School of Electrical and Computer Engineering

Intelligent Systems Laboratory

Dynamic Micro-Service Placement in Hybrid Cloud - Fog Infrastructures

Diploma Thesis

of

Prountzos Athanasios

Examination Committee:

Professor Euripides G.M. Petrakis (Supervisor)
Professor Michail G. Lagoudakis
Associate Professor Vasilis Samoladas

Chania, 2023

Abstract

This thesis delves into the problem of microservice placement and load balancing in distributed multicluster Fog-Cloud systems. Minimization of both application response times and utilization of resources is the primary objective of the proposed method. In this study, the systems under investigation are formulated as a three-layered hierarchical model. The top layer represents the Cloud, which encompasses the centralized computing infrastructure and resources. The middle layer represents the Fog, which consists of intermediate computing nodes located closer to the edge devices and data sources. Finally, the bottom layer represents the Edge, which includes the edge devices and sensors that generate and consume data while also providing a small computing capacity. Application users connect to the Edge. Applications are modeled as Service Oriented Architectures which comprise multiple (micro)services. Placing microservices closer to the Edge is deemed to reduce the response time of an application, as opposed to services placed at the upper layers. As many applications can be running on the same infrastructure and as the resources of the Edge and the Fog are limited, choosing services to run on the Edge or the Fog is the problem this work is dealing with. The top layer (the Cloud) exhibits virtually unlimited resources, but running the application on the cloud might not be optimal in terms of response time. Each layer consists of multiple clusters of machines, with resource constraints increasing, as we move towards lower layers, primarily in terms of CPU and RAM resources. The proposed system, DeFog, focuses on decentralized service placement within each cluster, eliminating the need for coordination between clusters. To gather essential data and monitor applications, Service Mesh technologies, such as Linkerd are utilized. DeFog utilizes this data to apply the proposed strategies and achieve efficient load balancing within the system. To evaluate the effectiveness of the strategies, two realistic microservices-based applications, Google's Online Boutique and iXen for processing IoT information, are deployed in a heterogeneous multicluster environment on the Google Cloud Platform. Through extensive experimentation and performance evaluation, the latencies of each strategy are measured and compared under various loads and for various placement policies. The experimental results demonstrate that the proposed strategies successfully reduce response latencies and achieve load balancing, preventing any single service from being overwhelmed and ensuring a smooth user experience. The utilization of realistic applications in a heterogeneous multicluster environment adds practicality and relevance to the evaluation, validating the effectiveness of the proposed strategies in real-world scenarios.

Περίληψη

Αυτή η διπλωματική εργασία εμβαθύνει στο πρόβλημα της τοποθέτησης μικροϋπηρεσιών και της εξισορρόπησης φορτίου σε κατανεμημένα συστήματα ομίχλης-νέφους πολλαπλών cluster. Η ελαχιστοποίηση και των δύο, των χρόνων απόκρισης εφαρμογής και της χρήσης των πόρων είναι ο πρωταρχικός στόχος της προτεινόμενης μεθόδου. Σε αυτή τη μελέτη, τα υπό διερεύνηση συστήματα διατυπώνονται ως ένα ιεραρχικό μοντέλο τριών επιπέδων. Το ανώτερο επίπεδο αντιπροσωπεύει το Cloud, το οποίο περιλαμβάνει την κεντρική υπολογιστική υποδομή και πόρους. Το μεσαίο στρώμα αντιπροσωπεύει το Fog, το οποίο αποτελείται από ενδιάμεσους υπολογιστικούς κόμβους που βρίσκονται πιο κοντά στις συσκευές του Edge και τις πηγές δεδομένων. Τέλος, το κάτω στρώμα αντιπροσωπεύει το Edge, το οποίο περιλαμβάνει τις Edge συσκευές και αισθητήρες που δημιουργούν και καταναλώνουν δεδομένα, ενώ παρέχουν επίσης μια μικρή υπολογιστική ικανότητα. Οι χρήστες εφαρμογών συνδέονται στο Edge. Οι εφαρμογές μοντελοποιούνται ως Service Oriented Architectures που περιλαμβάνουν πολλαπλές (μικρο)υπηρεσίες. Η τοποθέτηση μικροϋπηρεσιών πιο κοντά στο Edge θεωρείται ότι μειώνει τον χρόνο απόκρισης μιας εφαρμογής, σε αντίθεση με τις υπηρεσίες που τοποθετούνται στα ανώτερα επίπεδα. Καθώς πολλές εφαρμογές μπορούν να εκτελούνται στην ίδια υποδομή και καθώς οι πόροι του Edge και του Fog είναι περιορισμένοι, η επιλογή υπηρεσιών για εκτέλεση στο Edge ή το Fog είναι το πρόβλημα που αντιμετωπίζει αυτή η εργασία. Το ανώτερο στρώμα (το Cloud) εμφανίζει ουσιαστικά απεριόριστους πόρους, αλλά η εκτέλεση της εφαρμογής στο cloud μπορεί να μην είναι η βέλτιστη από την άποψη του χρόνου απόκρισης. Κάθε επίπεδο αποτελείται από πολλαπλά clusters μηχανών, με τους περιορισμούς πόρων να αυξάνονται καθώς προχωράμε προς τα χαμηλότερα επίπεδα, κυρίως όσον αφορά τους πόρους CPU και RAM. Το προτεινόμενο σύστημα, το DeFog, εστιάζει στην αποκεντρωμένη τοποθέτηση υπηρεσιών σε κάθε cluster, εξαλείφοντας την ανάγκη για συντονισμό μεταξύ των cluster. Για τη συλλογή βασικών δεδομένων και την παρακολούθηση εφαρμογών, χρησιμοποιούνται τεχνολογίες Service Mesh, όπως το Linkerd. Το DeFog χρησιμοποιεί αυτά τα δεδομένα για να εφαρμόσει τις προτεινόμενες στρατηγικές και να επιτύχει αποτελεσματική εξισορρόπηση φορτίου εντός του συστήματος. Για την αξιολόγηση της αποτελεσματικότητας των στρατηγικών, δύο ρεαλιστικές εφαρμογές που βασίζονται σε μικροϋπηρεσίες, η Online Boutique της Google και το iXen για την επεξεργασία πληροφοριών IoT, αναπτύσσονται σε ένα ετερογενές περιβάλλον πολλαπλών clusters στην πλατφόρμα Google Cloud. Μέσω εκτεταμένων πειραματισμών και αξιολογήσεων απόδοσης, οι καθυστερήσεις κάθε στρατηγικής μετρώνται και συγκρίνονται κάτω από διάφορα φορτία και για τις διάφορες πολιτικές τοποθετήσεων. Τα πειραματικά αποτελέσματα καταδεικνύουν ότι οι προτεινόμενες στρατηγικές μειώνουν επιτυχώς τις καθυστερήσεις απόκρισης και επιτυγχάνουν εξισορρόπηση φορτίου, αποτρέποντας την καταπόνηση οποιασδήποτε υπηρεσίας και διασφαλίζοντας μια ομαλή εμπειρία χρήστη. Η χρήση ρεαλιστικών εφαρμογών σε ένα ετερογενές περιβάλλον πολλαπλών clusters προσθέτει πρακτικότητα και συνάφεια στην αξιολόγηση, επικυρώνοντας την αποτελεσματικότητα των προτεινόμενων στρατηγικών σε σενάρια πραγματικού κόσμου.

Acknowledgments

I am sincerely grateful for the invaluable guidance and unwavering support provided by my supervisor, Professor Euripides Petrakis, throughout the entire journey of conducting my Diploma Thesis. His extensive expertise, profound knowledge, and unwavering dedication have played a crucial role in shaping the trajectory of my research and steering me toward achieving significant outcomes.

I would also like to express my heartfelt gratitude to my fellow colleagues who embarked on their diploma theses alongside me during this period. Engaging in discussions, exchanging ideas and sharing experiences have fostered a collaborative and dynamic learning environment.

I dedicate this work to my dear friends and beloved family for their support in each of my steps.

Table of Contents

List of Figures

List of Charts

List of Tables

1 Introduction

1.1 Problem Definition

In recent years, Cloud Computing assumes pivotal and dynamic factor in the seamless deployment of modern applications. By operating on top of a network of geographically distributed data centers owned by infrastructure owners such as Amazon Web Services and Google Cloud Platform, they showcase a diverse array of resources. These resources include virtual machines, containers, serverless functions and databases that can be easily provisioned and managed through web interfaces or APIs. This streamlined process reduces the complexity proposed by the traditional approach prompting developers to focus more on innovation and less on infrastructure management.

As cloud computing gains traction and becomes increasingly popular, a unique set of challenges emerges calling for innovative solutions. Undoubtedly, the existing Cloud infrastructure faces significant challenges supporting a multitude of current applications that introduce issues such as network latency, bandwidth limitations, data privacy concerns. To cope with these issues, promising paradigms including fog and edge computing have emerged. Figure 1.1 provides comprehensive visual representations of cloud and edge-fog-cloud infrastructures, offering detailed models that showcase the intricate components and relationships within these systems.

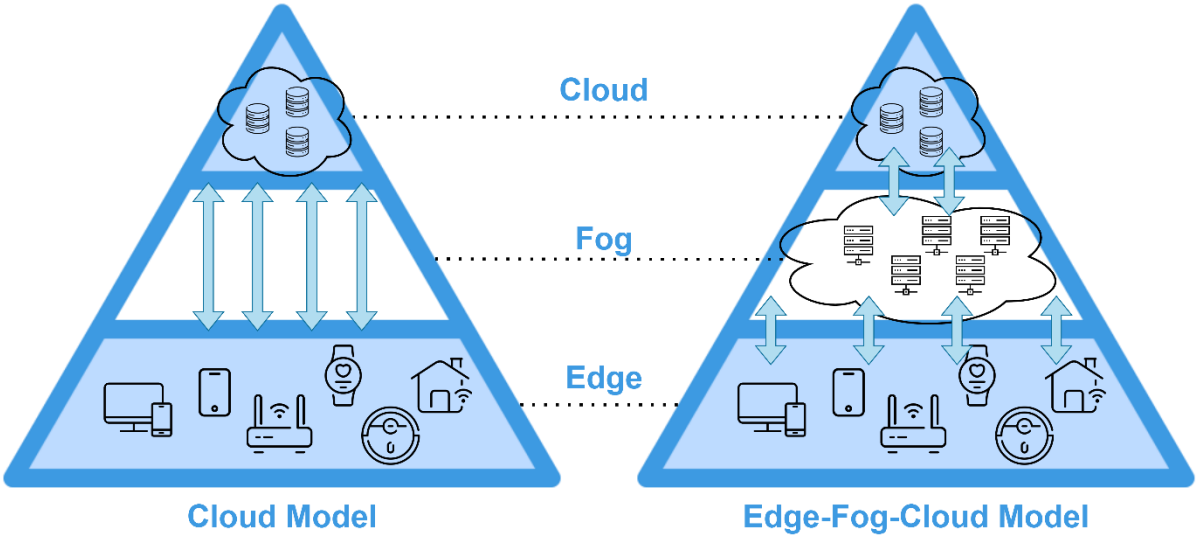


Figure 1.1: Infrastructures' Models

Fog computing is a distributed computing architecture that brings the computing, storage and networking capabilities of cloud closer to the edge of the network [1]. On the premises of fog computing, data is generated by IoT devices, sensors and other edge devices, and then processed by a network of

intermediate nodes located between the edge and the cloud, such as gateways, routers, servers, etc. These fog nodes propose various computing resources such as processing, storage and networking, also acting as an intermediate layer that filters, aggregates and processes data before sending it to the cloud.

Similarly, Edge computing introduces computation and storage closer to the end-users and edge devices, minimizing the data transfer to a central location as processing is managed close to the data sources. In this computational model, the processing is done on the device itself or a nearby server, while only relevant data is sent to the cloud for further analysis. This paradigm aims to reduce network latency, improve the user experience and enhance the privacy and security of distributed applications. By enabling data processing and analysis at the edge of the network, edge computing complements cloud and fog computing offering a range of benefits for various applications.

These approaches bring computation and storage closer to the network's edge, providing advantages such as decreased latency, enhanced user experience, and improved privacy and security. While fog computing brings the cloud closer to the edge, edge computing focuses on processing data closer to its source. Together, these paradigms complement cloud computing and offer a range of benefits for diverse applications. However, this hybrid cloud-fog-edge architecture entails the great challenge of the placement of microservices. Several studies have investigated different aspects of fog and edge computing to optimize their performance and efficiency. Specifically, references [2], [3], [4], [5] and [6] concentrate on the resource allocation in fog computing environments, proposing algorithms that consider the resource requirements and constraints of fog nodes and optimize the allocation and service placement process. Additionally, references [7] and [8] explore a range of techniques and methodologies for resource allocation, task scheduling, and data processing in edge computing environments.

In summary, the adoption of cloud-edge-fog computing architectures can provide a new paradigm for addressing the challenges of cloud computing. However, the complexity of these architectures presents significant challenges. The placement of microservices is just one aspect of this broader problem, which also encompasses issues related to network latency, data privacy, security, and transfer costs. To effectively handle these issues, a comprehensive approach is required that reflects on the specific needs and constraints of these underlying architectures. Ultimately, efficient deployment on such hybrid computing architectures will demand careful planning, implementation and ongoing management to ensure optimal performance.

1.2 Scope of Thesis

This thesis aims to explore, devise, and execute dynamic placement strategies for microservices in order to enhance the latency and performance of applications operating on a hybrid distributed cloud-fog-edge infrastructure. The focus lies specifically on multiple Kubernetes clusters within this infrastructure. The strategies will primarily target the reduction of network latency and the optimization of microservice performance by dynamically allocating them across various computing nodes within the multi-cluster network. The implementation process will involve the utilization of specialized tools tailored for managing and monitoring microservices within distributed environments. Furthermore, these strategies will be seamlessly integrated with existing cloud computing infrastructures to provide a comprehensive solution.

In order to address the challenges of service placement in distributed cloud-fog-edge architectures, this thesis puts forward a novel solution in the form of a tailored application called DeFog. DeFog is envisioned to leverage data gathered from specialized tools to inform and execute strategic decisions relevant to the evaluated placement strategies. The application will intelligently consider factors such as the volume of requests per service, network latency, and resource utilization. By incorporating these elements into its decision-making process, DeFog aims to optimize service placement and enhance the overall efficiency of the architecture.

The overarching objective is to revolutionize service placement methodologies not merely for monolithic applications comprising single services but also for applications comprising multiple services such as Service Oriented Architectures. There, the placement decisions should be taken not only based on resource constraints of individual services but also based on the dependence between communicating services. For example, a decision to place a service to the cloud and another closer to the edge based on CPU or RAM resources might not be optimal if the two services communicate with each other. Through the development and implementation of the DeFog application, this thesis endeavors to make a significant contribution to the broader field of cloud computing and service placement strategies for Service Oriented Applications and not only for single service applications. The envisioned outcome is a novel and customizable solution that can be tailored to meet the unique requirements of diverse applications and infrastructure frameworks, enabling them to effectively cater to the demands of contemporary applications operating within intricate distributed cloud-fog-edge infrastructures.

2 Background and related work

In this chapter, we bestow an overview of the theoretical background that forms the foundation of this thesis. We delve into the concepts and principles that underpin the implementation of service placement strategies. Additionally, we discuss the metric tools and agents that will be utilized and modified to support the implementation of these strategies.

2.1 Related work

The issue of service placement in distributed systems has garnered substantial research interest in academic and industrial circles. It has become a focal point for optimizing the allocation of services within these systems. Over the years, numerous studies have emerged, presenting a multitude of algorithms and strategies aimed at enhancing service placement decisions. In the realm of service placement in Cloud, Fog, and Edge environments, several authors have made significant contributions, as highlighted in [9], [10] and [11]. These works shed light on the existing body of research and showcase various approaches and methodologies employed to tackle the challenges associated with effective service placement.

Fog environments have emerged as a promising paradigm for distributed computing, bringing computational resources closer to the edge of the network. In [2] researchers focus on addressing the energy consumption challenges in the fog computing paradigm by formulating a service placement plan that maximizes resource utilization while considering the active and idle states of machines. To achieve this, they propose an architecture that combines cloud and fog computing, introducing a middleware known as the cloud fog control middleware. This middleware plays a key role in optimizing costs and determining the placement of services within the fog cluster. It achieves this by dynamically rearranging and scheduling the services based on continuous or on-demand events. By optimizing the placement of services, the researchers aim to achieve an optimal balance between cost, energy consumption, and time while meeting the Quality of Service (QoS) expectations of the applications.

In [3] the authors propose a hierarchical and autonomous fog architecture (HAFA) to address the challenges of fog computing. They emphasize the importance of fog nodes, which are physical computing resources in close proximity to devices, for deploying applications in domains like smart cities, healthcare, and autonomous vehicles. The authors develop a distributed approach that selects cost-efficient fog nodes for hosting application services, considering computation and communication costs. This approach does not require complete system state knowledge and offers cloud-like features such as self-service, scalability, and performance. They evaluate the solution through simulations and compare its performance with other approaches, including a centralized one.

In [4] the authors address the complex task of managing multi-service applications on dynamic and heterogeneous Fog infrastructures. They highlight the importance of making informed decisions to avoid negative impacts on application QoS, resource utilization, and costs. To improve the efficiency of searching for QoS-aware application deployments, the authors propose a centralized approach that combines Genetic Algorithms with Monte Carlo simulations. By leveraging this hybrid method, they

aim to optimize the placement process and achieve better overall system performance in the context of Fog computing.

In their study [5], researchers focus on two important objectives in the fog computing environment: ensuring the QoS of applications by meeting service delivery deadlines and optimizing resource utilization. To achieve these objectives, they propose a latency-aware decentralized approach for placing Application Modules on distributed fog nodes. By taking into account latency awareness, the placement strategy dynamically relocates modules to optimize the number of fog nodes that are computationally active. In [6], the authors introduce a decentralized microservices-based IoT application placement policy specifically designed for Fog environments that are characterized by heterogeneity and resource constraints. The proposed policy takes advantage of the horizontal scalability feature of microservices and incorporates important functionalities such as service discovery and load balancing. They aim to minimize latency and network usage by placing microservices as close as possible to the data source. However, both [5] and [6] model and evaluate their proposed policies in an iFogSim-simulated environment. While simulations provide a controlled and scalable environment for assessing the proposed policies, it is important to consider the limitations and assumptions of the simulation framework when interpreting the results.

In [7], the authors address the challenge of service placement and request scheduling for data-intensive applications in edge-clouds. They propose a two-time-scale framework that optimizes both service placement and request scheduling by formulating the problem as a mixed integer linear program (MILP). By utilizing this framework, they aim to achieve near-optimal performance in terms of service placement and request scheduling. On the other hand, [8] focuses on uncoordinated strategies for service placement in Edge-Clouds. The authors propose a set of techniques that enable opportunistic, on-path execution with uncoordinated resource allocation in the edge-cloud environment. The resource allocation process encompasses admission of requests, scheduling of the order these requests will be served, and placement of services at each computational spot. The authors demonstrate through their study that uncoordinated strategies can achieve near-optimal performance without the need for communication or coordination overhead that centralized solutions often entail. However, it is important to note that both studies, [7] and [8], use a combination of synthetic and trace-driven simulations. A summary of the reviewed related works is presented in Table 2.1.

Work	Infrastructure	Placement Approach	Platform	Application	Placement
[2]	Fog-Cloud	centralized	simulator	simulation	static
[3]	Fog	distributed	PFogSim simulator	simulation	dynamic
[4]	Fog-Cloud	centralized	FogTorchPI simulator	simulation	static
[5]	Fog	distributed	iFogSim simulator	simulation	dynamic
[6]	Fog	distributed	iFogSim simulator	simulation	static
[7]	Edge-Cloud	centralized	simulator	simulation	static
[8]	Edge-Cloud	distributed	Icarus simulator	simulation	dynamic
This Work	Edge-Fog-Cloud	distributed	K3s	iXen,eShop	dynamic

Table 2.1: Comparison Table

In this thesis, our focus is on improving the performance of Service Placement in a distributed Edge-Fog-Cloud environment and load balancing. To achieve this goal, we leverage a Kubernetes distribution as our orchestration platform for microservices. One of the main factors influencing our research is the Least Frequently Used (LFU) service placement strategy proposed in [8]. We adapt the LFU algorithm for our implementation and propose a set of optimization strategies for service placement that take into account various metrics such as Requests per Second (RPS), response latencies, and resource utilization. By using it as a benchmark, we can assess the effectiveness and efficiency of our proposed algorithm in comparison. Leveraging these data points, we aim to improve the overall efficiency and effectiveness of service placement in our distributed system. To achieve effective load balancing, we employ a strategy that involves efficiently distributing incoming requests across multiple clusters. This approach prevents any single service from becoming overwhelmed, ensuring a balanced workload distribution. Unlike previous studies that have primarily used synthetic environments, we take a different approach by utilizing realistic microservices-based applications for evaluating our strategies, obtaining more meaningful and practical insights into the performance of our proposed optimization strategies. Our ultimate goal is to reduce response times ensuring the Quality of Service (QoS) for end users and delivering an enhanced user experience.

2.2 Infrastructure and tools

2.2.1 Microservices

Microservices (or microservices architecture) is a cloud-native architectural approach in which a single application is composed of many loosely coupled and independently deployable smaller components, or services. These services typically communicate with one another over a combination of REST APIs [12]. Breaking down applications into these independent services ensures that any modifications or updates made to each one of them will have minimal impact on the rest of the system. This modular architecture also promotes better collaboration among development teams, as they can work on different microservices concurrently. Figure 2.1 provides a visual representation of a typical Microservice architecture, showcasing the intricate arrangement of interconnected components.

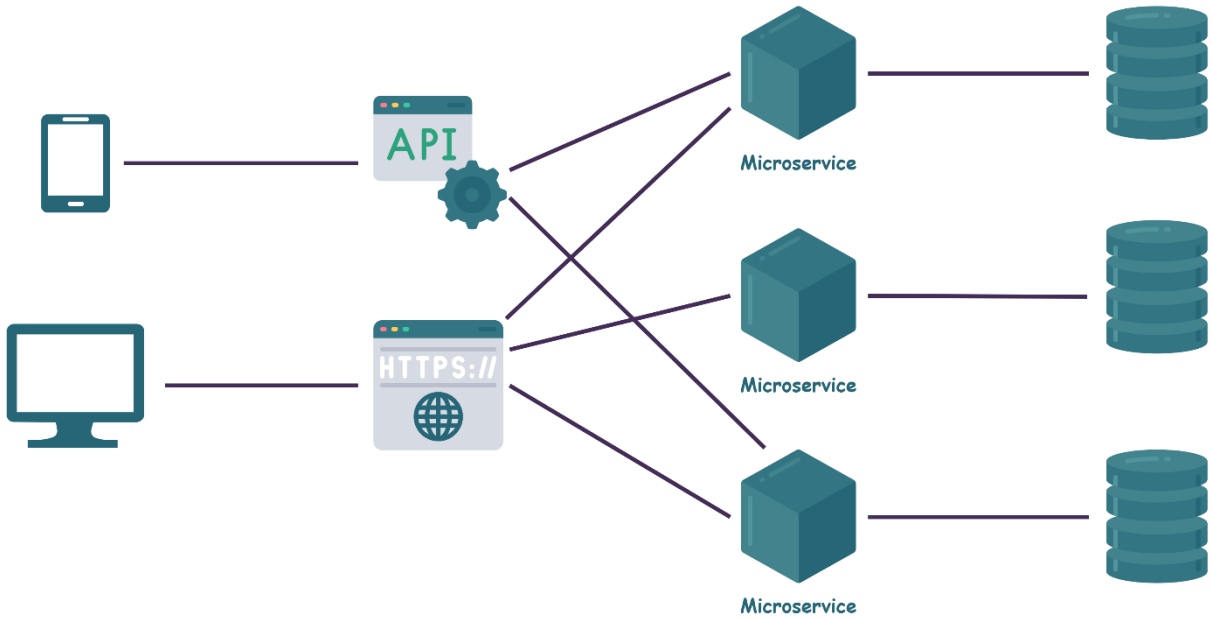


Figure 2.1: Microservices Architecture

The microservices approach is ideal for modern software development, as applications should be agile, scalable, responsive, and flexible to meet evolving requirements. While they present numerous benefits, they also propose challenges, particularly in terms of management and deployment. To overcome the challenges developers frequently rely on Kubernetes, commonly referred to as K8s. K8s is an open-source system for automating deployment, scaling, and management of containerized applications [13]. K8s allows developers to deploy and manage microservices regardless of the underlying infrastructure.

2.2.2 K3S

While K8s is a powerful tool for managing containerized applications, it may not always be the best option for every situation. In particular, when it comes to fog, and edge infrastructures, resource constraints may be encountered that impact performance and scalability. Under such circumstances, K3s offers a lightweight and streamlined solution that effectively addresses these limitations.

K3s is a fully CNCF (Cloud Native Computing Foundation) certified Kubernetes distribution that has been specifically designed to cater to resource-constrained environments such as edge, IoT, and low-powered devices [14]. Its deployment and management are easy to handle, and it has a smaller footprint as compared to the standard K8s distribution. The lightweight architecture of K3s provides an optimal solution for small businesses to operate more efficiently and with lower resource consumption while still enjoying the benefits of high availability, security, and other features offered by the complete K8s architecture.

K3s main features are:

- Lightweight: K3s is packaged as a small binary.
- High availability.
- Security: K3s provides several built-in security measures, such as automatic TLS certificate generation, RBAC (Role-Based Access Control), and network policies.
- Scalability: K3s supports the same scaling capabilities as Kubernetes.
- Simplicity: K3s is easy to use and maintain.
- Extensibility: K3s can be extended with custom plugins and integrations.

2.2.2.1 K3S' Architecture

Like Kubernetes, a K3s cluster consists of a set of worker machines called Agent nodes that run containerized applications and one or more Server nodes that run the lightweight control plane that manages the nodes and pods in the cluster. K3s offers four deployment architectures: single node, single node with multiple agents, where both architectures feature embedded database, high availability with external etcd database, and high availability with an embedded etcd database.

The single node architecture is designed for development or testing purposes, optimized for edge devices, and consists of a single Server node that serves both as the control plane and the worker node. The single node with multiple agents architecture includes multiple worker nodes that are managed by a single control plane. Both these architectures features an embedded SQLite database instead of etcd. High availability with an external etcd database architecture is ideal for production environments where this database is hosted separately from the control plane, and multiple Server nodes provide redundancy and high availability. Finally, high availability with an embedded database architecture is similar to the previous architecture but includes a built-in, lightweight etcd database [15].

The Server node in K3s is the control plane that manages the worker nodes of the cluster. It consists of several components, including the API server, SQLite, scheduler, controller manager and tunnel

proxy. The API Server is the central component of the control plane that serves as the management interface for the entire cluster. The SQLite used as is the default storage backend instead of etcd to store cluster state and configuration data for the components. The scheduler component is responsible for assigning workloads to the worker nodes in the cluster. The controller manager ensures the desired state of the cluster is maintained. Finally, the tunnel proxy is responsible for managing and securing network traffic between the server and the worker nodes in the cluster.

In K3s, the agent node serves as the worker node that executes the containerized applications within the cluster. This node comprises various components, including containerd, which functions as the container runtime, and kubelet, which operates as the primary node agent responsible for managing containers and facilitating communication with the API server. Additionally, kube-proxy is an integral component that manages network proxy and load balancing functions for services running within the node. The agent node also features Flannel, which provides a lightweight network fabric that allows networking between containers across different nodes. Moreover, the tunnel proxy component is responsible for establishing a secure and encrypted connection between the worker nodes and the server node in a K3s cluster. Figure 2.2 provides a visual representation of the basic architecture of a K3S cluster that includes a server and Agent nodes.

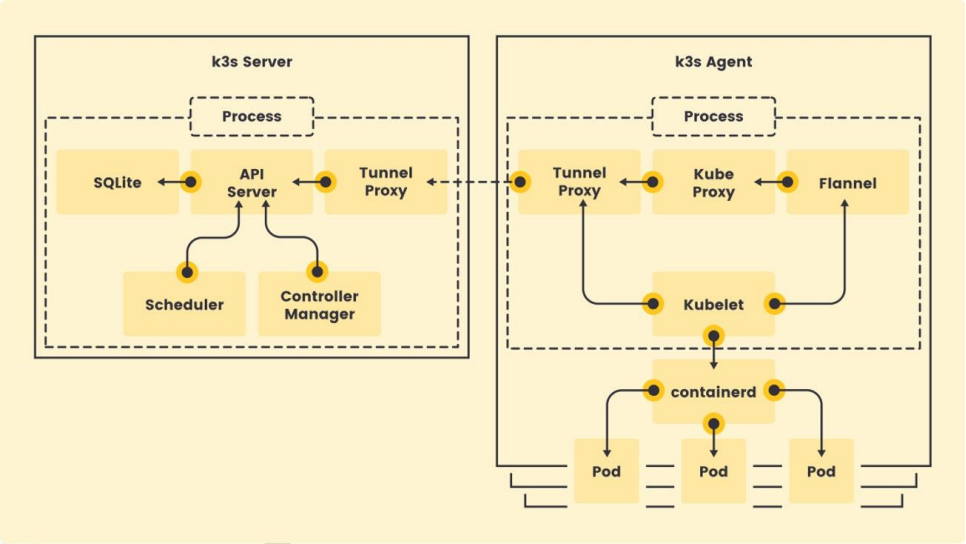


Figure 2.2: K3S Architecture [15]

2.2.3 Service Mesh

Microservices are a popular approach to building scalable and maintainable applications. However, as microservices architecture grows in size and complexity, real challenges start to appear. Some major challenges that arise include:

- Service-to-service communication
- Observability
- Failure handling

The solution to deal with these challenges is the Service Mesh. A service mesh is a dedicated infrastructure layer built right into an application [16]. This visible infrastructure layer can provide features such as traffic management, service discovery, error handling, observability, load balancing, authentication, and encryption [17].

A service mesh introduces an abstraction layer between services. This abstraction layer forms the data plane of the service mesh and is implemented as a set of network proxies that are deployed alongside each service instance, providing a standardized way to manage and control communication. On the other hand, the service mesh's control plane consists of a collection of services that can be deployed across one or multiple nodes within a cluster, taking care of the supporting functionalities and coordinating the proxies. Figure 2.3 depicts the general architecture of a service mesh.

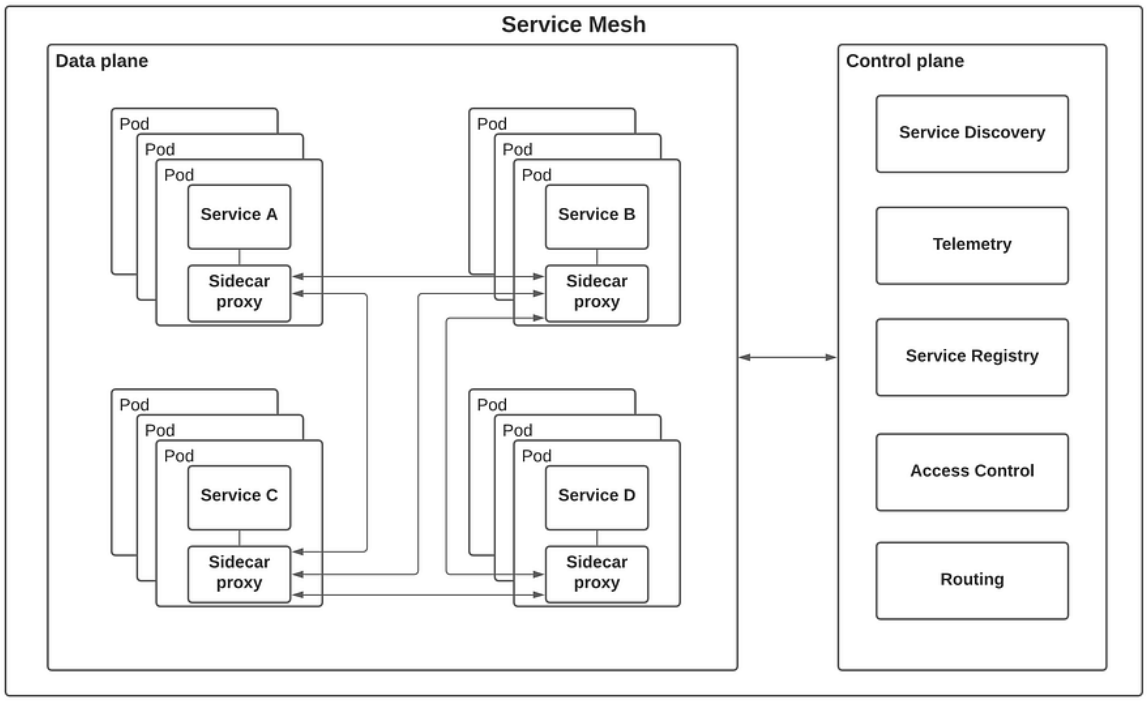


Figure 2.3: Service Mesh Architecture [17]

Service mesh technology is rapidly evolving, with new features and implementations being added all the time. One of the most popular service mesh implementations is Linkerd [18]. Linkerd is a highly regarded open-source project that is hosted by the Cloud Native Computing Foundation (CNCF),

specifically designed to be lightweight and user-friendly, also prioritizing reliability and performance. Linkerd provides features such as service discovery, load balancing, and observability [18]. Some other notable service mesh implementations include Istio [19], Consul [20], and Traefik Mesh [21].

In summary, a service mesh plays a critical role in enhancing the management and control of inter-service communication in a microservices architecture and offering a range of powerful features, which collectively contribute to improved communication between services, simplified management, and optimized performance. Adopting a service mesh helps simplify complex microservices architecture while gaining valuable insights into how services interact with each other, troubleshoot issues, and improve overall system reliability. Ultimately, by leveraging the benefits of a service mesh, the overall agility and scalability of a microservices architecture can be improved.

2.2.3.1 Linkerd

Linkerd is a versatile, open-source service mesh for cloud-native applications. Built on top of the Rust programming language, Linkerd's lightweight design is ideal for use in resource-constrained or complex environments. It provides security, observability, and reliability to Kubernetes [18], without additional complexity.

Linkerd can monitor and report per-service success rates and latencies, can automatically retry failed requests, and can encrypt and validate connections between services, all without requiring any modification of the application itself [18]. Linkerd is also highly reliable and performant, with an emphasis on minimizing latency and ensuring the high availability of services. Furthermore, it provides a wide range of extensions such as Linkerd-Viz and Linkerd Multicluster.

Linkerd offers a range of key features that are essential for managing and securing microservices-based architectures. One of these features is automatic service discovery, which enables services to discover one another dynamically and transparently. Another critical feature is load balancing and traffic splitting, which increases resource efficiency and enhances service reliability. Linkerd's load balancing refers to the distribution of network traffic across multiple instances of a service for resource optimization and high availability, while traffic splitting allows users to divide incoming traffic between different versions of a service or different services altogether, enabling controlled deployments. Additionally, Linkerd provides Dynamic Request Routing capabilities, allowing control over traffic flow by routing incoming requests based on various criteria, such as headers or request content. Moreover, it provides observability and monitoring tools, enabling real-time insights into the behavior and performance of services. Finally, it offers robust security features, including mutual TLS (mTLS) authentication, request-level authorization, and automatic TLS certificate rotation. All of these features work together to simplify complex microservices architectures while improving communication between services and ensuring high levels of security and reliability.

2.2.3.1.1 Linkerd's Architecture

Linkerd's architecture comprises two essential components: the data plane and the control plane. The data plane functions as the workhorse of the system and handles the critical task of directing network traffic between various services ensuring smooth communication with each other and without any disruptions or delays. The control plane is comprised of a set of services, responsible for the overall orchestration and management of the data plane. Its services can be deployed in one or multiple nodes of a cluster and their responsibilities include configuring, scaling, and monitoring the data plane, making sure that everything is always running optimally. Figure 2.4 provides a detailed overview of the architecture of Linkerd, showcasing the distinct grouping of mesh's control plane services separate from the meshed ones (Mesh's Data Plane), to highlight the clear separation and organization of components within Linkerd's architecture.

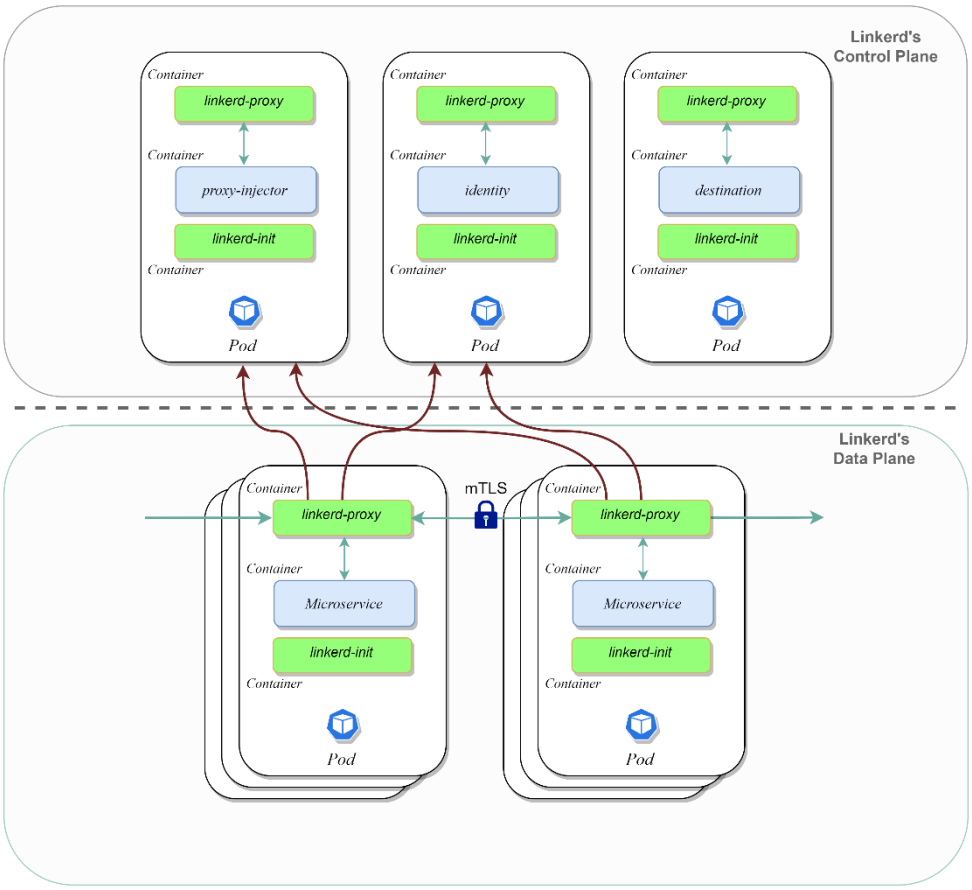


Figure 2.4: Linkerd's Architecture (basic installation)

The data plane is made up of a set of transparent micro-proxies that are deployed alongside each service instance, as sidecar containers in the pods [18]. These proxies handle all the incoming and outgoing TCP network traffic for the service, intercepting all requests and responses. When a request is intercepted, the proxy routes it to the appropriate destination based on the rules defined in the service mesh. The proxy is also responsible for adding metadata to the requests and responses, which is used for observability and monitoring purposes. This metadata can include information about the service name, version, request latency, success rate, and more. To configure the proxies with the

necessary settings, the data plane includes Linkerd-init container. This component is added to the meshed pod as a Kubernetes init container that executes before any other containers and performs one-time initialization tasks such as fetching certificates and setting up service discovery. Once the Linkerd-init process is complete, the Linkerd proxy is ready to perform its role, routing traffic and adding metadata for observability and monitoring.

On the other hand, the control plane is made up of a set of components that work together to manage and configure the proxies that form the data plane, including the identity, destination, and proxy-injector services. The identity service acts as a certificate authority that accepts certificate signing requests (CSRs) from the proxies in the data plane. These certificates are used for mutual TLS (mTLS) between the proxies. By implementing mTLS, Linkerd ensures that communication between proxies is secure and authenticated. This prevents eavesdroppers from getting access to data transferred (man-in-the-middle attacks or MITM), and unauthorized access to services. The destination service is responsible for managing and distributing information about the available services in the service mesh, allowing for dynamic service discovery and efficient load balancing. The data plane proxies use the destination service to determine various aspects of their behavior, such as where to send a particular request, which types of requests are allowed, and how to handle per-route metrics, retries, and timeouts. Lastly, the proxy-injector is responsible for automatically injecting the Linkerd proxy into the application pods in the data plane. This is achieved using a Kubernetes mutating admission webhook, which acts as an HTTP callback to receive and process admission requests [22]. Specifically, the mutating admission webhook used by Linkerd intercepts the creation of pods and modifies their YAML specification, adding the necessary proxy-init and linkerd-proxy containers.

2.2.3.1.2 Linkerd's Extensions

While Linkerd includes a comprehensive set of features, it also provides extensions that add further functionality and customizability to the service mesh. These extensions can be installed on top of the core components of the control plane, customizing its behavior, and adding new features. Some of the most popular Linkerd extensions include Linkerd Viz, Linkerd Multi-cluster, and Linkerd SMI.

Linkerd Viz [23] is a comprehensive visualization tool that consists of five components: Tap, Metrics-api, Web, Prometheus, and Grafana. The Tap component provides the ability to introspect live traffic in real-time. The Metrics-api provides a programmatic interface for accessing and retrieving metrics data collected by Linkerd proxies, while the Web component provides a dashboard for visualization of this metrics data, service dependencies and traffic. Prometheus is responsible for collecting, storing, and querying the metrics data generated by the proxies. Lastly, Grafana is an optional component that can be used to create custom dashboards and visualizations of that data. It's also worth noting that an external instance of Prometheus can be used instead of the bundled version, providing additional flexibility.

Linkerd's Multi-cluster [23] is implemented by the Service Mirror and the Gateway components. It works by mirroring service information between clusters, which ensures that services in one cluster can easily discover and communicate with services in another cluster. The Service Mirror component watches for updates to services in a target cluster and mirrors that information locally on a source

cluster. Meanwhile, the Gateway component provides a secure and efficient way for services in one cluster to communicate with services in another cluster by establishing encrypted connections between them. This extension provides a powerful solution for building and managing multi-cluster environments, enabling to deployment and scaling of applications across multiple geographic regions or cloud providers while maintaining consistent performance and security. Figure 2.5 illustrates the functionality of Linkerd's Multicluster, depicting two connected clusters, where Cluster A serves as the source cluster and Cluster B as the target. In this setup, Microservice B's Kubernetes service is mirrored at Cluster A, and a virtual IP address is created by Cluster A to provide a consistent response to pods resolving the mirrored service. Additionally, the gateway in Cluster B functions as an ingress controller, routing incoming requests to the appropriate service.

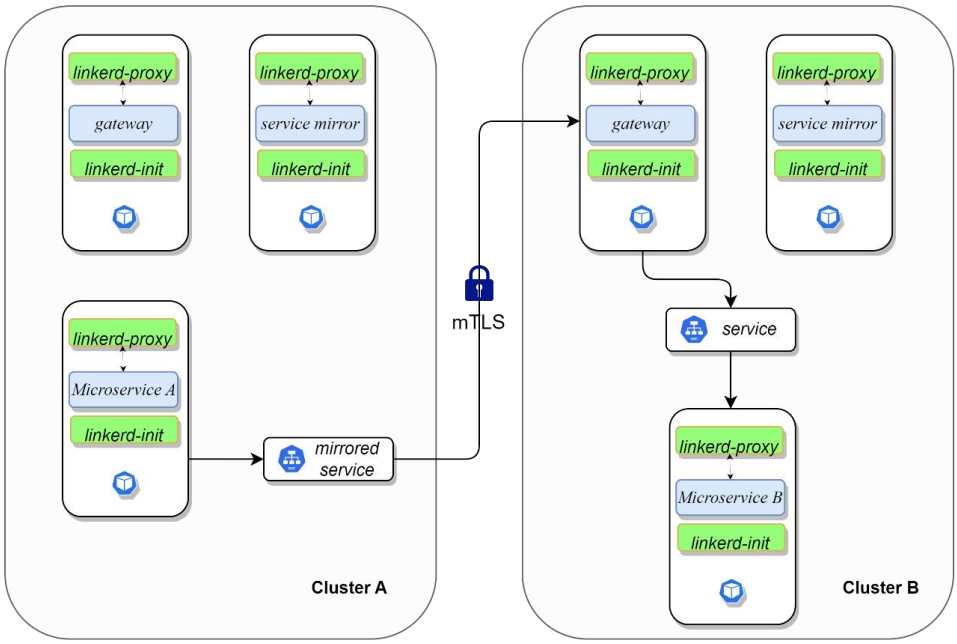


Figure 2.5: Linkerd's Multicluster's Functionality

SMI (Service Mesh Interface) is a standard interface for service meshes on Kubernetes, such as Linkerd. It defines a set of resources that could be used across service meshes that implement it [18]. It includes features such as traffic policy, traffic telemetry, and traffic management [24]. One of the key traffic management mechanisms defined by SMI is TrafficSplit, which enables controlled distribution of traffic across multiple versions or subsets of a service instance. This allows for precise traffic routing to different service instances based on predefined proportions-weights. By leveraging TrafficSplit, service mesh deployments can achieve fine-grained control over traffic distribution, optimizing resource utilization and enhancing the reliability and availability of services. Figure 2.6 demonstrates the implementation of a TrafficSplit configuration, allocating traffic between a local service instance and a mirrored service.

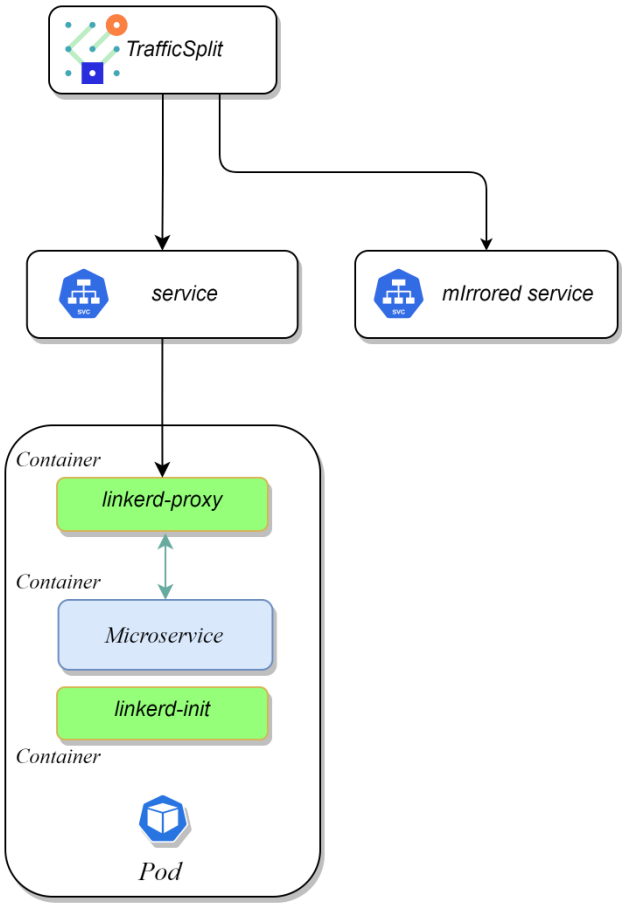


Figure 2.6: TrafficSplit

2.2.4 Benchmark Stressing Tool – Locust

Locust [25] is a popular open-source load testing tool. It enables developers to test the performance of web applications. It provides a user-friendly web interface for defining load testing parameters and reviewing the results. Its scalability and ability to simulate millions of concurrent users make it a perfect tool to test distributed systems and microservices architectures.

Locust generates detailed metrics and graphs for response times, error rates and other performance indicators to identify areas for optimization and potential bottlenecks. However, it requires a basic understanding of Python and web protocols to write test scripts and interpret results.

3 System Topology and Architecture

The effective placement and replacement of services within a distributed environment heavily rely on the intricate nature of system topology. In this section, we delve into the architecture and design of the proposed multicluster cloud-fog-edge infrastructure, offering a comprehensive overview of its topology. By gaining a profound understanding of the interconnections and structure of the system's various layers, we can grasp the seamless flow of requests, the deployment and initialization of services, and the mechanisms that facilitate dynamic service placement and migration. Our aim is to unravel the complexity inherent in the system's design and illuminate the multifaceted processes that drive its functionality.

3.1 Overview of System Topology

The system's topology plays a crucial role in achieving efficient service placement and replacement. It encompasses multiple layers, each serving a specific purpose in the overall architecture. The layers involved in the system include the cloud layer, edge layer, and fog layer. These layers are strategically designed to handle different aspects of service management and request processing. The traffic flow of requests follows a path of clusters through each layer. If a service is not instantiated in a cluster on the path, the request is forwarded to a connected cluster of the next layer. To visually represent the cloud-fog-edge model and illustrate the interplay among these layers, Figure 3.1 is provided in the context of the system's topology.

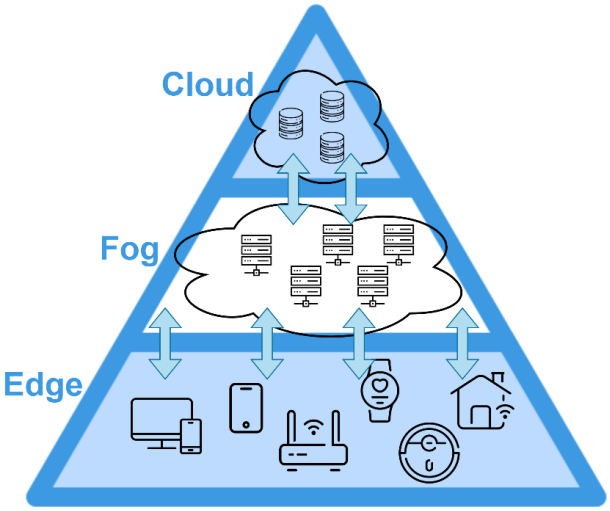


Figure 3.1: Topology Model

Within the system's topology, the cloud layer assumes a pivotal role as the centralized infrastructure, often housed in data centers or entrusted to public cloud providers. This layer offers a multitude of

advantages, including scalability, robust computing power, and ample resources necessary for managing and processing intricate workloads. Services deployed within the cloud layer serve as the bedrock of the entire system, acting as a fundamental starting point for the processing of incoming requests. By capitalizing on the cloud layer's capabilities, the system establishes a strong foundation that enables seamless request processing and sets the stage for subsequent layers to play their respective roles in the overall architecture.

Serving as a vital intermediary between the cloud layer and the edge layer, this layer encompasses additional computing resources and services that are strategically positioned closer to the edge. By doing so, it facilitates efficient processing and data management for edge devices while mitigating issues such as latency and network congestion. By leveraging the capabilities of the fog layer, the system can intelligently offload computation and storage tasks from both the cloud and edge layers. The proximity of the fog layer to edge devices allows for quicker data processing, reducing latency and ensuring timely responses. In essence, the fog layer acts as a crucial bridge within the system's topology, augmenting computational resources and services nearer to the edge.

At the foundation of the system's topology lies the edge layer, representing the closest tier to end-user devices. This layer encompasses a diverse array of devices, including sensors, IoT devices, and mobile devices. Notably, the edge layer is distinguished by its relatively constrained computational resources. However, its strategic placement in close proximity to users offers significant advantages. By positioning services closer to users, the edge layer effectively reduces the round-trip time for requests, resulting in enhanced response times and improved user experience. This attribute proves especially valuable in latency-sensitive applications and scenarios that necessitate real-time data processing.

3.2 Service Initialization

In terms of service deployment, the system's topology follows a strategic approach to ensure efficient operations. Initially, all services, excluding the user interfaces (Frontend), are deployed within the cloud layer. The cloud layer provides the necessary computational resources, scalability, and high availability required for hosting a diverse range of services and applications. Acting as the centralized backbone, services within the cloud layer offer essential functionalities and data access.

Following the deployment phase, services undergo initialization steps to prepare them for request handling. To simplify this initialization process, Linkerd Service Mesh is employed, leveraging its robust features. A crucial component of the service mesh, Linkerd Multicluster, assumes a vital role in establishing connections between the different layers of the system. By utilizing Linkerd Multicluster, mirrored services are created to facilitate seamless communication between the user interfaces deployed in the edge layer and the services within the cloud layer. This enables efficient interaction and access to functionality and data residing in the cloud layer. The mirrored services effectively act as a bridge, ensuring effective communication between the user interfaces and the services hosted in the cloud.

To route requests from the edge layer to the appropriate services, Linkerd's TrafficSplits mechanism comes into play. TrafficSplits enable the routing of traffic to mirrored services without requiring

additional configuration of the traffic source. The TrafficSplits configuration assigns weight values to each service, determining the proportion of traffic that each service will receive. Initially, TrafficSplits guide all the flow of requests from the edge layer to the fog layer and subsequently to the services located in the cloud layer. This routing mechanism ensures that requests originating from the user interfaces successfully reach the corresponding services residing in the cloud layer, making effective use of the available capabilities and resources.

3.3 Service Placement and Migration

Service placement and migration are critical components of the system's operation, contributing to efficient resource utilization, scalability, and responsiveness. The DeFog application, deployed in each cluster within the system's topology, plays a central role in these processes by leveraging decentralized strategies. Rather than relying on a central service, the decision-making process is distributed across the clusters, allowing each cluster to independently evaluate its resources and workload. Factors such as resource availability, including CPU and RAM capacities, are considered when making placement decisions to ensure efficient utilization of computational resources.

In addition to resource availability, DeFog utilizes metrics like Requests per Second (RPS) and Response Times to rank services and make informed decisions. Services with higher RPS and lower response times are more likely to be deployed, while those with lower rankings may be considered for deletion or replacement. By continuously monitoring and collecting relevant metrics data, DeFog can dynamically adjust service deployments and deletions to optimize performance.

The distributed deployment of DeFog allows services to be instantiated in multiple clusters, facilitating load balancing and traffic redistribution. If a stressed service is detected, DeFog adjusts the weights of the TrafficSplit resource, diverting a proportion of traffic to connected clusters. This dynamic load balancing mechanism ensures that the system efficiently handles varying workloads while maintaining performance.

Overall, the decentralized and adaptive approach of the DeFog application enables continuous evaluation, dynamic traffic routing, and orchestration of service deployments and deletions. This flexibility optimizes resource utilization, enhances system responsiveness, and ensures a reliable service environment.

4 System Design

In this chapter, we delve into the implementation phase of our service placement strategy, focusing on the display of the cluster's infrastructure and the essential components required for its successful execution. We begin by outlining the fundamental metrics that serve as the foundation for our placement and load-balancing strategies. Additionally, we present and thoroughly analyze the algorithms that form the core of our implementation framework. Next, we analyze the cluster's architecture and the application we implemented namely DeFog to perform the placement and load balancing. Lastly, we will present the benchmark applications, in which we will perform the proposed placement strategies.

4.1 Performance Metrics

Performance metrics are important when designing and evaluating a distributed system especially for the proposed service placement strategies. The fundamental metrics often used in measuring performance are Requests Per Second (RPS) and Response Latency. These metrics will be collected using appropriate metric tools and will be utilized by the proposed service placement strategies to dynamically place microservices on a hybrid cloud-edge-fog infrastructure, ensuring that the system meets the required performance goals.

The RPS metric showcases the number of requests that a system receives in one second. Its purpose is to provide insights into the traffic load on a specific microservice, effectively illustrating its significance and demand within the system. In this thesis, the RPS metric is acquired from Prometheus, which is collected by Prometheus API. Services with higher RPS values indicate a higher volume of incoming requests, suggesting their increased significance for deployment.

The response Latency metric is commonly referred to as the time-to-first-byte (TTFB) and includes network travel time and server processing time. For this thesis, we utilize the 95th percentile latency metric to assess the performance of microservices. It represents the response time at which 95% of the requests exhibit latency equal to or less than that value, allowing us to focus on the tail end of the response time distribution, where response times may have a more pronounced impact on the overall user experience. This metric is also collected using Prometheus. By measuring the 95th percentile latency for each microservice, we can rank the services based on their response times. Services with lower response times are given higher priority and are more likely to be deployed in appropriate clusters, optimizing their placement for enhanced performance.

4.2 System Architecture

In single-node K3s architecture, the cluster runs on a single virtual machine (VM) and manages containerized applications using Kubernetes primitives. The cluster is responsible for orchestrating containerized applications' deployment, scaling, and management. The resources available to the node are fixed and determined by the VM's hardware specifications. The K3s control plane components are encapsulated within a single process, simplifying the operation and management of the cluster. The data plane of the cluster, responsible for executing and managing containerized applications, is comprised of containerd and the pods it oversees. Similar to traditional Kubernetes, a K3s node can run multiple Pods. Each Pod consists of one or more containers that host individual microservices or components of an application. The containers communicate with each other using Kubernetes services, with a well-defined DNS name and port number. Kubernetes Services can expose a set of pods to external or internal consumers. They provide a stable network endpoint (IP address and port) that other components within or outside the cluster can use to access the pods. The architecture of a single-node K3S cluster is illustrated in Figure 4.1.

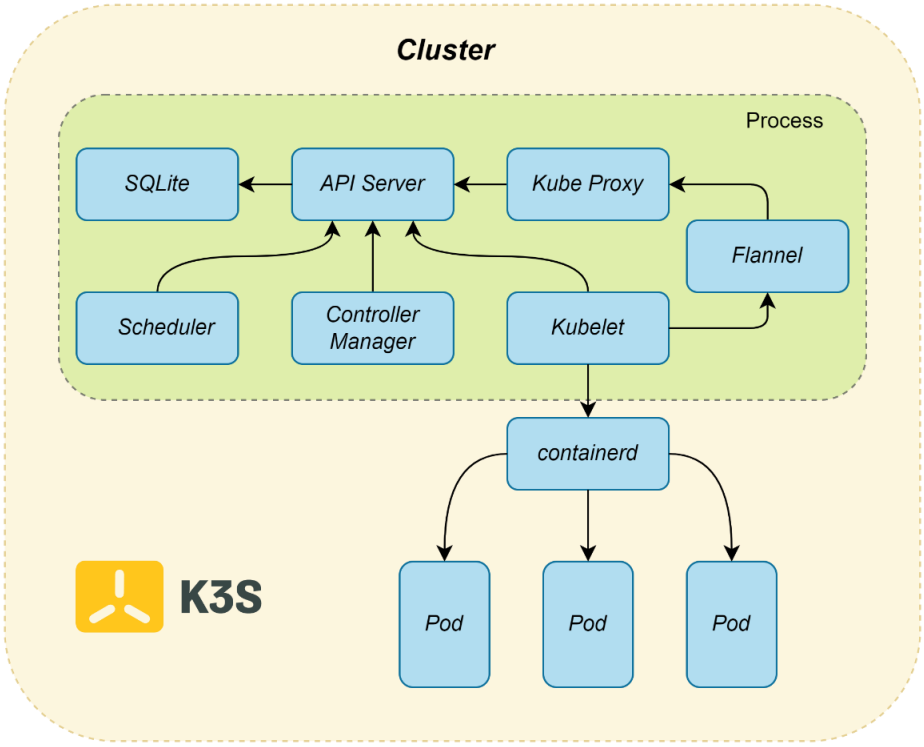


Figure 4.1: Single-Node K3s cluster

Deploying Linkerd service mesh in the cluster injects a sidecar proxy container alongside each pod in a Kubernetes deployment, which enables transparent traffic interception and routing. The sidecar proxy is responsible for managing all network traffic for the application pod, including traffic between other pods in the same namespace or across namespaces. When a request is made to a pod, the sidecar proxy intercepts the traffic and determines if it needs to be routed to another pod within the cluster. Figure 4.2 presents a Pod that has been injected by Linkerd.

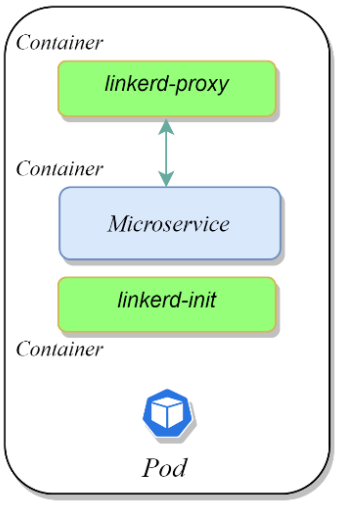


Figure 4.2: Injected Pod

Linkerd monitors the network traffic flowing through the proxies to provide observability and security features. With the installation of Linkerd Viz extension, we obtain enhanced observability into the service mesh such as visualizing the service topology. The built-in Prometheus instance extracts metrics by requesting data from the proxies, and the Grafana instance can be used to visualize this data. Moreover, Linkerd SMI provides us with the necessary TrafficSplit specification that allows to dynamically shift arbitrary portions of traffic destined for a Kubernetes Service to a different destination Kubernetes Service. Lastly, by installing Linkerd Multi-cluster we extend the benefits of Linkerd Service Mesh to multiple clusters, enabling seamless communication between microservices running on different clusters. In Figure 4.3, a comprehensive illustration showcases the installation of the Linkerd Service Mesh, including its extensions—Linkerd Viz and Linkerd Multi-cluster—in addition to other microservices. The illustration demonstrates the clear segregation of the Mesh's Control Plane microservices from the meshed ones, which comprise the Mesh's Data Plane. Notably, each pod belonging to the meshed microservices and the service mesh itself is equipped with the linkerd-proxy and linkerd-init. This configuration enables Linkerd to effectively monitor and observe its own constituent elements.

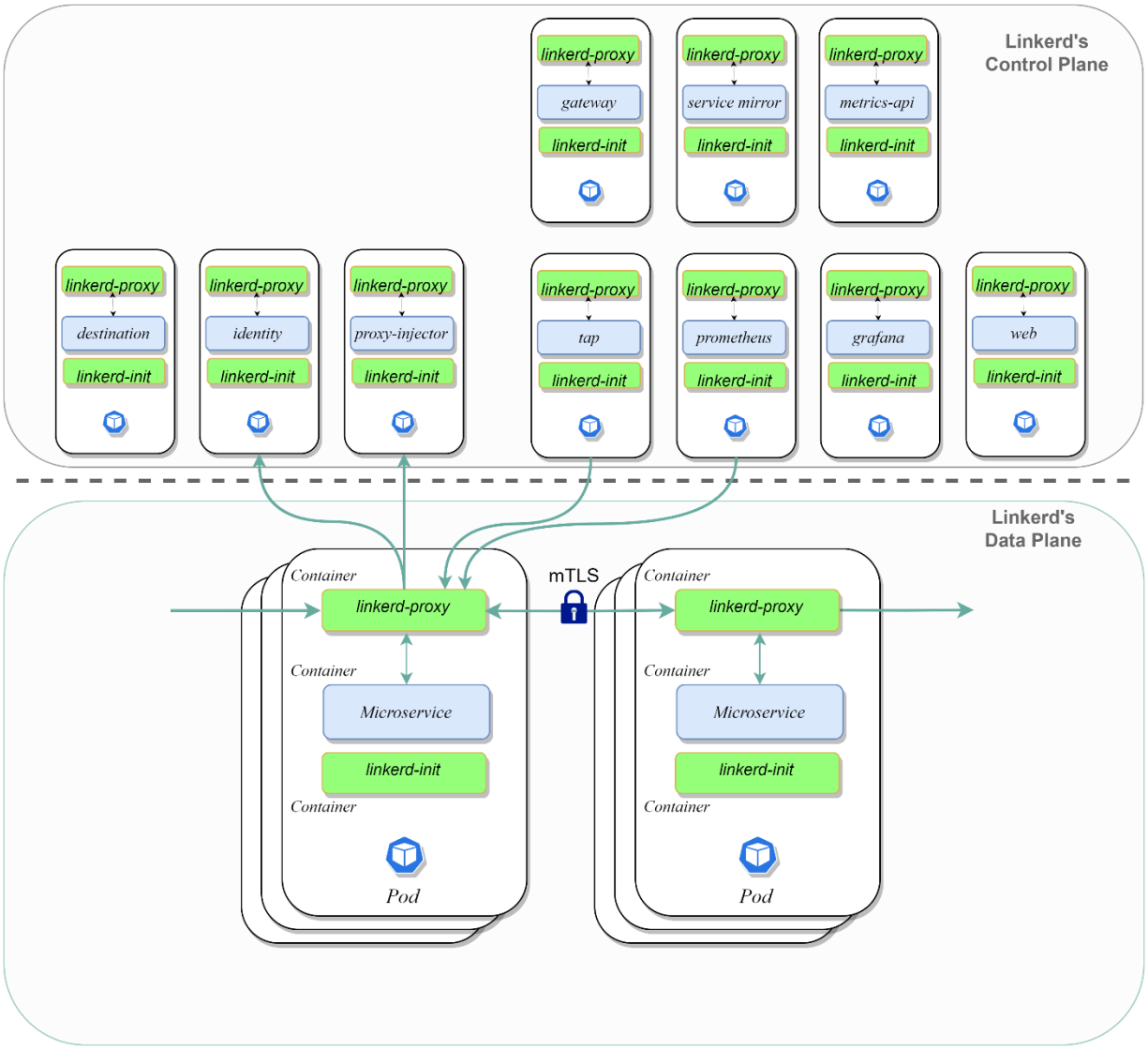


Figure 4.3: Linkerd's Architecture (Installation with extentions)

Taking full advantage of the powerful features provided by the Linkerd Service Mesh, in this thesis we are using multiple clusters of the above-described configuration, connected via the Multi-cluster extension. By leveraging this approach, we can seamlessly connect our microservices running on different clusters, enabling improved flexibility and scalability for our system architecture.

4.3 DeFog

DeFog represents a significant advancement in addressing the challenges of service placement in hybrid multicluster cloud-fog-edge infrastructures. Built using the Python programming language, DeFog leverages the power and versatility of two essential libraries: the Kubernetes Python client and the Prometheus API client. These libraries serve as fundamental building blocks, equipping Defog with the necessary tools to seamlessly engage with the Kubernetes API and harness the invaluable metrics furnished by Prometheus.

The Kubernetes Python client serves as a vital component in DeFog 's functionality by providing seamless integration with the Kubernetes API. Through this client, DeFog gains the ability to interact with the underlying Kubernetes infrastructure, including managing multiple Kubernetes resources, such as deployments, services, and traffic splits, and accessing resource information. The client empowers Defog to make informed decisions regarding the deployment and eviction of microservices based on real-time data and system requirements.

In parallel, DeFog capitalizes on the Prometheus API client, forging a vital link to the Prometheus HTTP API. Renowned as a robust monitoring and alerting solution in cloud-native environments, Prometheus provides an extensive array of metrics. By leveraging the Prometheus API client, DeFog gains access to rich and detailed metrics collected from the system, including RPS, response latency, resource utilization, and other performance indicators. These metrics serve as crucial inputs for the decision-making process employed by DeFog 's service deployment strategies.

DeFog incorporates two key features, efficient Service placement, and traffic management. To achieve efficient service placement, DeFog employs a continuous evaluation process that takes into account various system requirements. By analyzing metrics such as RPS, response latency, and RAM usage, DeFog dynamically determines the optimal placement of services based on their importance and resource demands. This evaluation process considers the Kubernetes resource requests and limits for CPU and RAM defined in the configuration files of the Services. By considering these specifications, DeFog accurately assesses the availability of resources in the cluster, ensuring that the deployment decisions align with the allocated resources and avoid resource contention. In addition to service placement, DeFog also focuses on traffic management, which plays a crucial role in load balancing and stress management. It intelligently splits the incoming traffic between the deployed instance within the local cluster and the mirrored services across other clusters. This dynamic traffic-splitting strategy helps achieve load balancing and efficiently manages stress.

These features operate periodically, constantly adapting to the system's needs to ensure optimal performance and resource utilization. The main process of DeFog is described in Algorithm 4.1.

Algorithm 1: DeFog main process.

```
1.  Establish Prometheus connection
2.  Create kubectl configuration
3.  Collect cluster data
4.  period <- 30 seconds
5.  While True:
6.          R,D <- getServices()
7.          If period then:
8.                  Run Service Placement Algorithm(R,D)
9.          Else:
10.                 currentResources, resourcesLimits <- gatherData()
11.                 For service in D:
12.                         weight <- calculateTrafficSplitWeight(service, currentResources, resourcesLimits)
13.                         updateServiceTrafficSplit(weight)
14.                 End For
15.         End If
16. End While
```

Algorithm 4.1: Defog main process

The algorithm described above focuses on the main process of DeFog. Upon establishing connection with Prometheus and creating the kubectl configuration by Kubernetes API client, DeFog initiates a loop that lists the deployed microservices (D), and the mirrored ones (R) whose requests are being forward connected clusters along with resource requirements. Then, the service placement algorithm is being executed at regular intervals of 30 seconds. For the remaining time, it gathers the current resources usage of each deployed microservice along with their limits defined at their configuration file and focuses updates TrafficSplits, based on the calculated weight.

In summary, DeFog combines algorithmic decision-making for service placement with traffic-splitting techniques to optimize resource utilization and improve service delivery. By considering metrics such as RPS, Response Latency, RAM usage, and Kubernetes resources' requests and limits, DeFog enables efficient deployment and eviction of services, while also ensuring load balancing and stress management through traffic splitting among mirrored services. The combination of these features makes Defog a powerful tool for enhancing the performance and scalability of service placement in complex infrastructures.

4.4 Service Placement Algorithms

Within a distributed system, the strategic placement of microservices across diverse infrastructures can significantly influence their overall performance. Hence, the implementation of effective service placement strategies becomes paramount in ensuring optimal system functionality. This thesis introduces a set of algorithms that dynamically places microservices on a hybrid infrastructure, taking into account comprehensive evaluations based on performance metrics such as RPS (Requests per Second), Response Latency and resource utilization.

4.4.1 Least Frequently Used (LFU)

Least Frequently Used (LFU) in general is a cache eviction [26] algorithm used to determine which items in a cache should be removed when the cache reaches its maximum capacity. The core concept behind LFU is that items that are least frequently accessed should be evicted first. When implementing the LFU strategy, DeFog aims to deploy services that have higher RPS values, indicating higher usage frequency. If there are available resources in the cluster, DeFog proceeds with deploying these services. However, if the cluster's resources are limited and cannot accommodate a requested service, DeFog compares the RPS values of the requested service with the already deployed services and evicts a deployed service if possible.

We define D as the already Deployed Services and R as the Requested Services to a given cluster. To perform the ranking, the RPS metrics are collected for each of the Services in D and R. Services with bigger RPS are ranked higher. LFU algorithm will be utilized as presented in Algorithm 4.2.

```
Algorithm 2: LFU (R, D)

1.  Sort R by RPS
2.  Sort D by RPS
3.  While R not empty do:
4.          serviceToDeploy <- R[0]
5.          If serviceToDeploy in D:
6.                  R.pop(0)
7.                  continue
8.          Else if checkAvailableResources(serviceToDeploy) == True then:
9.                  Deploy serviceToDeploy
10.                 R.pop(0)
11.         Else if D not empty then:
12.                 serviceToDelete <- D[Last]
13.                 If serviceToDelete.requests < serviceToDeploy.requests then:
14.                         Delete serviceToDelete
15.                         D.remove(Last)
16.                 End If
17.         End If
18. End While
```

Algorithm 4.2: LFU

The algorithm described above focuses on the dynamic placement of Services, considering their Requests Per Second (RPS) metric as a key factor. Here's a breakdown of the algorithm's steps:

1. Sorts the deployed Services and requested Services lists based on the RPS metric in descending order, indicating their priority.
2. Iterates through the requested Services list and attempt to deploy each service according to its priority.
3. If a requested Service is already deployed, proceed to the next one without taking any action.
4. Else it checks if there are sufficient available RAM and CPU resources to deploy the current requested Service.
5. If the required resources are available, it deploys the Service.
6. If the necessary resources are not available, the algorithm proceeds to evict a deployed Service from the list. However, eviction only occurs if the least important deployed Service has a lower RPS value than the Service scheduled for deployment. This condition ensures that the Service being deployed is of higher importance.
7. After eviction, the algorithm retries the deployment of the requested Service that was initially unable to be deployed.

By sorting the Services based on RPS and considering resource availability, the algorithm aims to effectively utilize the available resources while prioritizing the deployment of high-demand services. This approach ensures that critical services receive appropriate resources and are deployed efficiently in the infrastructure, resulting in optimal performance and resource utilization.

4.4.2 Response Latency-based Service Deployment (RLSD)

The proposed strategy, referred to as the “Response Latency based Service Deployment” (RLSD) takes advantage of the Response Latency metric to make informed decisions about service deployment. By considering response latency as a crucial factor, the RLSD strategy aims to optimize resource utilization and enhance user experience. When implementing the RLSD strategy, DeFog prioritizes the deployment of services with lower Response Latency values. If these services are deployed closer to the user end, it will enable faster response times and improved performance. If there are available resources in the cluster, DeFog proceeds with deploying these services. However, in scenarios where the cluster's resources are limited and cannot accommodate a new service, DeFog employs a decision-making process. It compares the Response Latency values of the new service with the already deployed services in the cluster. If feasible, DeFog evicts a deployed service with higher latency to make room for the deployment of a service with lower latency.

Similar to the LFU algorithm, the algorithm defines D as the set of already deployed services and R as the set of requested services within a specific cluster. To determine the ranking, response latency metrics are collected for each service in both D and R. Services with lower response latency are assigned higher priority in the ranking. This algorithm will be employed as outlined in Algorithm 4.3.

Algorithm 3: Response Latency based Service Deployment (R, D)

```
1.  Sort R by Response Latency
2.  Sort D by Response Latency
3.  While R not empty do:
4.          serviceToDeploy <- R[0]
5.          If serviceToDeploy in D:
6.                  R.pop(0)
7.                  continue
8.          Else if checkAvailableResources(serviceToDeploy) == True then:
9.                  Deploy serviceToDeploy
10.                 R.pop(0)
11.         Else if D not empty then:
12.                 serviceToDelete <- D[Last]
13.                 If serviceToDelete.responseLatency > serviceToDeploy. responseLatency then:
14.                         Delete serviceToDelete
15.                         D.remove(Last)
16.                 End If
17.         End If
18. End While
```

Algorithm 4.3: RLSD

The algorithm outlined above follows a similar approach to the LFU algorithm, but with a focus on the Response Latency metric of each service. Here is a breakdown of the algorithm's steps:

1. The deployed Services and requested Services lists are sorted based on their Response Latency metric in ascending order, reflecting their priority.
2. The algorithm iterates through the requested Services list, attempting to deploy each service according to its priority.
3. If a requested Service is already deployed, the algorithm moves on to the next one without taking any further action.
4. Else If the necessary RAM and CPU resources are available, the algorithm proceeds with the deployment of the current requested Service.
5. If the required resources are not available, the algorithm evaluates the possibility of evicting a deployed Service from the list. However, eviction only occurs if the least important deployed Service has a higher Response Latency value compared to the Service scheduled for deployment. This condition ensures that the Service being deployed is of higher importance.
6. After eviction, the algorithm retries the deployment of the requested Service that was initially unable to be deployed.

By considering the Response Latency metric, this algorithm prioritizes the deployment of services with lower response latency, aiming to enhance the overall performance and user experience.

4.4.3 Algorithm Variations

In low-resource systems like fog or edge computing, the implications of high RAM usage are amplified. These environments typically have limited resources compared to traditional data centers, making the impact of resource scarcity more significant. When a service consumes a substantial amount of RAM in such constrained settings, it exacerbates the competition for limited resources, resulting in resource contention. This contention manifests as challenges in allocating sufficient memory to each service, leading to delays in resource allocation and increased latency for affected services.

The consequences of high RAM usage and resource contention can directly affect the Quality of Service (QoS) requirements of applications in fog and edge computing. The increased latency caused by resource contention can result in missed deadlines, degraded user experience, and an overall decline in QoS. It's crucial to understand that when a service exhibits high RAM usage, it indicates its resource-intensive nature, which often translates to longer processing times for requests and generation of responses. Consequently, the latency of such a service tends to be higher compared to services with lower RAM usage.

However, by effectively managing high RAM usage, particularly through eviction strategies, it becomes possible to enhance overall latency performance. Evicting RAM-intensive services from the system liberates resources that can then be allocated to other services. This allocation optimization ensures that services receive the necessary resources, resulting in improved latency performance for the system.

Considering the significant impact of RAM usage on latency performance, this thesis introduces novel variations to the LFU and RLSD algorithms, named LFU-RAM and RLSD-RAM respectively. These algorithm variations maintain the strategic approach of LFU and RLSD in terms of service deployment based on RPS or Response Latency rankings. However, a key distinction lies in the eviction process when the cluster's resources reach their limits. If the cluster lacks sufficient resources to accommodate the new service, the algorithm compares the requested RAM of the new service with the deployed service that currently utilizes the most RAM resources in the cluster. If the new service requires less RAM, the algorithm evicts the deployed service with the highest RAM usage. This eviction process frees up resources and creates space for the new service to be deployed. This eviction process creates space and resources for other services that can utilize them more efficiently, alleviating resource contention, and ultimately improving overall latency performance.

4.5 Traffic Management Algorithm

In a distributed system, effective traffic management across diverse infrastructures is vital for achieving load balancing and stress management. This thesis presents an algorithm that dynamically handles traffic and intelligently splits it between deployed instances within the local cluster and mirrored services across other clusters. The algorithm takes into consideration the resource limits defined in the configuration of the Services. By utilizing the knowledge of these limits, DeFog calculates a weight that ensures that the traffic is distributed in a manner that optimizes resource usage and prevents the overloading of individual services. The calculation of this weight is described in Algorithm 3.4.

Algorithm 4: calculateTrafficSplitWeight (service, currentResources, resourceLimits)

```
1.  currentCPU <- currentResources.cpu[service]
2.  currentRAM <- currentResources.ram[service]
3.  desiredCPU <- resourceLimits.cpu[service] * THRESHOLD
4.  desiredRAM <- resourceLimits.ram[service] * THRESHOLD

5.  weight <- ( currentCPU / desiredCPU ) + ( currentRAM / desiredRAM)

6.  If weight > 1 then:
7.          weight <- weight – 1
8.  Else
9.          weight <- 0
10. End If
11. return weight
```

Algorithm 4.4: TrafficSplit Weight Calculation

In this algorithm, the current CPU and RAM utilization of the service is compared to the desired CPU and RAM limits multiplied by a threshold value. It aligns with the principles of the Kubernetes Horizontal Pod Autoscaler (HPA) which operates based on the ratio between the desired metric value and the current metric value [27]. If the weight exceeds 1, it indicates that the service's resource utilization has exceeded the desired limits by a certain margin. This excess weight signifies the additional load that needs to be distributed among the mirrored services.

By leveraging this concept, inspired by the Kubernetes HPA, DeFog dynamically adjusts the traffic splitting among mirrored services based on the weight calculation. This allows for efficient load balancing and ensures that services experiencing high CPU and RAM utilization are appropriately “calmed” by redistributing the excess load to the available computational resources distributed across multiple clusters.

4.6 Benchmark Applications

The present study employs two distinct applications to evaluate and validate the proposed service placement strategies. The chosen applications are Google's Online Boutique [28] and iXen [29], [30]. These benchmark applications are recognized for their versatility, complexity, and overall representativeness of modern cloud-based systems. They are co-deployed within multiple K3s clusters, accompanied by the Linkerd Service Mesh and DeFog, the application developed for this study. The DeFog will execute specific decisions based on each placement strategy that is being evaluated.

4.6.1 Google's Online Boutique

Online Boutique is an open-source application that acts as a reference application developed by Google. By leveraging this application, developers can be introduced to microservices, and experiment with different scaling strategies, fault tolerance techniques, and deployment models. Google uses this application to demonstrate the use of technologies like Kubernetes/GKE, Istio, Stackdriver, and Grpc [28].

The Online Boutique application consists of several microservices that work together to provide a complete e-commerce experience. These microservices' functionalities include product catalog, shopping cart, checkout, payment processing, and more. Each microservice is designed independently, allowing for greater flexibility and scalability.

It utilizes gRPC for communication between its services. gRPC is a high-performance remote procedure call (RPC) framework that enables efficient communication between distributed systems. In the Google Online Boutique architecture, each microservice is responsible for a specific function, and communication between services occurs via gRPC APIs. This approach allows for efficient and scalable communication between services providing a scalable e-commerce platform to its users.

As described on the official GitHub page of the application [28], the Online Boutique is composed of 12 microservices written in different languages. Figure 4.4 provides a detailed overview of Online Boutique's architecture. More specifically:

- **Frontend:** Written in Go. Exposes an HTTP server to serve the website. Does not require signup/login and generates session IDs for all users automatically.
- **Cart Service:** Written in C#. Stores the items in the user's shopping cart in Redis and retrieves it.
- **Redis Cart:** Redis database. Cart Service stores its data in this in-cluster Redis database.
- **Product Catalog Service:** Written in Go. Provides the list of products from a JSON file and the ability to search products and get individual products.
- **Currency Service:** Written in Node.js. Converts one money amount to another currency. Uses real values fetched from European Central Bank. It's the highest QPS service.
- **Payment Service:** Written in Node.js. Charges the given credit card info (mock) with the given amount and returns a transaction ID.

- **Shipping Service:** Written in Go. Gives shipping cost estimates based on the shopping cart. Ships items to the given address (mock)
- **Email Service:** Written in Python. Sends users an order confirmation email (mock).
- **Checkout Service:** Written in Go. Retrieves user's cart, prepares orders, and orchestrates the payment, shipping, and email notification.
- **Recommendation Service:** Written in Python. Recommends other products based on what's given in the cart.
- **Ad Service:** Written in Java. Provides text ads based on given context words.
- **Load Generator:** Written in Python. Continuously sends requests imitating realistic user shopping flows to the frontend.

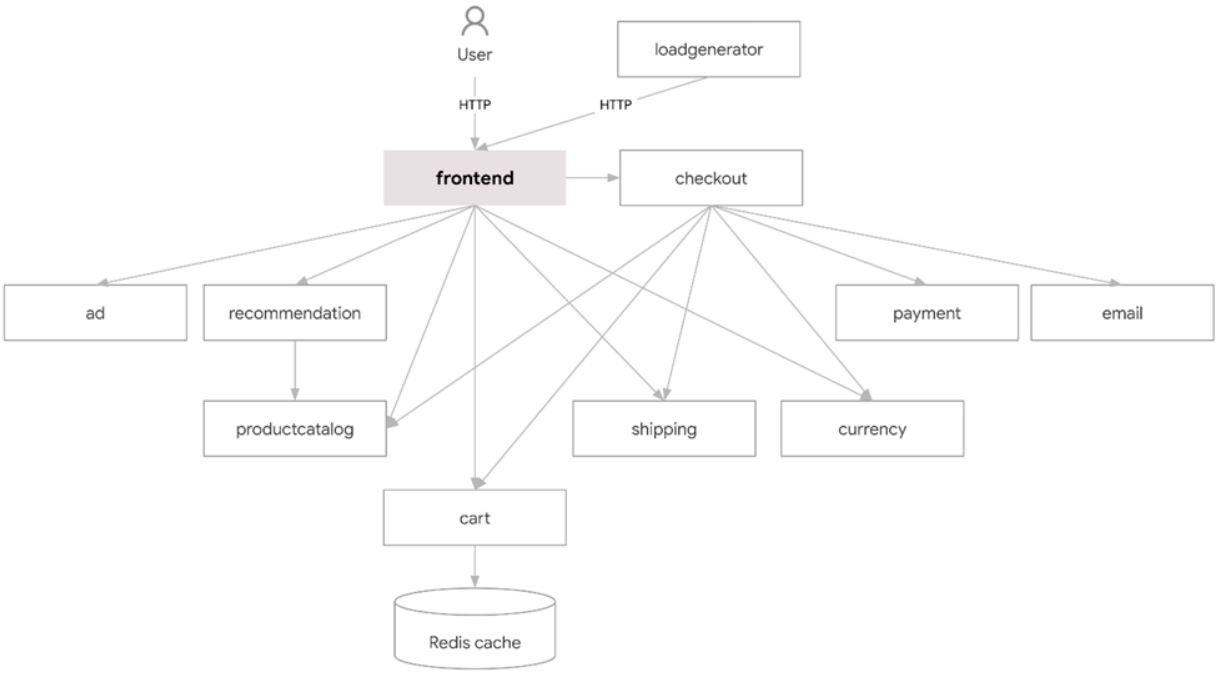


Figure 4.4: Google Online Boutique Architecture [28]

We must note that during the evaluation process, the Load Generator microservice is not used, as its sole purpose is to test the application, continuously sends requests to the frontend microservice. The Google Online Boutique is a well-established microservices reference application that has been a great resource for developers and architects seeking to enhance their knowledge of microservices architecture, container orchestration, scaling, and management. To summarize, with its versatile, complex, and representative design, the Google Online Boutique provides a powerful tool for evaluating and developing cloud-based systems, ensuring the reliability and validity of research findings.

4.6.2 iXen

iXen [29], [30], is a platform for an IoT scenario based on the Service Oriented Architecture (SOA) principles. The platform demonstrates a remarkable capability for efficient and effective management of a diverse range of sensors. These sensors, encompassing various models, seamlessly transmit their measurements to the cloud computing system of the iXen platform. Its architecture is founded upon a publish/subscribe model, which empowers users to effortlessly subscribe to the data from specific sensors. This subscription grants them the flexibility to utilize and access the data in their desired format, catering to their specific requirements.

The iXen platform is meticulously implemented as a comprehensive three-tiered IoT system, thoughtfully designed to cater to the distinctive needs of different user types. Each tier has its unique functions and serves a specific purpose for Infrastructure Owners, System Developers, and Customers. At the first tier of the system, Infrastructure Owners have the ability to install and connect their physical devices to the system. The second tier of iXen is designed to support System Developers who can subscribe to the devices installed by Infrastructure Owners. This allows them to access and use the devices for building various applications using the data gathered by them. Finally, the third tier of iXen is tailored for the Customers, who can subscribe to various applications offered by System Developers. This allows them to access these applications and utilize them for various purposes. Figure 4.5 shows the microservices architecture of iXen.

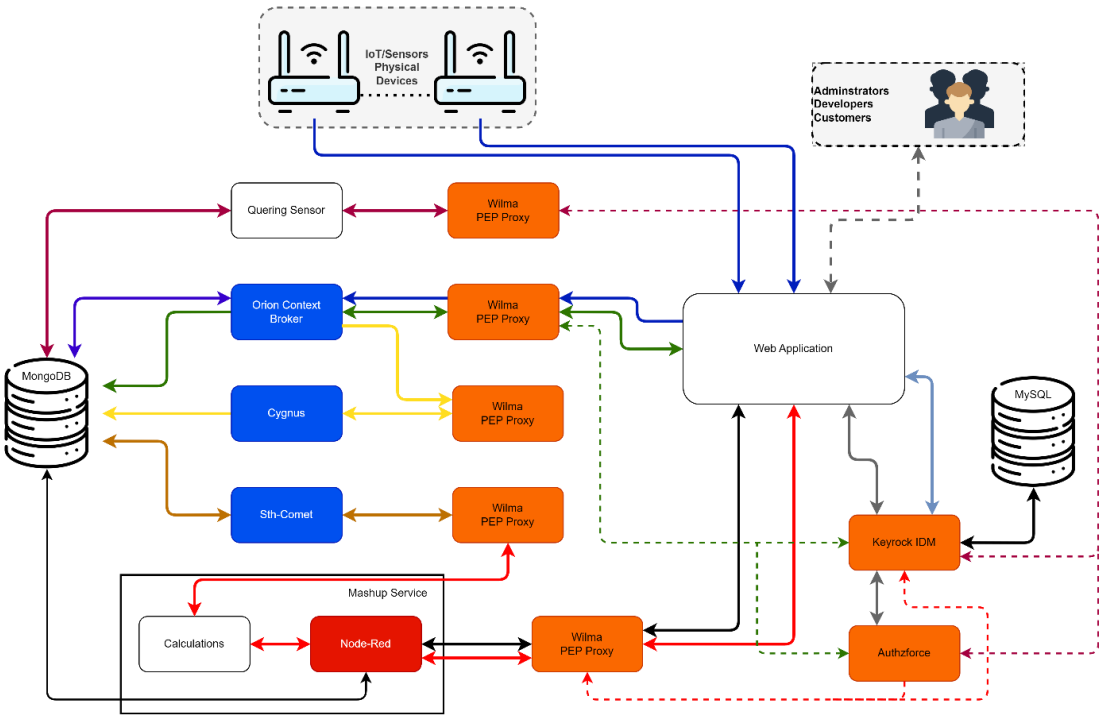


Figure 4.5: iXen's Architecture

As illustrated in Figure 4.5 of the system architecture, iXen is composed of 15 distinct microservices. The intercommunication between these services is facilitated through the use of Restful APIs, enabling

seamless data exchange and interaction. To facilitate clear comprehension of the system architecture, each edge of the diagram is color-coded to represent the different functions and processes of the system. The following is a comprehensive overview of the individual microservices that form the fundamental components of iXen, allowing for the delivery of an efficient and comprehensive IoT solution.

- **Web Application:** Written in PHP and Node.js, this service serves as the core of the entire system. It exposes a Web UI for users to access while orchestrating every other service and forwarding each user request to the proper services.
- **Quering Sensors:** This service provides translation of a custom query to a MongoDB query. It is used to search the MongoDB and retrieve devices registered in the system, information about these devices, and measurements they provide.
- **Mashup Service:** This component consists of two services, a Node-Red instance, and a Calculations service. Node-Red is used to create applications on the system while the Calculations service is responsible for retrieving data, needed for the applications, from the Sth-Comet service.
- **Keyrock IDM:** It is a project of FIWARE. This component is responsible for Identity Management. It provides the functionality of user and role management, storing the necessary information in a MySQL database.
- **AuthzForce:** It is a project of FIWARE. It provides an API to get authorization decisions based on authorization policies, and authorization requests from PEPs. The API follows the REST architecture style [31].
- **PEP Proxy - Wilma:** It is a project of FIWARE. Combined with the other security components, Keyrock and Authzforce, it enforces access control to the system. Every request to each service is forwarded by this proxy. This secures the backend system by allowing only permitted users with specific roles to access the requested service.
- **Orion Context Broker:** It is a project of FIWARE. It is an NGSIv2 server implementation to manage context information, providing a powerful mechanism for subscribing to and receiving real-time updates about changes to context data stored in MongoDB. This mechanism is based on the publish-subscribe pattern, where subscriptions to physical devices or applications can be created.
- **Cygnus:** It is a project of FIWARE. This component serves as a bridge between the Orion Context Broker and external big data storage systems. It can capture and store real-time data streams from the Orion Context Broker, and create a historical view of the data. Cygnus provides a common interface for working with different databases allowing developers to choose the database that best fits their needs and requirements, without being tied to a specific technology.
- **Sth-Comet:** It is a project of FIWARE. It is in charge of managing historical raw and aggregated time series information related to the evolution of context data registered in an Orion Context Broker instance [32]. Its primary function is to store and retrieve large volumes of historical data providing the ability to analyze and process historical trends and patterns over time.
- **MongoDB:** It is used to store required data about the registered physical devices and their measurements, and applications on the system.
- **MySQL:** Keyrock IDM uses this database to store and retrieve data.

5 Experimental results

This chapter presents a detailed analysis of the performance of the proposed DeFog application in the context of service placement within hybrid multicluster cloud-fog-edge infrastructures. Through a comprehensive examination of the test infrastructure, the applied request test plan, and the resulting outcomes in terms of latencies and the services deployed in each cluster under various load scenarios, valuable insights into the effectiveness and efficiency of the proposed Service Placement Algorithms are gained. By thoroughly evaluating the experimental results, we can better understand how the proposed Service Placement Algorithms manage service placement and optimize system performance.

5.1 Infrastructure

For the experiments, we utilize multiple Virtual Machines (VMs) in the Google Cloud Platform (GCP) for the Multi-cluster Kubernetes environment. In particular, this architecture has five VMs distributed across multiple regions. These VMs are virtualized instances running on GCP's infrastructure, providing a scalable and flexible environment for hosting applications and services. In each VM a k3s is installed representing a single-node cluster. K3s is a popular choice for deploying Kubernetes clusters on VMs due to its minimal resource footprint and simplified installation process. Table 5.1 displays the essential characteristics of the clusters.

Cluster Attributes	**Options**
VM Boot Disk Image	debian-cloud/debian-10
Location Type	Zonal
K3s Cluster Version Type	Stable
K3s Cluster Version	v1.19.3+k3s3
Horizontal Autoscaling	Disabled
Boot Disk Size	50 GB

Table 5.5.1: Cluster Characteristics (1)

In order to simulate the cloud-fog-edge infrastructure, we have carefully designed each cluster in the multicluster architecture with specific characteristics, such as available CPU, RAM, machine type, and placement in different zones. This meticulous approach enables us to closely mimic the diverse nature of cloud, fog, and edge computing environments. Each cluster within the architecture represents a distinct tier, offering varying resource capacities and geographical locations. By incorporating these varied characteristics, we can accurately replicate the complexities and challenges associated with managing services in such environments. Table 5.2 displays a detailed overview of the unique characteristics of each cluster.

Cluster Attributes	Cluster 1	Cluster 2 and 3	Cluster 4 and 5
Zone	europe-west2-a	europe-west3-a/b	europe-west6-a/b
Machine Type	e2-standard	e2-standard	e2-custom
vCPU	8	4	2
RAM (GB)	32	16	3

Table 5.5.2: Cluster Characteristics (2)

To establish the interconnectedness and hierarchical structure of the clusters, we begin by deploying Linkerd Service Mesh, along with its extensions, on each cluster within the multicluster architecture. The Linked Multi-cluster extension proves to be invaluable as it enables us to establish links between the clusters in a hierarchical placement order, resulting in a three-layered pyramid of clusters. At the topmost layer, we have Cluster 1, which represents the Cloud. The second layer consists of Clusters 2 and 3, representing the Fog layer. Lastly, on the third layer, we find Clusters 4 and 5, constituting the Edge layer. The hierarchical arrangement and interconnectedness of the clusters can be visualized in detail in Figure 5.1.

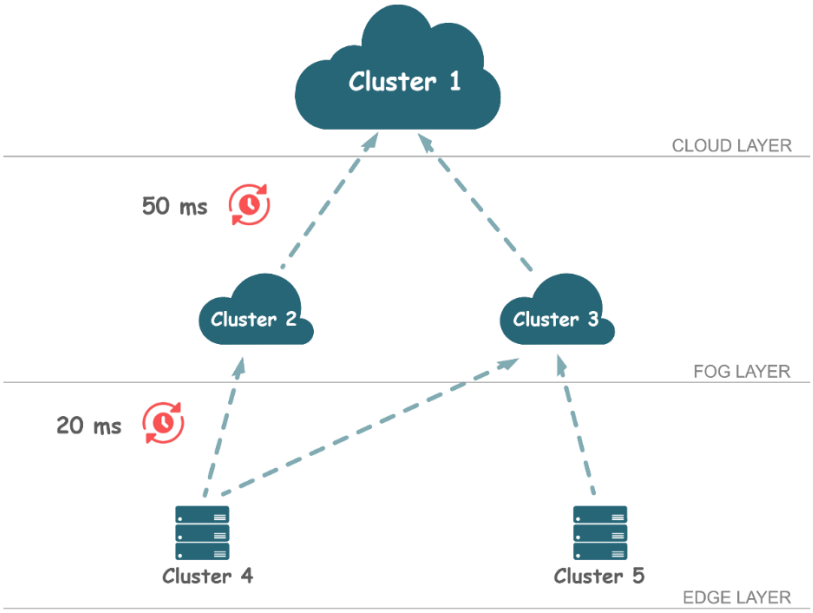


Figure 5.1: Cluster's arrangement

Furthermore, in our experimental setup, we made use of the Linux Traffic Control (tc) [33] command line tool. Leveraging the capabilities of tc, we had fine-grained control over network parameters, particularly latency. This allowed us to introduce realistic network conditions and simulate the effects of increased distances between clusters, thereby creating an accurate representation of a distributed cloud-fog-edge infrastructure. As depicted in Figure 5.1, we incorporated additional delays between each layer. Specifically, a delay of 50ms was applied between the Cloud and Fog layers, while a delay of 20ms was implemented between the Fog and Edge layers, in addition to the time for answering the request.

After configuring the infrastructure, the next step involved initializing the placement of our benchmark applications. As the Benchmark Applications section mentioned, we utilized two microservice-based applications: Google's online boutique, consisting of 12 microservices, and iXen, comprising 15 distinct microservices. For initialization of the applications, the majority of microservices were deployed in the Cloud cluster, taking advantage of its higher resource capacity and centralized nature. However, we adopted a different approach for the frontend microservices. Each application's frontend microservices (Frontend and Web Application) were deployed in both clusters of the Edge layer. In the final step of our experimental setup, we deployed the DeFog application on each cluster, excluding the Cloud cluster as it was not required for the deployment.

During this phase, the frontend microservices were made ready to handle incoming traffic and DeFog carried out the required deployments and TrafficSplit management across each cluster. However, it was imperative to consider specific microservices that should not be deployed in multiple clusters. Specifically, the database services of each application (Redis Cart, MongoDB, MySQL) were intentionally excluded from deployment in other clusters to prevent any potential data inconsistencies. Furthermore, the Keyrock IDM microservice necessitated reconfiguration every time it would be deployed in any cluster and was intentionally restricted from being deployed across multiple clusters. Lastly, the frontend microservices were also exempted from deployment in multiple clusters to uphold the integrity and consistency of the user interface.

5.2 Request Test Plan

Within this section, we present the load testing process conducted to assess the performance of the benchmark applications, iXen and Google's online boutique, under various workload intensities. To achieve this, we employed Locust, a powerful tool chosen for its suitability in simulating realistic user scenarios. By subjecting the benchmark applications to different workload intensities, we aimed to gain valuable insights into the placement decisions made by each algorithm and evaluate the system’s overall performance. This section provides a comprehensive overview of the load testing methodology, including the types of requests generated, the duration of each test, and the distribution of loads across the clusters. The experimental results obtained from these tests serve as a foundation for analyzing the system's performance and assessing its efficiency in managing the placement of services within the cloud-fog-edge infrastructure.

Using the Locust tool, to generate synthetic workloads, we implemented Python code to simulate user behaviors and swarm both benchmark applications, iXen and Google's online boutique, simultaneously with multiple concurrent users. Our load testing strategy involved three different request distributions in a sequence, each lasting five minutes, resulting in a total load testing duration of 15 minutes. Specifically, we designed a test plan where the requests are evenly distributed across both Edge Layer Clusters where the frontends of the applications reside. These requests were carefully selected from the available request types, with some requiring additional input parameters to configure their distribution and endpoints. For a comprehensive overview, Table 5.3 provides a detailed list of all the requests employed to simulate application workflows and interactions among the microservices.

Application	Request	Type	Requests Distribution		
			Load 1	Load 2	Load 3
iXen	Login into the App	POST	3.6%	5.2%	8.4%
	Index	GET	10.6%	17.6%	10%
	Search Application Subscriptions	GET	7%	0%	6.6%
	Search for an existing Application	POST	7%	1.8%	13.4%
	Search for an existing Sensor	POST	7%	12.4%	1.6%
	Search Sensor Subscriptions	GET	7%	1.8%	8.4%
	Access Device Measurements	POST	7%	14.2%	3.4%
	Subscribe to Application	POST	3.6%	0%	6.6%
	Subscribe to Sensor	POST	7%	7%	1.6%
Google's Online Boutique	Add to Cart	POST	4.2%	0%	9.2%
	Browse Product	GET	21%	10.6%	6.2%
	Checkout	POST	2.2%	0%	3%
	Index	GET	2.2%	10.6%	12.4%
	Set Currency	POST	4.2%	18.8%	3%
	View Cart	GET	6.4%	0%	6.2%

Table 5.5.3: iXen and Google's Online Boutique Requests and Test Plans

To evaluate the effectiveness of each placement strategy, we perform a series of tests based on the plan outlined in Table 5.3. With these tests, we measure the average, 90th and 95th percentile of the response latency produced by each load after the placement of services occurred. Specifically, we conduct a test to determine the optimal threshold for the calculateTrafficSplitWeight algorithm. This test aimed to find the threshold value that maximizes the effectiveness of the traffic splitting feature. Additional tests were conducted for the LFU, RLSD, LFU-RAM, and RLSD-RAM strategies. Each test was executed at a consistent rate of 50 requests per second (RPS), providing a standardized workload for comparative analysis. As a result, we can obtain valuable information about the performance and effectiveness of the different placement strategies as well as gain insights into the traffic splitting feature of DeFog.

5.3 Results

In our implemented infrastructure, we established five separate single-node K3s Clusters located in different Zones. DeFog, our service placement application, utilizes the Kubernetes API and Prometheus to gather crucial metrics such as current and available RAM and CPU usage, resource requirements, and request per second (RPS) statistics from each cluster. The placement process of DeFog is executed periodically, ensuring continuous optimization. To evaluate the effectiveness of each microservice placement strategy, we subjected the applications to synthetic workloads, generating network communication among the microservices. In the subsequent sections, we present the outcomes of each algorithm's placement at varying request load plans, alongside the corresponding response latency measurements before and after the placement occurred. These results shed light on the performance and efficiency of our service placement algorithms in the cloud-fog-edge environment.

5.3.1 Initial Microservice Placement

In this subsection, we provide an overview of the initial placement of services for both applications and examine the response latency at three different load levels. As previously mentioned, all microservices are initially deployed in the cluster belonging to the Cloud layer (Cluster 1), except for the frontend of each application, which are deployed in both clusters of the Edge Layer (Cluster 4 and Cluster 5). To provide a comprehensive understanding, Table 5.4 presents a detailed list of the deployed microservices, while Chart 5.1 illustrates the response latency data. This analysis allows us to gain insights into the initial placement and evaluate the performance of the system under different load conditions.

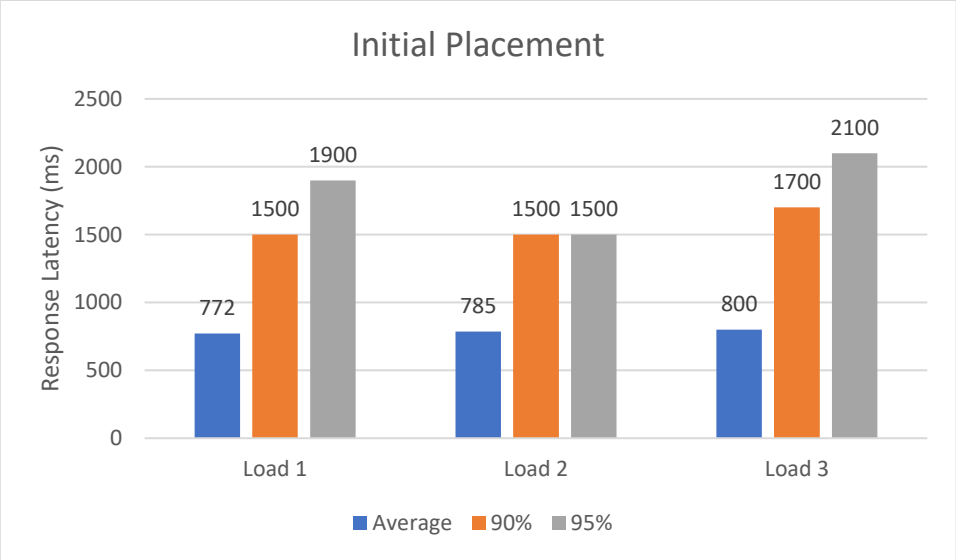


Chart 5.1: Initial Placement Response Latency

Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
adservice authzforceservice cartservice checkoutservice currencyservice cygnusservice cygnusserviceproxy emailservice keyrock noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice rediscart shippingservice sthcometservice sthcometserviceproxy	-	-	apache frontend	apache frontend

Table 5.4: Initial Service Placement

The distribution of requests in each load scenario has a noticeable impact on the response times of the initial placement. Load 1, with a significant proportion of “Browse Product” requests, maintains an average response time of around 772 ms. Load 2, characterized by a higher percentage of “Access Device Measurements” and “Set Currency” requests, shows a similar average response time of 785 ms. Load 3, featuring a larger percentage of “Search for an existing Application” and “Add to Cart” requests, exhibits a slightly higher average response time of 800 ms, with occasional spikes in the 95th percentile response time. These findings highlight how the distribution of requests within each load can influence response times demonstrating variations in performance for different types of requests.

5.3.2 Traffic Management Algorithm's Threshold Calculation

In the test aimed at determining the optimal threshold for the Traffic Management feature of Defog, we utilized the request distribution pattern from Load 1. The LFU Service Placement strategy, which had shown promising results in [8], was employed for this test. The objective was to assess the impact of different thresholds in the traffic splitting algorithm. This test was conducted with threshold values including 1, 0.9, 0.8, 0.7 and 0.6. These thresholds determined the maximum resource usage, such as RAM and CPU, that a microservice could utilize without becoming stressed. Based on these limits, the algorithm calculated the proportion of requests that would be split between the deployed services and the mirrored services. By adjusting these thresholds, we could observe the distribution of workload and its effect on system performance. Table 5.5 provides a comprehensive overview of the Services that were deployed using the LFU algorithm.

Cluster 2	Cluster 3	Cluster 4	Cluster 5
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice checkoutservice curren- cyservice frontend productcatalogservice shippingservice	adservice apache cartservice currencyservice frontend productcatalogservice recommendationservice

Table 5.5: Load 1's LFU Service Placement

The results of this experiment are presented in Chart 5.2, illustrating the latencies observed for each threshold of the traffic splitting algorithm. Additionally, Table 5.6 shows the percentage of requests that the affected deployed Services handled, providing insights into the workload distribution.

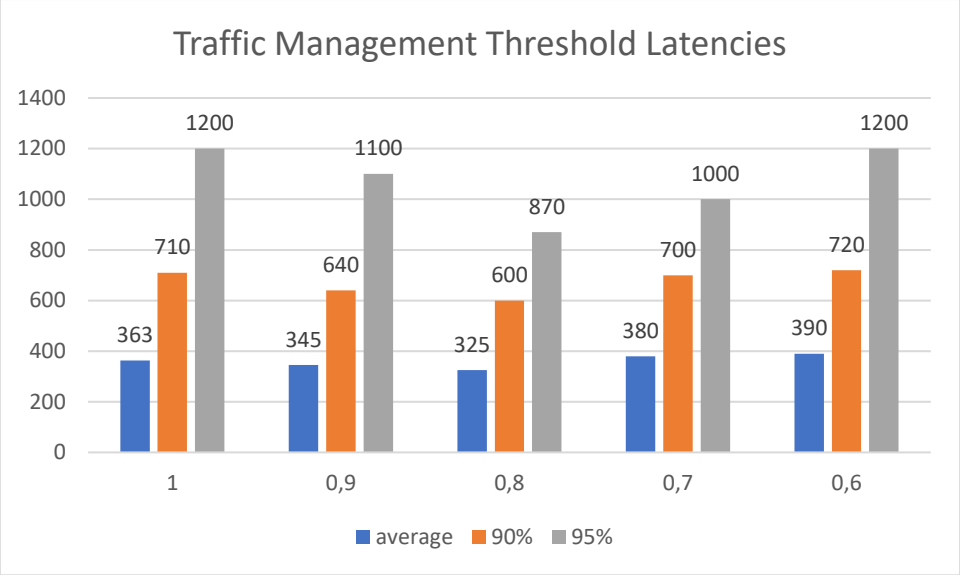


Chart 5.2: Traffic Management Threshold Latencies

Threshold	Cluster 2	Cluster 3	Cluster 4	Cluster 5
0.9	-	-	currencyservice – 99%	currencyservice – 98%
0.8	-	-	currencyservice – 86% productcatalogservice – 96%	currencyservice – 94% productcatalogservice – 98%
0.7	-	-	adservice – 95% currencyservice – 72% productcatalogservice – 91%	currencyservice – 75% productcatalogservice – 95%
0.6	-	adservice – 97% orionproxyservice – 88%	adservice – 88% currencyservice – 65% productcatalogservice – 80%	currencyservice – 66% productcatalogservice – 80%

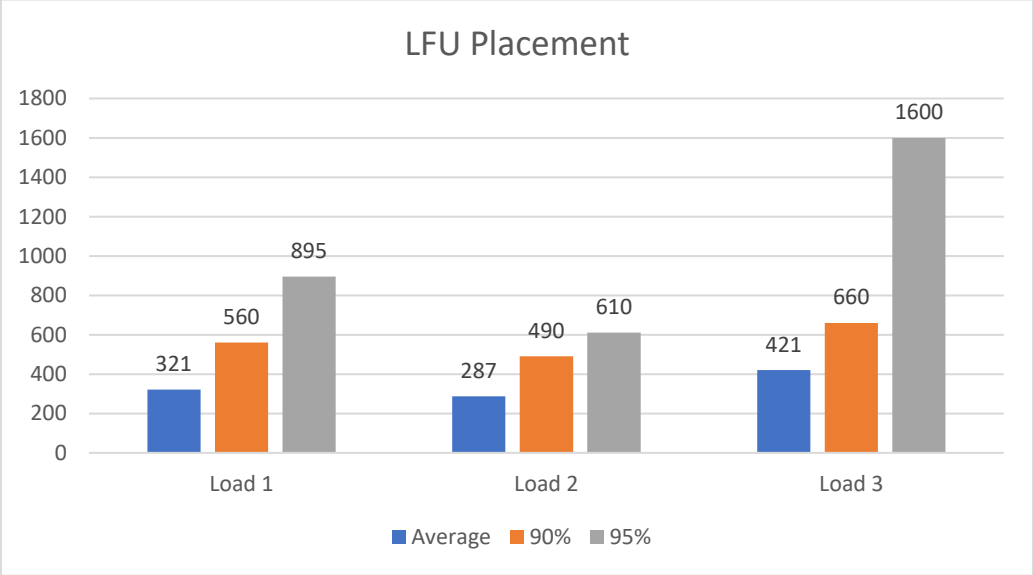
Table 5.6: Request percentages on deployed Services

Among the range of thresholds tested, the threshold of 0.8 emerged as the most promising in terms of performance improvement. Notably, this threshold substantially reduced the average response time, bringing it down to 325 ms. The 90th and 95th percentiles also exhibited notable improvements, with response times of 600 ms and 870 ms, respectively. The observed performance improvement can be attributed to the balanced distribution of requests across the deployed services. By strategically splitting the traffic based on the selected threshold, the workload was effectively distributed among the microservices. This optimized workload allocation led to more efficient handling of requests and, consequently, faster response times.

It is important to emphasize that the ideal threshold for traffic splitting may vary depending on the specific application and the characteristics of the workload it handles. The observed performance improvements at a threshold of 0.8 are promising but it is crucial to consider the unique requirements and constraints of different use cases.

5.3.3 Least frequently Used (LFU)

The LFU (Least Frequently Used) placement strategy demonstrated its effectiveness in improving response times compared to the initial placement, considering the frequency of Service usage. The response latency data in Chart 5.3 and the detailed list of deployed microservices in Table 5.7 provide a comprehensive overview of the improved placement achieved by the LFU strategy.



Cluster 2	Cluster 3	Cluster 4	Cluster 5
Load 1			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice checkoutservice currencyservice frontend productcatalogservice shippingservice	adservice apache cartservice currencyservice frontend productcatalogservice recommendationservice
Load 2			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice currencyservice frontend productcatalogservice	adservice apache cartservice currencyservice frontend productcata- logservice recommendationservice
Load 3			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice currencyservice frontend productcatalogservice shippingservice	adservice apache cartservice currencyservice frontend productcatalogservice recommendationservice

Table 5.7: LFU's Service Placement

In Load 1, the placement of the “productcatalogservice”, “cartservice” and “currencyservice” on the Edge Layer's clusters significantly impacted response times. By deploying this microservice closer to the users, the average response time decreased from 772 ms to 321 ms. Similarly, in Load 2, where “Set Currency” requests were prominent, key microservices such as “currencyservice”, played a crucial role in reducing the average response time from 785 ms to 287 ms. Lastly, in Load 3, there is a notable drop in the average response time from 800 ms to 421 ms. These observations highlight how the strategic placement of microservices based on request types can lead to improved response times.

5.3.4 LFU-RAM

The LFU-RAM placement strategy demonstrated its effectiveness in improving response times compared to the initial placement, considering the frequency of Service usage and resource contention. The response latency data in Chart 5.4 and the detailed list of deployed microservices in Table 5.8 provide a comprehensive overview of the improved placement achieved by the LFU-RAM strategy.

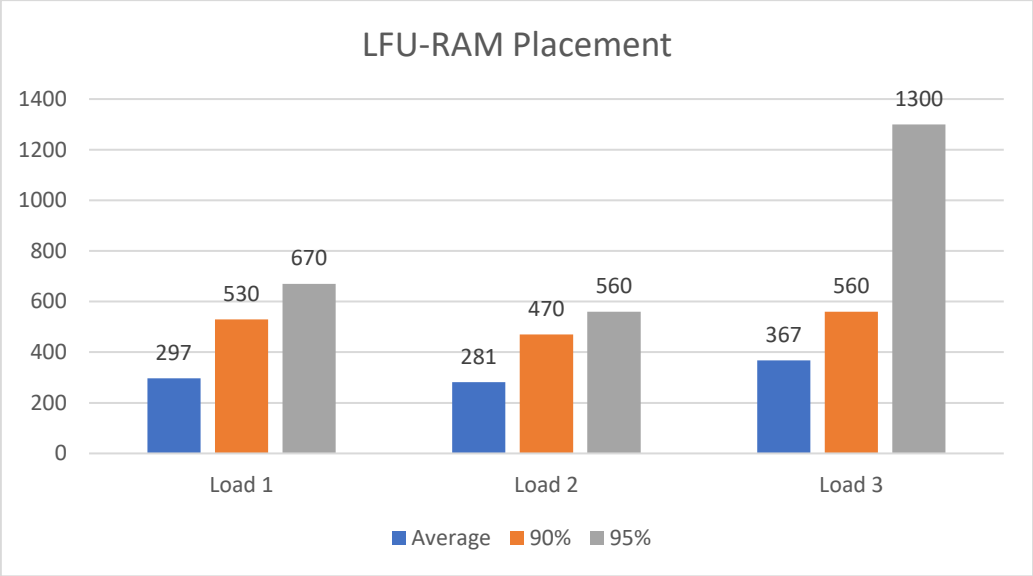


Chart 5.3: LFU-RAM's Service Placement Response Latency

Cluster 2	Cluster 3	Cluster 4	Cluster 5
Load 1			
adservice cartservice checkoutservice currencyservice noderedservice noderedserviceproxy orionservice orionserviceproxy queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice noderedservice noderedserviceproxy orionservice orionserviceproxy queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice currencyservice frontend productcatalogservice	apache cartservice currencyservice frontend orionservice orionserviceproxy productcatalogservice recommendationservice shippingservice
Load 2			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice checkoutservice currencyservice frontend productcatalogservice shippingservice	apache cartservice currencyservice frontend orionservice orionserviceproxy productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice
Load 3			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice checkoutservice currencyservice frontend productcatalogservice shippingservice	apache cartservice checkoutservice currencyservice emailservice frontend orionservice orionserviceproxy paymentservice productcatalogservice recommendationservice shippingservice

Table 5.8: LFU-RAM's Service Placement

In Load 1, the deployment of microservices such as “currencyservice”, “productcatalogservice”, and “cartservice” on both Cluster 4 and Cluster 5 significantly reduced the average response time from 772 ms to 297 ms, resulting in a noticeable improvement in performance. During Load 2, requests such as “Set Currency”, “Browse Product”, “Search for an existing Sensor” and “Access Device

Measurements" benefited from the presence of "currencyservice" and "productcatalogservice" in both Edge clusters and "orionservice", "orionserviceproxy", "queryingsensorsservice" and "queryings-ensorsserviceproxy" in Cluster 5, achieving an average response of 281 ms. Average response time in Load 3 also improvedfrom 800ms to 367ms.

5.3.5 Response Latency-based Deployment (RLSD)

The RLSD (Response Latency based Service Deployment) strategy demonstrated its effectiveness in optimizing response times by placing microservices based on the observed latencies. The response latency data in Chart 5.5 and the detailed list of deployed microservices in Table 5.9 provide a comprehensive overview of the improved placement achieved by the RLSD strategy.

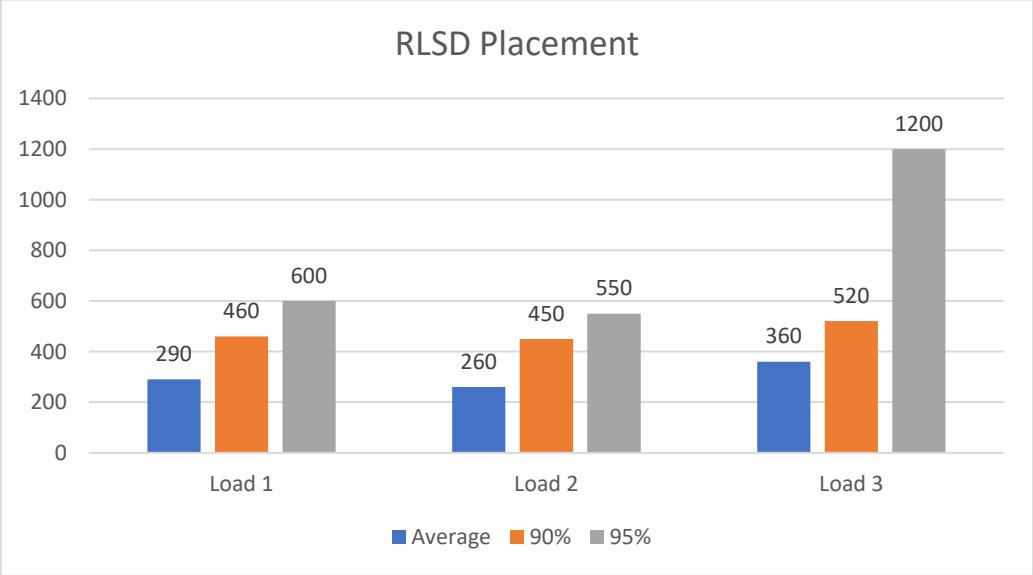


Chart 5.4: RLSD's Service Placement Response Latency

Cluster 2	Cluster 3	Cluster 4	Cluster 5
Load 1			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache frontend productcatalogservice shippingservice	apache currencyservice frontend orionservice orionserviceproxy productcatalogservice recommendationservice shippingservice
Load 2			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache currencyservice frontend productcatalogservice shippingservice	apache currencyservice frontend noderedservice noderedserviceproxy productcatalogservice recommendationservice shippingservice
Load 3			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache currencyservice frontend productcatalogservice shippingservice	apache currencyservice frontend noderedservice noderedserviceproxy productcatalogservice recommendationservice shippingservice

Table 5.9: RLSD's Service Placement

In Load 1, certain requests like “Set Currency”, “Browse Product”, and “Checkout” were benefited by microservices such as “currencyservice”, “productcatalogservice” and “shippingservice” deployed on the Edge Layer's clusters. This strategic deployment significantly reduced the average response time from 772 ms to 290 ms. During Load 2, “noderedservice” and “noderedserviceproxy” replaced “orion-service” and “orionserviceproxy” resulting in the improvement of the response time from 785 ms to 260 ms. During Load 3, there was no change in placement since Load 2, however average response time was reduced from 800 ms to 360 ms.

5.3.6 RLSD-RAM

The RLSD-RAM placement strategy demonstrated its effectiveness in improving response times compared to the initial placement. This strategy takes into account factors such as Response Latency by Services and resource contention to optimize the placement of microservices. The response latency data presented in Chart 5.6 demonstrates the significant improvements achieved through the implementation of the RLSD-RAM strategy. Additionally, Table 5.10 provides a detailed list of the deployed microservices, showing a comprehensive overview of the enhanced placement achieved by this strategy.

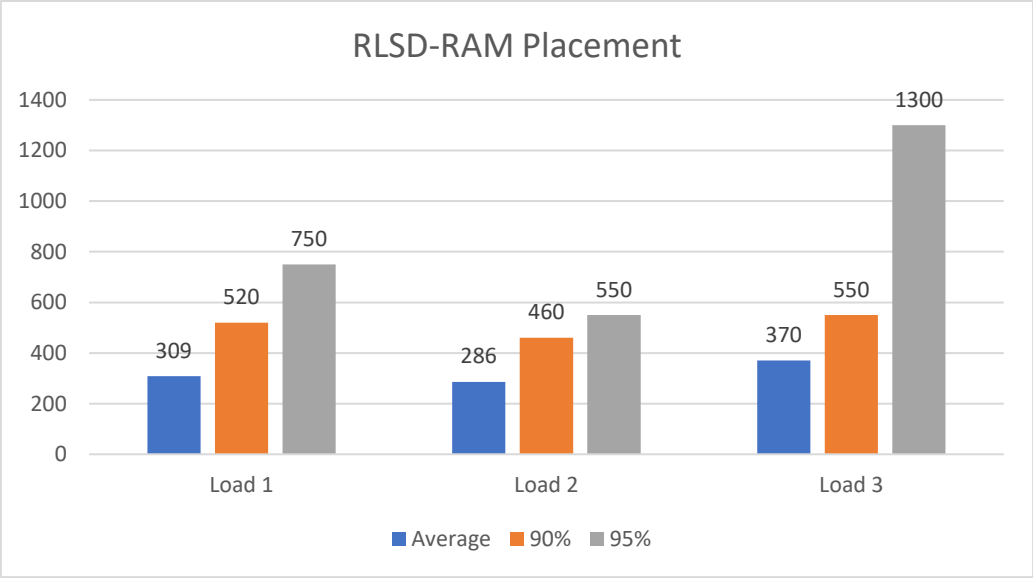


Chart 5.5: RLSD-RAM's Service Placement Response Latency

Cluster 2	Cluster 3	Cluster 4	Cluster 5
Load 1			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice currencyservice frontend productcatalogservice shippingservice	apache currencyservice frontend noderedservice noderedserviceproxy productcatalogservice queryingsensorsservice queryingsensorsserviceproxy shippingservice
Load 2			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice currencyservice frontend productcatalogservice shippingservice	apache currencyservice frontend noderedservice noderedserviceproxy productcatalogservice queryingsensorsservice queryingsensorsserviceproxy shippingservice
Load 3			
adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	adservice cartservice checkoutservice currencyservice emailservice noderedservice noderedserviceproxy orionservice orionserviceproxy paymentservice productcatalogservice queryingsensorsservice queryingsensorsserviceproxy recommendationservice shippingservice	apache cartservice currencyservice frontend productcatalogservice shippingservice	apache currencyservice frontend noderedservice noderedserviceproxy productcatalogservice queryingsensorsservice queryingsensorsserviceproxy shippingservice

Table 5.10: RLSD-RAM's Service Placement

In Load 1, the deployment of microservices like “productcatalogservice”, “shippingservice” and ”cartservice” significantly improved the average response time from 772 ms to 309 ms. In load 2, requests like “Set Currency”, “Browse Product”, benefited from “currencyservice” and “productcatalogservice” reducing the average response time to 286 ms. Similarly, Load 3, with requests like "Add to Cart," "View Cart," and "Checkout," experienced enhanced performance as “currencyservice”, “productcatalogservice”, and “shippingservice” were already deployed in Cluster 4 and Cluster 5, reducing the average response time to 370 ms.

5.4 Discussion

Comparing the performance of the Service placement strategies, LFU, RLSD, LFU-RAM, and RLSD-RAM, we can acquire valuable insights into how effectively they optimize response times, allowing us to assess their respective strengths and weaknesses and make informed decisions about their suitability for optimizing the system's overall performance.

The LFU strategy, which prioritizes request affinity, does not consistently exhibit the lowest latencies across all loads. While it performs well in some cases, the RLSD strategy generally achieves lower latencies by considering response latency for service placement decisions. This indicates that response latency-aware placement strategies can contribute to improved performance.

Taking resource contention into account, the LFU-RAM strategy combines the LFU algorithm with the eviction of the most RAM-intensive service. This approach aims to balance request affinity and resource utilization, resulting in average latencies comparable to LFU and RLSD. Similarly, the RLSD-RAM strategy combines the RLSD algorithm with resource contention consideration by evicting the most RAM-intensive service. The test results demonstrate that RLSD-RAM performs on par with LFU, RLSD, and LFU-RAM in terms of average latencies.

In addition to the service placement strategies, it is important to highlight the traffic splitting feature of DeFog, which plays a crucial role in optimizing system performance. By efficiently distributing incoming requests across multiple clusters, DeFog ensures load balancing and prevents any single Service from being overwhelmed. This traffic splitting capability improves resource utilization by evenly distributing the workload and avoiding bottlenecks. Moreover, it enhances response times and user experience by effectively managing incoming traffic. The traffic splitting feature of Defog adds an extra layer of flexibility and scalability to the system, allowing it to efficiently handle high volumes of requests while maintaining stability. Therefore, incorporating traffic splitting in the service placement decision-making process can further enhance the overall performance and responsiveness of the distributed system.

In conclusion, the RLSD strategy consistently demonstrates lower overall latencies compared to the LFU, LFU-RAM, and RLSD-RAM strategies, indicating its effectiveness in optimizing response times. The incorporation of resource contention considerations in the LFU-RAM and RLSD-RAM strategies enables them to perform competitively, effectively managing resource utilization while optimizing response times. These findings highlight the importance of considering request affinity, response latency, and resource awareness when making service placement decisions in distributed systems. By incorporating these factors into the placement strategy, optimal performance can be achieved. Furthermore, the inclusion of the traffic splitting feature in DeFog enhances system performance and responsiveness. By offloading a portion of requests to connected clusters, the system maintains better stability for each service and prevents resource usage from exceeding 80% of the service's limits. Based on the test results, using a threshold value of 0.8 for the traffic management feature of DeFog proves to be effective in optimizing resource usage and load balancing. The combination of the traffic management feature with the RLSD service placement algorithm yields the best results in terms of system performance and stability. Charts 5.7, 5.8, 5.9 demonstrate the average, 90^{th} and 95^{th} percentiles of response times for each placement strategy, showcasing the differences in performance. A

significant improvement can be observed in the 95th percentile response latencies when comparing LFU and RLSD strategies. In Load 1, the RLSD strategy demonstrates approximately a 33% reduction in the 95th percentile response latency compared to LFU. Similarly, in Load 3, the RLSD strategy exhibits a noteworthy 25% decrease in the 95th percentile response latency. These findings highlight the superior performance and effectiveness of the RLSD strategy.

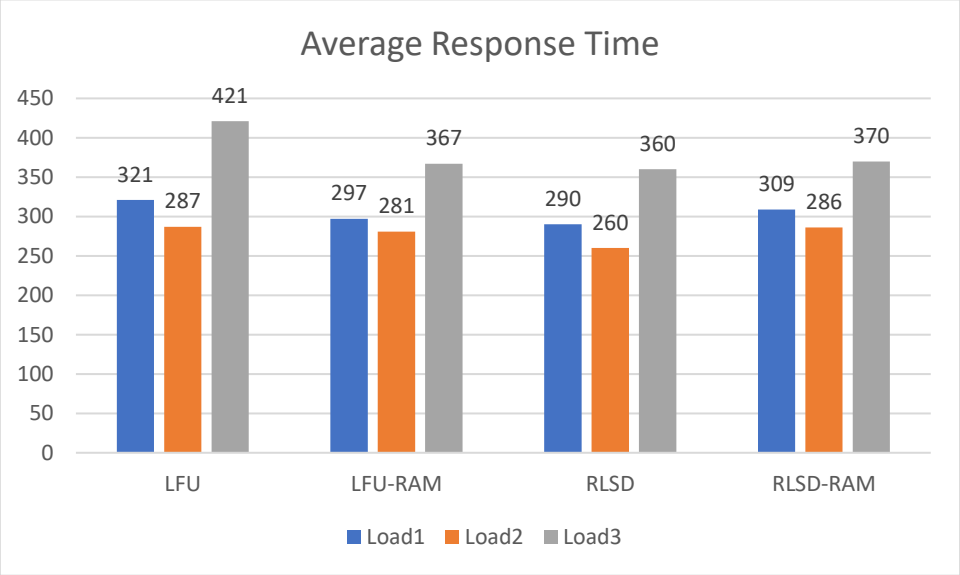


Chart 5.6: Average Response Times for each placement strategy

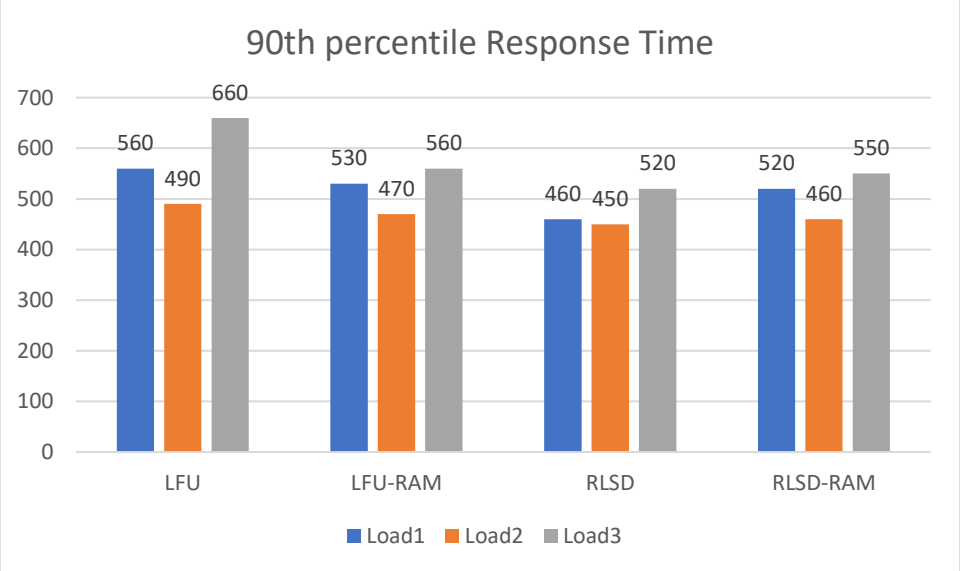


Chart 5.7: 90th percentile Response Times for each placement strategy

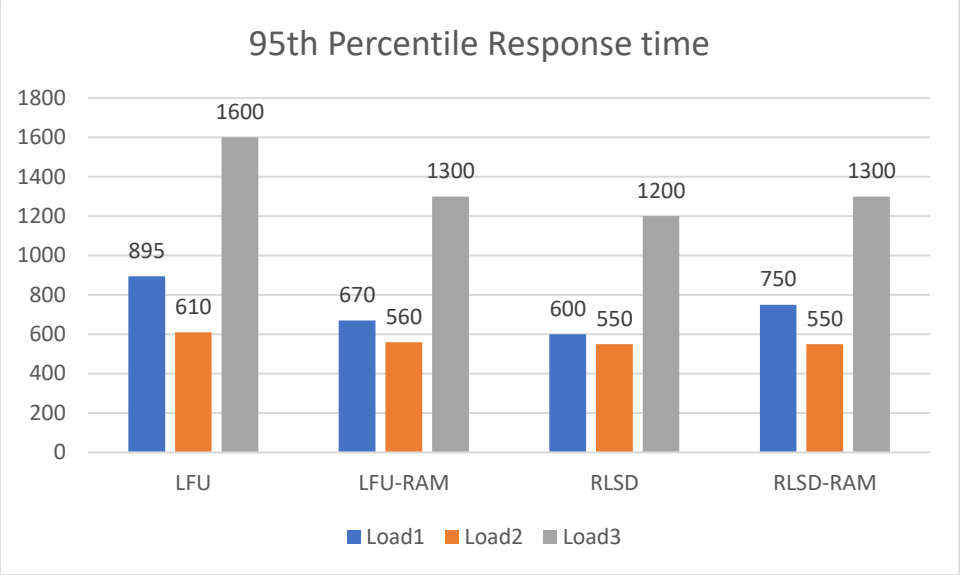


Chart 5.8: 95th percentile Response Times for each placement strategy

6 Conclusion and future work

In this final chapter, we will summarize the contents and the results of this Thesis and propose future work aimed at refining the service placement problem and investigating advanced optimization strategies on distributed systems.

The primary objective of this study was to reduce the total latency in a hybrid cloud-fog-edge multicluster environment by strategically placing services in the most appropriate clusters. To achieve this, we conducted a thorough analysis and evaluation of various service placement strategies, LFU, RLSD, LFU-RAM, and RLSD-RAM. In addition, we developed an application called DeFog, which implemented these algorithms and incorporated a traffic splitting feature to enhance system stability. To enable the service placement and traffic splitting features of DeFog, we utilized the capabilities of Linkerd, a service mesh framework.

To evaluate the performance of the proposed algorithms, we utilized two benchmark applications, namely Google's Online Boutique and iXen. Through extensive testing and analysis, we gained valuable insights into each strategy. Our findings revealed that the RLSD strategy mostly outperformed LFU in terms of achieving lower latencies, emphasizing the significance of considering response latency in service placement decisions. Furthermore, the resource-aware strategies, LFU-RAM and RLSD-RAM, showcased promising results by effectively managing resource utilization while optimizing response times.

Additionally, we explored the importance of traffic splitting in optimizing system performance. By distributing incoming requests across multiple clusters and ensuring load balancing, we observed improvements in resource utilization, enhanced response times, and ultimately, a better user experience.

Although this thesis has provided valuable insights and optimization strategies for service placement and traffic splitting, there are several areas that can be explored in future research.

First and foremost, we suggest investigating the potential of combining multiple placement strategies, such as a hybrid approach that leverages both request affinity and response latency-aware placement. This can provide a more comprehensive and flexible approach to optimizing performance.

Secondly, in this thesis, we have not considered Cost-Performance Trade-offs. Considering cost-performance trade-offs in service placement decisions and integrating cost factors, such as communication overhead between clusters, into the placement algorithms can help optimize resource utilization while minimizing operational costs.

Furthermore, considering the utilization of real-time data in this thesis, we suggest exploring the application of machine learning techniques to predict workload patterns and make proactive service placement decisions. Machine learning models can leverage historical data and real-time metrics to optimize placement decisions and traffic splitting algorithms.

Moreover, the threshold in the traffic splitting feature of DeFog is statically provided. If resource limits on microservices provided by their developers aren’t optimized, a static threshold may result in underutilization or stress of the microservices. To address this issue, investigating dynamic adaptation

mechanisms that adjust the threshold value can enable the system to continuously optimize performance in dynamic environments.

In the premises of this thesis, the DeFog application is deployed in each cluster separately and each instance of DeFog is only aware of the state of the cluster it is installed in. However, to further enhance the effectiveness of the placement strategies, we propose upgrading this decentralized approach by enabling communication between different instances of DeFog. By implementing a communication mechanism between DeFog instances, valuable insights and information about the overall system state can be shared. This would enable a more holistic view of the entire distributed system, allowing for coordinated decision-making and optimization of the placement strategies.

Lastly, in the context of the multicluster environment explored in this thesis, the focus has been on single-node clusters. We suggest a study on placement strategies for a multicluster environment where each cluster consists of multiple nodes. While focusing on minimizing latency for end users, efficient placement of microservices on each multi-node cluster becomes crucial to achieve optimal resource utilization, load balancing, and cost-effectiveness. By carefully assigning microservices to nodes within clusters, it is possible to reduce network traffic and maximize the utilization of available resources.

7 Bibliography

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” *MCC’12 - Proceedings of the 1st ACM Mobile Cloud Computing Workshop*, pp. 13–15, 2012, doi: 10.1145/2342509.2342513.

[2] H. K. Apat, B. Sahoo, and P. Maiti, “Service Placement in Fog Computing Environment,” in *2018 International Conference on Information Technology (ICIT)*, 2018, pp. 272–277. doi: 10.1109/ICIT.2018.00062.

[3] S. Shaik and S. Baskiyar, “A Scalable Approach to Service Placement in Fog/Cloud Environments,” in *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2021, pp. 1–8. doi: 10.1109/IPCCC51483.2021.9679396.

[4] A. Brogi, S. Forti, C. Guerrero, and I. Lera, “Meet Genetic Algorithms in Monte Carlo: Optimised Placement of Multi-Service Applications in the Fog,” in *2019 IEEE International Conference on Edge Computing (EDGE)*, 2019, pp. 13–17. doi: 10.1109/EDGE.2019.00016.

[5] R. Mahmud, K. Ramamohanarao, and R. Buyya, “Latency-Aware Application Module Management for Fog Computing Environments,” *ACM Transactions on Internet Technology*, vol. 19, no. 1, Nov. 2018, doi: 10.1145/3186592.

[6] S. Pallewatta, V. Kostakos, and R. Buyya, “Microservices-Based IoT Application Placement within Heterogeneous and Resource Constrained Fog Computing Environments,” in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, in UCC’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 71–81. doi: 10.1145/3344341.3368800.

[7] V. Farhadi *et al.*, “Service Placement and Request Scheduling for Data-intensive Applications in Edge Clouds,” in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1279–1287. doi: 10.1109/INFOCOM.2019.8737368.

[8] O. Ascigil, T. K. Phan, A. G. Tasiopoulos, V. Sourlas, I. Psaras, and G. Pavlou, “On Uncoordinated Service Placement in Edge-Clouds,” in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 41–48. doi: 10.1109/CloudCom.2017.46.

[9] A. Hedhli and H. Mezni, “A Survey of Service Placement in Cloud Environments,” *Journal of Grid Computing*, vol. 19, no. 3, p. 23, 2021, doi: 10.1007/s10723-021-09565-z.

[10] V. Cardellini, F. Lo Presti, M. Nardelli, and F. Rossi, “Self-adaptive Container Deployment in the Fog: A Survey,” in *Algorithmic Aspects of Cloud Computing*, I. Brandic, T. A. L. Genez, I. Pietri, and R. Sakellariou, Eds., Cham: Springer International Publishing, 2020, pp. 77–102.

[11] F. A. Salaht, F. Desprez, and A. Lebre, “An Overview of Service Placement Problem in Fog and Edge Computing,” *ACM Computing Surveys.*, vol. 53, no. 3, Jun. 2020, doi: 10.1145/3391196.

[12] “What are Microservices? | IBM.” https://www.ibm.com/topics/microservices (accessed Jun. 10, 2023).

[13] “Kubernetes.” https://kubernetes.io/ (accessed Jun. 10, 2023).

[14] “K3s - Lightweight Kubernetes | K3s.” https://docs.k3s.io/ (accessed Jun. 10, 2023).

[15] G. Hussein, “Introduction to K3s | SUSE Communities,” May 16, 2021. https://www.suse.com/c/rancher_blog/introduction-to-k3s/ (accessed Jun. 10, 2023).

[16] “What’s a service mesh?,” Jun. 29, 2018. https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh (accessed Jun. 10, 2023).

[17] F. Chanaka, “Building scalable microservices without a Service Mesh | Medium,” *Microservices Learning*, Oct. 08, 2021. https://medium.com/microservices-learning/building-scalable-microservices-without-a-service-mesh-86b916ba7e98 (accessed Jun. 10, 2023).

[18] “The world’s lightest, fastest service mesh. | Linkerd.” https://linkerd.io/ (accessed Jun. 10, 2023).

[19] The Istio Authors, “Istio,” 2023. https://istio.io/ (accessed Jun. 10, 2023).

[20] “Consul by HashiCorp.” https://www.consul.io/ (accessed Jun. 10, 2023).

[21] “Traefik Mesh, the Simplest Service Mesh | Traefik Labs.” https://traefik.io/traefik-mesh/ (accessed Jun. 10, 2023).

[22] “Dynamic Admission Control | Kubernetes.” https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#admission-webhooks (accessed Jun. 10, 2023).

[23] “Overview | Linkerd.” https://linkerd.io/2.11/overview/ (accessed Jun. 10, 2023).

[24] “SMI | A standard interface for service meshes on Kubernetes.” https://smi-spec.io/ (accessed Jun. 10, 2023).

[25] “Locust - A modern load testing framework.” https://locust.io/ (accessed Jun. 10, 2023).

[26] “Least frequently used - Wikipedia.” https://en.wikipedia.org/wiki/Least_frequently_used (accessed Jun. 10, 2023).

[27] “Horizontal Pod Autoscaling | Kubernetes.” https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (accessed Jun. 10, 2023).

[28] “GoogleCloudPlatform/microservices-demo: Sample cloud-first application with 10 microservices showcasing Kubernetes, Istio, and gRPC.” https://github.com/GoogleCloudPlatform/microservices-demo (accessed Jun. 10, 2023).

[29] X. Koundourakis and E. G. M. Petrakis, “iXen: context-driven service oriented architecture for the internet of things in the cloud,” *Procedia Computer Science*, vol. 170, pp. 145–152, Jan. 2020, doi: 10.1016/J.PROCS.2020.03.019.

[30] X. Koundourakis, “Design and implementation of service oriented architecture for deploying IoT applications in the cloud,” Diploma Thesis, Technical University of Crete, 2019. doi: https://doi.org/10.26233/heallink.tuc.81120.

[31] “1. Introduction — AuthzForce CE 11.0.0 documentation.” https://authzforce-ce-fiware.readthedocs.io/en/latest/Introduction.html (accessed Jun. 10, 2023).

[32] “Fiware-STH-Comet.” https://fiware-sth-comet.readthedocs.io/en/latest/ (accessed Jun. 10, 2023).

[33] “tc(8) - Linux manual page,” Dec. 16, 2001. https://man7.org/linux/man-pages/man8/tc.8.html (accessed Jun. 10, 2023).