cases} \tag{5.15c}$$

Eqs. 5.15 calculate the reliability improvement of deploying microservice instances on both d_1 and d_2 compared to deploying only on d_1, where t_{f,d_1} indicates the total failure duration of d_1 alone. To maintain the diversity among the generated chromosomes, results of the heuristic are made random by changing the order of considered mappings from TSP (line 13) and changing the order of the alternative devices (lines 19-20) to select the best alternative device out of a portion of the devices selected from D'.

2. Chromosome fitness and Reliability-aware Dominant Selection (**RDS**): We define the fitness of the chromosomes using 3 parameters including availability (Eq. 5.16a), TTF (Eq. 5.16b) and cost (Eq. 5.8) of the placement. Based on the problem formulation

Algorithm 12 AHI Algorithm

Input: Number of chromosomes (N) and Meta-data
Output: Initial population

$initPopulation \leftarrow \{\}$
▷***Calculate Per Device Alternatives***
$altDevices \leftarrow \{\}$ ▷***Alternative devices and scores per device***
for $d \in fogDevices$ **do**
  for $d' \in [fogDevices - \{d\}]$ **do**
    $altScore \leftarrow$ calculateAtlScore(d, d') ▷***Use Eqs. 5.15***
    if $altScore \neq 0$ **then** $altDevices$.add$(d, d', altScore)$
  Order $altDevices$.get(d) in descending fitness score
for $n \in N$ **do**
  $ordered \leftarrow (n \leq N/2)$?TRUE:FALSE
  $P \leftarrow$ fog layer placement from TSP (list of $\{m, d\}$)
  shuffle(P)
  for $(m, d) \in P$ **do**
    $D' \leftarrow altDevices$.get(d)
    if *ordered* is TRUE **then**
      $d' \leftarrow$ choose device with highest $altScore$ from first device of D' s.t resource constraints are met
    else
      shuffle(D')
      $d' \leftarrow$ choose device with highest $altScore$ from first x devices of D' s.t resource constraints are met
    if d' is null **then**
      $d' \leftarrow$ random device from $fogDevices$ s.t resource constraints are met
    $initPopulation$.getChromosome(n).place (m, d')
return $initPopulation$

in Section 5.3.7, the final fitness values are created as follows:

$$f_1 = \left[1 - \frac{\sum_{\forall S \in A_s} (Max[\rho_s - AV(S), 0])/\rho_s}{S^v_{num}}\right] DI(A_s) \tag{5.16a}$$

$$f_2 = \frac{\sum_{\forall S \in A_s} TTF(S)}{S_{num}.T} DI(A_s) \tag{5.16b}$$

where ρ_s indicates the reliability expectation of the service in terms of average uptime availability and S^v_{num} denotes the number of reliability expectation violated services. To maximise the reliability satisfaction while reducing the cost, we propose RDS (Algo-

rithm 13) for dominant selection where higher priority is given to satisfying ρ_s using f_1 (lines 1-4) and non-dominated sorting is used for f_2 and cost (lines 5-10).

Algorithm 13 RDS Algorithm

Input: Chromosomes C_i and C_j
Output: **TRUE** if C_i dominates C_j, **FALSE** otherwise

1: **if** $C_i.f_1 > C_j.f_1$ **then**
2: *dominates* $\leftarrow$ **TRUE**
3: **else if** $C_i.f_1 < C_j.f_1$ **then**
4: *dominates* $\leftarrow$ **FALSE**
5: **else**
6: **if** ($C_i.f_2 \geq C_j.f_2$ AND $C_i.cost \leq C_j.cost$) AND
7: ($C_i.f_2 > C_j.f_2$ OR $C_i.cost < C_j.cost$) **then**
8: *dominates* $\leftarrow$ **TRUE**
9: **else**
10: *dominates* $\leftarrow$ **FALSE**
return *dominates*

3. Generation of new population: RRP uses tournament selection, single-point crossover with random point selection and a custom mutation process to evolve the current population into the next. The mutation operator randomly selects between replica growth and replica removal. The device for replica growth is chosen by selecting a microservice placement and making a tournament selection on Alt_Score values of its alternative Fog devices. Resource constraints are validated afterwards, and chromosomes undergo a mending process in case of violation by moving microservice instances from resource-violated Fog devices.

Finally, RRP acquires the best chromosome of the final population by selecting the one with the highest weighted sum of the three objectives. To adjust the weighted sum as a maximisation objective, the cost is normalised using $(MaxCost - Cost)/(MaxCost - MinCost)$ for each chromosome. RRP combines the selected chromosome with TSP output and returns the final placement.

5.5 Performance Evaluation

5.5.1 Experimental Configurations

For the evaluations, we use iFogSim2 [8] simulated Fog environment. iFogSim2 provides support for modelling hierarchical fog-cloud architecture and microservice application architecture along with microservices-related functions such as horizontal scalability, load balancing and dynamic service discovery, which are essential in modelling and simulating our reliability-aware deployment scenario. Furthermore, the simulator is easily extendable to simulate failure scenarios of the Fog nodes.

We model the Fog environment according to the architecture presented in Section 5.3.2. Network parameters of the Fog environment include bandwidth and latency among different devices of the Fog architecture. We extract these values from previous studies on network performance of edge networks following novel communication technologies as follows: WLAN communication (150Mbps, 2ms) based on WiFi-6 [120] and 5G [121], LAN connections (1Gbps, 0.5ms) based on gigabit Ethernet technology [105], and fog-cloud connections with WAN (30ms, 100Mbps) [46]. Fog device resources are defined using three parameters: CPU (1500-3000 MIPS), RAM (2-8 GB) and storage (32-256 GB) [123, 124]. These values represent resource availability of heterogeneous Fog devices such as RaspberryPi, Dell PowerEdge, Jetson Nano, etc. The cost of the resources is modelled following the price model of AWS Fargate and extended to the Fog layer with an increase factor of 1.2-1.5 as proposed in [102].

Due to the novelty of the Fog computing paradigm, there's a lack of availability in Fog computing reliability data. Hence, following previous reliability studies in the area [138], we create synthetic failure traces based on real-world failure data available for distributed systems. In our work, we use the failure characteristics presented in [143], which analyses Google Cloud trace logs consisting of around 12,5000 servers monitored over 29 days. Failure characteristics of the Fog devices in our simulated environment are modelled based on the results of the empirical analysis done on the said data set and fed to our placement algorithms. Failure and repair events during the simulation time are also synthesized accordingly.

Workloads used in the performance evaluation are synthetically generated following

the microservices-based applications used in the literature [32, 38]. Workloads model multiple IoT applications such as smart health monitoring [106], smart parking [107], etc. and also follow general microservice composition patterns such as chained, aggregator, and hybrid patterns. Diversity among applications is ensured by varying microservice resource requirements in terms of CPU (300-900 MIPS), bandwidth (200-1500 bytes/packet), base request rate (100-200 requests/s) following previous IoT simulation benchmarks [8, 46].

5.5.2 RPM Algorithm Performance Evaluation

In this section, we evaluate RPM's ability to converge to a solution that can reach a trade-off between cost and reliability. To this end, we consider multiple design decisions made in our proposed algorithm (**RPM**) and evaluate their effect on the performance of the placement. For the comparison, the following variants of the algorithm are used,

1) **No_AFF**: In this approach, the fitness function of the **TSP** uses Eqs. 5.2a, 5.2b to calculate the availability, instead of the Availability Score proposed in **AFF**.

2) **No_AHI**: Creates random chromosomes for the initial population of RRP algorithm, without using Algorithm 12.

3) **No_RDS**: In this approach, reliability and cost have equal priority during dominant chromosome selection. Hence, generic non-dominated sorting is used instead of our proposed **RDS** approach.

4) No Cost-awareness (**No_CA**): Maximises reliability without having cost as a limiting factor for the redundant placement.

We carry out the experiments for 6 workloads covering both independent and correlated failures (see Table 5.2 and 5.3). The algorithm's search space depends on three main parameters: the number of composite services in the batch placement, the number of Fog devices eligible for placement and the time duration considered. We create the workloads to capture performance with variations in all three parameters. All variants use the same parameters for the algorithms: TSP with 100 particles, 300 iterations and RRP with 100 chromosomes, 300 iterations. Based on the results, we compare each approach with RPM to evaluate its ability to reach a better trade-off between reliability and

cost. To this end, we calculate the Trade-off Ratio of each approach with respect to the No_CA, where reliability degradation per unit cost reduction is calculated. Reliability Satisfaction, FTTF and Cost values are calculated with a confidence interval of 95%.

The aim of AFF is for the TSP (Stage 1) to produce a placement such that it is easier for the RRP (Stage 2) to find redundant placements that can improve the overall reliability of the final output. We can validate this by comparing No_AFF and RPM. Based on the results, it is evident that Reliability Satisfaction (R.S) and FTTF of No_AFF are lower than RPM for all considered workloads. Moreover, No_AFF does not provide sufficient cost advantage compared to RPM, which is further proven by the high trade-off ratio of the resultant placement. This shows that having AFF improves RRP's ability to find redundant placements that can easily enhance the reliability of the final placement while reducing the cost.

In RPM, we have introduced a heuristic to populate the initial population of RRP such that nodes selected for redundant placement try to complement the output from the TSP. The aim of introducing this method is to improve the convergence of the RRP by creating an initial population of better solutions. We verify this by comparing RPM with No_AHI, which randomly initialises the population. Results show that RPM can achieve higher reliability satisfaction and FTTF. The costs incurred by No_AHI vary depending on the scenarios showing slightly higher or lower cost values than RPM. However, No_AHI records a lower trade-off ratio demonstrating RPM's ability to reach a better trade-off between objectives.

In No_RDS, traditional non-dominated sorting gives equal priority to cost and reliability, which results in a lower cost at the expense of lower reliability (over 9% reliability violation for considered scenarios). Thus, for mission-critical services that usually expect availability higher than 99.99%, this approach fails to achieve a proper balance. With our proposed RDS approach, the placement algorithm handles multi-objective optimisation while giving the reliability aspect higher priority than cost. Considering these factors, RPM can reach a better trade-off between reliability and cost for services with high-reliability requirements.

Based on the above analysis, the introduced improvements (AFF, AHI and RDS) ensure RPM's ability to converge towards a placement with higher reliability while mini-

Table 5.2: Evaluation of different varients (under independent failures)

	Scenario1				**Scenario2**				**Scenario3**			
Approach	R.S (%)	FTTF (%)	Cost (0-1)	Trade Ratio	R.S (%)	FTTF (%)	Cost (0-1)	Trade Ratio	R.S (%)	FTTF (%)	Cost (0-1)	Trade Ratio
RPM	98.813 ±0.201	93.533 ±0.652	0.531 ±0.005	0.036	98.577 ±0.205	94.51 ±0.503	0.609 ±0.005	0.049	98.239 ±0.307	91.784 ±0.823	0.713 ±0.007	0.069
No_AFF	95.249 ±0.435	83.019 ±0.906	0.472 ±0.007	0.139	93.073 ±0.494	81.654 ±0.899	0.524 ±0.007	0.218	91.314 ±0.799	73.317 ±1.634	0.624 ±0.01	0.324
No_AHI	97.54 ±0.344	91.005 ±0.773	0.539 ±0.007	0.086	97.664 ±0.263	92.641 ±0.577	0.592 ±0.006	0.085	97.55 ±0.404	90.62 ±0.917	0.731 ±0.01	0.127
No_RDS	90.498 ±0.594	76.085 ±0.882	0.314 ±0.002	0.192	88.161 ±0.405	76.321 ±0.603	0.373 ±0.001	0.254	80.992 ±0.565	64.307 ±0.957	0.367 ±0.002	0.364
No_CA	99.753 ±0.134	98.808 ±0.314	0.795 ±0.009	N/A	99.637 ±0.114	98.624 ±0.29	0.825 ±0.008	N/A	99.328 ±0.161	96.16 ±0.568	0.871 ±0.008	N/A

Table 5.3: Evaluation of different varients (independent and correlated failures)

	Scenario4				**Scenario5**				**Scenario6**			
Approach	R.S (%)	FTTF (%)	Cost (0-1)	Trade Ratio	R.S (%)	FTTF (%)	Cost (0-1)	Trade Ratio	R.S (%)	FTTF (%)	Cost (0-1)	Trade Ratio
RPM	98.305 ±0.29	92.474 ±0.812	0.633 ±0.015	0.064	98.894 ±0.196	96.237 ±0.443	0.762 ±0.01	0.103	99.417 ±0.14	95.968 ±0.563	0.592 ±0.011	0.025
No_AFF	97.032 ±0.427	89.297 ±0.92	0.643 ±0.013	0.148	96.606 ±0.379	92.438 ±0.576	0.728 ±0.009	0.397	98.832 ±0.222	93.734 ±0.743	0.566 ±0.012	0.083
No_AHI	96.395 ±0.474	88.094 ±0.943	0.619 ±0.015	0.164	97.449 ±0.333	93.001 ±0.622	0.694 ±0.013	0.178	99.124 ±0.201	95.914 ±0.586	0.632 ±0.014	0.188
No_RDS	87.252 ±0.724	69.859 ±0.925	0.291 ±0.002	0.238	88.161 ±0.405	76.321 ±0.602	0.323 ±0.001	0.235	87.798 ±0.525	72.487 ±0.809	0.233 ±0.001	0.278
No_CA	99.376 ±0.178	96.753 ±0.584	0.801 ±0.01	N/A	99.225 ±0.158	97.268 ±0.39	0.794 ±0.008	N/A	99.576 ±0.122	97.337 ±0.464	0.656 ±0.011	N/A

mizing the deployment cost as a secondary objective. Thus, we use RPM in the following section to provide reliability-aware placements under different scenarios for further evaluation. However, the algorithms are designed flexibly to switch between these variations easily depending on the kind of trade-off required.

5.5.3 RPM Algorithm Placement Evaluation

In this section, we evaluate the efficiency of the placement generated by RPM under multiple aspects addressed by the algorithm: the effect of reliability-aware redundant placement, the impact of throughput-awareness, and finally, CCF consideration. To indicate the behaviour of the algorithms under different failure types, we start with independent failures in the first two experiments and add CCF to the final experiment to analyse the overall effect.

Effect of Redundant Placement - This section evaluates "proactive redundant placement" handled in stage 2 (RRP) of the hierarchical placement process. We compare our approach with multiple alternative approaches as follows:

1) **No_Red**: Does not consider the redundant placement of the microservices but tries to place the minimum required microservice instances to maximise the reliability of the placement using TSP.

2) **Even_Dist**: The placement method proposed in [32], where microservice instances are evenly replicated across the Fog resources while maximising Fog resource usage.

3) Reliability-aware Heuristic (**R_Heu)**: Uses the two heuristic approaches used in our placement policy to populate the initial populations of TSP and RRP algorithms. R_Heu represents an improved adaptation of primary-backup copy placement concept in [135] to our FSPP problem with load sharing.

For this evaluation, we use two workloads (WL1 with six composite services and 30 devices, WL2 with 12 composite services and 60 devices) and consider two time periods (20 days, 30 days). Such a selection of workloads covers all three parameters that affect the solution space. Figure 5.6 depicts the results of the different approaches. Results show that our policy is able to outperform other approaches in terms of reliability satisfaction while improving FTTF (up to 25% and 40% improvement in reliability satisfaction and FTTF, respectively). All the approaches except No_Red utilise independent scalability of the microservices to replicate them across Fog environments. Thus, No_Red records the lowest reliability at a lower cost. Due to redundant placements, R_Heu records improved reliability. However, being a heuristic approach, R_Heu lacks control over the number of redundant placements, which hinders it from achieving higher satisfaction compared to RPM. The reliability satisfaction of both of these

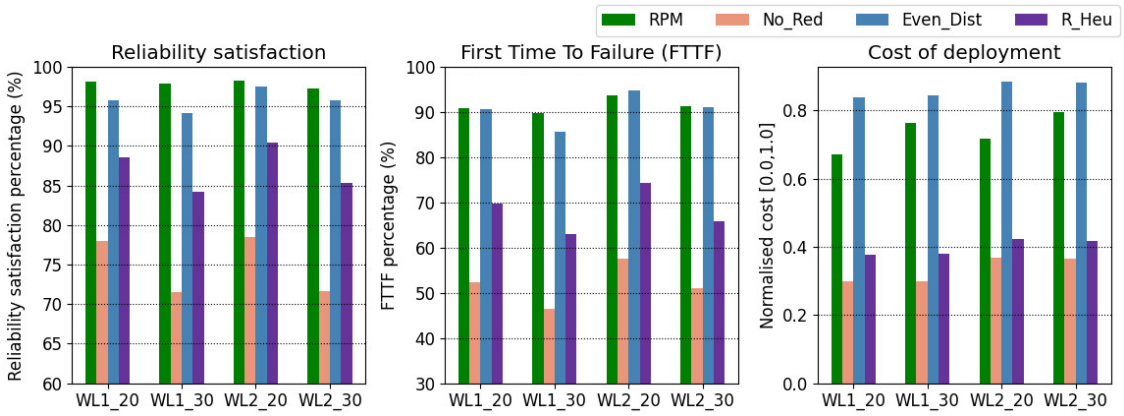


Figure 5.6: Evaluation of proactive redundant placement

approaches is unacceptable for mission-critical services with stringent reliability expectations. Even_Dist approach shows reliability metrics closer to RPM, especially in WL2 where the number of Fog devices is higher, allowing Even_Dist to deploy more replicas to ensure even distribution of instances. However, this approach incurs higher costs due to reliability-unaware replication and shows a higher reduction in reliability metrics as the considered time period increases. Although RPM incurs higher costs compared to No_Red and R_Heu due to higher flexibility in its replica placements, reliability and cost awareness of the algorithms allow it to reach higher reliability satisfaction (over 98%) while reducing the cost by more than 8% compared to Even_Dist which also makes use of independent scalability of microservices.

Throughput-aware Scalability of the Placement - In this section, we evaluate how throughput awareness, together with MSA, contributes to higher performance (see Figure 5.7). To this end, we use two workloads: a Uniform workload where all services have similar throughput requirements and a Varied workload having heterogeneous throughput requirements among services.

All considered approaches except Even_Dist incorporate throughput awareness into the placement. No_Red places the minimum required instances (k) to satisfy throughput requirements, whereas R_Heu deploys redundant microservice instances on top of that using the AHI algorithm. RPM formulates the problem as a k *out* n load balancing problem where n is determined robustly based on the failure characteristics of the environment. As a result, RPM reaches the highest reliability satisfaction in both sce-

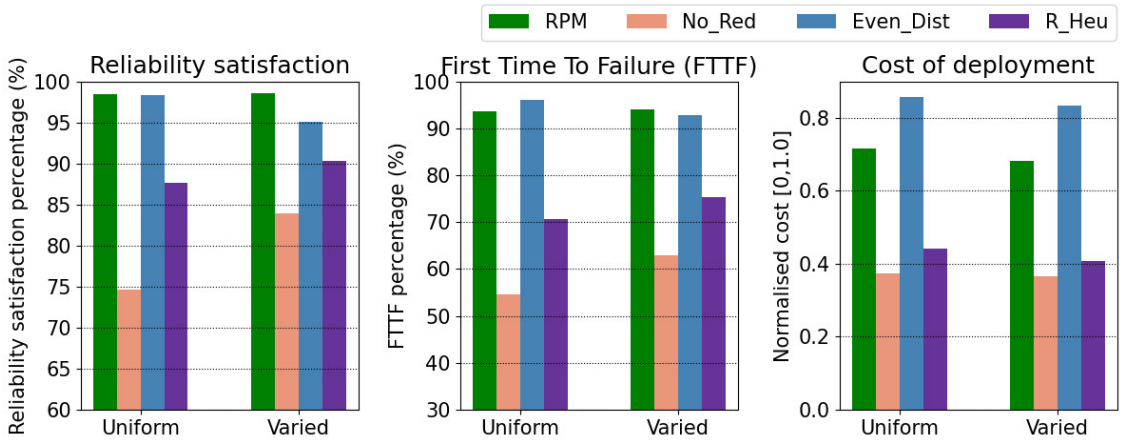


Figure 5.7: Evaluation of throughput-aware scalability

narios (around 98.5% in both), adapting well to the heterogeneous throughput needs. Although No_Redundancy and R_Heu have lower performance due to limitations in proactive redundant placement, they show an increase in reliability metrics in the Varied scenario. This is also a result of combining throughput and reliability awareness, where it's easier for these two approaches to ensure high reliability for low throughput services with less number of instances, which ultimately improves the average reliability compared to a uniform throughput scenario. Compared to the above three approaches, Even_Dist shows a considerable decline (98.4% in Uniform to 95.1% in Varied) in reliability metrics in the Varied scenario as this approach tries to replicate instances for all services evenly without prioritising the ones with higher throughput requirements. From the above results, it is evident that the incorporation of throughput awareness to proactive redundant placement decisions improves the robustness of the algorithm allowing proper utilisation of limited Fog resources to generate a scalable microservice placement (using both horizontal and vertical scalability).

Effect of CCF - In this section, we evaluate the effect of considering common cause failures along with independent device failures. To assess the robustness of the proposed fitness functions, two main categories of CCFGs are considered: a non-overlapping scenario where device groups can be isolated and overlapping scenarios where devices can belong to multiple CCFGs in an overlapping manner (see Figure 5.8). For these two scenarios, RPM is compared with CCF_Unaware variation of the RPM algorithm

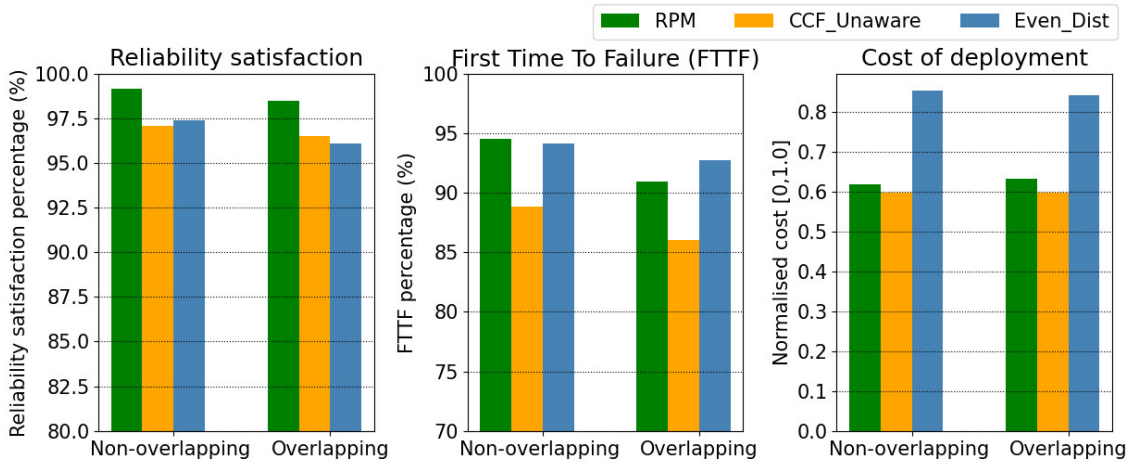


Figure 5.8: Evaluation of CCF effect

and Even_dist approach. In both scenarios, RPM is able to take the effect of CCFGs into consideration for the placement decisions and hence, records the highest reliability satisfaction (up to 2.5% improvement). Because of CCFGs, RPM spreads redundant microservice instances across CCFGs such that failures of such groups would be isolated. This results in a slight increase in cost compared to CCF_Unaware (up to 3.5%), but still able to achieve around 20% cost reduction compared to Even_dist.

Based on the experiments, it is evident that RPM provides a robust approach capable of delivering throughput-aware redundant placements under both independent and correlated failures of Fog environments, while achieving a balance between reliability and cost. Moreover, the proposed algorithm is capable of navigating solution spaces of different sizes successfully.

5.6 Summary

In this chapter, we proposed a reliability model for microservices-based IoT applications, considering their placement within resource-constrained and heterogeneous Fog devices where independent and correlated failures exist within the Fog environments. Accordingly, we proposed a proactive redundant placement policy that utilises the independently deployable and scalable nature of the microservices to support the high-

reliability requirements of the mission-critical IoT services in a throughput and cost aware manner. We implemented a hierarchical algorithm consisting of PSO and NSGA2-II algorithms and improved them with multiple approaches to improve the algorithm's convergence. Moreover, we evaluated our approach through extensive experiments under two main aspects: performance improvements of the algorithm compared with multiple alternative approaches and efficiency of the resultant placement compared to multiple benchmark placement policies. The obtained results show that our policy can successfully navigate different solution spaces and provide robust placements that can achieve high reliability (up to 25% improvement in reliability) considering independent and correlated failures.

In the next chapter, we investigate microservice placement within federated Fog environments and develop a software framework to enable scalable placement and dynamic composition of microservices within multi-fog multi-cloud environments.

Chapter 6

A Framework for Scalable Microservices Placement in Federated Fog Environments

The Federation of distributed Fog resource clusters and Cloud data centres is an effective solution to overcome the resource-constrained nature of Fog resources and to provide geo-distributed access to IoT services. Execution of placement policies for distributed microservice placement across Fog and Cloud resource clusters and their dynamic composition (service discovery and load balancing) to create composite services are the main challenges related to realising federated Fog computing. Thus, this chapter presents "MicroFog", a novel Fog computing framework designed and developed to support the placement and management of microservices-based applications within multi-fog multi-cloud environments. We design and implement a software framework that utilises the capabilities of cloud-native technologies (i.e., Kubernetes and Istio etc.) to facilitate distributed placement and composition of microservices-based applications across federated fog environments. We extend and integrate the distributed placement algorithm presented in Chapter 3 to evaluate the framework's features. Results demonstrate that MicroFog is a scalable, extensible and easy-to-configure framework that can be used to integrate and assess novel placement policies for deploying microservice-based applications within multi-fog multi-cloud environments. It enables horizontally scaled placement, service discovery and load balancing of microservices across federated environments, thus reducing the application service response time up to 54% compared to placements without horizontally scaling microservices across distributed resources.

This chapter is derived from:

- **Samodha Pallewatta**, Vassilis Kostakos, and Rajkumar Buyya, "MicroFog: A Framework for Scalable Placement of Microservices-based IoT Applications in Federated Fog Environments", *Journal of Systems and Software*, (revision, June 2023).

6.1 Introduction

The IoT is growing rapidly, and the ever-increasing number and variety of connected devices generate massive amounts of geo-distributed data to be processed using Fog resources. However, the resource-constrained nature of the Fog resources is the main drawback which limits realising the full potential Fog resources as the load on applications grows. This challenge can be overcome through the federation of geo-distributed Fog clusters and Cloud data centres. This includes cooperative use of distributed Fog computing cluster/ data centres and Cloud data centres for the placement of applications to satisfy their demands and meet QoS requirements [149]. Such an approach focuses on extending the hybrid Cloud to include Fog computing resources provided by multiple Fog Infrastructure Providers (FIP) and maintain seamless connectivity across different environments to achieve the best possible performance [150].

Furthermore, cloud-native characteristics of microservices make them perfect for such placement of large-scale IoT applications, which has given rise to novel paradigms like Osmotic Computing that proposes the convergence of IoT, MSA and Fog computing where microservices are dynamically moved and composed across hybrid fog-cloud environments [56]. To support such distributed and dynamic deployment, Microservices are containerised using technologies such as Docker and dynamically composed using container orchestration platforms like Kubernetes and service mesh technologies such as Istio, thus ensuring seamless connectivity among microservices deployed across distributed computing resources.

The development and integration of novel efficient placement algorithms are vital to harvesting the full potential of MSA in Fog computing environments. Existing literature contains works focusing on horizontally scaled placement of microservices to meet QoS parameters such as throughput, reliability and latency [32, 38, 40, 46], location-aware placements [73], etc. that place interconnected microservices across distributed resources. However, these algorithms require extensive and accurate evaluations and validations before applying them at the enterprise level [8]. Compared to Cloud computing, where Cloud resources can be acquired from commercial service providers like Amazon AWS, Google Cloud, etc., Fog computing lacks frameworks and platforms that

can be used for easy integration and evaluation of novel placement policies. Although several real-world frameworks are available to manage Fog resources [94, 151], they have limitations related to Microservices-based IoT application placement. They lack support for the dynamic composition of microservices across federated Fog and Cloud data centres, easy integration of distributed placement policies, compatibility with open-source cloud-native technologies, support for heterogeneous microservices-based applications, ease of setup and prototyping support, etc. To overcome these limitations, we propose MicroFog: an easily configurable software framework for microservice-based application placement within federated fog-cloud environments. MicroFog can be used by IoT application developers, Fog infrastructure providers, and researchers in Fog computing to create, integrate and evaluate novel placement policies to deploy and manage microservices-based IoT applications. MicroFog enables the users to create placement approaches that harvest the potential of MSA, thus improving the QoS of applications.

MicroFog provides a configurable control engine that executes placement policies in a distributed or centralised manner and deploys containerised microservices within Kubernetes and Istio-managed Fog and Cloud resource clusters. MicroFog abstracts Kubernetes and Istio resource deployment (i.e., pods, services, virtual services, gateways, etc.) while providing support for integrating novel placement algorithms and load-balancing policies. Moreover, MicroFog ensures the dynamic composition of microservices distributed across geo-distributed multi-fog multi-cloud environments by enabling service discovery and load balancing.

The major contributions of our work are as follows:

- A scalable and extensible framework is proposed for deploying and managing microservices-based IoT applications within the federated Fog and Cloud environments. The framework consists of multiple components, including a Control Engine (MicroFog-CE) for placement algorithms execution and application deployment, data stores to store required metadata, a monitoring component and a logging component.

- MicroFog-CE is designed and developed as an easy-to-configure microservice supporting different operation modes (centralised vs distributed), application place-

ment modes (periodic vs event-driven), integration of novel placement policies, load balancing policies, etc.

- Deployment architectures are proposed for the major components of the MicroFog framework to ensure their scalable and fault-tolerant deployment across federate Fog and Cloud environments.
- A proof-of-concept prototype of the framework is created, and the main features of the framework are demonstrated and evaluated using multiple use cases and benchmark policies integrated with the control engine.

The rest of the chapter is organised as follows. In Section 6.2, we provide a comprehensive background on microservices-based application placement, derive requirements of the framework based on that and analyse related research. Section 6.3 introduces the MicroFog framework, and Section 6.4 details the deployment architectures for the main components of the framework. APIs to access MicroFog-CE are presented in Section 6.5. Features of the framework are evaluated in Section 6.6. Finally, Section 6.7 concludes the chapter.

6.2 Background and Related works

In this section, we present a comprehensive background on the Fog computing paradigm, microservices-based applications, their deployment-related aspects and the Fog application placement problem to derive requirements of the frameworks for scalable Placement of Microservices-based IoT Applications within Federated Fog Environments. Moreover, we provide a qualitative comparison of existing frameworks to highlight the capabilities of our proposed framework.

6.2.1 Fog Computing

Fog computing introduces an intermediate layer between IoT devices and the Cloud, consisting of distributed, heterogeneous and resource-constrained resources compared to Cloud data centres [11]. With the rapid growth in IoT applications, Fog computing

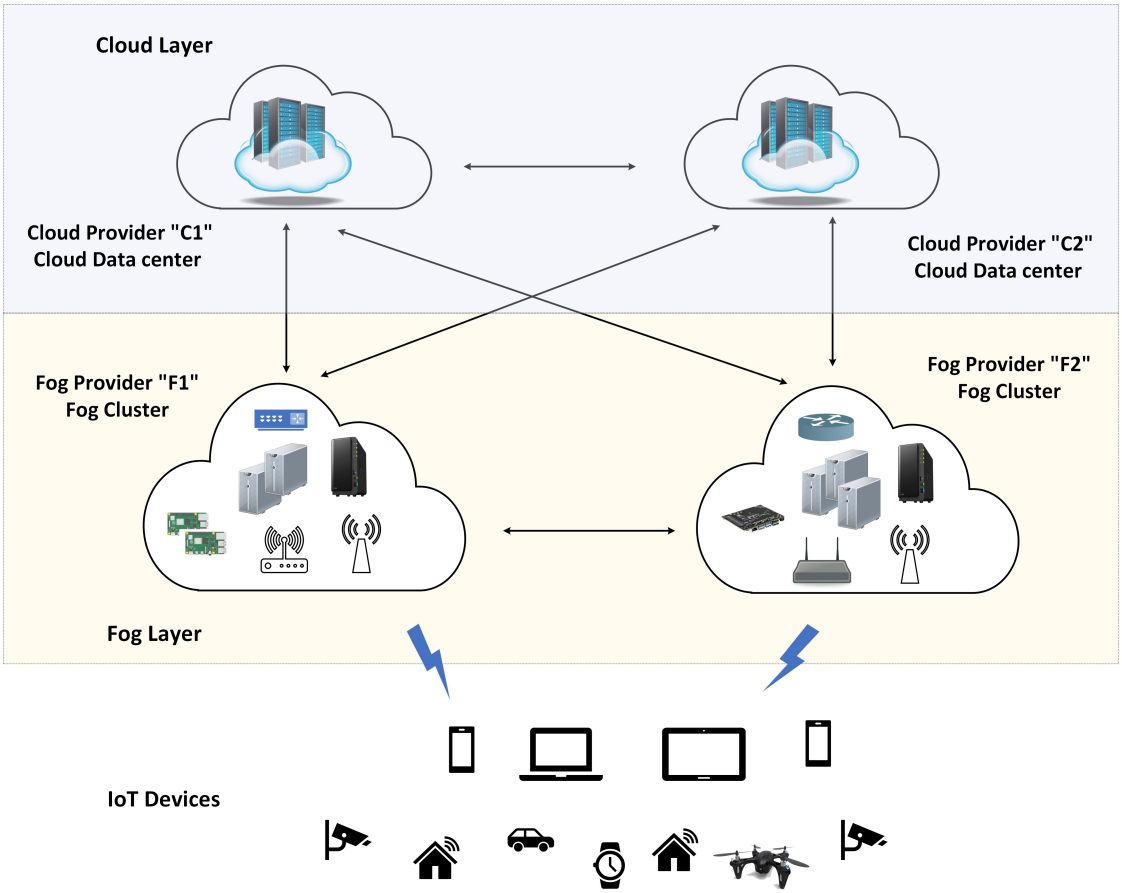


Figure 6.1: Federated multi-fog and multi-cloud architecture

is evolving towards a federated multi-fog multi-cloud architecture [150] where multiple Fog Providers provide infrastructure, including computing, storage and networking resources within the Fog layer. This helps to overcome the resource-constrained natures of the Fog devices, enables ubiquitous access, and supports location-aware placement of applications. In this work, we consider the existence of multiple such Fog clusters provided by various service providers where they maintain connectivity with neighbouring clusters and the Cloud (see Figure 6.1).

6.2.2 Microservices-based Applications

MSA decomposes an application into a set of independently deployable modules known as microservices designed around business logic to have well-defined business bound-

aries [22]. Microservices communicate with each other using lightweight APIs to create composite services that the end users access.

The loosely coupled nature of these microservices enables them to be deployed and scaled independently within distributed environments. Thus, dynamic service discovery and load-balancing mechanisms ensure seamless connectivity among microservices. To achieve such cloud-native behaviour, microservices are packaged as containers (i.e., Docker) that can be scaled (up and down) rapidly to meet the request demand. With such technologies, MSA can deploy microservices across distributed multi-fog multi-cloud environments while maintaining seamless connectivity and dynamic load balancing among horizontally scaled instances.

Modelling Microservice Application

As microservices-based applications have interactions among microservices, they can be modelled using Directed Acyclic Graphs (DAGs) [46] where the vertices of the DAG represent microservices ($m \in M_a$ where M_a is the set of microservices of application a). Directed edges in DAG represent microservice invocations such that the direction is from the client microservice (consumer) to the invoked microservice (consumed). Microservices are independently packaged and have heterogeneous resource requirements that can be defined in terms of required RAM, CPU, storage, etc., needed to satisfy a specific request rate/throughput. Due to the fine-grained nature of the microservices, they communicate to create composite services where each application provides multiple services (S_a: the set of services of application a) with heterogeneous QoS requirements that can be defined at the service level. As microservices can have complex interaction patterns to create composite services (i.e., chained, aggregator, hybrid), the dataflows among microservices can be uni-directional or bi-directional (df^a: set of dataflows among $m \in M_a$). Thus, each application can be denoted as a tuple of $< M_a, df^a, S_a >$ where each service $s \in S_a$ is depicted by a tuple containing its microservices, data paths within them and QoS requirements of the service; $< M_a^s, P_a^s, Req_a$¿. Data paths are collections of dataflows within a composite service that can be used to calculate the makespan of the service. It depends on the interaction pattern of the mi-

croservices within the composite service (i.e., the chained pattern has a single data path, whereas the aggregator invokes multiple datapaths).

6.2.3 Application Deployment Related Aspects

Microservices-based application deployment and management are aided by three cloud-native technologies: containerisation platforms (i.e., Docker), container orchestration systems (i.e., Kubernetes, Docker Swarm) and service mesh platforms (i.e., Istio, Consul). The MicroFog framework proposed in this work uses Docker, Kubernetes and Istio for the deployment and management of the microservices. Hence, we describe each technology and its aspects related to the federated fog-cloud deployment of applications as follows:

6.2.4 Containerisation using Docker

Microservices are packaged as Docker containers to make them independent of the host environments. Moreover, compared to earlier used virtual machines, containers are light-weigh with less startup time. Thus, containerisation of the microservices suits distributed deployment and scaling across heterogeneous and resource-constrained Fog nodes. Docker container images are stored and distributed using a container registry. Docker provides a fully managed container repository known as DockerHub. However, this is a centralised repository with limitations in privacy and security. Pulling images from a centralised repository can incur extra latency during microservice deployment in Fog environments. Thus, for Fog computing, it's important to explore distributed container image registries, depending on the resource availability of the Fog infrastructure to host the registry.

Kubernetes as Container Orchestration Platform

Decomposition of an application according to microservices architecture results in a large number of microservices and an even more significant number of containers due to horizontally scaled deployment of microservice instances to meet throughput demand,

redundant placement of microservice instances to ensure reliability, distributed placement across Fog cluster to support location-awareness, etc. Thus, a container management platform such as Kubernetes is required to manage the life cycle of thousands of containers. As one of the most popular open-source container orchestrators, Kubernetes is rapidly improved for use within heterogeneous computing environments through distributions like k3s which is a minimal Kubernetes distribution for extreme edge (i.e., resource-constrained IoT devices, Raspberry Pis, etc.). Thus, the use of k8s and k3s across multi-fog multi-cloud environments is exceeding explored by Cloud providers and Telco providers in their efforts to extend cloud-like services towards network edge [152–154]. Thus, we summarise the basic concepts used in Kubernetes. To deploy containers at a scale and to maintain communication among microservice containers, Kubernetes provides build-in **"resources"** (i.e., Pods, Service, etc.) that provide abstractions for underlying management operations. We discuss some of the most used resources in our framework below.

- Pod: A Pod is the smallest deployable unit supported by Kubernetes, where each pod can contain one or more containers (containers co-located with its sidecar containers). A pod represents a logical host where all co-located containers of the pod share the network resources and communicate through localhost. Pods provide fine-grained control over microservice instance deployment by enabling the deployment of pods on specific nodes by adding node selection constraints (i.e., node selectors, node name, etc.) to the pod.

- Service: Kubernetes service is an abstraction over a set of pods within a Kubernetes cluster that provides discovery and load balancing to those pods, thus allowing pods to get dynamically created and destroyed. Although in-cluster service discovery is handled through services, multi-cluster service discovery is not possible with Kubernetes alone.

- Namespace: Namespaces isolate name-spaced Kubernetes objects (i.e., pods, services, etc.), thus providing a way to isolate resources within multi-tenant Kubernetes clusters.

- ConfigMaps: ConfigMaps stores configurations as key-value pairs, thus separating configurations from the pods. This improves the flexibility and portability of containerised microservices.
- Secrets: Secrets are similar to ConfigMaps, but are designed to hold sensitive information that should not be stored within the application code.
- Roles and Rolebindings: They grant role-based access to Kubernetes resources (i.e., nodes, pods, configmaps, etc.)

Istio as Service Mesh

While Kubernetes provides basic functionalities required for container orchestration, it has limitations related to service discovery, load balancing, observability, fault tolerance and security management of the microservice applications. Thus, the service mesh is introduced as a software abstraction layer on top of Kubernetes to overcome these limitations. To this end, Istio implements multiple Custom Resource Definitions (CRDs) extending Kubernetes resource definitions as follows:

- Virtual Service (VS): Virtual Services provide more control over traffic routing by providing a way to define traffic routing rules to pods exposed through Kubernetes services.
- Destination Rules (DR): Once virtual service routing rules are applied, and the traffic is routed to the destination, Destination Rules are applied to perform load balancing, direct traffic towards service subsets, etc.
- Gateway: Gateway is an abstraction for a load-balancer for ingress and egress traffic of the cluster. Furthermore, to support inter-cluster traffic among Kubernetes clusters spread across different networks, Istio provides a specialised gateway known as the east-west gateway.

Kubernetes and Istio provide HTTP REST APIs to retrieve, create, update, and delete the above resources. Moreover, client libraries (i.e., Fabric8, client-go, etc.) are available for accessing these APIs through programming languages.

Example Application Deployment

In this section, we demonstrate the use of Kubernetes and Istio resources to deploy a microservices-based IoT application within Kubernetes and Istio available clusters. We use a Smart Health Monitoring Application (see Figure 6.2) [46] as a use case. The application consists of three microservices and two composite services accessed by the users: a latency-sensitive emergency event detection service ($S1$) where both its microservices ($m1, m1$) are placed in distributed Fog resources, a latency-tolerant predictive health warning service consisting two microservices ($m1, m3$). $m1$ is shared between both services and placed within the Fog layer to meet stringent latency requirements of service $S1$, whereas $m3$ is deployed within the Cloud.

Figure 6.2 demonstrates a logical view of how Kubernetes and Istio resources route external traffic from users to $m1$ and $m3$ and maintain communication between interconnected microservices (between $m1$ and $m2$, between $m1$ and $m3$). With the use of Istio, the ingress traffic received at the IP and port of the Istio ingress gateway are routed toward the desired pods based on the "host" header of the request. In Istio, the "host" value acts as the address of each set of pods exposed through Kubernetes services. Istio gateway, Virtual Service and Destination Rules are configured accordingly to enable proper traffic routing. Internal traffic among communicating microservices of the application is also routed by Virtual Services and Destination Rules based on "host" value. Moreover, these Istio resources together with Kubernetes services decouple service endpoint from the IP addresses of the individual pods, so that the pods can be dynamically placed and migrated to different nodes within and across clusters.

Kubernetes + Istio Multi cluster support

Istio supports deploying a single mesh to span multiple Kubernetes clusters, thus enabling cross-cluster service discovery and load balancing. The Istio deployment model for multi-cluster scenarios depends on the nature of the underlying network model. The simplest network model considers multiple clusters belonging to a single network where all nodes are fully connected through technologies like VPN. However, large-scale production systems that span multiple Kubernetes clusters belong to multiple net-

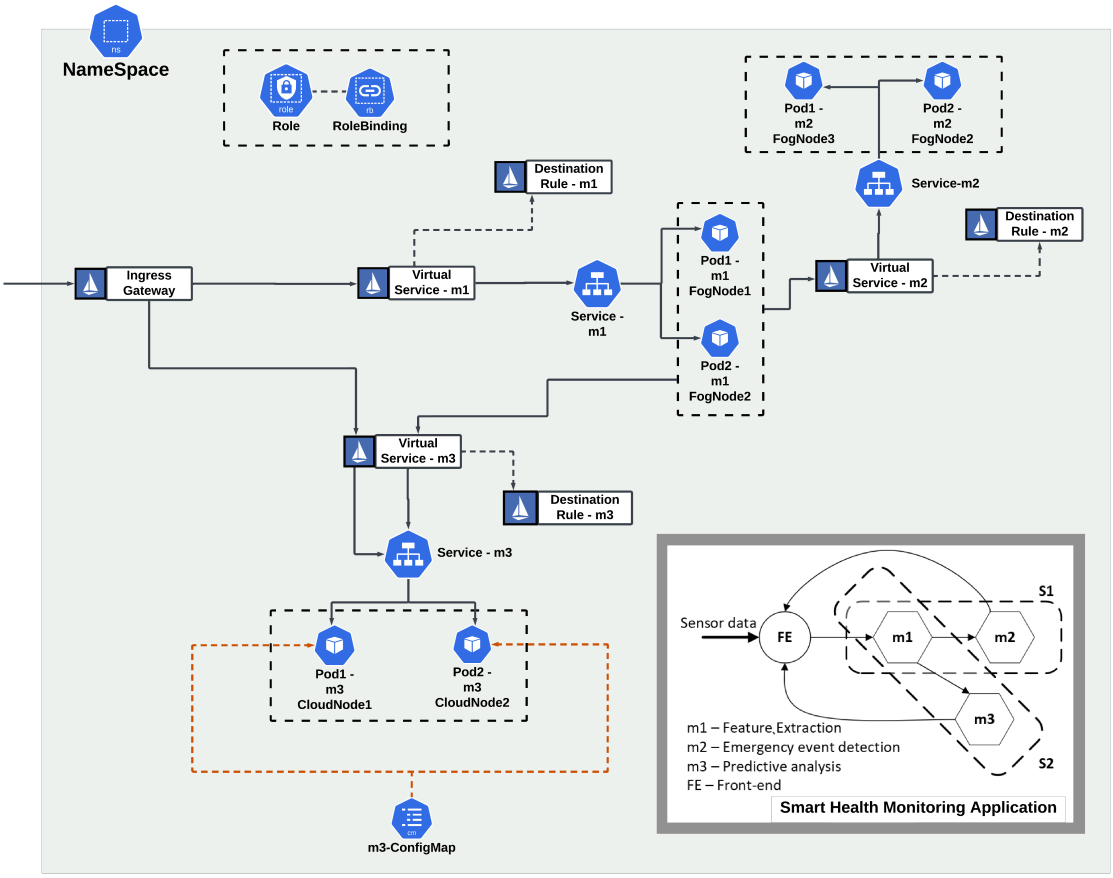


Figure 6.2: Example deployment of a smart health monitoring application

works with administrative boundaries where each cluster is exposed through load balancers. Fog computing architecture considered in this work (Section 6.2.1) maps to a multi-network model. Hence, in this work, we consider Istio multi-network deployment with multiple control planes to improve the resilience of the deployment. In this deployment mode, each Istio control plane connects to the API server of the connected clusters for service discovery across clusters.

Istio introduces an east-west gateway to expose the services within the cluster to other clusters to enable cross-cluster service discovery. Moreover, to ensure successful DNS lookup across clusters, consumer clusters need to have access to the Kubernetes Service resource, Istion DR and VS of the consumed service deployed in other clusters. As an example, for $S1$ an example application, for $m1$ to route traffic from its Fog cluster

to $m2$ deployed within a Cloud cluster, the above resources related to $m3$ should be deployed within both Fog and Cloud clusters.

6.2.5 Placement Problem

Microservice-based IoT application placement problem within Fog environments addresses deployment and maintenance of microservices within federated Fog and Cloud environemnts to meet the SLAs of the application services [32, 33].

Due to the flexibility provided by the microservices architecture, placement algorithms aim to incorporate horizontal scalability to meet throughput requirements [32, 38, 46], location-aware distribution [73], redundant placement to improve reliability [48], balanced placement across Fog clusters and Cloud depending on service discovery capabilities [41, 46], optimum load balancing and routing [43], etc. to efficiently utilise limited Fog resources while satisfying QoS parameters such as makespan, budget, reliability, availability, and throughput.

Execution of placement algorithms can take place as batch placements [46, 62] that process multiple application placement requests at once or sequential placements [36, 41] where queued placement requests are processed one after the other. Moreover, the placement policies can be developed as centralised [155] or distributed [60] algorithms to achieve placement across distributed Fog and Cloud resources provided by multiple infrastructure providers.

6.2.6 Framework Requirements

Based on the background, we summarise the functional and non-functional requirements of a framework for scalable placement of microservices-based IoT applications within federated Fog and Cloud computing environments, as follows:

- Multi-fog Multi-cloud microservice placement and deployment: Framework should support execution of placement algorithm across multiple Fog and Cloud clusters using either centralised or distributed operation modes. Accordingly, application microservices need to be deployed by using relevant Kubernetes and Istio

resources.

- Seamless microservice composition across hybrid environments: Kubernetes and Istio resource deployment should ensure cross-cluster service discovery and load balancing.
- Ability to integrate novel placement algorithms and load balancing policies easily.
- Support for heterogeneous cloud-native application deployment without any application-level changes.
- Compatibility with cloud-native technologies so that the framework can improve and evolve as the underlying technologies evolve (extensibility).
- A configurable control engine to support different operation modes like centralised or distributed operation, application placement modes such as event-driven or periodic placement request processing and batch or sequential placement request processing.
- Distributed storage solutions to store the data required for application placement and deployment (i.e., application models, Kubernetes and Istio resource definitions).
- Rapid prototyping support to enable evaluations of placement algorithms during their rapid design and development cycles.
- Framework should be flexible and scalable such that it can be deployed to operate across distributed Fog and Cloud clusters.

6.2.7 Existing Fog Frameworks

In this section, we compare existing Fog frameworks qualitatively based on the requirements identified in the previous section (see Table 6.1).

Yousefpour et al. [105] present a FogPlan, a framework for dynamic provisioning containerised Fog services using container orchestration platforms such as Kubernetes or OpenStack. FogPlan consists of a centralised Fog Service Controller responsible for

Table 6.1: Comparison of existing frameworks

Work	Architecture		Cloud-native Application Support					Microservice Composition Support					Control-engine			
Work	Integration	Multi-cluster	μservices	Containers	Container Orchestration	Service Mesh	Automated Deployment	Service Discovery		Load Balancing			Extensibility	Scalability	Configurability	Data Stores
								Avail.	cross-cluster	Avail.	configurable	Cross-cluster				
[105]	Fog, Cloud	-	✓	✓	-	-	∂	∂	-		-	-	✓	∂	∂	Centralised
[94]	Fog, Cloud	-	✓	✓	✓	-	∂	✓(Kubernetes)	-	✓(Kubernetes)	-	-	✓	∂	∂	Distributed
[78]	Fog, Cloud	-	✓	✓	✓	-	∂	✓(Kubernetes)	-	✓(Kubernetes)	-	-	✓	∂	∂	-
[156]	Edge	-	✓	✓	✓	-	-	✓	-	✓	-	-	✓	∂	∂	-
[157]	Fog,Cloud	-	✓	✓	✓	-	∂	✓(Docker Swarm)	-	-	-	-	∂	∂	∂	-
[158]	Fog, Cloud	-	-	-	-	-	∂	-	-	-	-	-	∂	∂	∂	Distributed
[151, 159]	Fog, Cloud	-	∂	✓	∂	-	∂	∂(*ProxyServer*)	-	-	-	-	∂	∂	∂	Distributed
[160]	Fog, Cloud	-	✓	✓	-	-	∂	∂	-	-	-	-	✓	∂	∂	Centralised
Our	Fog,Cloud	✓	✓	✓	✓	✓	✓	✓ (Kubernetes, Istio)	✓	✓ (Istio)	✓	✓	✓	✓	✓	Distributed, Replicated Fault-tolerant

✓: Supported by the framework, ∂: Partially supported

hosting the data stores, provisioning Fog services and deploying them within Fog nodes. Santoro et al. [94] provide an open-source technology-based (i.e., OpenStack, Kubernetes, Docker) platform named Foggy for workload placement in Fog computing environments. FogAtlas [78] extends Foggy platform by extending Kubernetes to orchestrate distributed Fog and Cloud resources in a user-friendly manner. Ermolenko et al. [156] also propose a framework based on Kubernetes and Docker where a Kubernetes cluster is deployed within a Mobile Edge Computing (MEC) environment. Bellavista et al. [157] create a microservice deployment framework based on Docker and Docker Swarm with a centralised control engine deployed in the Cloud to execute placement algorithms and deploy microservices accordingly. While they utilise Kubernetes and Docker Swarm features for container orchestration, they also have limitations in multi-cluster support, advanced microservice composition with service mesh technologies, and scalability of the control engine across multi-fog multi-cloud environments. Tuli et al. [158] introduce FogBus framework to harness edge/Fog and remote Cloud resources for the placement of applications developed as a collection of inter-connected modules. Deng et al. [151] propose FogBus2, a resource management framework for the deployment of containerised applications across edge and Cloud resources that are interconnected to each other using a VPN network. Wang et al. [159] improve FogBus2 and integrated container orchestration capabilities to the framework using Kubernetes. Their framework supports the integration of novel placement policies and their performance monitoring to evaluate novel placement policies. However, their framework lacks support for multi-cluster scenarios with multiple geo-distributed Kubernetes clusters. Moreover, they lack sup-

port for the dynamic composition of microservices due to limitation in service discovery and load balancing aspects and does not integrate service mesh technologies to fully leverage the capabilities of microservices architecture. Kubernetes resource usage in FogBus2 is limited only to Pods, which limits the framework's scalability. Furthermore, application-level changes are required for the containerised application modules to be deployed within the framework. Mahmud et al. [160] propose a fully distributed and scalable framework named Con-Pi to execute microservices-based applications. Con-Pi provides a centralised controller to execute integrated customised placement policies and deploy containerised microservices accordingly. However, Con-Pi does not provide advanced microservice composition, dynamic service discovery and load balancing for the deployed microservices and does not consider application deployment across multiple Fog resource clusters.

Based on the qualitative analysis provided in Table 6.1, existing frameworks have limitations in multiple requirements identified in Section 6.2.6 such as multi-fog multi-cloud placement and fully-automated deployment of applications, ensuring cross-cluster dynamic composition of microservices through container orchestrators and service mesh technologies, improving extensible of the framework through open-source technologies, scalability of the framework across highly distributed Fog environments, configurability to support different operation and placement modes, and distributed management of data required for application placement and deployment. Thus, this work introduces a novel framework for microservices-based application placement within federated Fog environments that satisfy the above requirements.

6.3 MicroFog Framework

In this section, we discuss the high-level architecture of the proposed MicroFog framework, its main components and workflow to highlight how MicroFog meets the requirements identified in Section 6.2.6.

6.3.1 High-level Architecture

Figure 6.3 presents the high-level architecture and the workflow of MicroFog. MicroFog provides a scalable and extensible Control Engine (CE) to execute placement algorithms and deploy IoT applications within Istio-installed Kubernetes clusters. CE communicates with three data stores: 1. YAML File Store containing YAML definitions (both Kubernetes and Istio) required for deployment of applications, 2. Meta Data Store for storing application models and links to related deployment resources stored within the YAML File Store, and 3. Docker registry hosting docker images for the application microservices. Application providers can submit Placement Requests (PRs) to the MicroFog-CE, defining the application for deployment and QoS requirements. CE receives application placement requests (PRs), processes them according to a selected placement policy (either an inbuilt placement algorithm or external algorithm accessed through an API), configures related Kubernetes and Istio YAML files according to the generated placement and the load balancing policy, and finally deploys them within Fog and Cloud resources using Kubernetes API. Furthermore, MicroFog integrates monitoring and logging tools to observe the performance of the MicroFog framework and applications deployed using it.

6.3.2 Main Components and Technologies

Control Engine (CE)

CE is designed to abstract microservices placement (execution of placement algorithms and deployment) and cross-cutting function handling (i.e., service discovery, load balancing) for the dynamic composition of microservices across multi-fog multi-cloud environments.

We implement CE as an independently deployable and scalable microservice developed using Quarkus [1], a novel Kubernetes-native lightweight Java framework designed to build cloud-native microservices. Quarkus reduces memory usage and improves deployment density [161], which is suitable for developing microservices for de-

[1] https://quarkus.io/

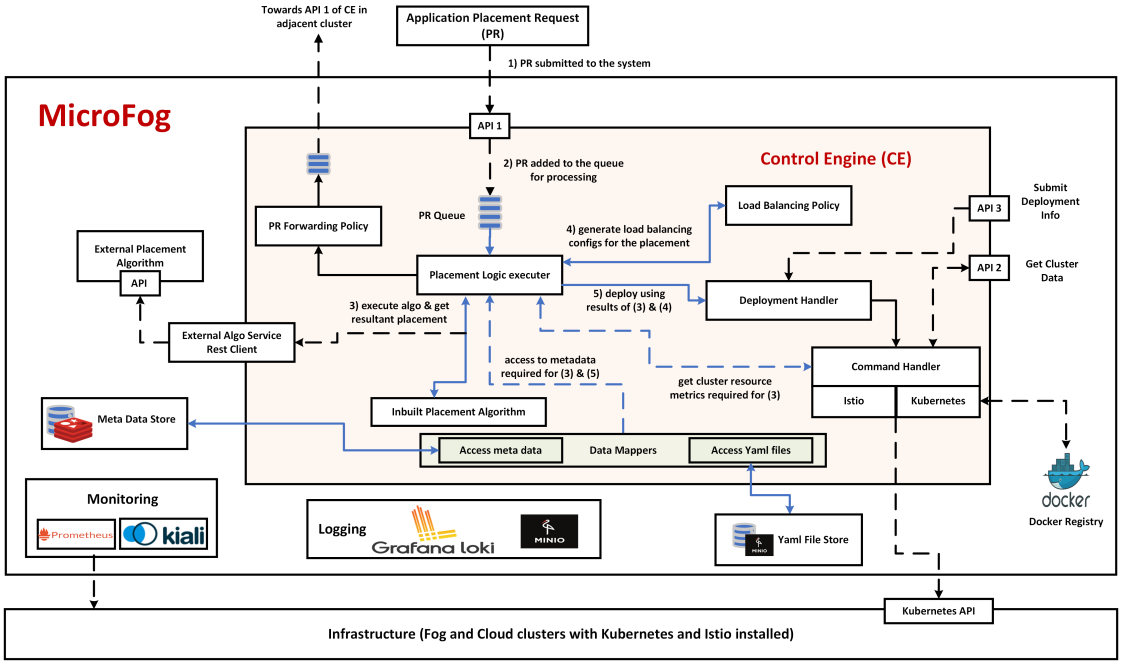


Figure 6.3: MicroFog: High-level architecture

ployment within resource-constrained Fog environments. As Quarkus is a Kubernetes-native framework, the development and deployment of the CE become straightforward and less time-consuming, thus allowing users to rapidly improve, extend and customise it with evolving needs. Thus, Quarkus is rapidly becoming popular as a lightweight Java framework for creating cloud-native microservices. Moreover, Quarkus allows easy access to Fabric8 Kubernetes and Istio clients [2] through its extensions. Fabric8 is a highly popular Kubernetes and Istio client that provides complete access to Kubernetes API. Fabric8 consists of a rich DSL (Domain Specific Language) for interacting with Kubernetes API, hence making it one of the most used open-source Kubernetes clients with an extremely active community using and continuously improving it. Thus, we have selected Qaurkus together with Fabric8 Kubernetes and Istio clients to create our controller.

We discuss the functional and non-functional features of the MicroFog-CE as follows:

1. *PR submission for placement:* Application providers can submit their PRs to the CE

[2] https://github.com/fabric8io/kubernetes-client

through an API which expects HTTP POST requests with the PRs represented in JSON format (API 1 shown in Figure 6.3). Each submitted PR can define multiple data fields related to the application, including application id, QoS parameters, any restrictions for application placement, and traffic entry clusters. Once submitted, CE uses such information to process the PR (i.e., the application id is the key to retrieving the application model and deployment resources from the data store, and entry clusters denote the clusters that act as the entry point for the ingress traffic for the considered application) and deploy the application microservices and deployment resources accordingly.

2. *Multiple operation and placement modes:* CE supports Centralised and Distributed operation modes. In centralised mode, a primary CE (i.e., deployed within the Cloud) with a global view of the infrastructure (i.e., Fog, Cloud clusters, their topology and resource availability) is responsible for executing the placement algorithm. In this mode, the primary CE queries the secondary CEs (through API 2) to gain information regarding the resources available within each cluster and their topology-related data (i.e., directly reachable Fog and Cloud clusters from each cluster) to construct the global view of the federated environment. Primary CE uses this information to generate placements for the applications requested by the PRs and send the output placement details to each relevant cluster (through API 3). The secondary CEs deployed within each cluster process the placement output and deploy Kubernetes and Istio resources accordingly. In contrast, in the distributed mode, all CEs are responsible for running the placement algorithm locally per cluster. They collaborate by forwarding the PRs among the clusters for distributed placement across multi-fog multi-cloud environments. MicroFog-CEs use API 1 for PR forwarding among clusters as well.

   Furthermore, the CE supports two placement modes: Periodic Placement and Event-driven Placement. Periodic placement invokes the placement algorithm periodically based on a configurable time period. Under this mode, the placement algorithms can be designed to process the PRs either as a batch (all PRs in the queue are processed simultaneously by the algorithm) or sequentially (either in

First-In-First-Out order or prioritised). In the event-driven mode, the placement algorithm is invoked upon receiving a new PR.

3. *Placement Algorithm Integration:* CE supports easy integration of novel placement algorithms. This can be done using two methods: in-built algorithm implementation where novel placement policies can be implemented by extending *PlacementAlgorithm.java* base class of the CE. The base class is initialised with the metadata required by the placement algorithms (i.e., resource availability of the devices, application model and topological information). Novel placement algorithms can extend this to implement customised placement logic that utilises the metadata to produce placement output (denoted by PlacementOutput.java) consisting of microservice-to-device mapping and PR completion data (completed PRs vs incomplete PRs that should go through a forwarding process to other clusters for placement completion). Moreover, CE provides capability to integrate external placement algorithms, which allows algorithms to be implemented in other programming languages (i.e., Python for placement algorithms that use Machine Learning). Such algorithms can be implemented as a separate microservice and integrate it to the MicroFog-CE by implementing an API that can be called by the *External Algo Service Rest Client in Figure 6.3* of the CE through an HTTP GET request. CE rest client is designed to send the metadata along with the GET request so that the external placement algorithm can generate the placement and return the deployment-related information back to the CE.

   By default, MicroFog-CE implements a Latency-aware Scalable Placement Policy proposed in [60]. The above algorithm aims to place microservices of latency-critical service as close as possible to the users who access them. We implement this algorithm in both distributed and centralised modes. We also implement it with and without horizontal scalability of the microservices to demonstrate the performance improvement MSA can provide within resource-limited Fog environments.

4. *Access Infrastructure Metrics:* To make placement decisions, placement algorithms require metrics related to infrastructure, such as resource availability within the

cluster. To this end, the current version of CE provides two measurements: 1. CE access Kubernetes Metric Server to obtain node metrics of current CPU and RAM usage, 2. CE also provides current resource allocation of the deployed pods by querying the Kubernetes API. Placement algorithms can utilise both types of metric information to make placement decisions. Metric collection can be further extended to use Prometheus as well to utilise time-series metric data for placement decision making.

5. *Load Balancing Policy Integration:* Due to the independently deployable and scalable nature of the microservices, load balancing plays a vital role in properly distributing the load across horizontally scaled microservices deployed across federated Fog and Cloud environments. By default, Istio use a round-robin load balancing method to route the requests. Moreover, Istio supports other load balancing methods like random, least request and weighted load balancing, which are already implemented in Envoy Proxy used by Istio for service discovery and load balancing purposes. They can be configured by updating the Istio DRs related to each microservice. In addition to thus, MicroFog-CE provides enhanced capabilities to support custom load-balancing policies, where weights of the weighted load-balancing approach can be updated based on custom load-balancing policies.

   As an example, the current version of the CE implements weighted round-robin load balancing policy. Once the weight for each microservice instance is calculated based on the placement, CE handles the updates related to subsets, weights, and routes in Istio VS and DR resources. While this update is straightforward for centralised operation mode, distribute placement has one main challenge. Load balancing information can only be calculated after all required microservice instances are placed. Moreover, to execute load-balancing policies properly, Istio needs VS and DR resources to be available in all clusters that host the particular microservice (consumed microservice) and any microservice that tries to interact with it (consumer microservices). Thus, in distributed placement mode, for each microservice, the CE waits until all its instances and its consumer microservices are placed. Afterwards, the information required for VS and DR updates (subset

names and weights) are sent to relevant clusters through API 3 of the distributed CEs.

6. *PR Forwarding Policy Integration:* Placement across multi-cloud multi-fog environments requires the use of distributed placement policies across infrastructure provided by multiple Cloud and Fog infrastructure providers. MicroFog-CE enables this by providing the ability to update the status of the partially processed PRs and forward them to adjacent Fog or Cloud clusters. Such PRs are submitted to the selected cluster's API 1. Moreover, novel forwarding policies can be integrated as well. The default implementation of the CE provides two forwarding policies where the PRs can be either forwarded to a random Fog cluster or to the Cloud. As CE instances are configured independently, it is possible to use different forwarding policies across clusters.

7. *Automated Application Deployment:* MicroFog CE abstracts the microservice deployment process from the framework users. For each application, YAML File Store is used to retrieve the Kubernetes and Istio resources related to the deployment of microservices. This includes resources at different abstraction levels such as 1. application level resources such as Namespaces, Roles and RoleBindings, 2. microservice level resources such as ConfigMaps, Secrets and Pod definition YAML files to create microservice instances on mapped nodes based on the placement algorithm output, 3. Services, Virtual Services and Destination Rules for service discovery across clusters and to load balance and route traffic to create composite services based on the load balancing policy and 4. Gateways to enable ingress traffic to reach root microservices of application DAG. Moreover, MicroFog-CE enables federation across multiple Fog and Cloud clusters by deploying microservice composition-related resources (i.e., Kubernetes Services, Virtual Services, Destination Rules) in relevant clusters. CE rules are designed to handle these functionalities, thus abstracting the underlying complexities from the framework users.

8. *Scalable and Distributed CE deployment:* As the CE is developed as a microservice using a Kubernetes-native microservice framework, it can be deployed within Kubernetes and Istio-enabled environments in a distributed manner. Each CE can

be configured separately and communicate across clusters using the REST APIs, thus making MicroFog scalable to operate across federated Fog and Cloud environments.

9. *Extensibility:* Design and architecture of the CE capture the problem domain of microservices-based application placement by implementing java objects as rich domain-specific objects. Figure. 6.4 domain diagram used in developing the MicroFog-CE, which adheres with the system models and placement problem formulated in the Section 6.2. This makes the CE implementation easy to comprehend and extend to incorporate novel features. Moreover, due to the compatibility of the MicroFog framework with open-source cloud-native technologies, the CE can evolve as the capabilities of the underlying technologies evolve.

10. *Configurability:* Quarkus enables application configuration properties to be acquired through Kubernetes ConfigMaps. This highly improves the configurability of the CE, where the users can update application configurations without creating new Docker images to rapidly use different configurations (policies, placement modes, operation modes, etc).

Data Stores

MicroFog uses three main data stores as follows:

1. *Meta Data Store:* Metadata store contains application-related information belonging to two main categories: 1) application model (as discussed in Section 6.2.2) which contains specification related to microservices, interconnections among microservices to create services, dataflows, etc. 2) application deployment related Kubernetes and Istio resources. This includes resource type (i.e., Namespaces, Pods, Services, etc.) and URL to the YAML file containing the specifications of each resource. We use Redis [3] as a primary database to store this information. Even though Redis was initially introduced as a cache, now it is increasingly used as a primary database to reduce the complexity of data retrieval and improve per-

[3]https://redis.io/

formance. Redis allows data to be stored as key-value pairs. With the use of Redisson, a Redis Java client, the *Application* domain objects of the CE can be easily serialised to store within the Redis metadata store and retrieve them back as Java objects.

2. *Yaml File Store:* This is used for storing Kubernetes and Istio resource configurations as YAML files. Due to the geo-distributed nature of the Fog clusters, a distributed object store is required for efficiently storing the YAML files. To meet this requirement, we use MinIO Object Store [4], an AWS S3 compatible, Kubernetes-native object store designed for multi-fog multi-cloud environments. For each Istion/Kubernetes resource to deploy, the CE retrieves the YAML file from the MinIO data store using an object URL and uses the Fabric8 Kubernetes client library to load it as a domain object representing the deployment resource.

3. *Docker Registry:* As IoT application microservices are containerised for deployment, the container images must be stored in a docker registry reachable by the CEs. In the current implementation, we use Docker Hub, a publicly available managed Docker store. However, this can be further improved by using local Docker stores in conjunction with Docker Hub, depending on the resource availability of each Fog cluster to host the images.

Monitoring and Log Management:

Due to their highly distributed and dynamic nature, monitoring and observability remain essential aspects of cloud-native microservices. To this end, Istio enables the integration of multiple tools in the form of pre-configured plugins. This includes metric collection and visualisation (Prometheus and Grafana), distributed tracing (Jaeger, Zipkin), and mesh visualisation using Kiali. In the current version of the MicroFog framework, we have integrated Prometheus, Kiali and Grafana to observe the traffic across clusters and to validate the functionalities of the MicroFog-CE. In addition, MicroFog uses a cluster-level logging architecture to manage the logs generated within each cluster. To this end, MicroFog uses Grafana Loki, a decentralised, lightweight logging stack

[4] https://min.io/

that compresses and stores data in object stores such as S3. As the MinIO object store used for YAML File storage is S3 compatible, MicroFog uses the same store for storing the logs. Compared to other cloud-native logging solutions like ElasticSearch, Loki has a less complex architecture, requires less storage and consumes less power, which makes it suitable for Fog deployment. Depending on the resource availability of the Fog clusters, the logs can be stored within the MinIO hosted in Cloud to save storage space. However, other tools also can be easily integrated depending on requirements. Moreover, the current architecture can be easily extended so that MicroFog-CE can use the metrics collected from monitoring and logging tools to execute dynamic placement algorithms or integrate machine-learning-based approaches.

Rapid Prototyping Support

Producing novel placement algorithms undergo multiple development and evaluation cycles to optimise their performance. Thus, rapid prototyping during different stages of policy development is beneficial before conducting large-scale evaluations or applying them in real-word application deployments. Due to the use of open-source cloud-native tools, MicroFog enables fast creation of underlying infrastructure using tools such as Kind and MetaLB to create Fog computing clusters consisting of heterogeneous nodes and route inter-cluster traffic through load balancers.

6.3.3 PR Processing flow of MicroFog-CE

In this section, we discuss the high-level pseudo-code (see Algorithm 14) of the MicroFog-CE with regards to processing received PRs. In an environment where each cluster contains a separate CE, the depicted PR processing procedure is executed in all CEs under the distributed placement mode and only in the primary CE if the placement mode is set to centralised placement.

PR processing begins with retrieving *PR*s from the *PRQueue* (line 1). The method of retrieval depends on the placement mode of the CE, where in periodic placement, all PRs collected in the *PRQueue* are retrieved for processing, whereas in event-driven mode, each PR is taken from the queue as its added. If the PR processing thread is busy, the

PR waits in the queue until the thread becomes free. The current implementation of the CE uses a single thread for the PR processing, whereas multiple threads add incoming requests to the *PRQueue* implemented using *Java ConcurrentLinkedQueue*, which is a non-blocking and thread-safe queue implementation.

Retrieved PRs undergo three main steps: Meta Data Retrieval, Placement Algorithm Execution, and finally, Deploying microservices-based applications using Kubernetes and Istio resources and handling uncompleted PRs. The first step of metadata retrieval is to generate cluster data required by the placement algorithm (lines 5-11). This includes details about the resource availability of each node in the cluster along with topological details such as adjacent Fog and Cloud clusters of each considered cluster. For centralised placement, the primary CE that is responsible for executing the placement algorithm needs to have a bird's eye view of all the Fog and Cloud clusters. Thus, the primary CE queries other clusters by sending requests to the API 2 of the connected clusters (lines 10-11). For this, we implement a Reactive REST Client that sends all requests simultaneously, waits for the results of all the sent requests, and retrieve each cluster's data from the reply. Reactive REST Clients supported by the Quarkus framework enable concurrent request sending, which improves the efficiency of collecting data from distributed clusters. As the second step of metadata retrieval, the CE queries the application model related to the application requested by each PR from the Redis metadata store (line 13). This retrieves a Java domain object of type *Application* (as depicted in domain model 6.4) which consists of Microservices, Composite Services, Datapaths, Dataflows, Resource Requirements and Commands used for microservice deployment, which are all depicted using serialisable Java objects.

Afterwards, the CE starts processing the *PR*s using the placement algorithm (lines 16-19). As the CE can support integration of placement algorithms either by extending the existing CE or as an external microservice, the algorithm can be configured as a property of the CE. The CE is designed to use the factory pattern to initialise placement algorithms based on the configured placement algorithm name. Thus, the internal integration of the placement algorithms requires them to be added to the factory. To use external algorithms, CE implements a REST client with a configurable URL that can be updated with the URL of the external algorithm (line 19).

Once the placement output is generated by the placement algorithm, the CE moves on to the application deployment stage. During this step, CE generates deployment information for each cluster under two main categories: basic deployment information and load balancing information. Basic deployment information includes pod-to-device mapping with required resource allocation, ingress clusters for each application for the deployment of Istio Gateway and related Virtual Service for ingress traffic routing, etc. Load balancing-related deployment information generation includes executing the load balancing policy for the placement of completed microservices and generating subsets and weights accordingly. This data will be used to update Virtual Services and Destination Rules to ensure desired load balancing.

After generating the deployment information, the CE invokes a new thread to forward incomplete PRs (in the distributed placement mode) based on the forwarding policy while the current thread continues with deployment. In the centralised placement mode, the CE uses a Reactive REST Client to send the deployment information to others concurrently while the deployment for the current cluster is carried out in parallel as well. This decision is made to improve the overall efficiency of the placement as the deployment of microservices as Docker containers can be time-consuming if carried out sequentially. Similarly, in the distributed placement mode, load balancing information relevant to previous clusters are also transmitted concurrently while one thread continues with deployments related to the current cluster.

6.4 MicroFog Deployment

Deployment of MicroFog within federated fog-cloud environments includes two main steps: 1. distributed setup for data stores, and 2. distributed deployment of the CE. As example deployment scenarios, we provide deployment architecture (Figure 6.5 and Figure 6.6 for each step. The demonstrated examples consider a federated fog-cloud environment consisting of two Fog clusters and one Cloud cluster. Three clusters belong to three separate networks and are three independent Kubernetes clusters interconnected through Istio multi-primary architecture to enable inter-cluster microservice composition and traffic.

Algorithm 14 MicroFog-CE PR processing

```
1: procedure PROCESSPRS(PRQueue)
2:     PRs ← get PRs from the PRQueue for processing
3:     # Step 1. Meta Data Retrieval wich consists of two sub-steps 1.1 and 1.2
4:     # Step 1.1 : Cluster data retrieval (including both resource availability within cluster and tipology information
5:     clusterData ← {}                                          ▷ Maps cluster name to its data
6:     inclusterDeviceData ← loadInClusterDeviceData()    ▷ Device data related to the current cluster is loaded
7:     currentClusterData ← inclusterDeviceData ∪ topologyData
8:     clusterData.add(currentClusterName, currentClusterData)
9:     # For centralised placement, request cluster data from other cluster using API 2
10:    if centralisedPlacement AND is primary CE then
11:        clusterData ← requestOtherClusterData()
12:    # Step 1.2 : Loading application meta data form the Meta Data Store
13:    appInfo ← loadRelatedAppInfo(prs)
14:    # Step 2: Execute the placement algorithm
15:    placementOutPut ← {}
16:    if is internalAlgo then
17:        placementOutPut ← placementAlgo.generatePlacement(PRs, appInfo, clusterData)
18:    if is externalAlgo then
19:        placementOutPut ← externalPlacementAlo.generatePlacement(PRs, appInfo, clusterData, externalUrl)
20:    # Step 3. Deploy using Istio + Kubernetes resources and handle incomplete PRs
21:    perClusterDeploymentInfo ← {}
22:    perClusterDeploymentInfo.add(generateBasicDeploymentInfo(placementOutPut))
23:    perClusterDeploymentInfo.add(generateLoadBalancingRelatedDeploymentInfo(placementOutPut))
24:    if is distributedPlacement then                           ▷ Uses a separate thread
25:        incompletePRs ← placementOutPut.getIncompletePRs()
26:        forwardIncompletePRs(incompletePRs)
27:    thisClusterDeployment ← perClusterDeploymentInfo.getThisCluster()
28:    deploymentHandler.deploycommands(thisClusterDeployment)
29:    sendToOtherClusters(perClusterDeploymentInfo − thisClusterDeployment)
```

6.4.1 MinIO YAML File Store Deployment

We provide an example deployment scenario in Figure 6.5 to demonstrate the distributed deployment of the MinIO YAML File Store within federated fog-cloud environments. For distributed storage and access of YAML files, we design the deployment architecture to meet the following requirements: 1) Distributed deployment across clusters to improve the latency of application deployment, 2) Replication across distributed data stores to maintain data consistency, 3) Fault-tolerance through a prioritised failover mechanism to ensure availability in a latency-aware manner.

To achieve these objectives, we create two traffic routing layers using Kubernetes and Istio resources, namely, the Management layer and the Data Access layer. The

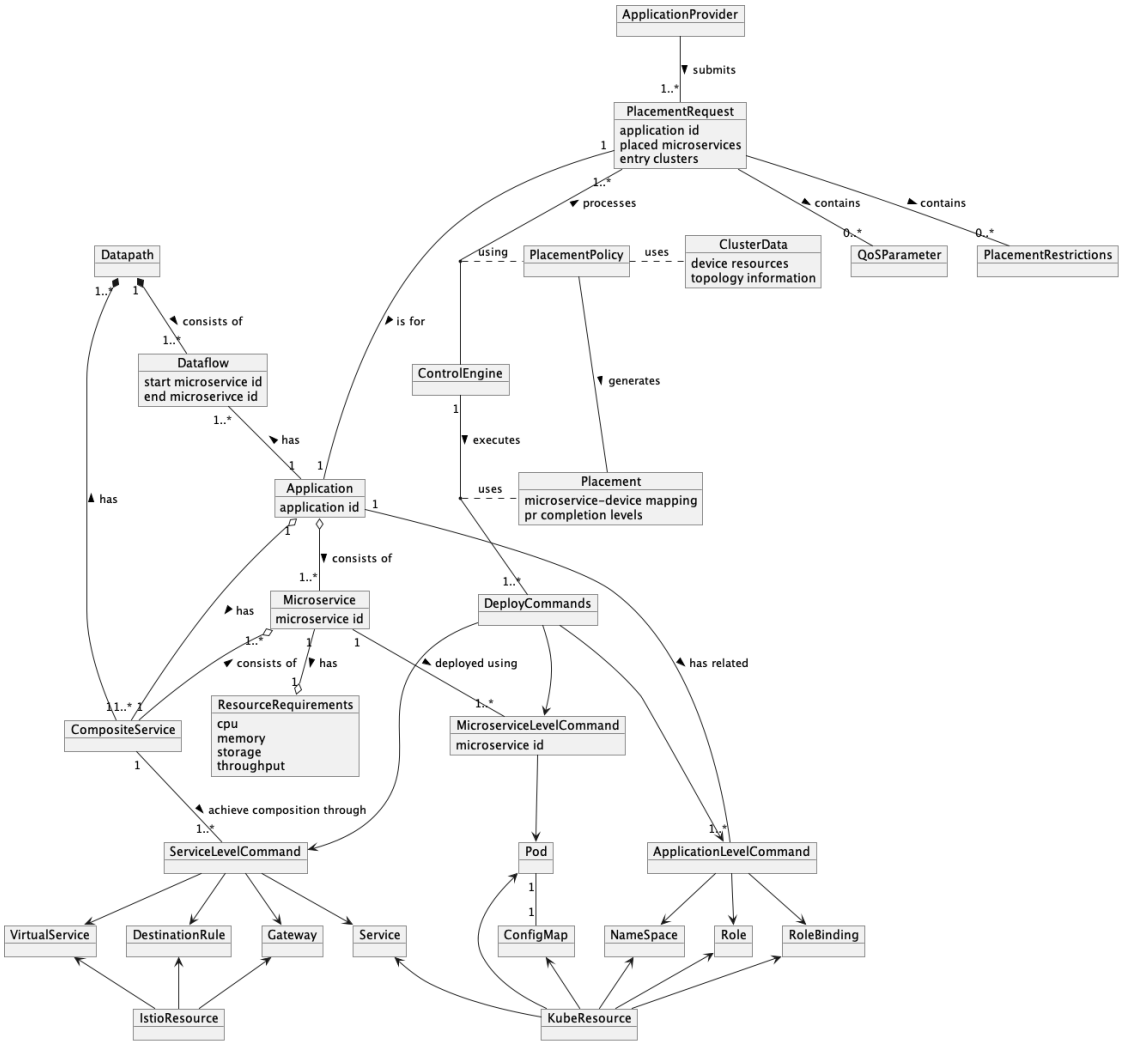


Figure 6.4: MicroFog: Domain diagram for CE

management layer is used for configuring individual MinIO servers deployed per cluster. Kubernetes service and Istio VS for the management layer expose default MinIO ports for management console access through ingress gateway (console port) and data replication among distributed MinIO instances (API port). The second layer of routing exposes the API port of the MinIO data store, for access by the CE to retrieve YAML files required for application deployment. This layer of traffic implements a two-tier failover policy to improve the reliability of the deployment. Istio supports locality-aware load-balancing to failover based on region (topology.kubernetes.io/region), zone

(topology.kubernetes.io/zone) and sub-zone (topology.istio.io/subzone) of the nodes. We use the region and zone to conduct the failover where all Fog level resources belong to the region "fog", where each Fog cluster is considered as a separate zone. Similarly, all Cloud clusters belong to the region "cloud". Istio default failover policy assigns high priority to failover within the same region (i.e., Fog clusters would fail over to adjacent Fog clusters). We further extend this by incorporating an Istio DR to ensure failover from Fog to Cloud if no Fog clusters are available. To ensure proper fault tolerance, each node in the Kubernetes clusters needs to be annotated with their related region and zone. Although the number of tiers is limited to two in the current implementation, it's possible to extend it to three tiers by implementing Istio sub-zones as well.

6.4.2 Redis Meta Data Store Deployment

Deployment of Redis Meta Data flow follows a similar approach with two traffic layers, one for data replication and the other for retrieving application information. We use the master-replica deployment supported by Redis. In our proposed architecture, we deploy the master Redis server in the Cloud cluster and deploy the rest as replicas where they sync with the master server to retrieve the available metadata. Similar to MinIO YAML Store, this deployment also uses locality load-balancing in Istio to ensure failover from the Fog layer to the Cloud to improve the availability of the data.

6.4.3 Control-Engine Deployment

Figure 6.6 depicts an example scenario for the distributed deployment of CEs across federated Fog and Cloud clusters. We discuss the main aspects of the deployment as follows:

- Distributed deployment of CEs and maintaining communication across clusters: In both centralised and decentralised placement modes, CEs need to access APIs of the other CEs deployed in different clusters for various functions, including querying cluster data, forwarding PRs, submitting deployment information. We enable this by using Istio DR and VS to route based on the header value of each

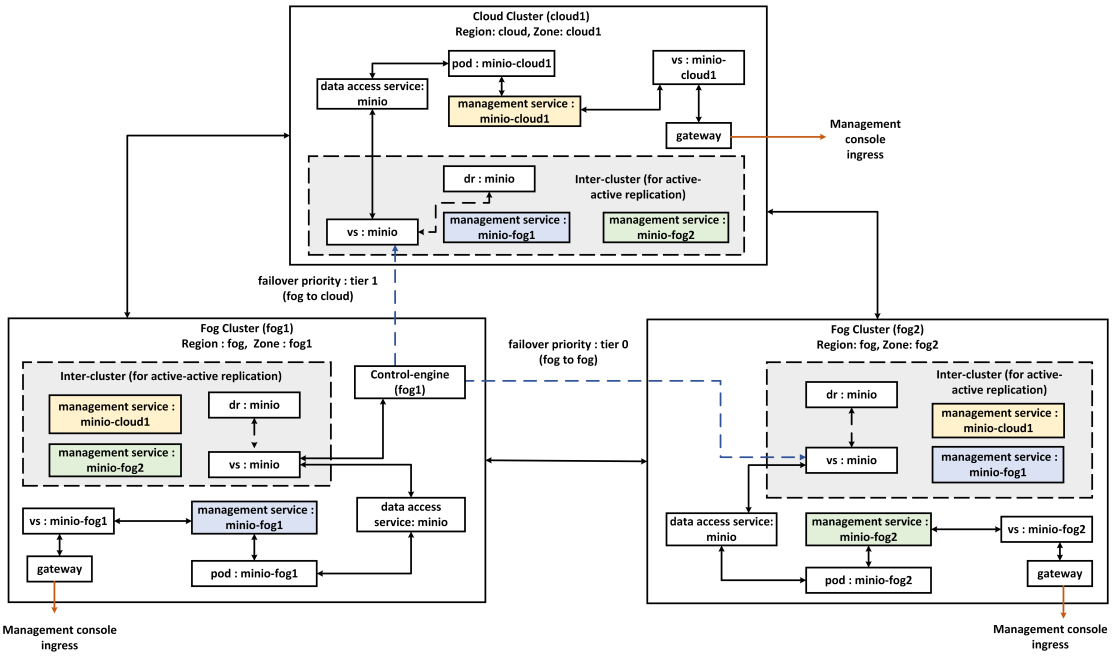


Figure 6.5: MinIO - YAML file store deployment

request. We introduce a header called "cluster", which defines the destination cluster to route the requests. To achieve proper routing, each pod of CE is labelled with its cluster name, and the DR creates subsets based on the cluster name. Following this implementation, the VS routes by matching the header value to the subset label.

- PR submission to a particular cluster: The above implementation enables not only inter-CE routing but enables ingress traffic to the CE (i.e., submitting PRs) to be routed to a specific CE based on the header value.

- Configure each CE separately during deployment: To improve the efficiency of configuring the CEs and to enable each CE to be configured independently, we use a Kubernetes CongfigMap to define the CE configurations. Due to its Kuenernetes-native nature, the Quarkus application is configured to retrieve the values for application.properties from the ConfigMap.

- Ensure access to underlying Kubernetes and Istio deployments: CE needs to access Kubernetes API for various actions (i.e., retrieve node data, retrieve resource

metrics, retrieve pod data, deploy Kubernetes and Istio resources). To this end, the proper level of permission should be granted to the CE. A dedicated service account is created and attached to a ClusterRoleBinding and a ClusterRole to grant the required access across the cluster.

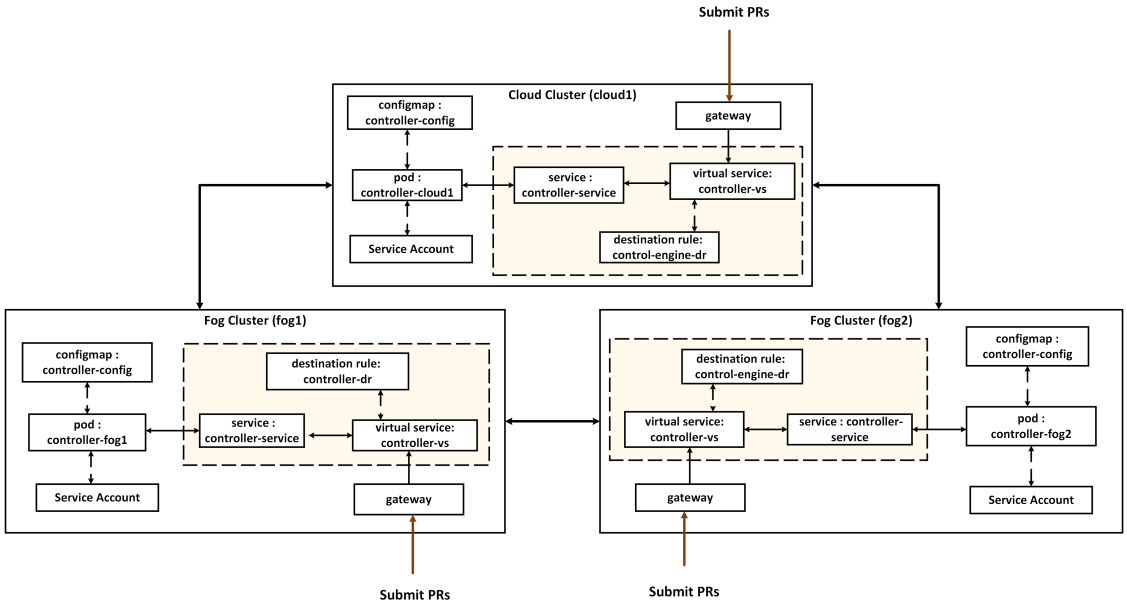


Figure 6.6: Distributed CE deployment

6.4.4 Deployment of Observability, Monitoring and Logging Tools

For the current implementation, we integrate Prometheus and Kiali to verify the feature supported by the CE. Kiali uses the Prometheus monitoring tool to create topology graphs, calculate health and show metrics. Istio add-on preconfigures it to visualise multi-cluster service mesh, including different views such as graphs (depicting application, services, microservice versions, etc.), traffic flows, metric details, and Istio configurations (YAML files related to each deployed Istio resource). Within the distributed architecture, Prometheus and Kiali components are deployed per cluster, and the Kiali dashboard is exposed through the Istio ingress gateway to access it remotely.

For log aggregation and visualisation we use Loki and Grafana. Loki is configured to use a object bucket from MinIO object store. As the MinIO deployment and request

routing is already handled (Section 6.4.1), logs can be directed either to a central Cloud or stored within the own cluster depending on the resource availability.

6.5 APIs of MicroFog-CE

In this section, we highlight the three main APIs provided by MicroFog-CE and also explain the API implementation required to integrate external algorithms into the CE.

- API 1 (see Figure 6.7): API 1 is designed for receiving PRs through POST requests, where the request is routed to the cluster defined in the header. The request contains data related to the PR in JSON format, which will be mapped into a Java-based domain object by using the Jackson framework upon receipt. *"applicationId"*, which is used to identify the application to be deployed (matched with the metadata available in the Redis Meta Data Store), and the *"entryClusters"*, which indicates the traffic entry points to the application are required fields for the request data whereas other fields are optional. The rest of the fields are optional and can be filled if relevant. *"placedMicroservices"* indicate already placed microservices and their status. Thus this is mostly used for forwarding requests and can also be used for initial PR submission if some of the application microservices are excluded for placement within Fog or Cloud (i.e., already placed within IoT devices or client devices). *"compositionOnlyPlacements* keep track of intermediate clusters that needs to host service level resources to enable compositing of microservices across non-adjacent clusters. Boolean for *"loadBalancingCompleted"* indicates if load balancing-related deployment information for the microservice has already been transmitted to relevant clusters, whereas *"subsetWeights"* indicate relative resource-allocation among devices to be used for executing load balancing policy. Due to complex dependencies among microservices, the QoS parameters can be defined at multiple granularity levels: per composite service, among microservices and per application [46]. *"qosParameters"* field allows detailed parameter definitions adhering to this.

- API 2 (see Figure 6.8): API 2 is used in centralised placement mode for querying

cluster data from each cluster by defining the cluster name in the header to ensure routing. The response returns two main types of data: 1) an array containing resource availability of each node in the cluster defining total resources, resource usage at the time of query and allocated resources (i.e., memory in bytes and CPU in the number of cores/ vCPUs), 2) data related to topology containing the names of adjacent Fog and Cloud clusters.

- API 3 (see Figure 6.9): API 3 is for transmitting deployment information to each cluster specified by the header field. For centralised mode, this includes both microservice deployment and load balancing related Istio resource deployment, whereas, in distributed mode, it is limited to load balancing related resources. This API also accepts some additional information, such as the Boolean indication if the cluster is the entry cluster for the application so that the Istio Gateway and VS resources can be deployed accordingly to enable ingress traffic to reach the application. The request also contains a file that includes a list of microservices (*additionalMForSLevel)*, where their service level resources (i.e., Kubernetes Service, Istio VS and DR) need to be deployed within the cluster to maintain seamless connectivity among microservices deployed within clusters that are not adjacent.

Due to the use of Jackson library for conversion between JSON data and JAVA domain objects, the data sent to/from APIs can be modified easily by updating the relevant domain objects accordingly.

6.6 MicroFog - Evaluation and Validation

In this section, we validate the main features and functions supported by MicroFog using multiple use cases.

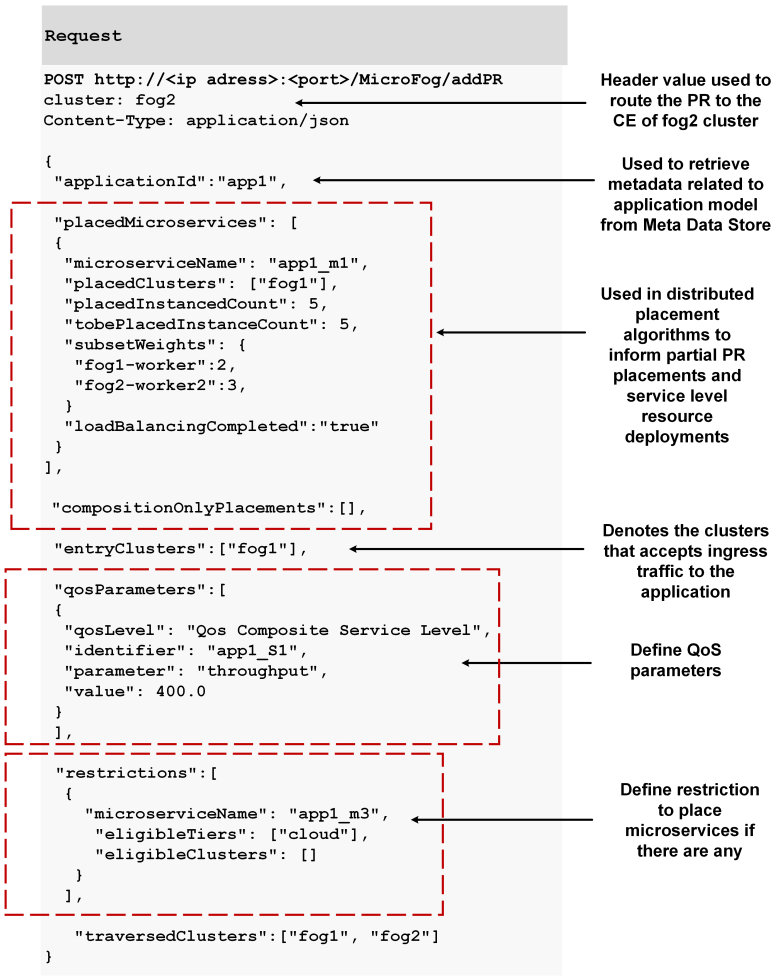


Figure 6.7: API 1 - For submitting PRs

6.6.1 Experimental Setup

Infrastructure and MicroFog setup

To evaluate the features supported by MicroFog, we create a prototype of a federated fog-cloud environment consisting of three Fog clusters (fog1, fog2 and fog3) and one Cloud cluster (cloud1) as depicted in Figure 6.10. Each cluster belongs to a separate network and communicates with each other through load balancers. For the prototype, we use MetalLB [5] as the load balancer that exposes each cluster to the outside. Each

[5]https://metallb.universe.tf/

Request

```
GET http://<ip adress>:<port>/MicroFog/getClusterData
cluster: fog1
```

Response

```
Content-Type: application/json
{
   "clusterName": "fog1",
   "fogDevices": [
      {
          "nodeName": "fog1-worker1",
          "resourcesTotal": {
          "memory": 3878420480,
          "cpu": 2
           },
          "resourcesUsed": {
          "memory": 569229312,
          "cpu": 1.014943195
           },
          "resourcesAllocated": {
          "memory": 2782920704,
          "cpu": 1.500
           }
       },
   ],
   "adjacentFogClusters": [fog2],
   "connectedCloudClusters": [cloud1]
}
```

Figure 6.8: API 2 - For querying cluster information

cluster is a separate Kubernetes cluster, and the communication among microservices running across different clusters is maintained by implementing an Istio service mesh across the clusters in multi-primary mode. Table 6.2 summarises the details of each cluster.

One of the main advantages of MicroFog is its compatibility with cloud-native technologies, which enables quick prototyping of federated fog-cloud architectures for placement algorithm development and evaluation to overcome the limitations due to the lack of publicly available Fog resources. To demonstrate this, we create the fog1, fog2 and fog3 clusters using virtualised resources available in the University of Melbourne's Queensberry Hall data centre, which is at the edge of the network and create cloud1 using AWS EC2 instances from ap-southeast-2 accessed through the internet. To repli-

Figure 6.9: API 3 - For submitting placement output for deployment

cate the behaviour of Fog clusters where Fog nodes are connected to each other through high bandwidth LAN links, we implement fog1, fog2 clusters as KinD Kubernetes (containerised k8s) clusters and fog3 as a k3d (containerised k3s) cluster belonging to separate sub-nets within the data centre. Their communication to the Cloud cluster occurs over the WAN network.

Table 6.2: Federated fog-cloud infrastructure setup

Cluster	Resources	
Details	**CPU (VCPUs)**	**Memory (GB)**
Cluster - fog 1 :		
node1 (control-node)	3	6
node2 (worker 1)	4	9
node3 (worker 2)	5	16
node4 (worker 3)	3	8
Cluster - fog 2 :		
node1 (control-node)	3	6
node2 (worker 1)	3	9
node3 (worker 2)	2	6
node4 (worker 3)	4	12
node5 (worker 4)	4	8
Cluster - fog 3 :		
node1 (server)	3	6
node2 (agent 0)	2	4
node3 (agent 1)	2	4
Cluster - cloud 1 :		
node1 (control-node)	8	14
node2 (worker)	8	14

Workload Creation

Due to the lack of diverse microservices-based IoT application benchmarks, we implement a tool to generate microservices-based mock applications [6] that can capture different characteristics of MSA and generate heterogeneous applications for placement policy evaluation purposes. In designing the tool, we analyse the model of the microservice application presented in Section 6.2.2. Based on this, our proposed tool provides a base

[6] https://github.com/Cloudslab/MicroFog/tree/main/Workload_Generator

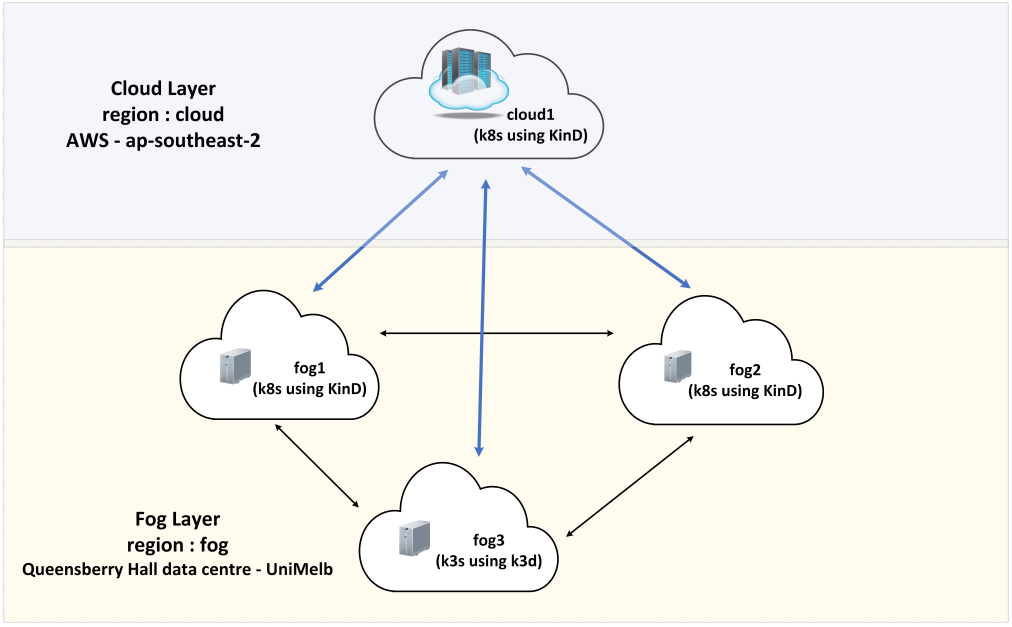


Figure 6.10: Multi-fog multi-cloud infrastructure

microservice as a template that can be configured (using a Kubernetes ConfigMap) to create microservices that can interact with other microservices to to create microservices-based applications having composite services that the users can access. To this end, microservices can create patterns such as chained, aggregate, hybrid or microservice candidate pattern. Furthermore, the microservices created using the template, can be configured to have different processing times and inter-microservice message sizes to fabricate the behaviour of heterogeneous IoT applications in terms of data processing and transmission. Using this tool, we create multiple microservices-based applications containing chained and aggregator interaction patterns to evaluate and verify different functionalities supported by the MicroFog framework. Moreover, we use the template microservice and configure it to create the Smart Health Monitoring IoT Application presented in Section 6.2.4. This demonstrates the tool's ability support creation of IoT applications based on their DAG representations.

Placement Algorithm

To highlight the main features supported by MicroFog, we adapt and implement different variations of the placement algorithm proposed in [60]. The algorithm in [60] aims to place the latency-critical IoT application services as close as possible to the user such that the resource requirements of the microservices are met. To this end, the placement policy starts placement from the traffic entry Fog clusters, moves towards adjacent Fog clusters and finally considers Cloud if the Fog resources are insufficient. We extend the policy in [60] to incorporate throughput awareness where the throughput of the composite services can be provided during PR submission, and the placement algorithm calculates the number of microservice instances and resources requirement to support the throughput. We use the calculation provided in [46] for this. We create three variations of this approach to evaluate and validate multiple configurations and features of MicroFog as follows:

1. Version 1 (V1) - Vertically Scaled Distributed Placement: The placement algorithm retrieves already placed microservices from the PR and calculates the next microservice to place based on the DAG representation of the application. Afterwards, the algorithm tries to place the microservice within the cluster in a resource-aware manner. In this approach, since vertical scalability is considered, a single instance is placed for each microservice so that their resource allocation suffices the throughput requirement. If the cluster doesn't have enough resources to complete the application placement, the PR is updated and forwarded to the next cluster to place the rest of the microservices.

2. Version 2 (V2) - Horizontally Scaled Distributed Placement: This follows a similar approach to V1 but supports the horizontal scalability of the microservices. Thus, instead of a single instance, multiple instances of each microservice are placed to support the throughput requirement.

3. Version 3 (V3) - Centralised Placement: In this version, the placement algorithm maintains a view of all available clusters. Once the request is received, the algorithm selects one of the entry clusters defined in the PR. Next, the algorithm traverses the DAG and places microservices starting from the selected Fog cluster,

then consider adjacent clusters if no resources are available and finally considers Cloud for placement.

As discussed above, V1 and V2 algorithms are designed specifically to support distributed operation mode of the MicroFog-CE whereas V3 is designed for centralised operation mode and can not carry out placement in distributed mode. To operate in distributed mode V1 and V2 algorithms are designed with additional functionalities such as processing partially placed PRs and forwarding partially completed PRs to adjacent clusters for completion.

6.6.2 Use cases and results

Analysing Flexibility and Scalability of MicroFog Architecture

Flexibility and scalability of the MicroFog architecture is denoted by its ability to operate within distributed multi-fog multi-cloud enviornments. We explore distributed deployment architecture of the MicroFog framework under different configurations to demonstrate this.

- **Distributed Data management and access :**

  In this section, we analyse and validate the deployment architectures proposed in this work for accessing MinIo Yaml File Store and Redis Meta Data Store. Our proposed deployment architectures aim to ensure lower latency and high availability of the data stores to ensure reliable placement and deployment of applications. To evaluate this, we consider three data access scenarios. Relative data retrieval latency is measured for each scenario as shown in Figure 6.11(a) and Figure 6.11(b) for MinIO YAML Store and Redis Meta Data Store, respectively. We submit placement requests to the CE placed in fog1 and observer behaviour under distributed placement mode. In Scenario 1, both data stores are deployed within all 3 clusters following the proposed architecture in Figure 6.5. Scenario 2 considers the unavailability of fog1 data stores, whereas Scenario 3 considers the unavailability of data stores in both fog1 and fog2.

Results demonstrate that the deployment architecture manages request routing to data stores as intended. The failover policy is configured to prioritise the closest data store in case of data store failures. Accordingly, if all data stores are available, the CE deployed within cluster fog1 accesses the data stored deployed within the same Fog cluster, thus resulting in the lowest data retrieval latency. If the data stores within the cluster are unavailable, the routing policy prioritises the closest adjacent Fog cluster over the Cloud cluster and only accesses the Cloud cluster in case the data stores in both Fog clusters are unavailable. This behaviour is depicted by the obtained latency values, which show a slight increase in latency due to failover triggered among Fog clusters (Scenario 2 - FO to Fog) and a relatively larger increase with failover from Fog to Cloud (Scenario 3 - FO to Cloud). Thus, the proposed deployment architecture is robust to ensure data access while aiming to improve performance. Furthermore, in the case of resource-constrained Fog clusters, it would be more feasible to host the data stores in adjacent resource-rich Fog clusters or Cloud clusters at the cost of data access performance. Our proposed architecture is flexible enough to support this behaviour and ensure data access across federated multi-fog multi-cloud environments.

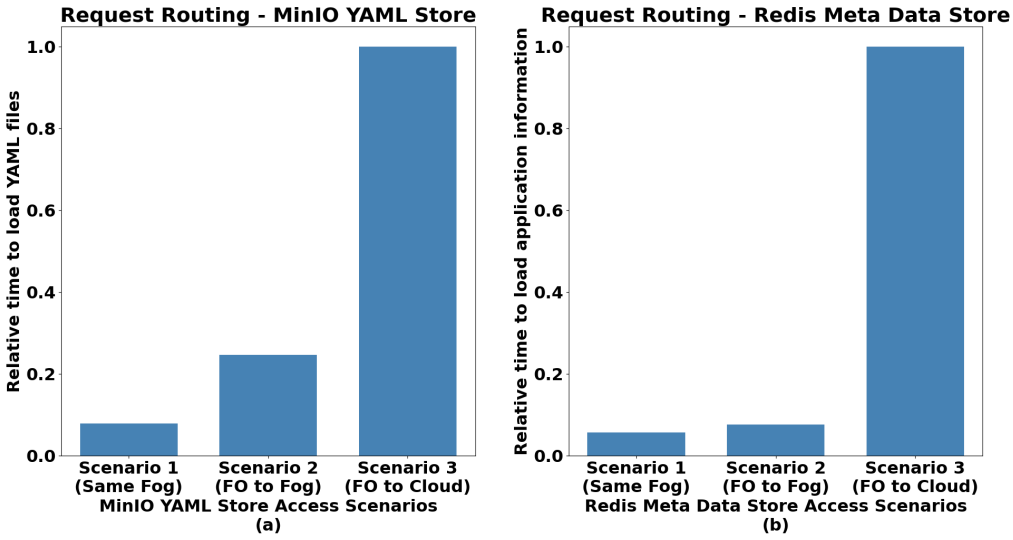


Figure 6.11: Availability analysis of data stores

- **Analysis on Distributed Deployment of CE and its Operation Modes**

MicroFog-CE is designed for scalable deployment across distributed Fog and Cloud clusters. To this end, CE supports distributed operation mode of the CE, where all CEs execute placement algorithms independently and the centralised mode, where the primary CE executes the placement algorithms and sends placement output to individual clusters. In both approaches connectivity among CEs are maintained using prososed deployment architecture (Section 6.4.3) to achieve successful placement of applications.

In the distributed mode, PRs can be forwarded to adjacent Fog or Cloud clusters, and MicroFog-CE supports the integration of different forwarding policies, thus providing the users of the framework with the flexibility to control distributed placement policies. We demonstrate this by implementing two forwarding policies, 1) FP1: if the current cluster does not have enough resources to complete PR placement, PR is forwarded to an adjacent Fog cluster, 2) FP2: if the current cluster does not have enough resources to complete PR placement, the PR is forwarded to a connected Cloud cluster. To route the PR to the selected cluster, the header of the PR forwarding request is updated with the destination cluster name. The deployment architecture proposed in Figure 6.6 routes to the correct destination based on that. Figure 6.12 shows three scenarios where in Scenario 1, the entry Fog cluster for the PR contains enough resources to host the application, thus resulting in the lowest response time out of the three scenarios. Scenario 2 and Scenario 3 consider a situation where the entry Fog cluster does not have enough resources to host the entire application. Scenario 2 uses FP1, thus placing the application across two adjacent Fog clusters, which results in a higher response time than the prior scenario due to inter-fog communication delay. However, FP1 performs better than Scenario 3, which uses FP2, where the request is forwarded to the Cloud. This incurs the highest response time among the three scenarios. The above use case demonstrates the scalability of the CE deployment architecture to tackled multiple Fog and Cloud clusters and also the ability to configure distributed placement policies by integrating forwarding policies.

MicroFog-CE also supports centralised placement algorithm execution as well. In Figure 6.13, we consider three placement scenarios and analyse time to application

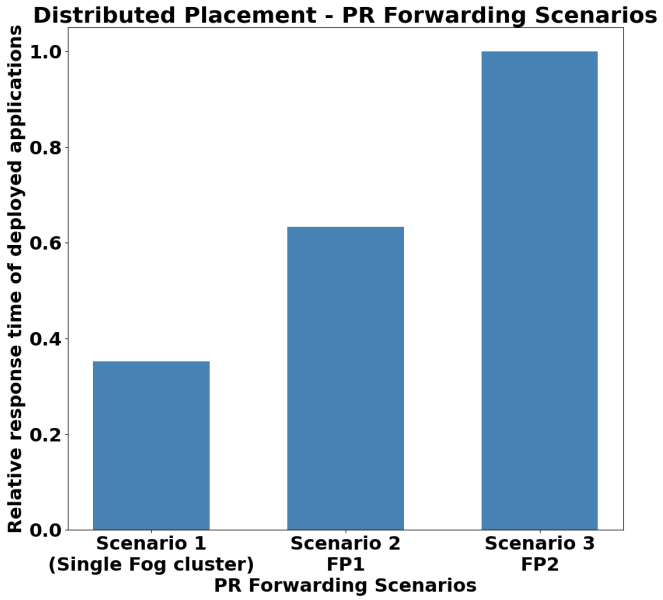


Figure 6.12: Distributed placement algorithm execution

placement under the CE's distributed and centralised operation mode. The three scenarios are as follows: Scenario 1 - 5 PRs are submitted to the system simultaneously such that three have *fog1* as the entry cluster and the other two have *fog2* as the entry cluster; scenario 2 - 10 PRs are submitted to the system simultaneously such that each receives 5PRs; scenario 3 - 15 PRs in total simultaneously submitted to fog1, fog2, fog3 such that each received 5 PRs. In the distributed operation mode PRs are submitted to the CE of their entry cluster, whereas in the centralised mode, all PRs are submitted to the primary CE deployed within the Cloud. Furthermore, the centralised mode uses V3, whereas distributed mode uses V2 as the placement policy. Figure 6.13 depicts the total time for PR deployment, calculated from when the CE receives the PR to application deployment completion under event-driven placement mode. According to the results, the distributed mode takes lesser time to complete application placement in all three scenarios. Moreover, experiment results depict that as the PR rate grows (i.e., PR arrival rate at each cluster increases in Scenario 2 compared to Scenario 1) or as the scale of the federated Fog environment grows (i.e., Scenario 2 with 3 Clusters and Scenario 3 with 4 Clusters), the relative increase in completion time is higher for centralised mode. This is because,

in the centralised mode, a single controller is processing the received PRs whereas in decentralised mode all controllers contribute to PR processing, thus reducing the load on each controller deployed per cluster. Thus, as the PR arrival rate and the scale of the environment increase, the distributed operation mode performs better.

However, the selection between the two modes depends also on the design of the placement algorithm (i.e., V2 is designed to operate in distributed mode, whereas V3 supports the centralised operation mode). Thus, MicroFog-CE is designed in an easy-to-configure manner, so that the users can use centralised or distributed operation modes depending on the PR arrival rate, the design of the placement policy and the scale of the federated Fog environments.

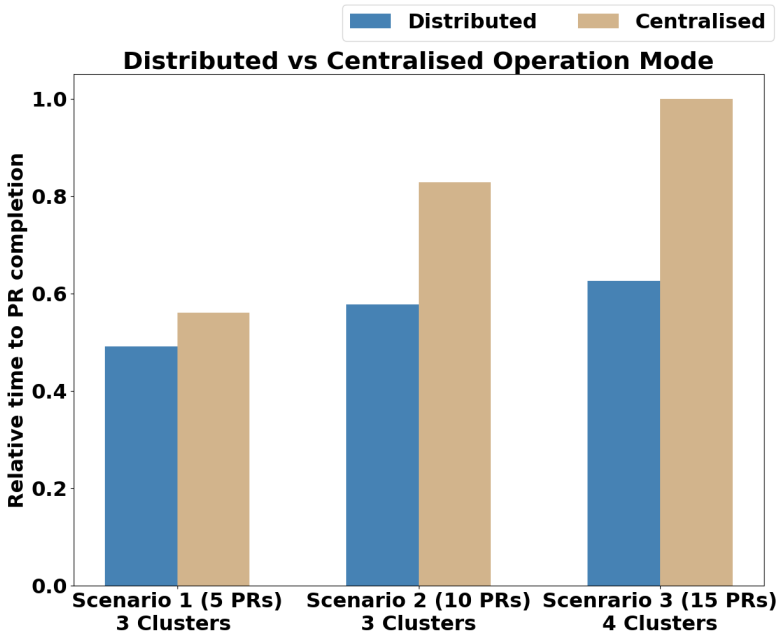


Figure 6.13: Analysis of CE operation modes

- **Analysis on Using Different Kubernetes Distributions**

  Due to heterogeneous resource availability, Fog and Cloud clusters can run different Kubernetes distributions (i.e., k8s for resource-rich clusters and k3s for resource-constrained clusters). To analyse the ability of MicroFog to operate across different

distributions. Results show that PR deployment time is lesser in fog3 (Scenario 2), which uses k3s due to its light architecture, whereas fog1 (Scenario 1) deployment time is higher. Furthermore, scenario 3 depicts a cross-cluster PR placement scenario, which takes longer than the k3s cluster but less time than the k8s deployment due to deployment across both. This demonstrates MicroFog-CEs' flexibility to operate across clusters with different Kubernetes distributions.

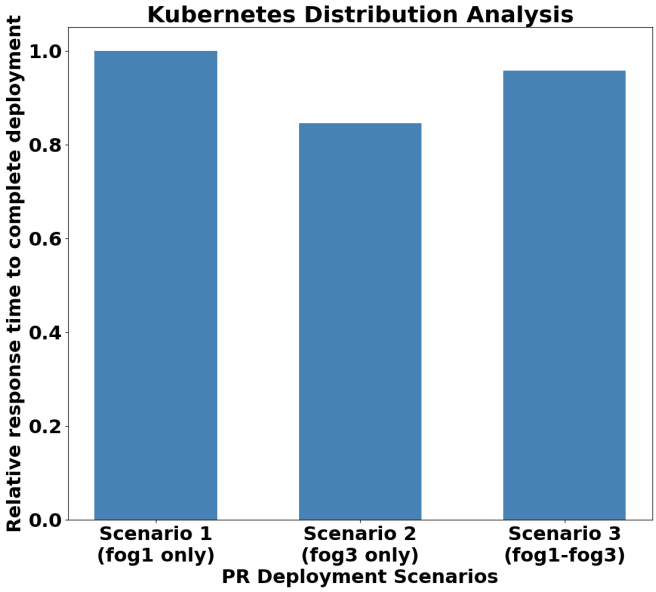


Figure 6.14: Analysis of Kubernetes distributions

The above results demonstrate the ability of MicroFog to handle placement across multiple clusters (scalable architecture) and configurability (integration of different placement algorithms, forwarding policies, and operation modes) of the MicroFog-CE, which enables it to successfully execute placement policies and deploy applications across distributed Fog and Cloud clusters.

Federated fog-cloud deployment and compositing (service discovery and load balancing) of microservices

One of the main advantages of MSA is the ability to independently scale microservices across distributed computing resources while ensuring their dynamic composition

through service mesh technologies. As MicroFog-CE supports easy integration of multiple placement algorithms, we implement V1 and V2 to demonstrate the effect of scalable microservice placement and validate dynamic composition and load-balancing enabled by MicroFog.

We consider the placement of two microservices-based applications generated using workload generator: smart healthcare application (application id: *hcapp*) discussed as an example IoT application in Section 6.2.4 (see Figure 6.2) consisting of two composite services, and a DAG-based application (application id: *app2*) which consists of a single composite service that can be accessed by the user (see Figure 6.16). The service consists of 4 microservices, where a2m1 and a2m2 form a chained invocation pattern and a2m2, a2m3, and a2m4 form an aggregator pattern such that a2m1 invokes a2m3 and a2m4, aggregates their results and return it back to a2m1 for further processing. The resultant placements generated by the two versions of the placement algorithm for *app2* and *hcapp* are recorded in Table 6.3. As V1 does not consider horizontal scalability, resource-constrained natures of the heterogeneous Fog nodes force the placement to move towards the Cloud, thus resulting in higher latency, as shown in Figure 6.15. In comparison to that, V2 utilises the ability to scale microservices horizontally. This results in better utilisation of limited Fog resources, thus resulting in lower latencies, as shown under scalable placement in Figure 6.15. Results demonstrate that, V2 improves latency by 44% for *app2* and 54% for *hcapp*.

However, dynamic service discovery and load balancing across clusters are required to ensure connectivity among microservices and maintain the expected level of performance. To this end, MicroFog-CE supports the integration of new load-balancing policies. In this experiment, we implement a Weighted Round Robin Load Balancing policy. Deployment rules of the MicroFog-CE deploy Istio VSs and DRs according to the output of the load balancing policy. For the above placement, we verify this based on the Kiali workload graph, which depicts the traffic distribution across different horizontally scaled instances of the same microservice. Table 6.3 shows that for the horizontally scaled microservice a2m2 in *app2*, the resource distribution is 1:2:1 among instances deployed within fog1-worker3, fog2-worker1 and fog2-worker2, respectively. Obtained graph (see Figure 6.16 shows that traffic for a2m2 is divided with a 1:3 ratio among two

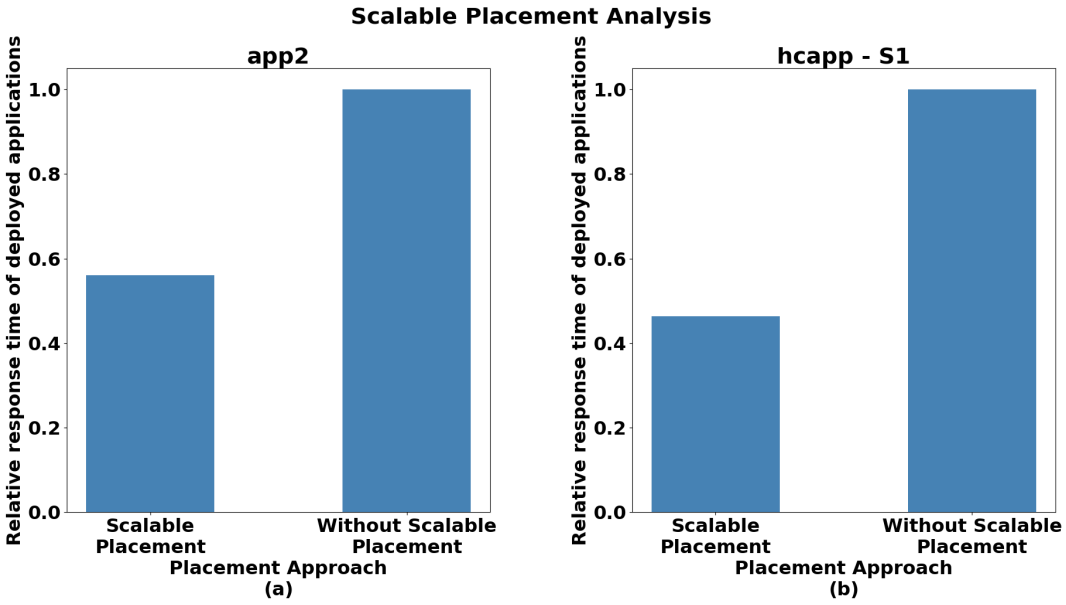


Figure 6.15: Scalable microservice placement

Table 6.3: Generated placement for example applications (app2 and hcapp)

Placement	app2		hcapp	
Algorithm	Microservice	Deployed Nodes	Microservice	Deployed Nodes
Version 1 (V1)	a2m1	fog1-worker2	hcm1	fog2-worker3
	a2m2	fog2-worker4	hcm2	cloud1-worker1
	a2m3	cloud1-worker1	hcm3	cloud1-control-node
	a2m4	cloud1-worker1		
Version 2 (V2)	a2m1	fog1-worker2	hcm1	fog1-worker1, fog1-worker3
	a2m2	fog1-worker3, fog2-worker1, fog2-worker2		Allocated Resource Ratio - 1:1
		Allocated Resource Ratio - 1:2:1	hcm2	fog1-worker1, fog2-worker1, fog2-worker3
	a2m3	fog2-worker3		Allocated Resource Ratio - 1:2:3
	a2m4	fog2-worker4	hcm1	cloud1-control-node

clusters and 2:1 within the fog2 cluster, thus diving a2m2 traffic with an approximate ratio of 1:2:1 among its three instances. This matches with the expected traffic distribution of Weighted Round Robin load balancing, thus confirming the ability of the MicroFog to automate Istio resource deployment to ensure the custom load balancing capabilities across clusters. This is further demonstrated by Figure 6.17, which reflects the traffic dis-

tribution of *hcapp*. The traffic distributions of microservices hcm1 (1:1) and hcm2 (1:2:3) adheres to their resource distribution of hcm1 (1:1) and hcm2 (1:2:3).

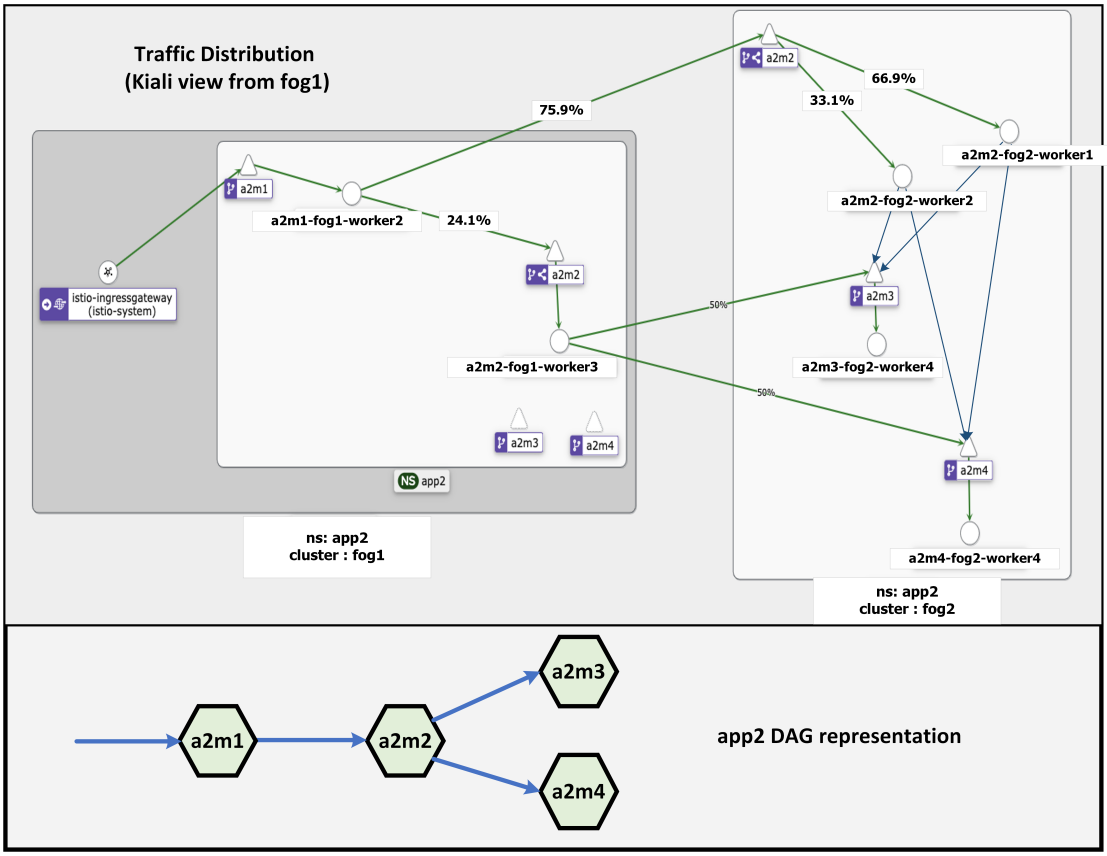


Figure 6.16: Multi-cluster service discovery and load balancing scenario - app2

Results obtained from the above use cases capture different features supported by MicroFog and verify that MicroFog is a scalable and easy-to-configure framework that can deploy microservices across federated Fog computing environments and ensure dynamic microservice composition across clusters. Hence, the MicroFog framework can be successfully used and extended for integrating and evaluating the performance of novel placement algorithms designed for the placement of microservices-based IoT applications.

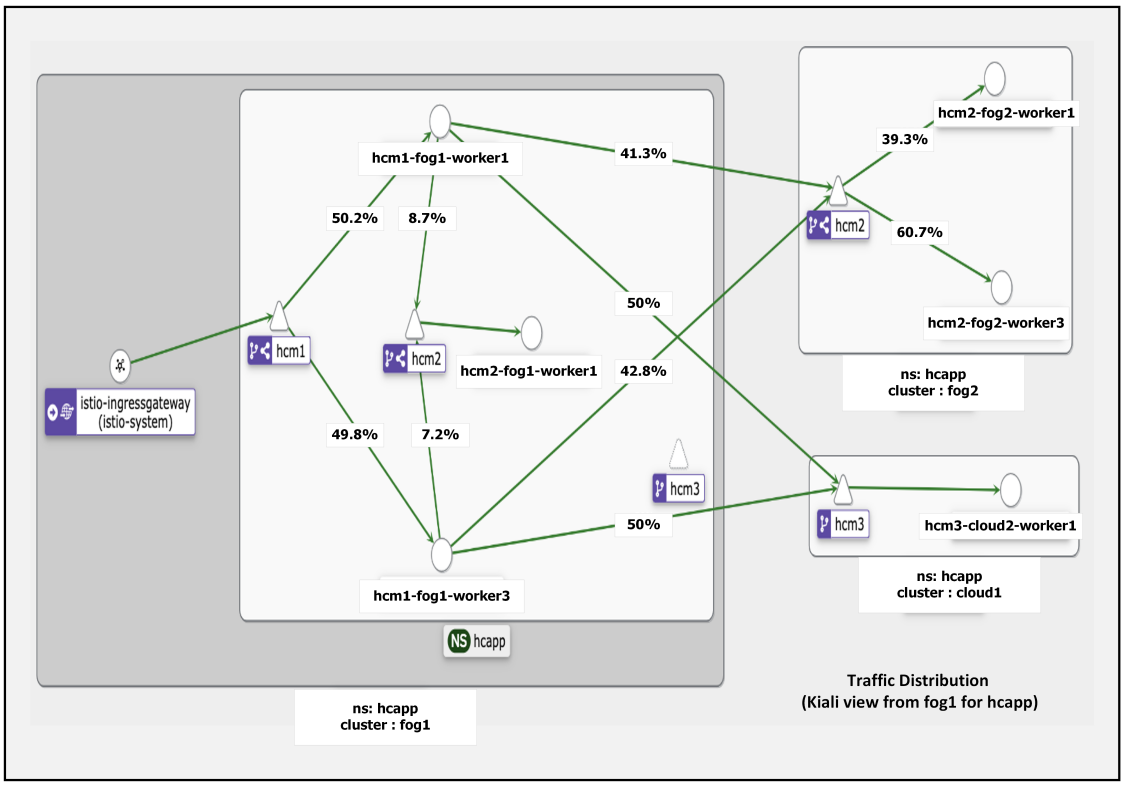


Figure 6.17: Multi-cluster service discovery and load balancing scenario - hcapp

Software Availability

The source code and documentation of the MicroFog framework is accessible from: https://github.com/Cloudslab/MicroFog

6.7 Summary

In this work, we proposed a framework for the scalable placement of microservices-based IoT Applications in federated Fog environments. First, we implemented a novel control engine for placement policy execution, microservice deployment and dynamic composition across multi-fog multi-cloud environments. Also, we proposed multiple deployment architectures to improve the distributed deployment of the framework. Also, we implemented multiple placement policies to demonstrate the framework's features. Next, we created a prototype of the proposed framework within a federated Fog

computing environment and evaluated the framework's performance and its ability to integrate placement algorithms for scalable placement of microservices, thus reducing the service latency of the microservices-based applications.

Chapter 7
Conclusions and Future Directions

This chapter concludes the thesis and provides a summary of works and key contributions. Next, it highlights several future research directions for further improvement of Microservices-based application placement in Fog computing environments.

7.1 Summary of Contributions

The IoT paradigm has gained tremendous popularity within diverse application domains containing heterogeneous services ranging from computation intensive and bandwidth-hungry to latency-sensitive and mission-critical. To meet the data processing demands of ever-growing IoT applications, Fog computing has emerged as a distributed computing paradigm that extends cloud-like services toward the edge to improve service latency and reduce network congestion. Meanwhile, Microservice Architecture has risen as a powerful software architecture that can meet the rapid development and deployment needs of fast-evolving IoT applications. Moreover, microservices' independent deployability, scalability, distributed deployment, and dynamic composition capabilities make them suitable for deployment within distributed computing paradigms such as Fog computing. Thus, the execution of microservices-based IoT applications within Fog computing environments has attracted significant interest from both industry and academia. With the ever-increasing number of IoT devices and various application services, the resource-constrained nature of the Fog resources becomes one of the main challenges for hosting large-scale IoT applications. Moreover, application execution within distributed Fog environments is affected by reliability and interoperability issues. In Fog computing environments, these issues can be resolved by identifying suit-

able placement techniques for application microservices. In this thesis, we investigated optimal placement techniques for microservices-based IoT applications within Fog computing environments.

Chapter 1 presented the basic concepts of Fog computing and Microservice Architecture and detailed the problem definition for "Microservices-based IoT Application Placement in Fog Computing Environments". Next, the challenges related to Fog application placement are highlighted and discussed. This chapter also discussed the resource questions identified in this thesis and summarised the thesis contributions.

Chapter 2 analysed the existing placement techniques for microservices-based IoT applications in Fog computing from different aspects, namely the accurate modelling of microservice architecture, developing microservices placement policy, incorporating microservice composition-related features, and performance evaluation. Next, taxonomies are created for each aspect, and the recent literature is reviewed according to the taxonomies, along with comprehensive discussions on research gaps.

Chapter 3 proposed a distributed placement approach for the scalable placement of microservices within resource-constrained and heterogeneous Fog devices. The proposed placement technique aims to reduce the service latency and network usage of the IoT applications services through the optimum use of heterogeneous Fog devices by utilising horizontal scalability and decentralised management of the microservices. Furthermore, to ensure distributed placement of microservices, this technique proposed a Fog node architecture to support distributed placement algorithm execution along with dynamic service discovery and load balancing.

Chapter 4 investigated a batch placement technique for the placement of IoT application services with heterogeneous QoS requirements in terms of makespan, budget and throughput. Moreover, the proposed approach dynamically utilises Fog and Cloud resources to ensure the optimum use of computation and network resources. To achieve this, the placement problem is formulated as a Lexicographic Combinatorial Optimisation Problem, considering QoS satisfaction (in terms of makespan, budget, and throughput) as the primary objective and optimum resource usage as the secondary objective. Afterwards, an improved meta-heuristic technique based on Set-based Comprehensive Learning Particle Swarm Optimisation (S-CLPSO) is proposed for the batch placement

of applications. Also, multiple novel heuristic techniques are integrated into the meta-heuristic algorithm to improve its convergence speed and avoid premature convergence to local optima.

Chapter 5 proposed a batch placement technique to improve the reliability satisfaction of mission-critical IoT applications using a throughput-aware proactive redundant placement method. Also, it aims to reduce the cost of deployment as a secondary objective. First, the reliability model of the microservices-based application services is developed as a *k out of n* serial-parallel system, and the placement problem is formulated to capture reliability, throughput awareness, and cost at the composite service level. Also, the failures of the Fog computing environment are modelled to capture both independent and correlated failures of the Fog devices. Next, a hierarchical placement approach is proposed, which consists of two meta-heuristics. Finally, multiple methods, including novel heuristic techniques and fitness functions, are introduced to improve the convergence of the meta-heuristics to an optimum microservice placement.

Chapter 6 proposed a framework for the scalable placement of microservices within federated Fog computing environments. Firstly, the functional and non-functional requirements of the framework were identified, and the framework architecture was proposed to meet the identified requirements. Also, a novel control engine is designed and implemented as a microservice to enable placement policy execution and microservice deployment within multi-fog multi-cloud environments. Next, multiple deployment architectures are provided for major components of the framework to ensure their scalable and fault-tolerant deployments. Finally, the features and performance of the framework are validated by creating a prototype of the framework and integrating multiple microservice placement policies.

These chapters proposed multiple algorithms and systems for optimal placement of microservices to meet the QoS requirements of IoT applications within Fog computing environments, which is a timely contribution to the state-of-the-art.

7.2 Future Research Directions

Based on the research carried out in this thesis, we identify and propose potential future directions for microservices-based IoT application placement in Fog environments.

7.2.1 Dynamic Application Management

To maintain application QoS under the dynamic nature of the Fog infrastructure and fluctuating workloads, application management algorithms must adapt and make decisions accordingly. Online placement algorithms that carry out continuous re-evaluations of application placements can achieve this. Furthermore, algorithms can exploit the independently deployable nature of the microservices, which adds dynamic behaviour to them through auto-scaling, migration, proactive placements, etc. To this end, the placement techniques can benefit from AI-based techniques, such as evolutionary algorithms, ML techniques, and Reinforcement Learning approaches that can adapt to dynamic environmental changes. Application placement algorithms proposed in this thesis, together with dynamic application management algorithms for microservice auto-scaling and migration, can provide holistic approaches to maintain QoS requirements throughout the application lifecycle under dynamic conditions.

7.2.2 Placement within Federated Multi-fog Multi-cloud Environments

Federation among Fog resources provided by multiple Fog infrastructure providers (multi-fog) and multi-cloud environments is emerging as an approach better suited for utilising geo-distributed and resource-constrained Fog computing resources to meet the non-functional requirements of IoT applications. This results in a large-scale distributed environment with Fog and Cloud resources separated by administrative boundaries and geographical locations. The loosely coupled nature of the microservices allows them to span across such environments while maintaining seamless connectivity among them. However, for such scenarios, placement policies need to consider costs, resource availability, security, composition-related overheads, fault-tolerance and fail-over mechanisms, and limitations based on the infrastructure provider and location of the Fog

resources. Moreover, placement policies must evolve towards distributed approaches to handle placement within multi-fog environments governed by multiple infrastructure providers.

7.2.3 Software Frameworks and Platforms for Fog Environments

For the evaluation of placement policies within Cloud environments, commercial platforms such as AWS, Google Cloud, and Microsoft Azure are available. As Fog computing is still in its early stages of industrial adaptation, current research uses small, custom-built test beds. However, they lack support for large-scale experiments, thus failing to capture important aspects related to MSA, such as distributed, location-aware deployments, load balancing, reliability, security and interoperability of services within the large-scale IoT ecosystem. Moreover, they should reflect novel technologies (i.e., container orchestration, service mesh, monitoring tools, overlay networks, etc.). Hence, scalable and extensible frameworks and platforms for Fog environments should be implemented for rapid integration and evaluation of placement policies. We have developed MicroFog as a distributed software framework to enable microservice placement and composition across federated environments. However, this framework can be further extended with lightweight security mechanisms for data transmission across clusters, scalable architectures to store and use observability-related data to improve placement algorithms, and the ability to integrate novel fault-tolerance policies for applications.

7.2.4 IoT Workloads/Benchmarks Related to MSA

The lack of microservices-based IoT workload traces from large-scale deployments and benchmark IoT applications that follow MSA hinder the large-scale evaluation of placement policies. Enterprise workload traces of IoT applications can be used to derive accurate data related to request volumes, patterns, diversity in usage of services, etc. Collecting such data over a long time within large-scale IoT deployments and making them accessible to the research community is significant for improving and accurately evaluating placement and management algorithms.

7.2.5 Security-aware Placement

Data privacy is one of the main concerns of data-driven IoT applications. Distributed deployment of microservices across fog-cloud, along with the vulnerability of open microservice interfaces, poses a considerable security threat to sensitive data transmission and processing. The independently deployable nature of microservices enables the migration of microservices easily across federated Fog environments and between Fog and the Cloud. However, placement algorithms have to incorporate data privacy and security threats related to such migrations in making deployment decisions.

7.2.6 Scalable Placement under state management constraints

Microservices can be stateless or stateful. Stateless microservice do not retain state data between requests, which makes them highly scalable. Stateful microservices persist state data (e.g., in-memory, databases, distributed caching, etc.) to be accessed and used when processing subsequent requests. Thus, the scalability of stateful microservices is constrained by the challenges related to data consistency and state synchronisation across multiple instances, especially across geo-distributed multi-fog environments. Application placement algorithms proposed in this thesis can be further extended to include the scalability constraints related to state maintenance of stateful microservices depending on the state persistence approach used. Moreover, the integration of dynamic application management approaches can further improve QoS in such scenarios by ensuring dynamic memory and storage requirements for retaining state data across instances and considering state synchronisation overheads (i.e., increased network traffic and latency) in real-time.

7.2.7 Resource Contention Handling

Due to the concurrent execution of multiple containers within the same device, resource limitation in the devices and complex interaction patterns of the microservices, resource contention among microservices can affect application performance negatively. The development of intelligent algorithms that can proactively identify resource contentions

during microservice placement, dynamically auto-tune container parameters or migrate containers across the Fog-Cloud continuum has the potential to mitigate this challenge.

7.2.8 Observability and Monitoring Driven Maintenance

Deployment of microservices-based IoT applications creates a distributed system of many microservice instances that can be dynamically created and destroyed. Observability and monitoring can be used to detect performance anomalies within such systems. This requires distributed tracing, monitoring and analysis of the system at both application and platform levels, which would create massive amounts of data of different metrics, logs and traces. This poses a big data analysis challenge where data mining and artificial intelligence models can be integrated with the placement policy to make performance-aware decisions in handling performance anomalies and failures within the system.

7.2.9 Placement within NFV-enabled Networks

Network Function Virtualisation (NFV) has become a key enabler for 5G and 6G mobile networks. NFV virtualises network functions (i.e., firewalls, routers, load balancers, etc.) to provide flexible management and orchestration of network resources, thus supporting IoT applications with large, fluctuating traffic volumes to meet expected QoS levels. To this end, virtualised network functions are developed as containerised microservices that become part of the composite application services (known as service function chains). This requires microservices-based application placement and request routing to be solved in the context of service function chains, where the dynamic placement of containerised network functions becomes a significant part of microservices-based IoT application placement to improve dynamic traffic routing, security, etc., to satisfy the non-functional requirements of the application services.

7.2.10 Fault Tolerant Placement and Management of Microservices

MSA avoids a single point of failure by decomposing applications into loosely coupled microservices. While this improves fault tolerance, it creates more points of failure and also cascading failures due to interacting microservices. Root-cause tracing, predictive redundant placements, awareness of microservice level stability patterns (i.e., circuit breaking, timeouts, retries, etc.) and their effect can be used to generate more resilient placement, migration and request load balancing approaches to improve fault-tolerance of IoT application services.

7.2.11 Availability Assurance under Continuous Integration and Delivery

One of the main reasons for the rising popularity of MSA is its ability to support rapid development and deployment cycles to keep up with the fast-evolving IoT domain. However, this requires smooth integration and deployment of novel updates and changes while minimising service downtime. MSA handles this through multiple deployment strategies such as Canary deployments, Rolling deployments, Blue-Green deployments and A/B testing. To support continuous integration and delivery requirements in an availability-aware manner, continuous placement policies need to be developed, including combinations of multiple deployment strategies supported by MSA.

7.3 Final Remarks

The Fog computing paradigm has emerged as a leading facilitator for IoT-driven solutions spanning a vast range of domains such as healthcare, smart city, intelligent transportation and industrial IoT. Moreover, MSA has risen as an application architecture that can support the rapid development and deployment of IoT applications. Also, MSA is designed with the ability to fully take advantage of the distributed, scalable and flexible nature of the Fog and Cloud resources. The efficient placement of microservices is vital in harvesting the full potential Fog computing paradigm. In this thesis, we investigated and developed algorithms and systems for optimal placement of microservices to meet the Quality of Service (QoS) requirements of IoT applications within Fog computing en-

vironments. The algorithms, mathematical models, and system architectures proposed in the thesis improve services latency, budget satisfaction, throughput, and reliability of the IoT application services while ensuring optimum use of Fog and Cloud resources. Moreover, the research outcomes of this thesis offer opportunities for further innovation and evolution in IoT and Fog computing domains.

Bibliography

[1] S. Bhardwaj and A. Kole, "Review and study of Internet of Things: It's the future," in Proceedings of the 2016 International Conference on Intelligent Control Power and Instrumentation (ICICPI). IEEE, 2016, pp. 47–50.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," Future Generation Computer Systems, vol. 29, no. 7, pp. 1645–1660, 2013.

[3] M. Waseem, P. Liang, and M. Shahin, "A systematic mapping study on microservices architecture in devops," Journal of Systems and Software, vol. 170, p. 110798, 2020.

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

[5] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, "Fog computing: Survey of trends, architectures, requirements, and research directions," IEEE Access, vol. 6, pp. 47 980–48 009, 2018.

[6] The International Market Analysis Research and Consulting Group (IMARC Group), "Microservices Architecture Market: Global Industry Trends, Share, Size, Growth, Opportunity and Forecast 2022-2027," https://www.imarcgroup.com/microservices-architecture-market, June 2022, [Online; accessed Jan-2023].

[7] A. Kaur, R. Kumar, and S. Saxena, "Osmotic computing and related challenges: a survey," in Proceedings of the 2020 Sixth International Conference on Parallel, Distributed and Grid Computing (PDGC). IEEE, 2020, pp. 378–383.

[8] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, "iFogSim2: An extended iFogSim simulator for mobility, clustering, and microservice management in edge and fog computing environments," Journal of Systems and Software, p. 111351, 2022.

[9] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments," Software: Practice and Experience, vol. 47, no. 9, pp. 1275–1296, 2017.

[10] International Data Corporation, "Worldwide Global DataSphere IoT Device and Data Forecast, 2021–2025," July 2021.

[11] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog computing: A taxonomy, survey and future directions," in Internet of Everything. Springer, 2018, pp. 103–130.

[12] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in Proceedings of the first edition of the MCC workshop on Mobile cloud computing, 2012, pp. 13–16.

[13] J. Ren, D. Zhang, S. He, Y. Zhang, and T. Li, "A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet," ACM Computing Surveys (CSUR), vol. 52, no. 6, pp. 1–36, 2019.

[14] A. M. Alqahtani, B. Yosuf, S. H. Mohamed, T. E. El-Gorashi, and J. M. Elmirghani, "Energy Minimized Federated Fog Computing over Passive Optical Networks," in 2021 International Symposium on Networks, Computers and Communications (ISNCC). IEEE, 2021, pp. 1–6.

[15] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," Journal of Systems Architecture, vol. 98, pp. 289–330, 2019.

[16] IoT For All, "The Big Three Make a Play for the Fog," https://www.iotforall.com/big-three-make-play-fog/, 2018, [Online; accessed 20-Jan-2023].

[17] VMware, "VMware Edge Compute Stack," https://www.vmware.com/products/edge-compute-stack.html, 2023, [Online; accessed 24-Jan-2023].

[18] Grand View Research, "Global Edge Computing Market Size, Share Trends Analysis Report by Component (Hardware, Software, Services, Edge-managed Platforms), by Application, by Industry Vertical, by Region, and Segment Forecasts, 2022-2030," June 2022.

[19] S. Newman, Monolith to microservices: evolutionary patterns to transform your monolith. O'Reilly Media, 2019.

[20] C. Richardson, Microservices Patterns. Manning Publications Company, 2018.

[21] M. Fowler and J. Lewis. (2014, March) Microservices a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. [Online; accessed Dec-2022].

[22] C. T. Joseph and K. Chandrasekaran, "Straddling the crevasse: A review of microservice software architecture foundations and recent advancements," Software: Practice and Experience, vol. 49, no. 10, pp. 1448–1484, 2019.

[23] F. Pachinger. (2022, June) Edge Native Applications Are Conquering the Smart Device Edge. https://blogs.cisco.com/developer/smartdeviceedge01. [Online; accessed Dec-2022].

[24] E. Al-Masri, "Enhancing the microservices architecture for the Internet of Things," in Proceedings of the 2018 IEEE International Conference on Big Data (Big Data). IEEE, 2018, pp. 5119–5125.

[25] C. Santana, B. Alencar, and C. Prazeres, "Microservices: A mapping study for Internet of Things solutions," in Proceedings of the 2018 IEEE 17th international symposium on network computing and applications (NCA). IEEE, 2018, pp. 1–4.

[26] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the Internet of Things," in Proceedings of the 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, 2016, pp. 1–6.

[27] M. Luksa, Kubernetes in Action. Simon and Schuster, 2017.

[28] A. Razzaq, "A systematic review on software architectures for IoT systems and future direction to the adoption of microservices architecture," SN Computer Science, vol. 1, no. 6, pp. 1–30, 2020.

[29] L. N. T. Thanh, N. N. Phien, H. K. Vo, H. H. Luong, T. D. Anh, K. N. H. Tuan, H. X. Son et al., "IoHT-MBA: an internet of healthcare things (IoHT) platform based on microservice and brokerless architecture," International Journal of Advanced Computer Science and Applications, vol. 12, no. 7, 2021.

[30] A. De Iasio, A. Futno, L. Goglia, and E. Zimeo, "A microservices platform for monitoring and analysis of IoT traffic data in smart cities," in Proceedings of the 2019 IEEE International Conference on Big Data (Big Data). IEEE, 2019, pp. 5223–5232.

[31] N. Bugshan, I. Khalil, N. Moustafa, and M. S. Rahman, "Privacy-preserving microservices in industrial Internet of Things driven smart applications," IEEE Internet of Things Journal, vol. 10, no. 4, pp. 2821–2831, 2023.

[32] C. Guerrero, I. Lera, and C. Juiz, "Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures," Future Generation Computer Systems, vol. 97, pp. 131–144, 2019.

[33] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards QoS-aware fog service placement," in Proceedings of the 1st IEEE international conference on Fog and Edge Computing (ICFEC). IEEE, 2017, pp. 89–96.

[34] A. Brogi, S. Forti, C. Guerrero, and I. Lera, "How to place your apps in the fog: State of the art and open challenges," Software: Practice and Experience, vol. 50, no. 5, pp. 719–740, 2020.

[35] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," ACM Computing Surveys (CSUR), vol. 53, no. 3, pp. 1–35, 2020.

[36] I. Lera, C. Guerrero, and C. Juiz, "Availability-aware service placement policy in

fog computing based on graph partitions," IEEE Internet of Things Journal, vol. 6, no. 2, pp. 3641–3651, 2018.

[37] J. Paul Martin, A. Kandasamy, and K. Chandrasekaran, "CREW: Cost and Reliability aware Eagle-Whale optimiser for service placement in Fog," Software: Practice and Experience, vol. 50, no. 12, pp. 2337–2360, 2020.

[38] S. Deng, Z. Xiang, J. Taheri, M. A. Khoshkholghi, J. Yin, A. Y. Zomaya, and S. Dustdar, "Optimal application deployment in resource constrained distributed edges," IEEE Transactions on Mobile Computing, vol. 20, no. 5, pp. 1907–1923, 2020.

[39] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 8, pp. 1825–1840, 2021.

[40] F. Faticanti, F. De Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Cutting throughput with the edge: App-aware placement in fog computing," in Proceedings of the 2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom). IEEE, 2019, pp. 196–203.

[41] C. Guerrero, I. Lera, and C. Juiz, "A lightweight decentralized service placement policy for performance optimization in fog computing," Journal of Ambient Intelligence and Humanized Computing, vol. 10, no. 6, pp. 2435–2452, 2019.

[42] F. Faticanti, F. De Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Throughput-aware partitioning and placement of applications in fog computing," IEEE Transactions on Network and Service Management, vol. 17, no. 4, pp. 2436–2450, 2020.

[43] J. L. Herrera, J. Galán-Jiménez, P. Bellavista, L. Foschini, J. Garcia-Alonso, J. M. Murillo, and J. Berrocal, "Optimal Deployment of Fog Nodes, Microservices and SDN Controllers in Time-Sensitive IoT Scenarios," in Proceedings of the 2021 IEEE Global Communications Conference (GLOBECOM). IEEE, 2021, pp. 1–6.

[44] V. Armani, F. Faticanti, S. Cretti, S. Kum, and D. Siracusa, "A Cost-Effective Workload Allocation Strategy for Cloud-Native Edge Services," arXiv preprint arXiv:2110.12788, 2021.

[45] H. Zhao, S. Deng, Z. Liu, x. Yin, and S. Dustdar, "Distributed redundant placement for microservice-based applications at the edge," IEEE Transactions on Services Computing, vol. 15, no. 3, pp. 1732–1745, 2020.

[46] S. Pallewatta, V. Kostakos, and R. Buyya, "QoS-aware placement of microservices-based IoT applications in Fog computing environments," Future Generation Computer Systems, vol. 131, pp. 121–136, 2022.

[47] T. Huang, W. Lin, C. Xiong, R. Pan, and J. Huang, "An ant colony optimization-based multiobjective service replicas placement strategy for fog computing," IEEE Transactions on Cybernetics, vol. 51, no. 11, pp. 5595–5608, 2020.

[48] F. Xu, Z. Yin, A. Gu, F. Zhang, and Y. Li, "A Service Redundancy Strategy and Ant Colony Optimization Algorithm for Multiservice Fog Nodes," in Proceedings of the 2020 IEEE 6th International Conference on Computer and Communications (ICCC). IEEE, 2020, pp. 1567–1572.

[49] C.-H. Hong and B. Varghese, "Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms," ACM Computing Surveys (CSUR), vol. 52, no. 5, pp. 1–37, 2019.

[50] B. Jamil, H. Ijaz, M. Shojafar, K. Munir, and R. Buyya, "Resource Allocation and Task Scheduling in Fog Computing and Internet of Everything Environments: A Taxonomy, Review, and Future Directions," ACM Computing Surveys (CSUR), Jan 2022. [Online]. Available: https://doi.org/10.1145/3513002

[51] M. M. Islam, F. Ramezani, H. Y. Lu, and M. Naderpour, "Optimal Placement of Applications in the Fog Environment: A Systematic Literature Review," Journal of Parallel and Distributed Computing, vol. 174, pp. 46–69, 2023.

[52] R. Mahmud, K. Ramamohanarao, and R. Buyya, "Application management in

fog computing environments: A taxonomy, review and future directions," ACM Computing Surveys (CSUR), vol. 53, no. 4, pp. 1–43, 2020.

[53] M. Goudarzi, M. Palaniswami, and R. Buyya, "Scheduling IoT applications in edge and fog computing environments: a taxonomy and future directions," ACM Computing Surveys (CSUR), vol. 55, no. 7, pp. 1–41, 2022.

[54] P. Varshney and Y. Simmhan, "Characterizing application scheduling on edge, fog, and cloud computing resources," Software: Practice and Experience, vol. 50, no. 5, pp. 558–595, 2020.

[55] M. Garriga, "Towards a taxonomy of microservices architectures," in Proceedings of the International conference on software engineering and formal methods. Springer, 2017, pp. 203–218.

[56] B. Neha, S. K. Panda, P. K. Sahu, K. S. Sahoo, and A. H. Gandomi, "A Systematic Review on Osmotic Computing," ACM Transactions on Internet of Things, vol. 3, no. 2, pp. 1–30, 2022.

[57] D. Andročec, "Systematic mapping study on osmotic computing," in Proceedings of the Central European Conference on Information and Intelligent Systems. Faculty of Organization and Informatics Varazdin, 2019, pp. 79–84.

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

[59] I.-D. Filip, F. Pop, C. Serbanescu, and C. Choi, "Microservices scheduling model over heterogeneous cloud-edge environments as support for IoT applications," IEEE Internet of Things Journal, vol. 5, no. 4, pp. 2672–2681, 2018.

[60] S. Pallewatta, V. Kostakos, and R. Buyya, "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, 2019, pp. 71–81.

[61] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," IEEE Transactions on Mobile Computing, vol. 20, no. 3, pp. 939–951, 2019.

[62] A. Samanta and J. Tang, "Dyme: Dynamic microservice scheduling in edge computing enabled IoT," IEEE Internet of Things Journal, vol. 7, no. 7, pp. 6164–6174, 2020.

[63] J. Fang and A. Ma, "IoT application modules placement and dynamic task processing in edge-cloud computing," IEEE Internet of Things Journal, vol. 8, no. 16, pp. 12 771–12 781, 2020.

[64] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari, and A. Erradi, "Predictive autoscaling of microservices hosted in fog microdata center," IEEE Systems Journal, vol. 15, no. 1, pp. 1275–1286, 2020.

[65] F. Faticanti, M. Savi, F. De Pellegrini, P. Kochovski, V. Stankovski, and D. Siracusa, "Deployment of Application Microservices in Multi-Domain Federated Fog Environments," in Proceedings of the 2020 International Conference on Omni-layer Intelligent Systems (COINS). IEEE, 2020, pp. 1–6.

[66] C. Lei and H. Dai, "A Heuristic Services Binding Algorithm to Improve Fault-Tolerance in Microservice based Edge Computing Architecture," in Proceedings of the 2020 IEEE World Congress on Services (SERVICES). IEEE, 2020, pp. 83–88.

[67] Y. Xu, L. Chen, Z. Lu, X. Du, J. Wu, and P. C. Hung, "An Adaptive Mechanism for Dynamically Collaborative Computing Power and Task Scheduling in Edge Environment," IEEE Internet of Things Journal, vol. 10, no. 4, pp. 3118–3129, 2023.

[68] D. Baburao, T. Pavankumar, and C. Prabhu, "Load balancing in the fog nodes using particle swarm optimization-based enhanced dynamic resource allocation method," Applied Nanoscience, pp. 1–10, 2021.

[69] X. He, Z. Tu, M. Wagner, X. Xu, and Z. Wang, "Online Deployment Algorithms for Microservice Systems with Complex Dependencies," IEEE Transactions on Cloud Computing, 2022.

[70] H. Watanabe, T. Sato, T. Kondo, and F. Teraoka, "AFC: A Mechanism for Distributed Data Processing in Edge/Fog Computing," in Proceedings of the 2021 IEEE Global Communications Conference (GLOBECOM). IEEE, 2021, pp. 01–07.

[71] K. Fu, W. Zhang, Q. Chen, D. Zeng, X. Peng, W. Zheng, and M. Guo, "QoS-aware and resource efficient microservice deployment in cloud-edge continuum," in 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2021, pp. 932–941.

[72] D. Alencar, C. Both, R. Antunes, H. Oliveira, E. Cerqueira, and D. Rosário, "Dynamic microservice allocation for virtual reality distribution with QoE support," IEEE Transactions on Network and Service Management, vol. 19, no. 1, pp. 729–740, 2021.

[73] F. Guo, B. Tang, and M. Tang, "Joint optimization of delay and cost for microservice composition in mobile edge computing," World Wide Web, pp. 1–29, 2022.

[74] W. Lv, Q. Wang, P. Yang, Y. Ding, B. Yi, Z. Wang, and C. Lin, "Microservice deployment in Edge Computing Based on Deep Q Learning," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 11, pp. 2968–2978, 2022.

[75] W. Zhang, Q. Chen, K. Fu, N. Zheng, Z. Huang, J. Leng, and M. Guo, "Astraea: towards QoS-aware and resource-efficient multi-stage GPU services," in Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022, pp. 570–582.

[76] K. Kaur, F. Guillemin, V. Q. Rodriguez, and F. Sailhan, "Latency and network aware placement for cloud-native 5G/6G services," in Proceedings of the 2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC). IEEE, 2022, pp. 114–119.

[77] M. G. Mortazavi, M. H. Shirvani, and A. Dana, "A Discrete Cuckoo Search Algorithm for Reliability-aware Energy-efficient IoT Applications Multi-service Deployment in Fog Environment," in Proceedings of the 2022 International

Conference on Electrical, Computer and Energy Technologies (ICECET). IEEE, 2022, pp. 1–6.

[78] RiSING unit of FBK , "FogAtlas," https://fogatlas.fbk.eu/, [Online; accessed Dec-2022].

[79] S. Yang, Y. Ren, J. Zhang, J. Guan, and B. Li, "KubeHICE: Performance-aware Container Orchestration on Heterogeneous-ISA Architectures in Cloud-Edge Platforms," in 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). IEEE, 2021, pp. 81–91.

[80] C. J. L. de Santana, B. de Mello Alencar, and C. V. S. Prazeres, "Reactive microservices for the Internet of Things: A case study in fog computing," in Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, 2019, pp. 1243–1251.

[81] A. Buzachis, A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and M. Villari, "Towards osmotic computing: Analyzing overlay network solutions to optimize the deployment of container-based microservices in fog, edge and IoT environments," in Proceedings of the 2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC). IEEE, 2018, pp. 1–10.

[82] O. R. C. Rodríguez, C. Pahl, N. El Ioini, H. R. Barzegar et al., "Improvement of edge computing workload placement using multi objective particle swarm optimization," in Proceedings of the 2021 8th International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE, 2021, pp. 1–8.

[83] I. Lera, C. Guerrero, and C. Juiz, "YAFS: A simulator for IoT scenarios in fog computing," IEEE Access, vol. 7, pp. 91 745–91 758, 2019.

[84] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in

Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 3–18.

[85] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," IEEE Transactions on Software Engineering, vol. 47, no. 2, pp. 243–260, 2018.

[86] M. I. Rahman, S. Panichella, and D. Taibi, "A curated dataset of microservices-based systems," SSSME-2019, 2019.

[87] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in Proceedings of the fifteenth European conference on computer systems, 2020, pp. 1–14.

[88] M. Taneja and A. Davy, "Resource aware placement of IoT application modules in Fog-Cloud Computing Paradigm," in Proceedings of the 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). IEEE, 2017, pp. 1222–1228.

[89] R. Mahmud, K. Ramamohanarao, and R. Buyya, "Latency-aware application module management for fog computing environments," ACM Transactions on Internet Technology (TOIT), vol. 19, no. 1, p. 9, 2018.

[90] T. Vresk and I. Čavrak, "Architecture of an interoperable IoT platform based on microservices," in Proceedings of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, 2016, pp. 1196–1201.

[91] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a smart city Internet of Things platform with microservice architecture," in Proceedings of the 3rd International Conference on Future Internet of Things and Cloud. IEEE, 2015, pp. 25–30.

[92] S. Nastic, M. Vögler, C. Inzinger, H.-L. Truong, and S. Dustdar, "rtGovOps: A Runtime Framework for Governance in Large-Scale Software-Defined IoT Cloud Sys-

tems," in Proceedings of the 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. IEEE, 2015, pp. 24–33.

[93] K. Vandikas and V. Tsiatsis, "Microservices in IoT clouds," in Proceedings of the 2016 Cloudification of the Internet of Things (CIoT). IEEE, 2016, pp. 1–6.

[94] D. Santoro, D. Zozin, D. Pizzolli, F. De Pellegrini, and S. Cretti, "Foggy: a platform for workload orchestration in a fog computing environment," in Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2017, pp. 231–234.

[95] P. Hu, S. Dhelim, H. Ning, and T. Qiu, "Survey on fog computing: architecture, key technologies, applications and open issues," Journal of network and computer applications, vol. 98, pp. 27–42, 2017.

[96] M. Slabicki and K. Grochla, "Performance evaluation of CoAP, SNMP and NETCONF protocols in fog computing architecture," in Proceedings of the NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium. IEEE, 2016, pp. 1315–1319.

[97] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," Software: Practice and experience, vol. 41, no. 1, pp. 23–50, 2011.

[98] T. N. Gia, M. Jiang, A.-M. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen, "Fog computing in healthcare internet of things: A case study on ecg feature extraction," in Proceedings of the 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing. IEEE, 2015, pp. 356–363.

[99] Z. Yang, Q. Zhou, L. Lei, K. Zheng, and W. Xiang, "An IoT-cloud based wearable ECG monitoring system for smart healthcare," Journal of Medical Systems, vol. 40, no. 12, p. 286, 2016.

[100] R. Mahmud and R. Buyya, "Modelling and simulation of fog and edge computing environments using iFogSim toolkit," Fog and Edge Computing: Principles and Paradigms, pp. 1–35, 2019.

[101] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers," Software: Practice and Experience, vol. 47, no. 4, pp. 505–521, 2017.

[102] D. T. Nguyen, H. T. Nguyen, N. Trieu, and V. K. Bhargava, "Two-stage robust edge service placement and sizing under demand uncertainty," IEEE Internet of Things Journal, vol. 9, no. 2, pp. 1560–1574, 2021.

[103] R. Mahmud, S. N. Srirama, K. Ramamohanarao, and R. Buyya, "Profit-aware application placement for integrated fog–cloud computing environments," Journal of Parallel and Distributed Computing, vol. 135, pp. 177–190, 2020.

[104] Y. Xie, Y. Zhu, Y. Wang, Y. Cheng, R. Xu, A. S. Sani, D. Yuan, and Y. Yang, "A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloud–edge environment," Future Generation Computer Systems, vol. 97, pp. 361–378, 2019.

[105] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," Journal of Systems Architecture, vol. 98, pp. 289–330, 2019.

[106] A. A. Abdellatif, A. Mohamed, C. F. Chiasserini, M. Tlili, and A. Erbad, "Edge computing for smart health: Context-aware approaches, opportunities, and challenges," IEEE Network, vol. 33, no. 3, pp. 196–203, 2019.

[107] R. Ke, Y. Zhuang, Z. Pu, and Y. Wang, "A smart, efficient, and reliable parking surveillance system with edge artificial intelligence on IoT devices," IEEE Transactions on Intelligent Transportation Systems, vol. 22, no. 8, pp. 4962–4974, 2020.

[108] A. Brogi and S. Forti, "QoS-aware deployment of IoT applications through the fog," IEEE Internet of Things Journal, vol. 4, no. 5, pp. 1185–1192, 2017.

[109] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized IoT service placement in the fog," Service Oriented Computing and Applications, vol. 11, no. 4, pp. 427–443, 2017.

[110] W.-N. Chen and J. Zhang, "A set-based discrete PSO for cloud workflow scheduling with user-defined QoS constraints," in Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, 2012, pp. 773–778.

[111] A. Verma and S. Kaushal, "A hybrid multi-objective particle swarm optimization for scientific workflow scheduling," Parallel Computing, vol. 62, pp. 1–19, 2017.

[112] J. Kennedy and R. Eberhart, "Particle swarm optimization," in Proceedings of the ICNN'95-International Conference on Neural Networks, vol. 4. IEEE, 1995, pp. 1942–1948.

[113] W.-N. Chen, J. Zhang, H. S. Chung, W.-L. Zhong, W.-G. Wu, and Y.-H. Shi, "A novel set-based particle swarm optimization method for discrete optimization problems," IEEE Transactions on Evolutionary Computation, vol. 14, no. 2, pp. 278–300, 2009.

[114] W.-N. Chen and D.-Z. Tan, "Set-based discrete particle swarm optimization and its applications: a survey," Frontiers of Computer Science, vol. 12, no. 2, pp. 203–216, 2018.

[115] J. J. Liang, A. K. Qin, P. N. Suganthan, and S. Baskar, "Comprehensive learning particle swarm optimizer for global optimization of multimodal functions," IEEE Transactions on Evolutionary Computation, vol. 10, no. 3, pp. 281–295, 2006.

[116] AWS, "AWS Fargate Pricing," https://aws.amazon.com/fargate/pricing/, 2021, [Online; accessed Sep-2021].

[117] Azure, "Container Instances pricing," https://azure.microsoft.com/en-au/pricing/details/container-instances/, [Online; accessed Sep-2021].

[118] IBM, "IBM ILOG CPLEX Optimization Studio V12.10.0 documentation," https://www.ibm.com/docs/en/icos/12.10.0?topic=SSSA5P_12.10.0/ilog.odms.studio.help/Optimization_Studio/topics/COS_home.htm, march 2021, [Online; accessed Sep-2021].

[119] O. Grodzevich and O. Romanko, "Normalization and other topics in multi-objective optimization," in Proceedings of the Fields–MITACS Industrial Problems Workshop, 2006.

[120] S. Edirisinghe, C. Ranaweera, C. Lim, A. Nirmalathas, and E. Wong, "Universal optical network architecture for future wireless LANs," Journal of Optical Communications and Networking, vol. 13, no. 9, pp. D93–D102, 2021.

[121] D. Minovski, N. Ogren, C. Ahlund, and K. Mitra, "Throughput Prediction Using Machine Learning in LTE and 5G Networks," IEEE Transactions on Mobile Computing, vol. 22, no. 3, pp. 1825–1840, 2023.

[122] M. Goudarzi, H. Wu, M. Palaniswami, and R. Buyya, "An application placement technique for concurrent IoT applications in edge and fog computing environments," IEEE Transactions on Mobile Computing, vol. 20, no. 4, pp. 1298–1311, 2020.

[123] M. Goudarzi, M. S. Palaniswami, and R. Buyya, "A Distributed Deep Reinforcement Learning Technique for Application Placement in Edge and Fog Computing Environments," IEEE Transactions on Mobile Computing, 2021.

[124] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic Scheduling for Stochastic Edge-Cloud Computing Environments Using A3C Learning and Residual Recurrent Neural Networks," IEEE Transactions on Mobile Computing, vol. 21, no. 3, pp. 940–954, 2020.

[125] E. El Haber, H. A. Alameddine, C. Assi, and S. Sharafeddine, "UAV-aided ultra-reliable low-latency computation offloading in future IoT networks," IEEE Transactions on Communications, vol. 69, no. 10, pp. 6838–6851, 2021.

[126] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial Internet of Things: Challenges, opportunities, and directions," IEEE Transactions on Industrial Informatics, vol. 14, no. 11, pp. 4724–4734, 2018.

[127] Azure, "Move mainframe compute to Azure," https://docs.microsoft.com/en-us/azure/virtual-machines/workloads/mainframe-rehosting/concepts/mainframe-compute-azure, 2021, [Online; accessed Sep-2021].

[128] L. Xing, "Reliability in Internet of Things: Current status and future perspectives," IEEE Internet of Things Journal, vol. 7, no. 8, pp. 6704–6721, 2020.

[129] S. Bagchi, M.-B. Siddiqui, P. Wood, and H. Zhang, "Dependability in edge computing," Communications of the ACM, vol. 63, no. 1, pp. 58–66, 2019.

[130] Z. Bakhshi, G. Rodriguez-Navas, and H. Hansson, "Dependable fog computing: A systematic literature review," in Proceedings of the 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2019, pp. 395–403.

[131] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in Proceedings of the 2018 IEEE 11th international conference on cloud computing (CLOUD). IEEE, 2018, pp. 970–973.

[132] N. Rehani and R. Garg, "Reliability-aware workflow scheduling using monte carlo failure estimation in cloud," in Proceedings of international conference on communication and networks. Springer, 2017, pp. 139–153.

[133] X. Tang, "Reliability-aware cost-efficient scientific workflows scheduling strategy on multi-cloud systems," IEEE Transactions on Cloud Computing, vol. 10, no. 4, pp. 2909–2919, 2021.

[134] X. Zhu, J. Wang, H. Guo, D. Zhu, L. T. Yang, and L. Liu, "Fault-tolerant scheduling for real-time scientific workflows with elastic resource provisioning in virtualized clouds," IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 12, pp. 3501–3517, 2016.

[135] G. Yao, X. Li, Q. Ren, and R. Ruiz, "Failure-aware Elastic Cloud Workflow Scheduling," IEEE Transactions on Services Computing, 2022.

[136] J. Yao and N. Ansari, "Fog resource provisioning in reliability-aware IoT networks," IEEE Internet of Things Journal, vol. 6, no. 5, pp. 8262–8269, 2019.

[137] J. Liu, A. Zhou, C. Liu, T. Zhang, L. Qi, S. Wang, and R. Buyya, "Reliability-enhanced task offloading in mobile edge computing environments," IEEE Internet of Things Journal, vol. 9, no. 13, pp. 10 382–10 396, 2021.

[138] A. Aral and I. Brandić, "Learning Spatiotemporal Failure Dependencies for Resilient Edge Computing Services," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 7, pp. 1578–1590, 2020.

[139] A. M. Maia, Y. Ghamri-Doudane, D. Vieira, and M. F. de Castro, "Optimized placement of scalable IoT services in edge computing," in Proceedings of the 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). IEEE, 2019, pp. 189–197.

[140] A. M. Maia, Y. GhamriDoudane, D. Vieira, and M. F. de Castro, "Dynamic service placement and load distribution in edge computing," in Proceedings of the 2020 16th International Conference on Network and Service Management (CNSM). IEEE, 2020, pp. 1–9.

[141] W. R. Blischke and D. P. Murthy, Reliability: modeling, prediction, and optimization. John Wiley & Sons, 2011, vol. 767.

[142] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," IEEE Transactions on Dependable and Secure Computing, vol. 7, no. 4, pp. 337–350, 2009.

[143] P. Garraghan, P. Townend, and J. Xu, "An empirical failure-analysis of a large-scale cloud computing environment," in 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering. IEEE, 2014, pp. 113–120.

[144] E. Zio, "Monte carlo simulation: The method," in The Monte Carlo simulation method for system reliability and risk analysis. Springer, 2013, pp. 19–58.

[145] M. Kijima, "Some results for repairable systems with general repair," Journal of Applied probability, vol. 26, no. 1, pp. 89–102, 1989.

[146] A. Mettas and W. Zhao, "Modeling and analysis of repairable systems with general repair," in Annual Reliability and Maintainability Symposium, 2005. Proceedings. IEEE, 2005, pp. 176–182.

[147] M. Yanez, F. Joglar, and M. Modarres, "Generalized renewal process for analysis of repairable systems with limited failure experience," Reliability Engineering & System Safety, vol. 77, no. 2, pp. 167–180, 2002.

[148] M. Tanwar, R. N. Rai, and N. Bolia, "Imperfect repair modeling using Kijima type generalized renewal process," Reliability Engineering & System Safety, vol. 124, pp. 24–31, 2014.

[149] W. do Espírito Santo, R. d. S. M. Júnior, A. d. R. L. Ribeiro, D. S. Silva, and R. Santos, "Systematic mapping on orchestration of container-based applications in fog computing," in Proceedings of the 2019 15th International Conference on Network and Service Management (CNSM). IEEE, 2019, pp. 1–7.

[150] P. Farzin, S. Azizi, M. Shojafar, O. Rana, and M. Singhal, "FLEX: a platform for scalable service placement in multi-fog and multi-cloud environments," in Australasian Computer Science Week 2022, 2022, pp. 106–114.

[151] Q. Deng, M. Goudarzi, and R. Buyya, "Fogbus2: a lightweight and distributed container-based framework for integration of IoT-enabled systems with edge and cloud computing," in Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, 2021, pp. 1–8.

[152] "Google", ""google distributed cloud edge overview"," https://cloud.google.com/distributed-cloud/edge/latest/docs/overview, [Online; accessed Feb-2023].

[153] M. Komu and T. Kauppinen, "Enhancing service mobility in the 5G edge cloud and beyond," https://www.ericsson.com/en/blog/2022/11/service-mobility-in-the-edge-cloud, [Online; accessed Feb-2023].

[154] "IBM", ""edge clusters"," https://www.ibm.com/docs/en/eam/4.2?topic=nodes-edge-clusters, [Online; accessed Feb-2023].

[155] P. Farhat, H. Sami, and A. Mourad, "Reinforcement R-learning model for time scheduling of on-demand fog placement," The Journal of Supercomputing, vol. 76, pp. 388–410, 2020.

[156] D. Ermolenko, C. Kilicheva, A. Muthanna, and A. Khakimov, "Internet of Things services orchestration framework based on Kubernetes and edge computing," in Proceedings of the 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus). IEEE, 2021, pp. 12–17.

[157] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over raspberrypi," in Proceedings of the 18th international conference on distributed computing and networking, 2017, pp. 1–10.

[158] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "Fogbus: A blockchain-based lightweight framework for edge and fog computing," Journal of Systems and Software, vol. 154, pp. 22–36, 2019.

[159] Z. Wang, M. Goudarzi, J. Aryal, and R. Buyya, "Container orchestration in edge and fog computing environments for real-time IoT applications," in Computational Intelligence and Data Analytics: Proceedings of ICCIDA 2022. Springer, 2022, pp. 1–21.

[160] R. Mahmud and A. N. Toosi, "Con-Pi: A distributed container-based edge and fog computing framework," IEEE Internet of Things Journal, vol. 9, no. 6, pp. 4125–4138, 2021.

[161] J. "Falkner, ""key findings from idc red hat quarkus lab validation"," https://www.redhat.com/en/blog/key-findings-idc-red-hat-quarkus-lab-validation, October 2020, [Online; accessed Oct-2022].