The Journal of Systems and Software 209 (2024) 111910

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

MicroFog: A framework for scalable placement of microservices-based IoT applications in federated Fog environments☆

Samodha Pallewatta *, Vassilis Kostakos, Rajkumar Buyya

The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

ARTICLE INFO

Keywords:
Fog computing
Microservices
Application placement
Internet of things
Microservice composition

ABSTRACT

MicroService Architecture (MSA) is gaining rapid popularity for developing large-scale IoT applications for deployment within distributed and resource-constrained Fog computing environments. As a cloud-native application architecture, the true power of microservices comes from their loosely coupled, independently deployable and scalable nature, enabling distributed placement and dynamic composition across federated Fog and Cloud clusters. Thus, it is necessary to develop novel placement algorithms that utilise these microservice characteristics to improve the performance of the applications. However, existing Fog computing frameworks lack support for integrating such placement policies due to their shortcomings in multiple areas, including MSA application placement and deployment across multi-fog multi-cloud environments, dynamic microservice composition across multiple distributed clusters, scalability of the framework to operate within federated environments, support for deploying heterogeneous microservice applications, etc. To this end, we design and implement MicroFog, a Fog computing framework compatible with cloud-native technologies such as Docker, Kubernetes and Istio. MicroFog provides an extensible and configurable control engine that executes placement algorithms and deploys applications across federated Fog environments. Furthermore, MicroFog provides a sufficient abstraction over container orchestration and dynamic microservice composition, thus enabling users to easily incorporate new placement policies and evaluate their performance. The capabilities of the MicroFog framework, such as the scalability and flexibility of the design and deployment architecture of MicroFog and its ability to ensure the deployment and composition of microservices across distributed fog–cloud environments, are validated using multiple use cases. Experiments also demonstrate MicroFog's ability to integrate and evaluate novel placement policies and load-balancing techniques. To this end, we integrate multiple microservice placement policies to demonstrate MicroFog's ability to support horizontally scaled placement, service discovery and load balancing of microservices across federated environments, thus reducing the application service response time up to 54%.

1. Introduction

The Internet of Things (IoT) is growing rapidly, and the ever-increasing number and variety of connected devices generate massive amounts of data related to a wide range of smart application domains, such as smart cities, smart healthcare, Industrial IoT, and smart transportation, to mention a few. Hence, IoT application development is adapting Microservices Architecture (MSA) to support the rapid evolution of IoT application development towards creating an IoT ecosystem. Being a cloud-native application architecture, MSA builds applications as collections of modules known as microservices that are independently deployable and scalable (Fowler and Lewis, 2014). 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.

Meanwhile, Fog computing is emerging as a powerful distributed computing paradigm for hosting latency-critical and bandwidth-hungry IoT applications. Fog computing extends cloud-like services towards the edge of the network by using the computing, networking and storage resources residing within the path connecting IoT devices to the centralised Cloud data centres (Mahmud et al., 2018). With the increasing use of IoT applications, sending large amounts of data towards the centralised Cloud incurs high latency and bandwidth congestion. Moreover, distributed Fog resources provide the location awareness, data security, mobility awareness, and scalability required by the IoT

☆ Editor: J.C. Duenas.

* Corresponding author.
E-mail address: ppallewatta@student.unimelb.edu.au (S. Pallewatta).

https://doi.org/10.1016/j.jss.2023.111910
Received 11 February 2023; Received in revised form 18 June 2023; Accepted 20 November 2023
Available online 30 November 2023

applications, with geo-distributed users seeking ubiquitous access to the application services (Goudarzi et al., 2022).

While the use of distributed Fog computing resources is a solution for this, the resource-constrained nature of the Fog resources is the main drawback which 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 (Santo et al., 2019). 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 (Farzin et al., 2022). 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 (Neha et al., 2022).

To harvest the full potential of MSA in Fog computing environments, the development of efficient placement algorithms is of vital importance. Thus, research on designing, developing and evaluating algorithms for the placement of microservice-based IoT applications is attracting a lot of attention. Existing literature contains works focusing on horizontally scaled placement of microservices to meet QoS parameters such as throughput, reliability and latency (Faticanti et al., 2019; Guerrero et al., 2019a; Deng et al., 2020; Pallewatta et al., 2022b), location-aware placements (Guo et al., 2022), 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 (Mahmud et al., 2022).

Evaluation of the placement policies can be conducted using numerical evaluations (Guo et al., 2022; Guerrero et al., 2019a), simulators (Fang and Ma, 2020; Pallewatta et al., 2022b; Paul Martin et al., 2020) and real-world deployments through small-scale testbeds (Faticanti et al., 2019; Fu et al., 2021). Cloud computing-related policy evaluation can be conducted using Cloud infrastructure provided by commercial service providers like Amazon AWS, Google Cloud, etc., through their Infrastructure as a Service (IaaS) offerings through a rental model. Due to the lack of such platforms for Fog computing, Fog application placement policies are primarily evaluated using numerical evaluations and simulators.

However, several frameworks are proposed in the literature to evaluate placement policies through real-world deployments (Deng et al., 2021; FogAtlas, 2023; Santoro et al., 2017; Bellavista and Zanni, 2017). These frameworks enable the creation of small-scale testbeds for the management of Fog and Cloud resources. Many of the frameworks support containerised application deployment using container orchestration platforms such as Docker Swarm (Bellavista and Zanni, 2017) and Kubernetes (FogAtlas, 2023; Santoro et al., 2017; Wang et al., 2022). Several works use service mesh technologies like Istio (Ruuskanen et al., 2021; Marchese and Tomarchio, 2023) on top of Kubernetes to support microservices. However, they have limitations related to the dynamic and automated deployment of Microservices-based IoT applications across federate Fog environments. Existing frameworks 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 placement 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 paper is organised as follows. In Section 2, we provide a comprehensive background on microservices-based application placement and analyse related research. Section 3 derives the requirements of the framework based on the background and introduces the MicroFog framework, and Section 4 details the deployment architectures for the main components of the framework. APIs to access MicroFog-CE are presented in Section 5. Features of the framework are evaluated in Section 6. Finally, Section 7 concludes the paper.

2. Background and related works

In this section, we present a comprehensive background on the Fog computing paradigm, microservices-based applications and how to model them, and deployment-related aspects of the microservice applications including use of cloud-native technologies for their deployment and composition. We also discuss Fog application placement problem. Moreover, we provide a qualitative comparison of existing frameworks to highlight the capabilities of our proposed framework. These related background concepts will be used in later sections to derive requirements of the framework and to make design and implementation decisions of our proposed framework.

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 (Mahmud et al., 2018). With the rapid growth in IoT applications, Fog computing is evolving towards a federated multi-fog multi-cloud architecture (Farzin et al., 2022) where multiple FIPs provide infrastructure, including computing, storage and networking resources within the Fog layer. In this work, we consider the existence of multiple such Fog

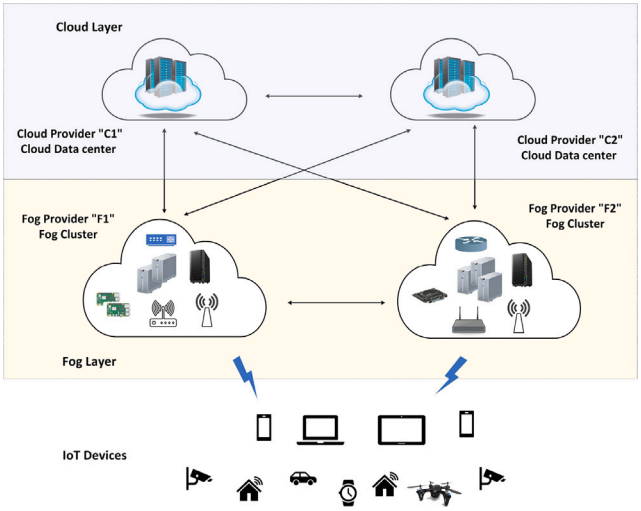

Fig. 1. Federated multi-fog and multi-cloud architecture.

clusters provided by various service providers where they maintain connectivity with neighbouring clusters and the Cloud. Fig. 1 depicts the architecture of the federated multi-fog and multi-cloud environment considered in our work.

2.2. Microservices-based applications

This section discusses the Microservice Architecture and how to model a microservices-based application based on its architectural characteristics. MicroService Architecture (MSA) decomposes an application into a set of independently deployable modules known as microservices designed around business logic to have well-defined business boundaries (Joseph and Chandrasekaran, 2019). Microservices communicate with each other using lightweight APIs to create composite services that the end users access.

As microservices-based applications have interactions among microservices, they can be modelled using Directed Acyclic Graphs (DAGs) (Pallewatta et al., 2022b) 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 $\langle M_a, df^a, S_a \rangle$ 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; $\langle M_a^s, P_a^s, Req_a \rangle$. 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 microservices within the composite service (i.e., the chained pattern has a single data path, whereas the aggregator invokes multiple datapaths).

2.3. Application deployment related aspects

The loosely coupled nature of these microservices enables them to be deployed and scaled independently within distributed environments. Thus, MSA require dynamic service discovery and load-balancing mechanisms to ensure seamless connectivity among microservices deployed dynamically across distributed multi-fog multi-cloud environments. To this end, Microservices-based application deployment and management are aided by three cloud-native technologies: containerisation platforms (i.e., Docker, LXC), 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 in the following sections.

2.3.1. Containerisation using Docker

Microservices are packaged as 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.

While there are multiple containerisation technologies in use (e.g., Docker, LXC, etc.), Docker has gained rapid popularity for the deployment of microservices-based enterprise applications. Being an "application-centric container technology", Docker supports higher portability, higher scalability, lightweight container image creations, etc. As the focus of MicroFog is to support rapidly growing microservices-based IoT application deployment, we use Docker as the containerisation technology.

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 is important to explore distributed container image registries, depending on the resource availability of the Fog infrastructure to host the registry.

2.3.2. 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 (Google, 2023; Ericsson, 2023; IBM, 2023). 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 decouples service end-point from IP addresses of the individual pods. Use of Services provide dynamic 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.)

2.3.3. 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.

2.3.4. 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 Fig. 2) (Pallewatta et al., 2022b) 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.

Fig. 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 towards 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 end-point 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.

2.3.5. 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 networks with administrative boundaries where each cluster is exposed through load balancers. Fog computing architecture considered in this work (Section 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.

2.4. Placement problem

Microservice-based IoT application placement problem within Fog environments addresses deployment and maintenance of microservices within federated Fog and Cloud environments to meet the Service Level Agreements (SLA) of the application services (Guerrero et al., 2019a; Skarlat et al., 2017). To this end, placement algorithms are developed to consider resource requirements of the microservices (i.e., CPU, RAM, Storage, Bandwidth) and map them to available Fog or Cloud resources while ensuring the satisfaction of QoS parameters such as makespan, budget, reliability, availability, and throughput of the application services.

Furthermore, due to the flexibility provided by the microservices architecture (i.e., independently deployable and scalable nature of microservices), placement algorithms aim to incorporate horizontal scalability to meet throughput requirements (Guerrero et al., 2019a; Deng et al., 2020; Pallewatta et al., 2022b), location-aware distribution (Guo et al., 2022), redundant placement to improve reliability (Xu et al., 2020), balanced placement across Fog clusters and Cloud depending on service discovery capabilities (Guerrero et al., 2019b; Pallewatta et al., 2022b), optimum load balancing and routing (Herrera et al., 2021), etc. to efficiently utilise limited Fog resources while satisfying QoS parameters.

Execution of placement algorithms can take place as batch placements (Samanta and Tang, 2020; Pallewatta et al., 2022b) that process multiple application placement requests at once or sequential placements (Lera et al., 2018; Guerrero et al., 2019b) where queued placement requests are processed one after the other. Moreover, the

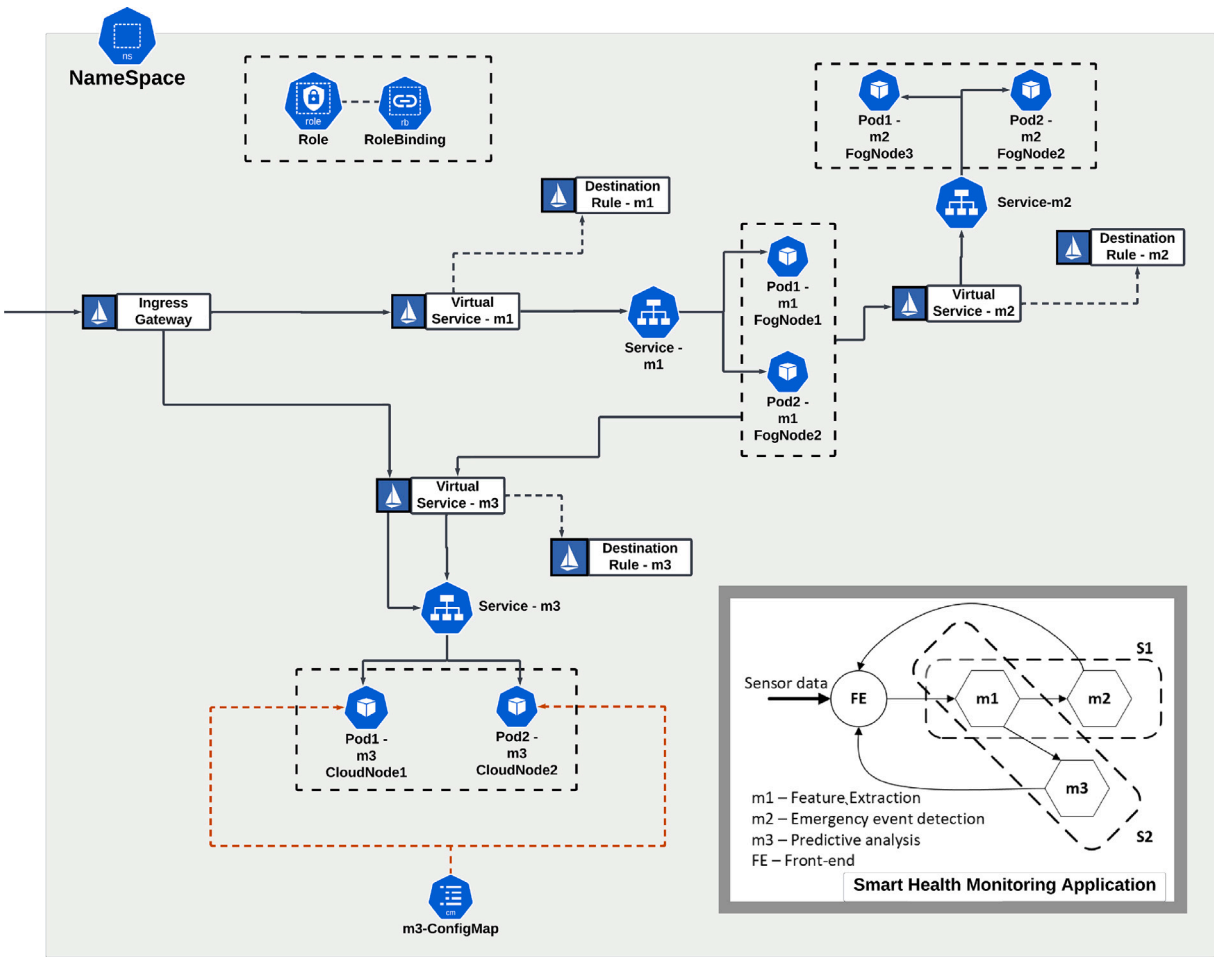

Fig. 2. Example deployment of a Smart Health Monitoring Application.

placement policies can be developed as centralised (Farhat et al., 2020) or distributed (Pallewatta et al., 2019) algorithms to achieve placement across distributed Fog and Cloud resources provided by multiple infrastructure providers.

2.5. Existing fog frameworks

In this section, we compare existing Fog frameworks qualitatively based on their supported features related to microservice-based IoT application deployment within federated fog environments (see Table 1).

Yousefpour et al. (2019) present "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 hosting the data stores, provisioning Fog services and deploying them within Fog nodes. Santoro et al. (2017) provide an open-source technology-based (i.e., OpenStack, Kubernetes, Docker) platform named Foggy for workload placement in Fog computing environments. FogAtlas (2023) extends Foggy platform by extending Kubernetes to orchestrate distributed Fog and Cloud resources in a user-friendly manner. Ermolenko et al. (2021) also propose a framework based on Kubernetes and Docker where a Kubernetes cluster is deployed within a Mobile Edge Computing (MEC) environment. Bellavista and Zanni (2017) 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. (2019) introduced 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. (2021) proposed 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. (2022) improved 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 support 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 and Toosi (2021) 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. Marchese and Tomarchio (2023) propose a framework for microservice orchestration in the Cloud-to-Edge continuum using

Table 1
Comparison of existing frameworks.

Work	Architecture		Cloud-native application support					Microservice composition support					Control-engine			Data stores
	Integration	Multi-cluster	μservices	Containers	Container orchestration	Service mesh	Automated deployment	Service discovery: Avail.	Service discovery: Cross-cluster	Load balancing: Avail.	Load balancing: Configurable	Load balancing: Cross-cluster	Extensibility	Scalability	Configurability	
Yousefpour et al. (2019)	Fog, Cloud	–	✓	✓	–	–	∂	∂	–		–	–	✓	∂	∂	Centralised
Santoro et al. (2017)	Fog, Cloud	–	✓	✓	✓	–	∂	✓(Kubernetes)	–	✓(Kubernetes)	–	–	✓	∂	∂	Distributed
FogAtlas (2023)	Fog, Cloud	–	✓	✓	✓	–	∂	✓(Kubernetes)	–	✓(Kubernetes)	–	–	✓	∂	∂	–
Ermolenko et al. (2021)	Edge	–	✓	✓	✓	–	–	✓	–	✓	–	–	✓	∂	∂	–
Bellavista and Zanni (2017)	Fog, Cloud	–	✓	✓	✓	–	∂	✓(Docker Swarm)	–	–	–	–	∂	∂	∂	–
Tuli et al. (2019)	Fog, Cloud	–	–	–	–	–	∂	–	–	–	–	–	∂	∂	∂	Distributed
Deng et al. (2021) and Wang et al. (2022)	Fog, Cloud	–	∂	✓	∂	–	∂	∂ (Proxy Server)	–	–	–	–	∂	∂	∂	Distributed
Mahmud and Toosi (2021)	Fog, Cloud	–	✓	✓	–	–	∂	∂	–	–	–	–	✓	∂	∂	Centralised
Ruuskanen et al. (2021)	Fog, Cloud	✓	✓	✓	✓	✓	–	✓ (Kubernetes, Istio)	∂	✓ (Istio)	∂	∂	–	–	–	–
Marchese and Tomarchio (2023)	Fog, Cloud	–	✓	✓	✓	✓	∂	✓ (Kubernetes, Istio)	–	✓ (Istio)	–	–	∂	∂	–	–
Our	Fog, Cloud	✓	✓	✓	✓	✓	✓	✓ (Kubernetes, Istio)	✓	✓ (Istio)	✓	✓	✓	✓	✓	Distributed, Replicated Fault-tolerant

✓: Supported by the framework, ∂: Partially supported.

Docker, Kubernetes and Istio service mesh and extends Kubernetes scheduler to support network aware scheduling. However, their architecture does not consider multi-cluster scenarios. Thus, their work does not use service mesh for microservice orchestration across clusters and does not automate application deployment within federated fog environments. Ruuskanen et al. (2021) create an open-source sandbox for deploying virtual clusters where Kubernetes is used for container orchestration and Istio is used for inter-cluster communication. However the aim of their work is to create an emulated federated cloud environment. Thus, application deployment related aspects are out of scope for their work and does not provide a configurable and scalable controller to abstract and automate the application deployment across multi-cluster environment.

Based on the qualitative analysis provided in Table 1, existing frameworks have limitations in multiple areas such as multi-fog multi-cloud placement, 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.

3. MicroFog framework

In this section, we discuss the functional and non-functional requirements addressed by the proposed MicroFog framework, its high-level architecture, main components and workflow to highlight how MicroFog meets the requirements identified in Section 3.1.

3.1. Framework requirements

Based on the background analysis, 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.

3.2. High-level architecture

Fig. 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

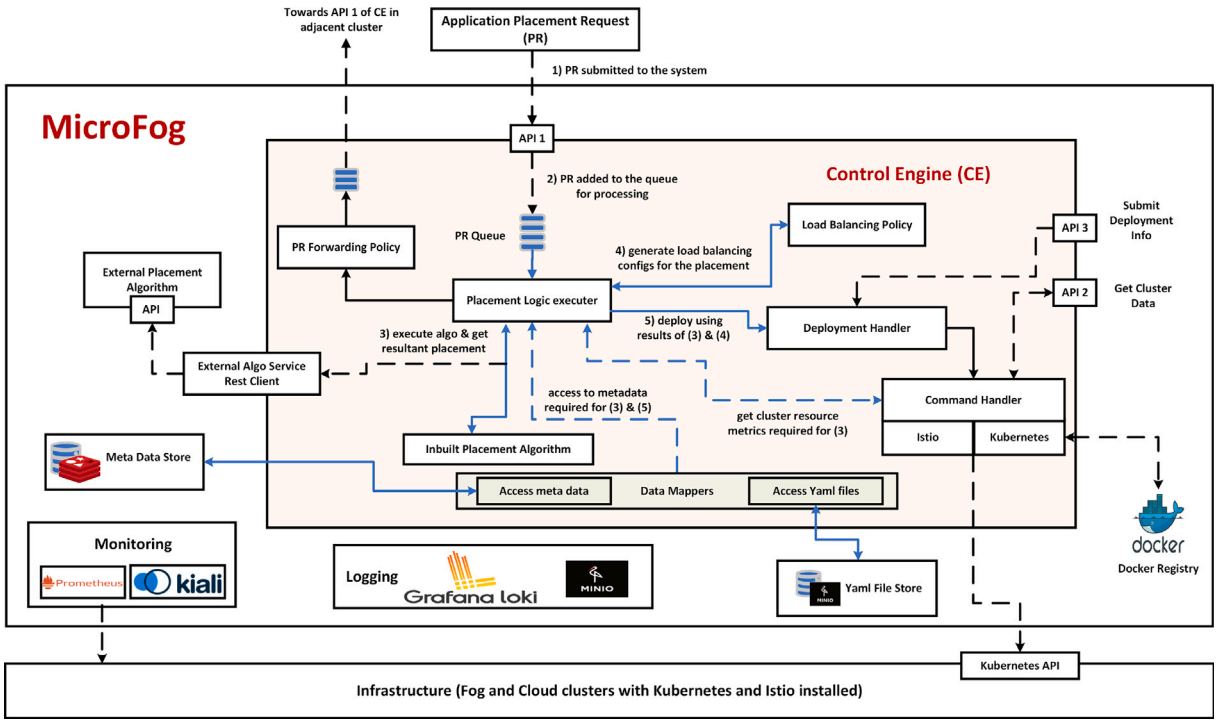

Fig. 3. MicroFog: High-level architecture.

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), process them according to a selected placement policy (either an inbuilt placement algorithm or external algorithm accessed through an API), configure related Kubernetes and Istio YAML files according to the generated placement and the load balancing policy, and finally deploy 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.

3.3. Main components and technologies

3.3.1. 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 (Falkner, 2020), which is suitable for developing microservices for deployment 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 through an API which expects HTTP POST requests with the PRs represented in JSON format (API 1 shown in Fig. 3). Each submitted PR can define multiple data fields related to the application, including application id, QoS parameters, any restrictions for application placement, traffic entry clusters, etc. 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 (Fig. 4). 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

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

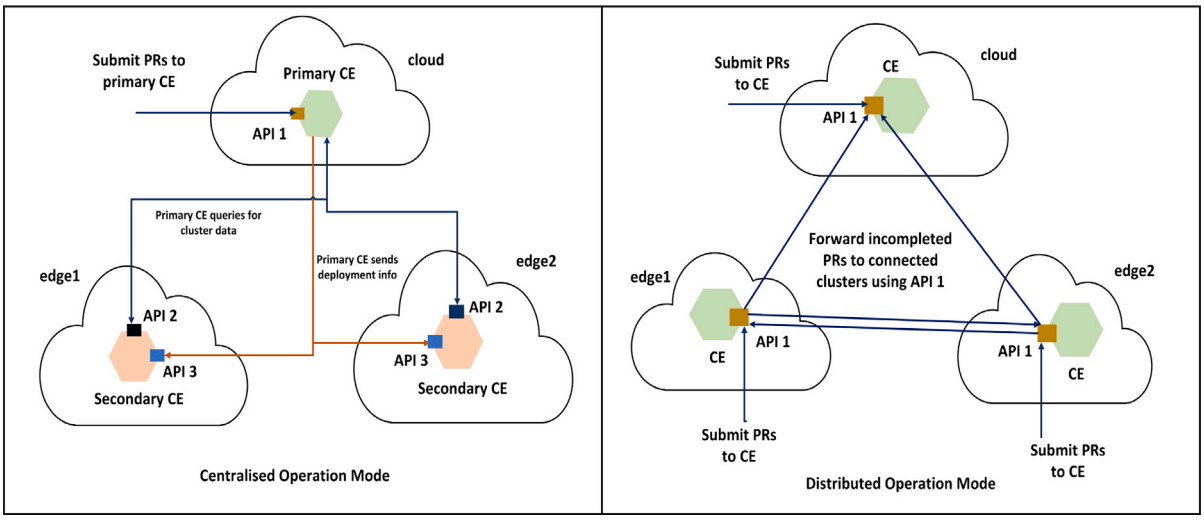

Fig. 4. CE operation modes.

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* Fig. 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 Pallewatta et al. (2019). 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. Fig. 5 domain diagram used in developing the MicroFog-CE, which adheres with the system models and placement problem formulated in Section 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.).

3.3.2. 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 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 performance. 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.

3.3.3. 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 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.

3.3.4. 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.

[3] https://redis.io/.
[4] https://min.io/.

3.4. PR processing flow of MicroFog-CE

In this section, we discuss the high-level pseudo-code (see Algorithm 1) 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 in Fig. 5) 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.

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 (see Figs. 6 and 7) 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.

4.1. MinIO YAML File Store deployment

We provide an example deployment scenario in Fig. 6 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 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 is possible to extend it to three tiers by implementing Istio sub-zones as well.

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

Algorithm 1 MicroFog-CE PR processing

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

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.

4.3. Control-engine deployment

Fig. 7 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 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.

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 4.1), logs can be directed either to a central Cloud or stored within the own cluster depending on the resource availability.

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 Fig. 8): 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"*,

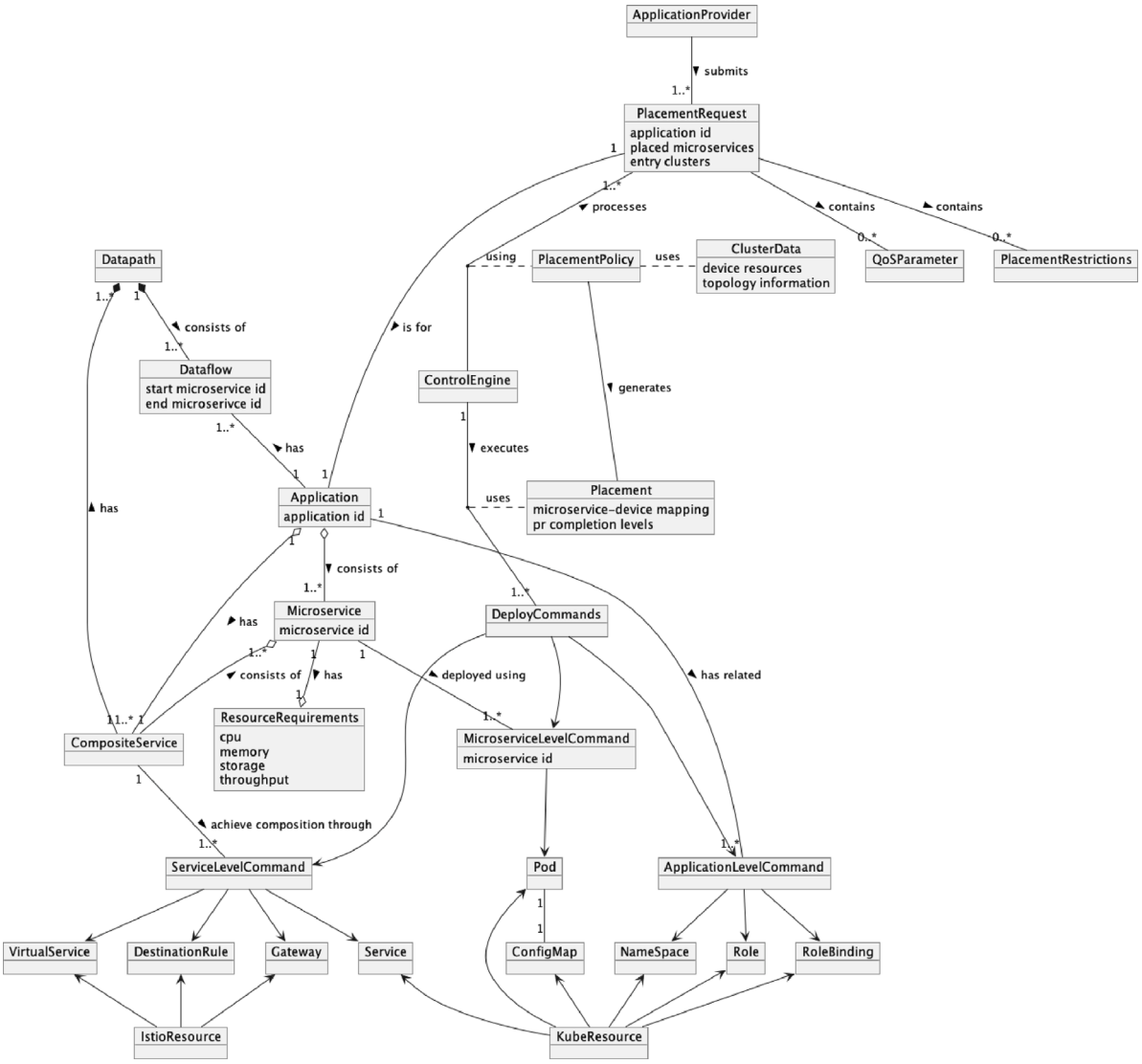

Fig. 5. MicroFog: Domain diagram for control engine.

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 (Pallewatta et al., 2022a). *"qosParameters"* field allows detailed parameter definitions adhering to this.

- API 2 (see Fig. 9): 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 Fig. 10): 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.

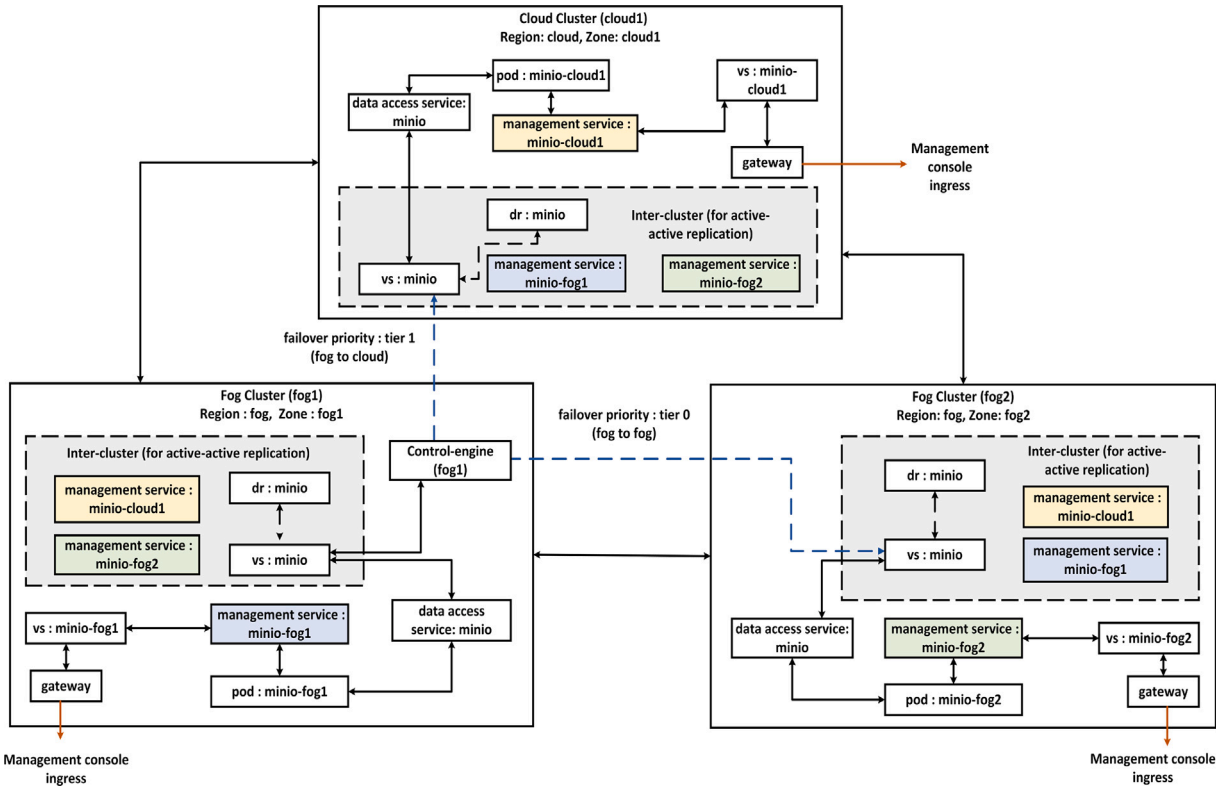

Fig. 6. MinIO - YAML File Store - Deployment.

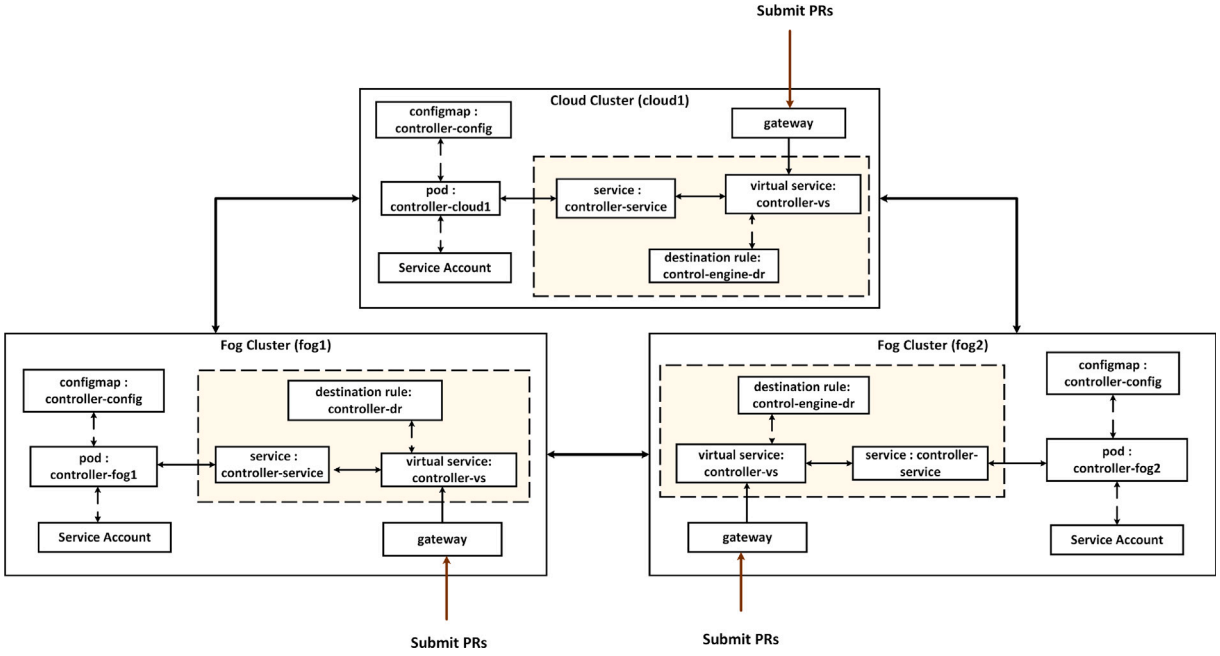

Fig. 7. Distributed control engine deployment.

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. MicroFog - Evaluation and validation

In this section, we validate the main features and functions supported by MicroFog using multiple use cases.

6.1. Experimental setup

6.1.1. 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). Each cluster belongs to a separate network and communicates with each

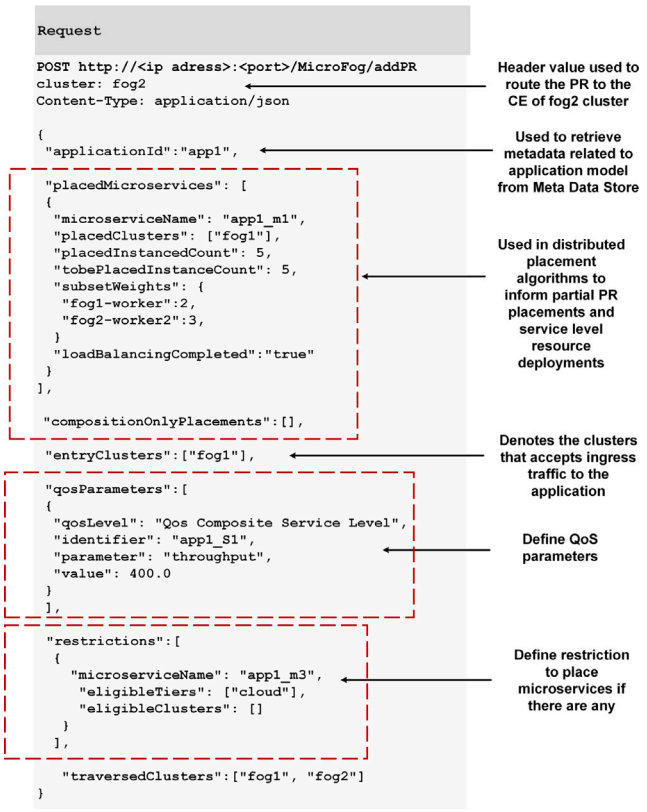

Fig. 8. API 1 - For submitting PRs.

Fig. 10. API 3 - For submitting placement output for deployment.

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]
}
```

Fig. 9. API 2 - For querying cluster information.

other through load balancers. For the prototype, we use MetalLB[5] as the load balancer that exposes each cluster to the outside. Each 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 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 replicate 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.

6.1.2. 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. The tool provides a base microservice as a template that can be configured (using a Kubernetes ConfigMap) to create microservices that have multiple interaction patterns among them

[5] https://metallb.universe.tf/.
[6] https://github.com/Cloudslab/MicroFog/tree/main/Workload_Generator.

Table 2
Federated fog–cloud infrastructure setup.

Cluster details	Resources	
	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

(i.e., chained, aggregate, or microservice candidate patterns) to create microservices-based applications having composite services that the users can access. 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 applications. 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.

6.1.3. Placement algorithm

To highlight the main features supported by MicroFog, we adapt and implement different variations of the placement algorithm proposed in Pallewatta et al. (2019). The algorithm in Pallewatta et al. (2019) 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 Pallewatta et al. (2019) 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 Pallewatta et al. (2022b) 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 does not 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 cannot 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.

6.2. Use cases and results

6.2.1. 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 environments. 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 paper 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 Fig. 11(a) and (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 Fig. 6. 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.

- **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 proposed deployment architecture (Section 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 Fig. 7 routes to the correct destination based on that. Fig. 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 Fig. 13, we consider three placement scenarios and analyse time to application 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. Fig. 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.
- **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.

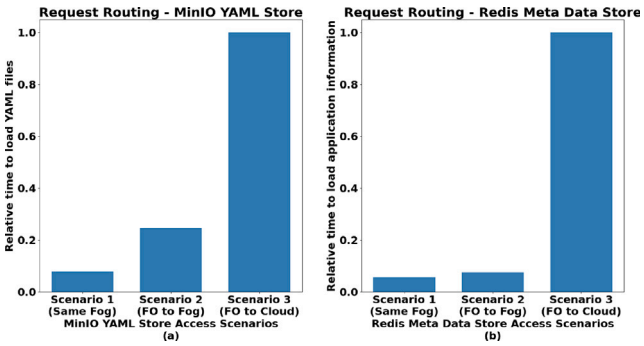

Fig. 11. Availability analysis of data stores.

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 (see Fig. 14).

6.2.2. 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 2.3.4 (see Fig. 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 Fig. 15). 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

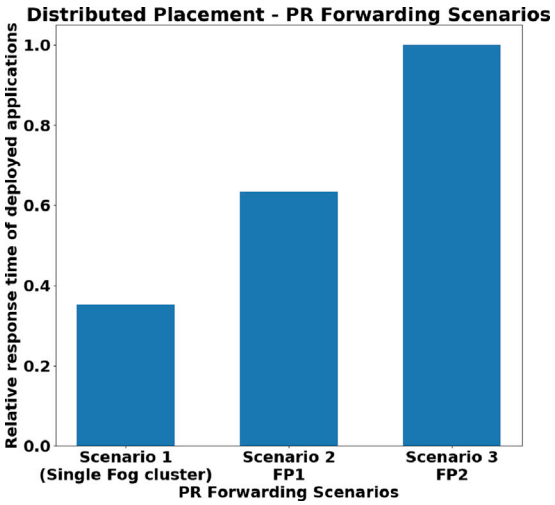

Fig. 12. Distributed placement algorithm execution.

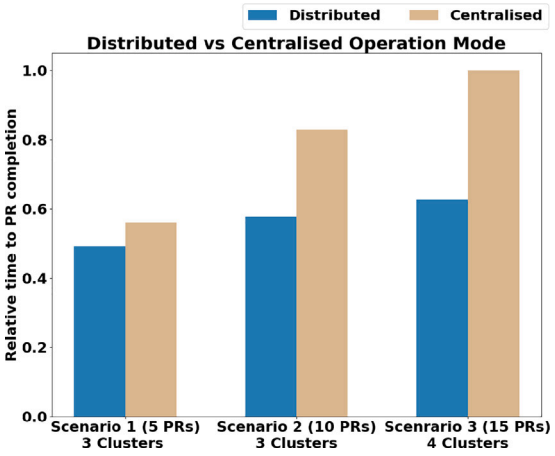

Fig. 13. Analysis of CE operation modes.

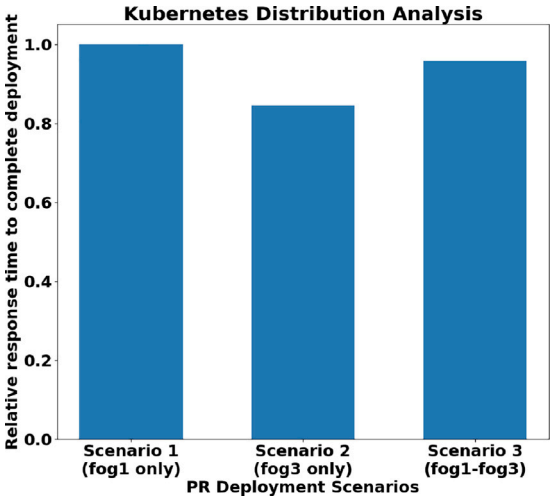

Fig. 14. Analysis of Kubernetes distributions.

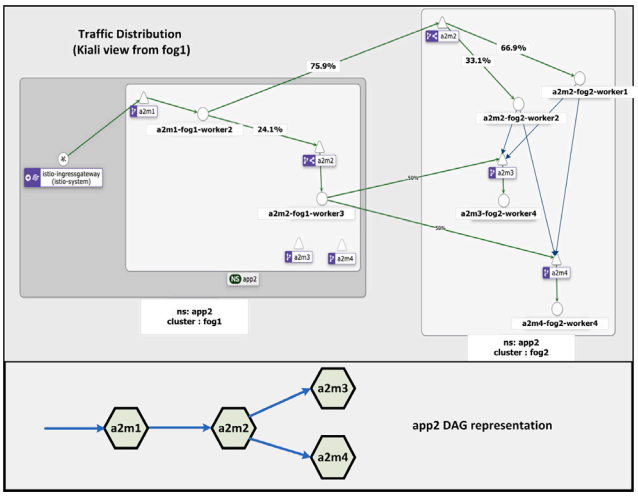

Fig. 15. Multi-cluster service discovery and load balancing scenario - app2.

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 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 Fig. 17. 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 Fig. 17. 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 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 Fig. 15) shows that traffic for a2m2 is divided with a 1:3 ratio among two clusters and 2:1 within the fog2 cluster, thus dividing 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 Fig. 16, which reflects the traffic distribution 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).

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.

Table 3
Generated placement for example applications (app2 and hcapp).

Placement algorithm	app2		hcapp	
	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

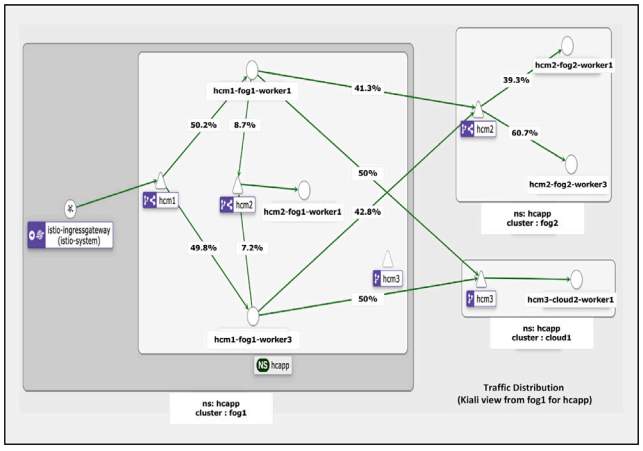

Fig. 16. Multi-cluster service discovery and load balancing scenario - hcapp.

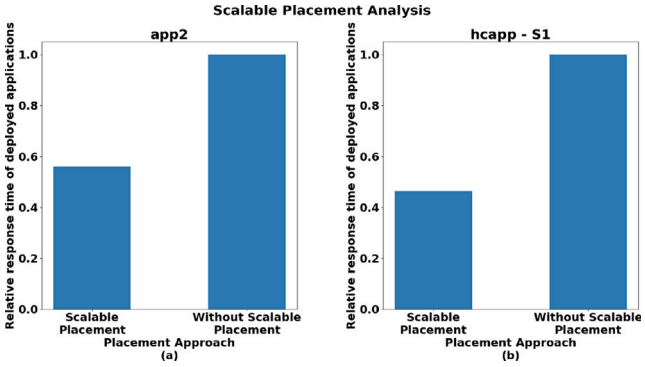

Fig. 17. Scalable microservice placement.

7. Conclusions and future work

In this work, we proposed a framework for the scalable placement of microservices-based IoT Applications in federated Fog environments. The proposed framework is scalable, extensible and configurable to execute placement algorithms and deploy applications across Kubernetes and Istio-enabled multi-fog multi-cloud environments. Moreover, the framework provides the ability to integrate novel placement policies, load balancing policies and PR forwarding policies. Thus, placement algorithm developers and IoT application developers can use the framework to deploy their applications within federated Fog environments and monitor their performance. Furthermore, the framework provides rapid prototyping support, and the applications developed following MSA do not require any application-level changes to be deployed using the framework. Thus, the framework abstracts the underlying deployment-related functionalities from the users, giving them a chance to focus more on placement policy development and IoT application development.

Due to the use of open-source technologies, modular design and architecture, developers can easily extend the framework to add novel functionalities. As future work, the MicroFog framework can be further improved with lightweight security mechanisms for data transmission across clusters, a scalable architecture to store and use observability-related data to improve placement algorithms, integrate dynamic microservice scaling and migration approaches based on observability data, ability to integrate novel fault-tolerance policies for applications.

CRediT authorship contribution statement

Samodha Pallewatta: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Vassilis Kostakos:** Conceptualization, Methodology, Supervision, Writing – review & editing. **Rajkumar Buyya:** Conceptualization, Supervision, Methodology, Visualization, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Link to the source code is includes in the "Software Availability" section of the submitted manuscript.

Acknowledgements

We thank Melbourne Research Cloud (MRC) for providing the infrastructure used for implementing the MicroFog prototype.

Software availability

The source code and documentation of the MicroFog framework is accessible from: https://github.com/Cloudslab/MicroFog

References

Bellavista, P., Zanni, A., 2017. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In: Proceedings of the 18th International Conference on Distributed Computing and Networking. pp. 1–10.

Deng, Q., Goudarzi, M., Buyya, R., 2021. 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. pp. 1–8.

Deng, S., Xiang, Z., Taheri, J., Khoshkholghi, M.A., Yin, J., Zomaya, A.Y., Dustdar, S., 2020. Optimal application deployment in resource constrained distributed edges. IEEE Trans. Mob. Comput. 20 (5), 1907–1923.

Ericsson, 2023. Enhancing service mobility in the 5g edge cloud and beyond. URL https://www.ericsson.com/en/blog/2022/11/service-mobility-in-the-edge-cloud. Accessed February, 2023 [Online].

Ermolenko, D., Kilicheva, C., Muthanna, A., Khakimov, A., 2021. 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, pp. 12–17.

Falkner, J., 2020. Key findings from IDC red hat quarkus lab validation. URL https://www.redhat.com/en/blog/key-findings-idc-red-hat-quarkus-lab-validation.

Fang, J., Ma, A., 2020. Iot application modules placement and dynamic task processing in edge-cloud computing. IEEE Internet Things J. 8 (16), 12771–12781.

Farhat, P., Sami, H., Mourad, A., 2020. Reinforcement R-learning model for time scheduling of on-demand fog placement. J. Supercomput. 76, 388–410.

Farzin, P., Azizi, S., Shojafar, M., Rana, O., Singhal, M., 2022. FLEX: a platform for scalable service placement in multi-fog and multi-cloud environments. In: Australasian Computer Science Week 2022. pp. 106–114.

Faticanti, F., De Pellegrini, F., Siracusa, D., Santoro, D., Cretti, S., 2019. 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, pp. 196–203.

2023. Fogatlas. URL https://fogatlas.fbk.eu/. Accessed February, 2023 [Online].

Fowler, M., Lewis, J., 2014. Microservices a definition of this new architectural term. URL https://martinfowler.com/articles/microservices.html.

Fu, K., Zhang, W., Chen, Q., Zeng, D., Peng, X., Zheng, W., Guo, M., 2021. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In: Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, pp. 932–941.

Google, 2023. Google distributed cloud edge overview. URL https://cloud.google.com/distributed-cloud/edge/latest/docs/overview. Accessed February, 2023 [Online].

Goudarzi, M., Palaniswami, M., Buyya, R., 2022. Scheduling IoT applications in edge and fog computing environments: a taxonomy and future directions. ACM Comput. Surv. 55 (7), 1–41.

Guerrero, C., Lera, I., Juiz, C., 2019a. Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. Future Gener. Comput. Syst. 97, 131–144.

Guerrero, C., Lera, I., Juiz, C., 2019b. A lightweight decentralized service placement policy for performance optimization in fog computing. J. Ambient Intell. Humaniz. Comput. 10 (6), 2435–2452.

Guo, F., Tang, B., Tang, M., 2022. Joint optimization of delay and cost for microservice composition in mobile edge computing. World Wide Web 1–29.

Herrera, J.L., Galán-Jiménez, J., Bellavista, P., Foschini, L., Garcia-Alonso, J., Murillo, J.M., Berrocal, J., 2021. 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, pp. 1–6.

IBM, 2023. Edge clusters. URL https://www.ibm.com/docs/en/eam/4.2?topic=nodes-edge-clusters. Accessed February, 2023 [Online].

Joseph, C.T., Chandrasekaran, K., 2019. Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. Softw. - Pract. Exp. 49 (10), 1448–1484.

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

Mahmud, R., Kotagiri, R., Buyya, R., 2018. Fog computing: A taxonomy, survey and future directions. In: Internet of Everything. Springer, pp. 103–130.

Mahmud, R., Pallewatta, S., Goudarzi, M., Buyya, R., 2022. Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments. J. Syst. Softw. 111351.

Mahmud, R., Toosi, A.N., 2021. Con-pi: A distributed container-based edge and fog computing framework. IEEE Internet Things J. 9 (6), 4125–4138.

Marchese, A., Tomarchio, O., 2023. Sophos: A framework for application orchestration in the cloud-to-edge continuum. In: Proceedings of 13th International Conference on Cloud Computing and Services Science. pp. 261–268.

Neha, B., Panda, S.K., Sahu, P.K., Sahoo, K.S., Gandomi, A.H., 2022. A systematic review on osmotic computing. ACM Trans. Internet Things 3 (2), 1–30.

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

Pallewatta, S., Kostakos, V., Buyya, R., 2022a. Microservices-based IoT applications scheduling in edge and fog computing: A taxonomy and future directions. arXiv preprint arXiv:2207.05399.

Pallewatta, S., Kostakos, V., Buyya, R., 2022b. Qos-aware placement of microservices-based IoT applications in fog computing environments. Future Gener. Comput. Syst..

Paul Martin, J., Kandasamy, A., Chandrasekaran, K., 2020. CREW: Cost and reliability aware eagle-whale optimiser for service placement in fog. Softw. - Pract. Exp. 50 (12), 2337–2360.

Ruuskanen, J., Peng, H., Å kesson, A., Larsson, L., Kihl, M., 2021. Fedapp: a research sandbox for application orchestration in federated clouds using openstack. arXiv preprint arXiv:2109.01480.

Samanta, A., Tang, J., 2020. Dyme: Dynamic microservice scheduling in edge computing enabled IoT. IEEE Internet Things J. 7 (7), 6164–6174.

Santo, W.d.E., Júnior, R.d.S.M., Ribeiro, A.d.R.L., Silva, D.S., Santos, R., 2019. 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, pp. 1–7.

Santoro, D., Zozin, D., Pizzolli, D., De Pellegrini, F., Cretti, S., 2017. 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, pp. 231–234.

Skarlat, O., Nardelli, M., Schulte, S., Dustdar, S., 2017. Towards qos-aware fog service placement. In: Proceedings of the 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC). IEEE, pp. 89–96.

Tuli, S., Mahmud, R., Tuli, S., Buyya, R., 2019. Fogbus: A blockchain-based lightweight framework for edge and fog computing. J. Syst. Softw. 154, 22–36.

Wang, Z., Goudarzi, M., Aryal, J., Buyya, R., 2022. Container orchestration in edge and fog computing environments for real-time iot applications. In: Computational Intelligence and Data Analytics: Proceedings of ICCIDA 2022. Springer, pp. 1–21.

Xu, F., Yin, Z., Gu, A., Zhang, F., Li, Y., 2020. 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, pp. 1567–1572.

Yousefpour, A., Patil, A., Ishigaki, G., Kim, I., Wang, X., Cankaya, H.C., Zhang, Q., Xie, W., Jue, J.P., 2019. Fogplan: A lightweight qos-aware dynamic fog service provisioning framework. IEEE Internet Things J. 6 (3), 5080–5096.

Samodha Pallewatta completed her Ph.D. from the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Australia. Her research interests encompass Fog/Edge Computing, Internet of Things (IoT) and Distributed Systems. She is one of the contributors of the iFogSim simulator, used extensively for resource management research in Fog/Edge computing.

Dr. Vassilis Kostakos is a professor at the School of Computing and Information Systems, University of Melbourne, Melbourne, Australia. His research includes Internet of Things, ubiquitous computing, human–computer interaction and social computing. Dr. Kostakos is a Marie Curie Fellow, a Fellow in the Academy of Finland Distinguished Professor Program, and a Founding Editor of the Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies.

Dr. Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He has authored over 625 publications and seven text books including "Mastering Cloud Computing" published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=160, g-index=334, 137500+ citations).