

TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

Intelligent Systems Laboratory



DIPLOMA THESIS

An Autonomous, Dynamic and Decentralised Minimum Latency Service Placement Method for Dynamic Workloads in Hybrid Fog - Edge Infrastructures

by

Dimitrios Chamarousios

Chania October 2023

THESIS COMMITTEE

Supervisor Prof. Euripides Petrakis

Assoc. Prof. Vasileios Samoladas

Prof. Michail Lagoudakis

Abstract

Latest applications supported by state-of-the-art technologies, like Internet of Things and Smart Cities, are highly dependent on Fog and Edge cloud Infrastructures. In order for the user to obtain the benefits of these applications in a proper manner, we should make sure that the underlying structure is capable of dealing with the demands of these applications. This implies that the substructure should allow users to experience an increased performance without the large communication delay that occurs due to cloud architecture configuration. In order for these cloud infrastructures to successfully answer users's requests, they need CPU cycles and memory for computation, storage resources and network bandwidth. Furthermore, due to the lack of resources at the edge and fog compared to cloud infrastructures/big data centers the application's components need to be placed adopting this restriction and depending on the time-variable type of requests. Moreover, another constraint is the stability insurance and the way that the infrastructure will react to a possible machine failure during peak-hours requests. We address the components/service placement problem, taking into consideration the CPU restrictions and the low-latency demand, by proposing an autonomous, dynamic, decentralized algorithm for service placement and a dynamic cloud infrastructure architecture capable of handling user's requests offering a low-latency experience to the end users. Through extensive experiments using real cloud infrastructure and applications, we demonstrate that our proposal can decrease latency compared to a static placement solution, while ensuring infrastructure stability.

Index Terms— Edge Fog computing, dynamic service placement, resource allocation.

Διπλωματική Εργασία με θέμα:

Μια αυτόνομη, δυναμική και αποκεντρωμένη ελάχιστης απόκρισης μέθοδος τοποθέτησης υπηρεσιών για δυναμικά φορτία, σε υβριδικές υποδομές υπολογιστικής ακμής και ομίχλης.

Συγγραφέας: Δημήτριος Χαμαρούσιος

Περίληψη

Οι πιο σύγχρονες εφαρμογές που υποστηρίζονται από τεχνολογίες αιχμής, όπως το διαδίκτυο των πραγμάτων και οι έξυπνες πόλεις, εξαρτώνται σε μεγάλο βαθμό από τις υποδομές υπολογιστικού νέφους ομίχλης. Προκειμένου οι χρήστες να αποκομίσουν τα οφέλη αυτών των εφαρμογών, ώστε να απολαμβάνουν μια αυξημένη απόδοση χωρίς τη μεγάλη καθυστέρηση επικοινωνίας που εμφανίζεται λόγω της εκάστοτε διαμόρφωσης της αρχιτεκτονικής νέφους, θα πρέπει να βεβαιωθούμε ότι η υποδομή είναι ικανή να ανταπεξέλθει στις απαιτήσεις τους. Οι εν λόγω υποδομές νέφους, για να ανταποκρίνονται επιτυχώς στα αιτήματα των χρηστών, χρειάζονται κύκλους CPU και μνήμη για υπολογισμούς, πόρους αποθήκευσης και εύρος ζώνης δικτύου. Λόγω της έλλειψης αυτών των πόρων στην ακμή (edge) και την ομίχλη (fog) σε σύγκριση με τις υποδομές νέφους/μεγάλα κέντρα δεδομένων (cloud), πρέπει οι υπηρεσίες της εφαρμογής να τοποθετούνται λαμβάνοντας υπόψη τον προαναφερόμενο περιορισμό, καθώς και τον τύπο και αριθμό των αιτημάτων που μεταβάλλονται με την πάροδο του χρόνου. Επιπλέον, μια άλλη πρόκληση είναι η διασφάλιση της σταθερότητας και ο τρόπος με τον οποίο η υποδομή θα αντιδράσει σε μια πιθανή βλάβη κατά τη διάρκεια λήψης αιτημάτων σε ώρες αιχμής. Αντιμετωπίζουμε το πρόβλημα της τοποθέτησης υπηρεσιών, λαμβάνοντας υπόψη τους περιορισμούς της CPU και τη απαίτηση χαμηλής απόκρισης, προτείνοντας έναν αυτόνομο, δυναμικό, αποκεντρωμένο αλγόριθμο για την τοποθέτηση υπηρεσιών και μια δυναμική αρχιτεκτονική υποδομής νέφους ικανή να διαχειρίζεται τα αιτήματα των χρηστών προσφέροντας μια εμπειρία χαμηλής καθυστέρησης στους τελικούς χρήστες. Μέσω εκτεταμένων πειραμάτων με τη χρήση πραγματικών υποδομών και εφαρμογών νέφους, αποδεικνύουμε ότι η πρότασή μας μπορεί να μειώσει την καθυστέρηση σε σύγκριση με μια στατική λύση τοποθέτησης, ενώ παράλληλα διασφαλίζει τη σταθερότητα της υποδομής.

Acknowledgments

I am grateful to my family, friends, and professor for their constant support during my thesis and my overall studies. To my family, thank you for your love, encouragement and your patience. To my friends, thank you for understanding and inspiration. To my professor, Dr Euripides Petrakis, thank you for your guidance. Your support has been invaluable.

Contents

TECHNICAL UNIVERSITY OF CRETE	1
Abstract	2
Contents	5
Introduction	7
1.1 Internet Of Things (IoT) and low latency demand applications.	7
1.2 Problem Definition	7
1.3 Proposed Solution	7
Background	8
2.1 Related Work	8
2.2. Layers of Computing	10
2.2.1 Cloud Computing	10
2.2.2 Fog Computing	10
2.2.3 Edge Computing	10
2.3 Related Technologies	11
2.3.1 Virtualization	11
2.3.2 Docker and Containers	12
2.3.2.1 Docker Images	12
2.3.2.2 Docker Registries	12
2.3.2.3 Container Orchestration	12
2.3.3 Kubernetes	13
2.3.3.1 Kubernetes Core Components	13
2.3.3.2 Kubernetes Functional Components	14
i. Pods	14
ii. Services	14
iii. Deployments	15
iv. Service Accounts, RBAC Role & RoleBinding	15
2.3.4 Linkerd	15
2.3.4.1 Linkerd Service Mesh	15
2.3.4.2 Linkerd Traffic Splits	17
2.3.4.3 Linkerd Multicluster Communication	18
Scheduler's Algorithm	20
3.1 The Core Logic	20
3.1.1 Example - Equation 1	21
3.1.2 Multiple Remote Clusters Communication	21
3.1.3 Example - Equation 2	21
3.1.4 Final Equation	22

3.2 The Algorithm	22
3.2.2 Function 1 (monitoring)	24
3.2.3 Example - Algorithm	26
3.2.4 Algorithms Details	29
3.2.5 Scheduler - Prometheus Communication	31
Implementation	34
4.1 Architecture	34
4.2 Choice of tools	36
4.2.1 Chaos Mesh	36
4.2.2 DNS Server	38
4.2.3 The tooling-cluster	38
4.2.4 Ambassador Ingress controller	39
4.2.5 Users-Workload	39
4.3 Controller Cluster & Experiment Cost	40
4.4 Benchmark Applications	40
4.4.1 Google's Online Boutique demo application	40
4.4.2 IXEN	41
Experiments & Results	43
Performance Evaluation	46
5.1 Experiments 1-2-3	46
5.1.1 Experiment 1	47
5.1.2 Experiment 2	54
5.1.3 Experiment 3	60
5.1.4 Experiments 1-2-3 Overview	67
5.2 Experiment 4	68
5.3 Experiment 5	75
Conclusions & Future Work	84
6.1 Conclusions	84
6.2 Future Work	84
References	85

Chapter 1

Introduction

1.1 Internet Of Things (IoT) and low latency demand applications.

In our all-time growing and fast moving society more and more devices ("things") that are embedded with sensors require internet connection, so they can exchange data with each other or with third sources. These devices can be household objects or even sophisticated industrial tools. [1]

Moreover, a network of these kinds of devices can create a "Smart City". Citizens and businesses can obtain benefits, while using applications, which are based on or heavily use these structures. Some common integration of this approach are fire hazard detection systems; autonomous electric vehicles that are connected to edge/fog infrastructures in order to self navigate and avoid crashes; cloud based gaming and virtually reality applications.

1.2 Problem Definition

In order for the proper functionality of these kinds of applications, low-latency response is strongly required. Apart from the low-latency requirement, high availability is crucial. Requests should be answered continuously with minimal disruption. To illustrate, autonomous vehicles can generate hazards, if they don't receive the necessary data in time or in time. Cloud based virtual reality applications can not operate with high latency this would destroy users experience. Therefore it is useful to create an infrastructure architecture able to cope with these demands. This architecture should be fast and guarantee high availability.

1.3 Proposed Solution

In this work, we propose the creation of a decentralized scheduler, which will be deployed individually in every cluster of a distributed cloud architecture. This scheduler, based on the number of incoming requests, their type and the available CPU resources of each host cluster, will dynamically deploy/delete applications services, forward requests towards the neighbor clusters with the goal of maximum exploitation of the edge and fog clusters layer CPU power, while ensuring faster response time to the final users and providing a dynamic way to deal with clusters outages/failure with minimum impacts to the end users. Additionally, we describe a cloud architecture implementation that can support the operation of this scheduler.

Chapter 2

Background

2.1 Related Work

The challenge of service placement within distributed cloud systems has gained significant attention within the academic field. This problem has to do with the best way of organizing and placing these services within a distributed cloud system to ensure that they operate normally and that they have acceptable response time. Beyond the emphasis on high performance, certain studies have introduced some extra variables like environmental, cost and energy consumption factors into the equation. This challenge can be subdivided into two categories: dynamic (involving real-time service relocation) and static placement. Another way of organizing these approaches could be the location that the corresponding algorithm is operating; distributed(decentralized) or centralized.

In [2] the authors propose a static dynamic service placement and load distribution strategy that uses limited look-ahead prediction to handle the workload fluctuations while considering performance cost trade-off service migrations. They take into consideration the difference of the nodes and requests (e.g., resource capacity, response deadlines) and using LLC (Limited Lookahead Control) policies they estimate how the Edge Computing system will change based on the service placement and the user-requests. Then they are trying to optimize the placement of services and the distribution of the workload by using a look-ahead prediction window. They evaluate the solution by conducting simulations and contrasting its performance with alternative methods, including a centralized approach.

In the study [3], authors introduce a hierarchical and autonomous fog architecture (HAFA) to place the services in a fog environment using a static algorithm. The specific algorithm does not require the knowledge of the whole cloud infrastructure system, but only a part of it. Their criteria are to satisfy the resources requirements of the app, ensuring minimum response time and taking into consideration the cost efficiency as well. They are splitting the placement process in two phases; during the first one they are organizing the nodes into layers and then they are organizing smaller groups of nodes inside these layers. Every group is monitored by a local authority which can communicate with neighboring authorities as well. In the second phase they start to search the most suitable node to place a service, upon a user request. Then using two two search operations in the topology, utilizing the local authorities, they are selecting the proper node to place the service based on the available resources and costs. To validate their study they are performing simulations using the CloudSim tool.

In their study [4], researchers focus on edge computing service placement. Thus, due to the fact that edge nodes most of the time are running with low energy, researchers' main consideration is the low energy consumption service placement. They used a centralized online service placement algorithm which is based on Lyapunov optimization to place the services dynamically, while ensuring that the average battery energy of the mobile devices (edge nodes) will remain stable under a certain value. In order to test their algorithm they are using a simulator on a real dataset. They are comparing this centralized algorithm with multiple other approaches like a distributed algorithm. They aim to compare this centralized energy-efficiency service placement solution with other approaches.

In this [5], authors present a central dynamic service placement strategy for fog nodes with the goal of better performance; lower latency and better energy consumption. In their algorithm they are taking into consideration the size of the user's requests as well and they know beforehand the size of the workload of every request. Thus their approach requires complete system state knowledge. They are testing their algorithm using a simulator, comparing it with a random service placement method and another already existing method which uses the ideas of the shortest path. They aim to prove that their proposal can achieve better performance.

In [6] research paper, the authors introduce multiple independent strategies for placing and replacing services within edge and cloud infrastructures. They propose a decentralized approach where each node can autonomously decide on service placement and replacement. These decisions are influenced by factors like request deadlines and usage patterns. Four ranking policies are presented: SDF (Strictest Deadline First) prioritizes services with strict deadlines, LFU (Least Frequently Used) is based on the total number of requests, Hybrid combines elements of both SDF and LFU policies, and LRU (Least Recently Used) replaces services based on their historical usage. These policies are aiming to optimize resource utilization and ensure timely request handling. To validate these policies, the researchers conducted simulations.

This thesis is particularly inspired by the LFU policy, incorporating elements of it into their proposed algorithm. Additionally, we considered the CPU availability of each node and introduced the functionality for forwarding requests to multiple upper neighbor nodes based on response times.

Importantly, this thesis goes beyond simulations by evaluating the proposed algorithm using real service-oriented cloud applications in actual cloud infrastructures, mimicking real-life scenarios, including situations like physical disasters. This approach is one of the main factors, which distinguishes it from previous research that primarily relied on simulation-based evaluations.

2.2. Layers of Computing

Cloud, fog and edge computing architectures are the three main layers of a distributed computing approach. Exploiting the varied range of resources of every layer, they are working together to execute calculations, process data and host applications.

2.2.1 Cloud Computing

Cloud Computing is a computer infrastructure hosted at a remote data center managed by a cloud services provider.[7]

It enables the delivery of computing services over the internet. Typically organizations that use cloud computing pay only for the cloud services they use, limiting the operating costs. Another benefit is the elasticity that cloud computing comes with; organizations can scale their resources up or down instantly according to their needs. Moreover it allows the organizations to expand to new geographical locations fast.

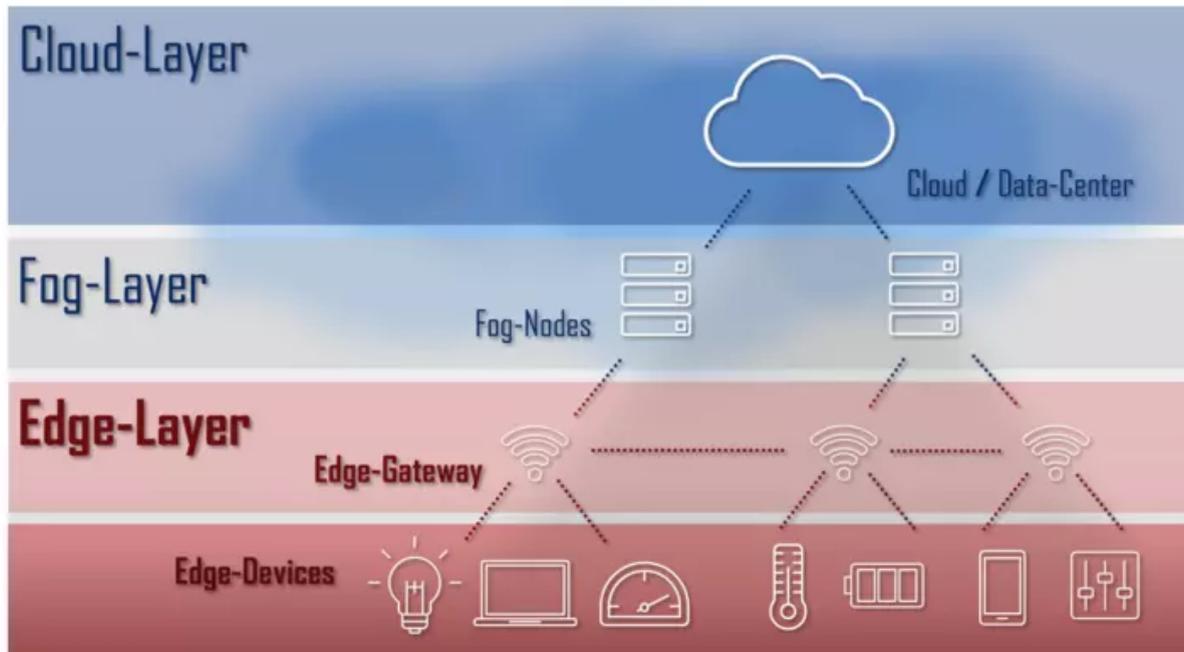
2.2.2 Fog Computing

Fog Computing by definition is a decentralized computing infrastructure located between Cloud and Edge Computing.[8]

The goal of it is to provide analytic services at the edge computing. Although it's not as physically close to the end user as Edge computing does, they are more powerful compared to them and bring the advantage and the cloud computing power of the cloud closer to where data is created. Although edge infrastructure is closer to the data creation location, it doesn't always have the compute power and resources to perform advanced calculations, required by the host applications. It combines both the advantages and the disadvantages of cloud and edge computing.

2.2.3 Edge Computing

Edge computing is a distributed computing infrastructure located close to the data creation source. Its goal is to process data closer to where they are being generated [9], providing improved response times, due to its location. Some examples of these devices are routers, phones, vehicles, electronic compute units (ECUs).



[10] Schematic representation of a cloud architecture with cloud, fog and edge layers.

2.3 Related Technologies

This specific form of distributed computing architecture leverages an array of diverse technologies to achieve cost-effective operations, secure functioning, and resilient performance. By combining a variety of cutting-edge tools and methodologies, it effectively maintains affordability and delivers rapid and reliable functionality.

2.3.1 Virtualization

A lot of organizations have the need to deploy multiple servers, each of them running a different operating system with different capacities. This would drive to a very high cost and inefficiency.

Using Virtualization technologies, they are capable of deploying several operating systems that run different software in the same physical hardware.

The particular process is using software to create an abstract layer on the physical host that is able to bisect the CPU, memory and all the resources into fragments. [11] Its goal is to deliver scalability; it's transforming traditional computing to make it more scalable.

Most of the time this software is called hypervisor. The hypervisor isolates each operating system from the underlying physical hardware. Although all the operating systems are sharing the same hardware, they are totally separated from one another. The operating systems are running in Virtual machines (VMs), created and run by the hypervisor. Every VM may have different amounts of resources allocated. Due to this separation, if a VM crashes it doesn't affect the rest of the VMs. The hypervisor can stop and start these VMs. Because of all these characteristics it constitutes the core of cloud computing.

2.3.2 Docker and Containers

Docker is a software platform that can achieve virtualization at the operating system level. It is the tool that creates containers. Containers are packaging applications and all of their libraries and dependencies and they can run in every operating system that supports Docker in a restricted environment. This means that these containers run in the same host as the Docker daemon and they share the same kernel, but they don't know that other containers can run in the same host. This enables users to run their applications without having to worry about operating system and environment compatibility. Another great benefit is the fact that if one container crashes, it doesn't affect the rest of the containers or the host operating system, because they are restricted. Moreover they are lightweight and they consume less resources compared to VMs. Containers are built based on docker images.

2.3.2.1 Docker Images

Docker images are made by read-only instructions-template layers that describe the container that should be deployed. Due to the layering, a lot of commonly used layers can be reused to build different images. A docker image can be based on another image with some extra customization. Docker images are generated from Dockerfiles. They describe which operating system the container should simulate, all of the dependencies, files and resources. Furthermore it allows the creator to define CPU and memory limits to the container. If the container exceeds the CPU limits, it would be throttled and if it exceeds memory limits it becomes a candidate for termination. Furthermore they can be configured to share a part or all the file-system with the host operating system or other containers.

2.3.2.2 Docker Registries

Docker registry is a stateless system that allows the storing, distributing and versioning of docker images. Someone can pull an image from a docker registry and use it as a base to build his own image. Docker registry can be public, accessible by everyone, or private, restricted to authenticated users. A common public docker registry is the Docker Hub. [12]

2.3.2.3 Container Orchestration

Organizations that need to deploy and manage a big number of containers need an environment to control the overall process. Using Container orchestration tools, they are able to automate complex processes, monitor and manage all of their containers.[13]Furthermore, some other benefits coming with the usage of these tools are:

- Configuration: Containers can be started automatically using scripts. This can help to avoid manual actions and can solve synchronization issues, ex. Huge number of containers that should start at the same time.
- Health-monitoring: This includes the failover and the recovery process. If a container crashes, the tools/framework will try to redeploy the container.

- Scalability: Organizations can scale up or down the number of containers and their resources based on the traffic they are receiving. Moreover they can automate this procedure using code.
- Effective Resource Management: For example if a container starts to consume a lot of CPU leaving the other containers starving, this container will be throttled or it will be moved to another node that can satisfy the containers resources.
- Networking: Trying to connect a huge number of containers with each other or with the outer world and adding restrictions at the same time is considered to be a difficult/time-consuming problem. Container Orchestration tools offer out of the box solutions that help with these network problems.

2.3.3 Kubernetes

Kubernetes is an open-source container orchestration platform for managing containerized workloads and services. [14]

It was developed by the company "Google" and it became an open source project in 2014.

2.3.3.1 Kubernetes Core Components

A kubernetes Cluster [15] consists of various components. A cluster in Kubernetes is made up of a number of nodes, or worker machines, which are in charge of running containerized applications. A worker node is present in each cluster at a minimum. Worker nodes are the nodes that host the containers (pods). A kubernetes cluster is composed of:

- the Control Plane: is the central decision made component. It takes decisions for the pod scheduling and reacts to any cluster event, like the initialization of a new deployment. It can run on any of the cluster's machines. This component is consists out of other components, like:
 - kube-apiserver: it provides the REST frontend mechanism so the admin or other clusters can interact with it. It is scalable, which means that several instances of the kube-apiserver can run at the same time, so the traffic can be balanced between these instances.
 - etcd: It's a distributed key-value store. Cluster's nodes are constantly using it to store critical information so they can run in harmony.
 - kube-scheduler: It is constantly monitoring for pods that are created, but they are not assigned to any node and assign them to an adequate node. During the assignment it takes into consideration the resources that the new pod is asking for and the available resources that every node has.
 - cloud-controller-manager: it allows cloud providers to evolve their code separately from the core Kubernetes code. It allows the link of the cluster into the cloud provider's API. If someone is running Kubenertes on his own premises, the cluster doesn't need to have a cloud-controller-manager.
 - kube-controller-manager: it's the component that actually runs the controller processes. It contains multiple controllers like node-controller, job-controller, service-account-controller. Each of these controllers is in charge of monitoring

the shared state of the cluster using the api-server and making the necessary changes to move the cluster's state from the current to the desired.

- the Nodes: Every Cluster has at least one node. Nodes can be either physical machines or virtual machines. They are the components which run, create, manage and control pods. Every node is consist of:
 - kube-proxy: It is in charge of managing the network rules of the host node. These rules can allow external network communication with the pods and the communication between separate pods. It is based on the operating system's packet filtering.
 - kubelet: Every node has a kubelet. Its goal is to manage pods and their containers. It has to make sure the containers are running in a pod. Furthermore It is also responsible for the communication between the Control Plane and the host node.
 - container runtime: The software that is responsible for running the containers.

2.3.3.2 Kubernetes Functional Components

i. Pods

Pods are the smallest deployable compute units and they are grouping containers. Every pod consists of at least one container. Pods provide the containers they host with storage, an unique IP address and they contain the specifications for how to run the containers. Due to this, containers running in the same pod can share the data storage or communicate with each other through a local network (localhost). Apart from the communication of the containers, pods can be used by Kubernetes to scale up or down an application; during peak hours of an application, kubernetes can automatically replicate the application's pods to keep up with the increased demand. Furthermore pods offer a form of failure safety to the system.

Finally another functionality of the pods are the CPU and RAM limitations.

In Kubernetes, in order to measure CPU resources is measured in CPU units. 1 CPU unit can be one physical or one virtual core depending on the infrastructure that the node provides (physical or virtual machine). This unit is always referred to as an absolute unit and never as a relative. Furthermore in Kubernetes 1 CPU unit is equivalent to 1000 millicores. Memory (RAM) is measured in bytes. For both of the CPU and memory, requests and limits can be defined in the pod description for every container individually. Requests are used by the kube-scheduler in order to place a pod in a node with sufficient resources. Pods may use more resources than their request specifies, but they are never allowed to use more than the resource limits. For example, if a container's memory exceeds its memory limits, the system kernel will terminate the process.

ii. Services

Since pods are ephemeral, in case of a crash and re-deployment they might be assigned with a different IP address than they used to have. This would create issues, because pods

are providing specific functions (web services, data-bases etc.) and the change of their IP address would make the rest of the pods unable to communicate with it. Services are solving this problem by being a logical abstraction for a deployed group of pods. This means that most of the time pods don't communicate directly, but they are communicating first with the corresponding service of the target pod, which will forward their request to the target type of pod. A service may be in charge of multiple pods of the same type, so it can load-balance the requests between these pods.

iii. Deployments

They are one of the basic building blocks of Kubernetes. Deployments are useful to define the desired state of the pods someone wants to deploy. After this deployment-controller will make sure that the described pods in the deployment are in the desired state and if they are not, it will contact the necessary changes to reach it.

iv. Service Accounts, RBAC Role & RoleBinding

Pods, system components and other entities outside of the cluster can use specific service accounts, providing the necessary credentials to authenticate to the Kubernetes API, so they can manipulate Kubernetes API objects. EX. A pod can read, delete or edit another deployment in the cluster using the right Service Account.

RBAC (Role-Based Access Control) Role is security control that manages who has access to each API resource. They contain a set of permissions describing what can be done in the cluster.

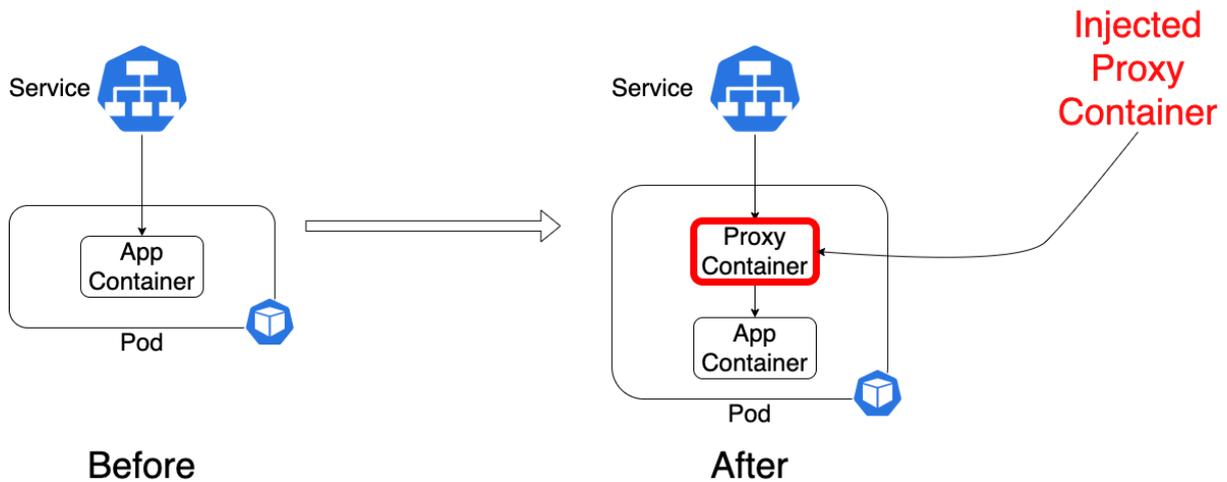
RoleBinding is binding a role to a user. It holds a set of subjects(Service Accounts) and a reference to the role to be granted. EX. Someone can "bind" the role of the list pods to a specific service account. This means that users belonging to the specific service account can list the pods running on the cluster.

2.3.4 Linkerd

Linkerd, created by "Buoyant", is an open-source network proxy that serves as a service mesh installation. Linkerd was one of the first technologies that inserted the term service mesh to cloud computing. It provides support for popular platforms including Docker and Kubernetes.[16]

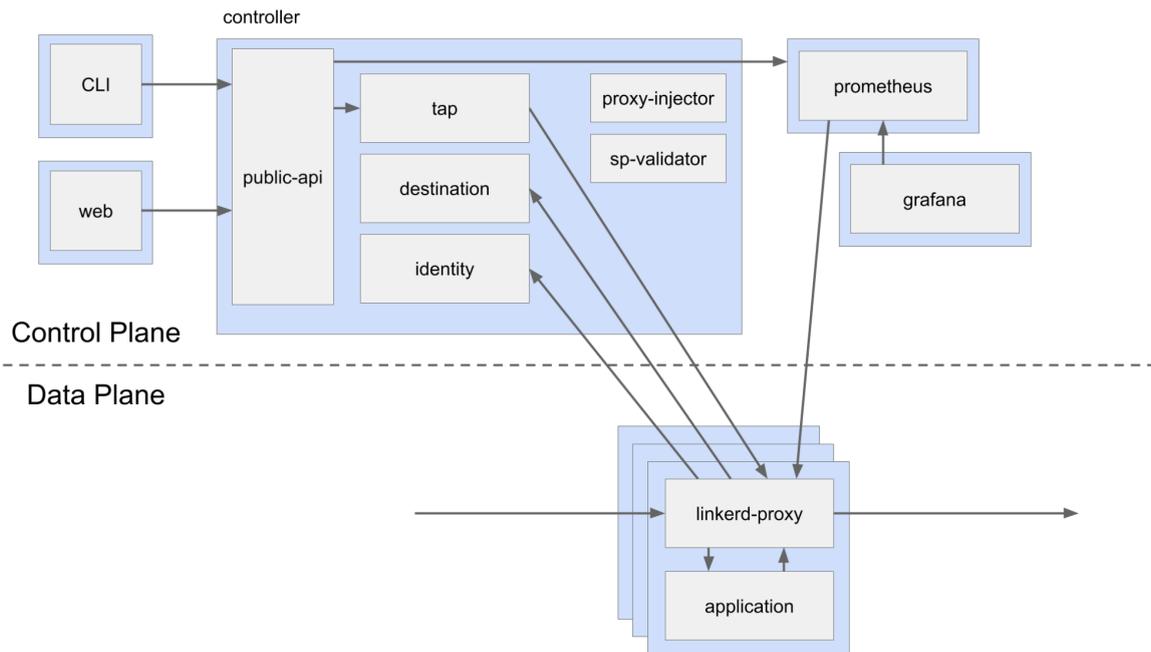
2.3.4.1 Linkerd Service Mesh

Service Mesh is a tool that enables additional observability, security and reliability features to applications by managing the way that communications happens between the services. It works by injecting a set of network proxies(sidecars) alongside the application code (containers). These network proxies are handling the communication between services.



This layer that was described before it's called "Data Plane". It consists of these injected sidecars, which are sending telemetry and receiving control signals from the Linkerd control plane.

Linkerd control plane is formed by a set of services, which are performing an array of things, like: proving an API so it can be configured, sending control signals to data plane's side-cars, consuming side-car's telemetry.



[17] Linkerd Control Plane

Some of these fundamental services are:

- Destination: This is the component that side-cars are using to lookup where to send requests
- Proxy Injector: Every time a pod is created, this module is in charge of injecting the side-car to the pod.
- Controller: It contains the public API so infrastructure administrators can interact and configure Linkerd.

- Identity: This part is in charge of creating and distributing the identity credentials to the side-cars. Their usage has the goal of the authentication and encryption between services.
- Tap: It watches requests and responses time of the injected pods in real time, providing them to the CLI and the dashboard when it is asked for.
- Prometheus: it's an open source system that is capable of monitoring and storing Cluster's data in time-series form. It comes with its own flexible querying language: PromQL. It also provides alerting functionalities and visualization –it offers multiple types of graphs—. It's working by scraping and saving the statistics every ten (10) seconds. These statistics then are exposed through an endpoint (/metrics) and they are available to all the other Linkerd components, like the CLI and the dashboard.
- Grafana: Grafana is an open-source data visualization and monitoring tool. It is using Prometheus' end-point in order to retrieve the necessary statistics to create dashboards. In the Linkerd project it is already configured with various dashboards that provide a high level metrics overview about Linkerd components and the Cluster's Workload.

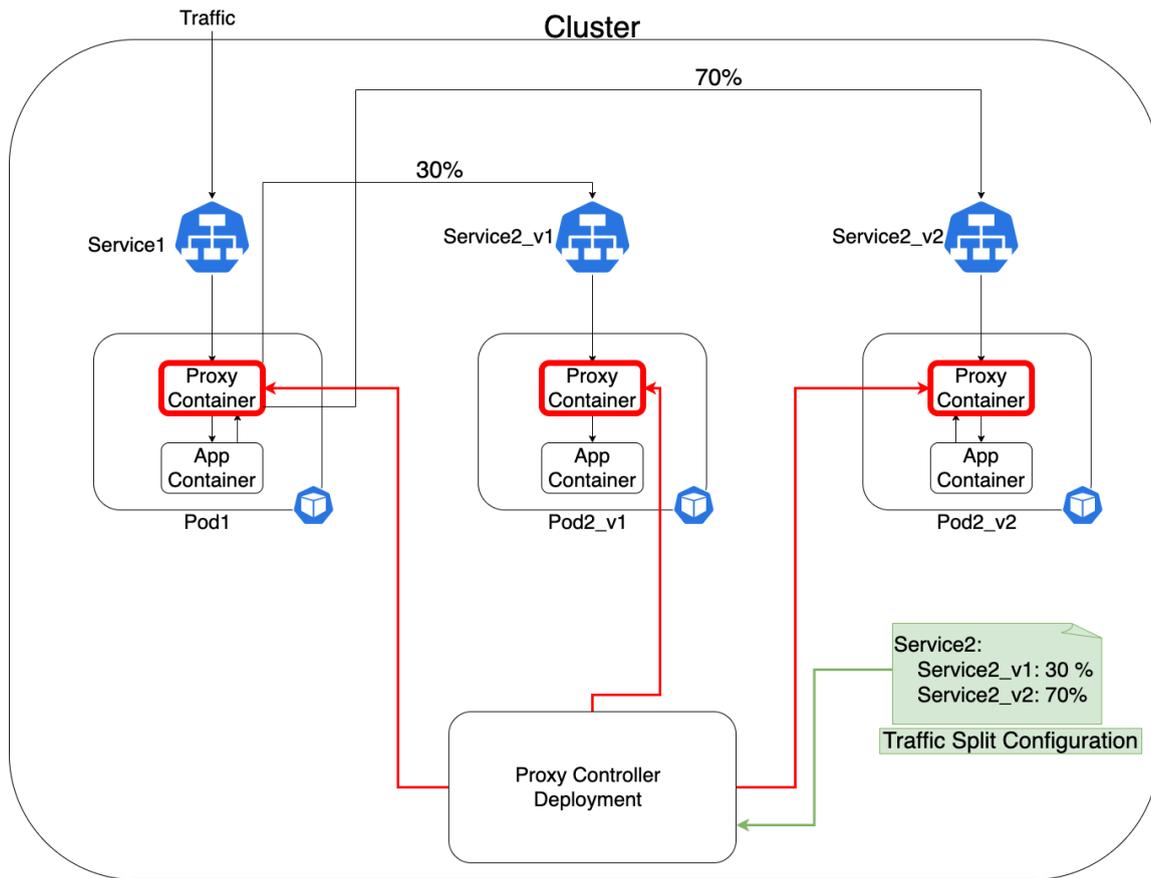
For the interactions with the administrators of the infrastructure, Linkerd is equipped with:

- CLI: it runs locally and interacts with the Linkerd API, which is located in the control plane. It is useful for debugging, installing and upgrading the control lane components.
- Dashboard: Contains an abstract view of the status of injected pods as well Linkerd components in real time.

Linkerd is an open-source implementation of this technology with contributions from various individuals and organizations from around the world.

2.3.4.2 Linkerd Traffic Splits

Among all the functionalities, Linkerd is offering the traffic-split option. By using this feature someone can split and forward the incoming requests to different services deployed on the same or remote Clusters. It is very useful for sophisticated canary and gradual deployments and a variety of traffic management scenarios. It is managed and configured through the Linkerd control plane. Using the control plane someone can describe the traffic split behavior, like the percentages between the targeted splitted services. Then the control plane configures the side-cars (data-plane) providing them with the necessary routing and balancing instructions to split the traffic based on the configuration. So when a request is made to a service, its side car will intercept this request at the network level. It will examine the target service of the request and based on the instructions it has already received by the control plane and it will decide how to split the request. Then based on this decision it will forward the request accordingly. Furthermore another feature is that someone is allowed to configure the traffic-splits on the fly. This means that in any given time Linkerd administrators can adjust/modify the traffic splits in real-time, even during a heavy workload time.

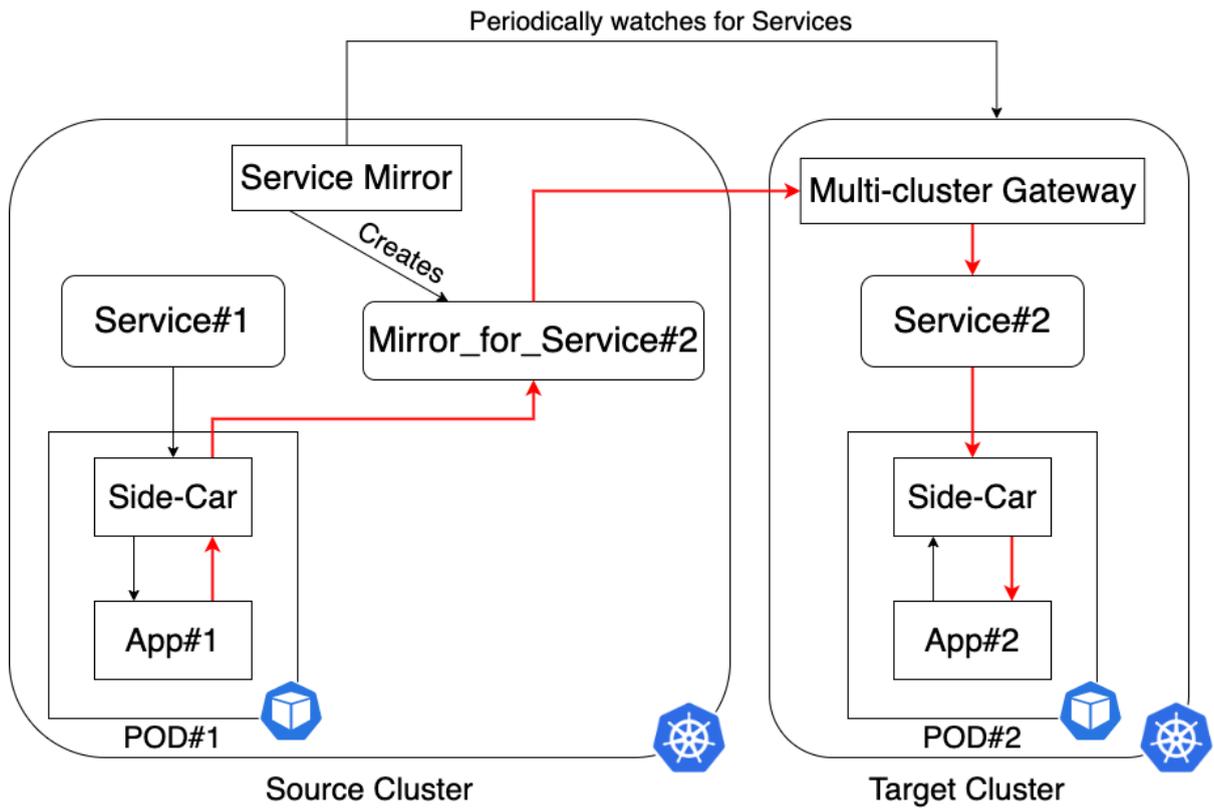


Example of a Traffic Split between Service2_v1 and Service2_v2.

2.3.4.3 Linkerd Multicluster Communication

Linkerd provides the feature of communication between remote clusters while maintaining the observability, reliability and security characteristics just like with in-cluster communications. Furthermore the overall process is transparent to the application code and the communication between clusters is secured and verified on both ends with mutual Layer Security mTLS. Linkerd is achieving this using two extra components:

- Service mirror: Service mirror is running on the host cluster and it is mirroring the Kubernetes services (not the pods) of the remote cluster. Due to this the host cluster has visibility on the services names of the target cluster and it can forward requests to it.
- Multi-cluster gateway: This component is running on the target remote cluster and it is accepting requests coming from the host/source cluster. Once it receives them it is forwarding them to the proper service.



Chapter 3

Scheduler's Algorithm

Once we have a solid understanding of the setup of a Kubernetes Cloud environment with Linkerd project enabled, we can now dive into the main focus of this project: the dynamic service placement algorithm. This chapter provides a detailed analysis of the algorithm itself and how it is packaged and deployed to the relevant clusters.

3.1 The Core Logic

As it was described before, the algorithm was designed with the goal of maximum CPU utilization of the host Cluster. Having this in mind, the algorithm is gathering frequently the total number of requests and the CPU utilization of the host CLuster. Based on these, it decides which services/deployments should remain on the cluster, which should be deleted and which one should be initialized. In the case of the new initialization of a service, the scheduler doesn't know the CPU resources it will consume. To solve this problem, the scheduler is calculating approximately the resources of this new service based on statistics it collected when the same service was running in the Cluster in the past. If the service never ran on this Cluster in the past, then the scheduler is calculating the resources based on inputted data that the infrastructure administrator provided during the initialization phase.

Another challenge with this approach is that if the workload of the Cluster is huge compared to cluster's resources then, maybe all the services (pods) consume individually more CPU resources, compared to what the Cluster can provide. Based on what it was described before, this means that this cluster should be left empty –without any services–. To address this problem the scheduler is utilizing Linkerd's traffic-split feature. If a pod's CPU utilization is so high that it can not fit in the Cluster, the scheduler will split the requests coming to this service based on its total number of requests, CPU usage estimation and Cluster's available CPU resources. The formula that the scheduler is applying is this:

$$\text{splitPercentageForNeighborCluster}_D = \frac{NMPDCPU - CRCPU}{NMPDCPU} * 100\%$$

$$\begin{aligned} \text{splitPercentageForHostCluster}_D &= \left(1 - \frac{NMPDCPU - CRCPU}{NMPDCPU}\right) * 100\% = \\ &= \frac{CRCPU}{NMPDCPU} * 100\% \end{aligned}$$

where:

- D = *The Deployment*
- $NMPDCPU$ = *Next Most Popular Deployment CPU Estimation*
- $CRCPU$ = *Cluster's Remaining CPU*
- $NMPDCPU > CRCPU$

Equation 1

(In this thesis, the terms 'services' and 'deployments' are synonymous.)

3.1.1 Example - Equation 1

To illustrate, if the next most popular service name is X and the CPU estimation is 200millicores and the Cluster's remaining CPU is 150 millicores, then the scheduler will deploy this service, but it will split the incoming requests for this service. A part will be answered by the new service and the rest of the requests will be forwarded to the neighbor Cluster.

$$splitPercentageForNeighborCluster_x = \frac{200-150}{200} * 100\% = 25\%$$

$$splitPercentageForHostCluster_x = \frac{150}{200} * 100\% = 75\%$$

So, 25% of incoming requests will be forwarded to the neighbor Cluster, and 75% will be answered by the local service.

Using this method, scheduler will utilize the most of the host Cluster, before starting forwarding requests to the neighbor Cluster.

3.1.2 Multiple Remote Clusters Communication

Another feature of the scheduler is that it supports the connection with multiple remote clusters. The main problem with this feature is the selection of the cluster to forward the requests. To tackle this issue, scheduler is regularly collecting and calculating average response time for every pod's requests running in a neighbor/remote Cluster using Prometheus. Using this data it computes a specific percentage for every individual deployment/service. This percentage is calculated based on this formula:

$$PercentageToCluster_{DC} = \frac{1}{l-1} \left(\frac{\sum_{i=1, i \neq C}^l (avgResponseTime_{Di})}{\sum_{i=1}^l (avgResponseTime_{Di})} * 100 \right) \%$$

where:

D = ID of The Deployment (Deployment1 = 1, Deployment2 = 2, ...)

C = ID of The Remote Cluster (Cluster1 = 1, Cluster2 = 2, ...)

l = Number of Topology's Remotes Clusters

Equation 2

3.1.3 Example - Equation 2

To illustrate, let's suppose that we have a cluster C1 which is able to forward requests to 3 (three) neighbor/remote Clusters –naming C2, C3 and C4 and we have a deployment X, for which we want to calculate the percentages to forward the requests.

Let Prometheus Data be:

Average Response Time for Deployment X from Cluster C2 = 30ms

Average Response Time for Deployment X from Cluster C3 = 40ms

Average Response Time for Deployment X from Cluster C4 = 60ms

Then:

$$PercentageToCluster_{XC2} = \frac{1}{l-1} \left(\frac{\sum_{i=1, i \neq C2}^l (avgResponseTime_{Di})}{\sum_{i=1}^l (avgResponseTime_{Di})} * 100\% \right) =$$

$$\frac{1}{3-1} * \frac{(40+60)}{(30+40+60)} * 100\% \Rightarrow PercentageToCluster_{XC2} = 38.45\%$$

$$PercentageToCluster_{XC3} = \frac{1}{3-1} * \frac{(30+60)}{(30+40+60)} * 100\% \Rightarrow PercentageToCluster_{XC3} = 34.6\%$$

$$PercentageToCluster_{XC4} = \frac{1}{3-1} * \frac{(30+40)}{(30+40+60)} * 100\% \Rightarrow PercentageToCluster_{XC4} = 26.9\%$$

So, someone can notice that the higher response time a cluster has, the smaller requests percentage it receives.

3.1.4 Final Equation

By combining Equation1 and Equation2, we can have the exact percentage of requests forwarded to a remote cluster C for a deployment D:

$$FinalPercentage_{DC} = splitPercentageForNeighborCluster_D * PercentageToCluster_{DC}$$

Equation 3

Through this equation it is clear that the final percentage of the traffic that will be forwarded to the neighboring Clusters, depends on the response time, Cluster CPU Utilization and the time delay between these Clusters.

3.2 The Algorithm

Now that we have gained a clear understanding of the underlying principles driving the scheduler, let's move on to the practical implementation of the algorithm.

3.2.1 MyScheduler Pseudocode

Algorithm 1: myscheduler

Input: RP, REQUESTS, D, RPRD, CLUSTER_CPU, CLUSTERS

Start:

1. **while True:**
2. D = order(REQUESTS, D)
3. calculated_CPU = 0
4. calculated_deployments = []
5. **for d in D:**
6. deployment_CPU = calculate_CPU(d, RPRD)
7. **if** (calculated_CPU + deployment_CPU) > CLUSTER_CPU:
8. **break**
9. **else:**
10. calculated_CPU+ = deployment_CPU
11. calculated_deployments.append(d)

12. RD = initialise_or_delete(calculated_deployments, D, CLUSTERS)
 13. splitted_deployment = **None**
 14. **while not RP:**
 15. RD, splitted_deployment = monitoring(RD) # Function1
 16. RPRD = update_RPRD(RPRD)
- (main algorithm)

PARAMETER	DESCRIPTION
RP	Replacement Period
REQUESTS	List with every Deployment and total number of its requests
CLUSTER_CPU	Cluster's CPU resources
RPRD	Resources/Requests/Deployment
D	List of ALL deployments
LP	Lowest CPU Percentage

(brief description of the "myscheduler" parameters)

Algorithm is running continuously. It starts by ordering all the existing infrastructure's deployments based on the number of requests in a list (line 2). It doesn't check if the deployments are running in the specific Cluster; if requests for a deployment is forwarded through this Cluster, then they will be included to the list. Then, for every of this list item (deployment), it computes a new variable which contains the deployment (pod) name and the CPU resources it is consuming. (line 6). For the deployments that are not running in the Cluster, but their requests are forwarded to a neighbor cluster, it will calculate the CPU consumption based on the **RPRD (Resources/Requests/Deployment)**. This is a list that contains the name of every running deployment and the CPU Resources required to answer one request for each deployment respectively. For the first run it is given to the algorithm as a parameter, but later as the algorithm will run some times, it will update this list with more accurate data. Then it checks if this consumption plus the consumption that the Cluster already has due to the previous deployments is higher than the Cluster's CPU limit (line 7). If this is true then it stops.

If not, it adds this CPU Consumption estimation to the already existing Cluster's CPU consumption. The loop is continuing until Cluster's CPU Consumption estimation is about to be more than the actual Cluster's CPU ability. Afterwards, it is deploying or deleting the deployments in the Cluster so the running deployments can match the deployment list it calculated (line 12). In case it has to delete a deployment, it will first create a traffic split, so it can forward all of the incoming requests for the specific deployment to a neighbor Cluster. After the traffic split deletion it proceeds to the deletion of the deployment. In case it has to deploy a new deployment, it will first deploy the deployment and then it will delete the traffic

split which was forwarding the requests to the neighbor Cluster, so host Cluster can start answering the requests. Following that, the Function #1 (monitoring) is called continuously for at least **RP** duration of time (lines 15-16). After every call the algorithm is calculating and updating the **RPRD**, so it can be more accurate with the future estimations of CPU consumption for deployments that are not running to this cluster.

3.2.2 Function 1 (monitoring)

This function is in charge of maintaining the order in the Cluster based on the previous decisions made (which deployments should the Cluster run); If it is required, it may apply some traffic splits, to keep the CPU Consumption between some limits, so it is never under a specific limit or more than the desired limit. These upper and lower limits are set by the system administrator and are inserted to the algorithm through the parameter **CPU_TOL (CPU TOLERANCE)**, which is represented by a percentage. At anytime #Function 1 makes sure that CPU Consumption is between this range.

Function1 : monitoring

Input: RD, REQUESTS, CLUSTER_CPU, RPRD, D, LP

Start:

```

1.   RD = order(REQUESTS, RD)
2.   if CPU_NOW > CLUSTER_CPU * (1 + CPU_TOL):
3.       least_used_deployment = RD[LAST]
4.       extra_CPU = CPU_NOW - CLUSTER_CPU
5.       percentage_to_forward = calculate_split_percentage
6.           (least_used_deployment, RPRD, extra_CPU)
7.       if percentage_to_forward >= 100:
8.           delete(least_used_deployment)
9.           RD.remove(least_used_deployment)
10.      else:
11.          apply_traffic_split
12.              (least_used_deployment, percentage_to_forward)
13.          splitted_deployment = least_used_deployment
14.      if CPU_NOW <= CLUSTER_CPU * (1 - CPU_TOL):
15.          if splitted_deployment:
16.              extra_CPU = CLUSTER_CPU - CPU_NOW
17.              percentage_to_forward = calculate_split_percentage
18.                  (splitted_deployment, RPRD, extra_CPU)
19.              apply_traffic_split(splitted_deployment, percentage_to_forward)
20.          else:
21.              most_used_deployment = D, RD[FIRST]
22.              initialize(most_used_deployment)
23.              RD.append(most_used_deployment)
24.      return RD, splitted_deployment

```

(Function #1 monitoring)

PARAMETER	DESCRIPTION
RD	Replacement Deployments
REQUESTS	List with every Deployment and total number of its requests
CPU_TOL	CPU Tolerance
CLUSTER_CPU	Cluster's CPU resources
RPRD	Resources/Requests/Deployment
D	List of ALL deployments
LP	Lowest CPU Percentage

(brief description of the "monitoring" function parameters)

It starts by ordering the deployments/services that are already running in the Cluster by the number of the receiving requests. Then it compares the current CPU utilization of the host Cluster with the Maximum CPU Percentage (desired limit of CPU, which is less than the overall CPU) of the Cluster.

If the last comparison is true, this means that during this last period maybe myscheduler underestimated some deployments CPU consumption, or the overall traffic suddenly increased. To cope with this problem Function #1 will try to create and apply a traffic split. At this traffic-split, it will set the percentage of the traffic it wants to forward to the neighbor Clusters. It will do this for the least used deployment (the deployment with the lowest number of requests - line 3). The percentage will be calculated based on the equation 3 we described earlier. Then it will apply this traffic-split and it will wait for some seconds in order for the changes to take effect. If equation 3 calculates that the percentage of the traffic that should be forward is more than 100%, then this means that the whole deployment should be deleted and its requests should be forwarded to the neighbor Clusters, because Cluster's CPU power is not enough to deal with this amount of traffic.

On the other hand, if the CPU utilization is less than the lower limit (line 13), then this means that myscheduler overestimated the amount of CPU power needed. This implies that there is sufficient CPU power to decrease the percentage of the forwarded requests of the least used deployment in order to increase cluster's CPU consumption.

If there is not splitted deployment, then it is going to initialize the next most popular deployment (line 20)

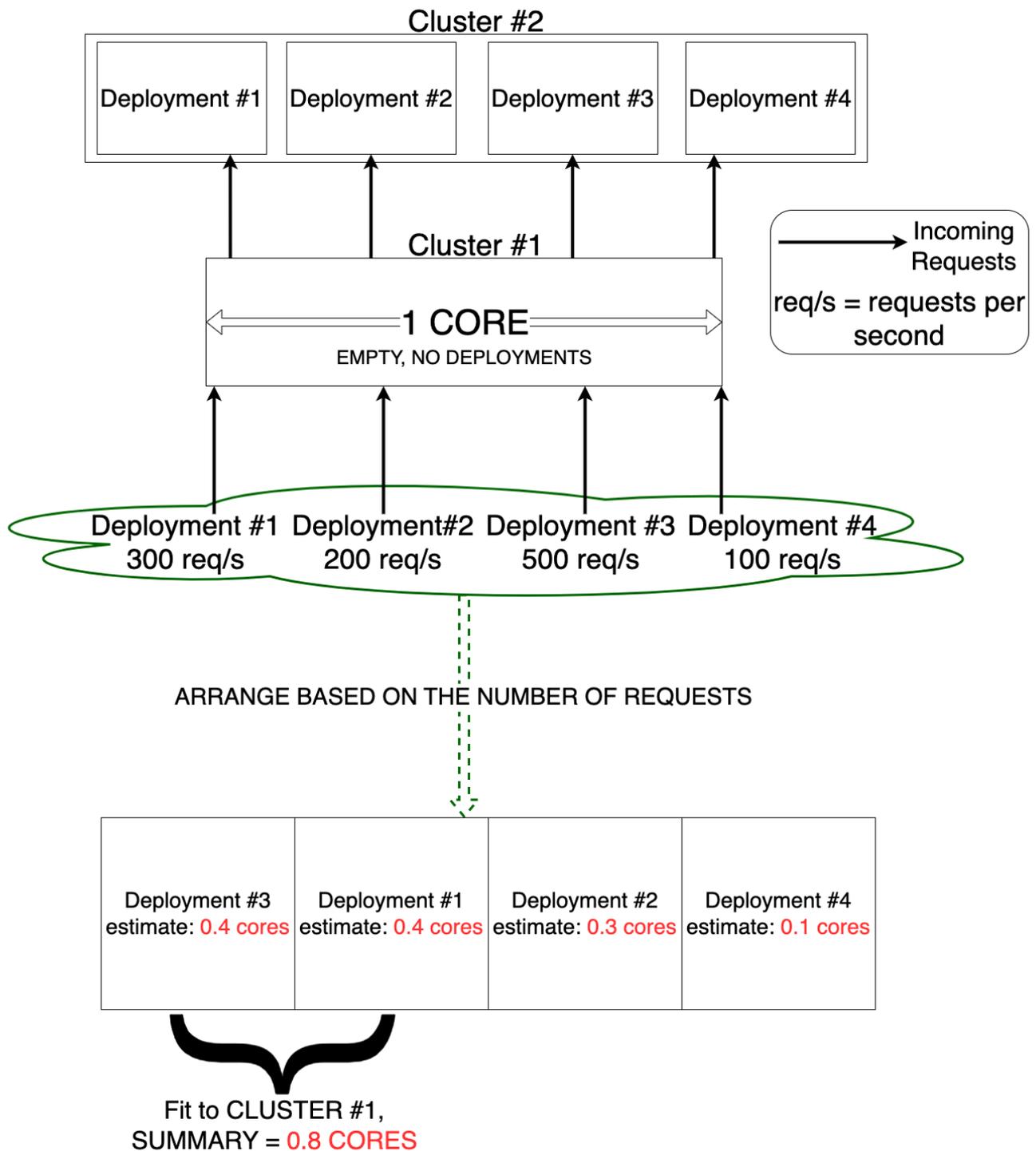
Finally the whole Function #1, will return the Running Deployments and the splitted deployment (if any) and the whole function will run again until the RP period is over ("myscheduler" algorithm line 13)

When this period ends, myscheduler algorithm will run again calculating which of the new new most popular deployments should run to the cluster, initializing or deleting deployments,

and it will call again Function #1, passing its decisions, so it can monitor and make the necessary corrections to ensure that CPU Consumption is between the desired limits.

3.2.3 Example - Algorithm

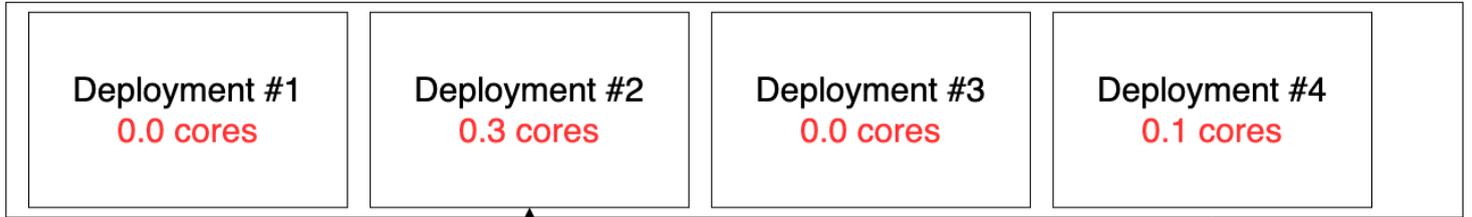
To illustrate, let's suppose that we have an infrastructure of two (2) Clusters, Cluster #1 and Cluster #2. Cluster #1 is running the algorithm and can forward requests to Cluster #2. We also have four (4) deployments that we want to place to our infrastructure. Cluster #2 is running all of the deployments. We are going to demonstrate how the algorithm will choose which deployments are going to be placed to Cluster #1.



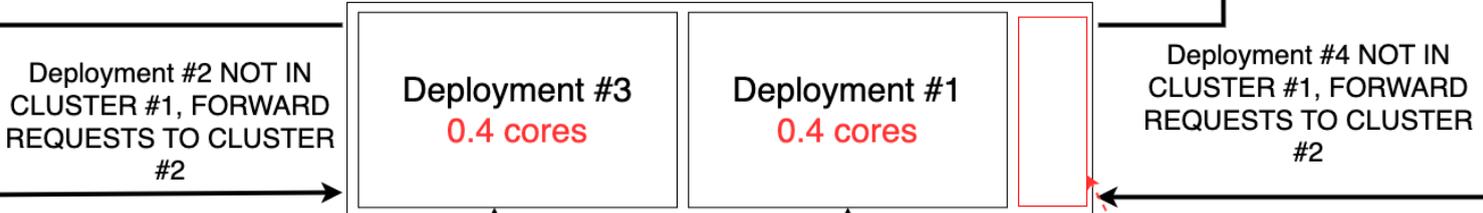
Starting, Cluster #1 is receiving requests for all deployments/services, but it is forwarding them to Cluster #2, because it doesn't have any of them deployed. Algorithm will arrange the deployments based on the number of requests. It has the knowledge of the request's numbers, because all the request are passing by it before being forwarded to Cluster #2. Then it will compute the CPU resources that every deployment is using. If the deployment does not run on the cluster, it will calculate an estimation based on the number of requests. Afterwards it will select which of them can fit in the Cluster (Cluster #1). In our case, Deployment #3 and Deployment #4 selected, because they are the first most popular

deployments and the summary of their CPU Consumption is 0.8 which is less than the 1 core that the host Cluster has.

Cluster #2



Cluster #1



1 CORE

500 req/s

300 req/s

0.2 CORES, NOT UTILIZED

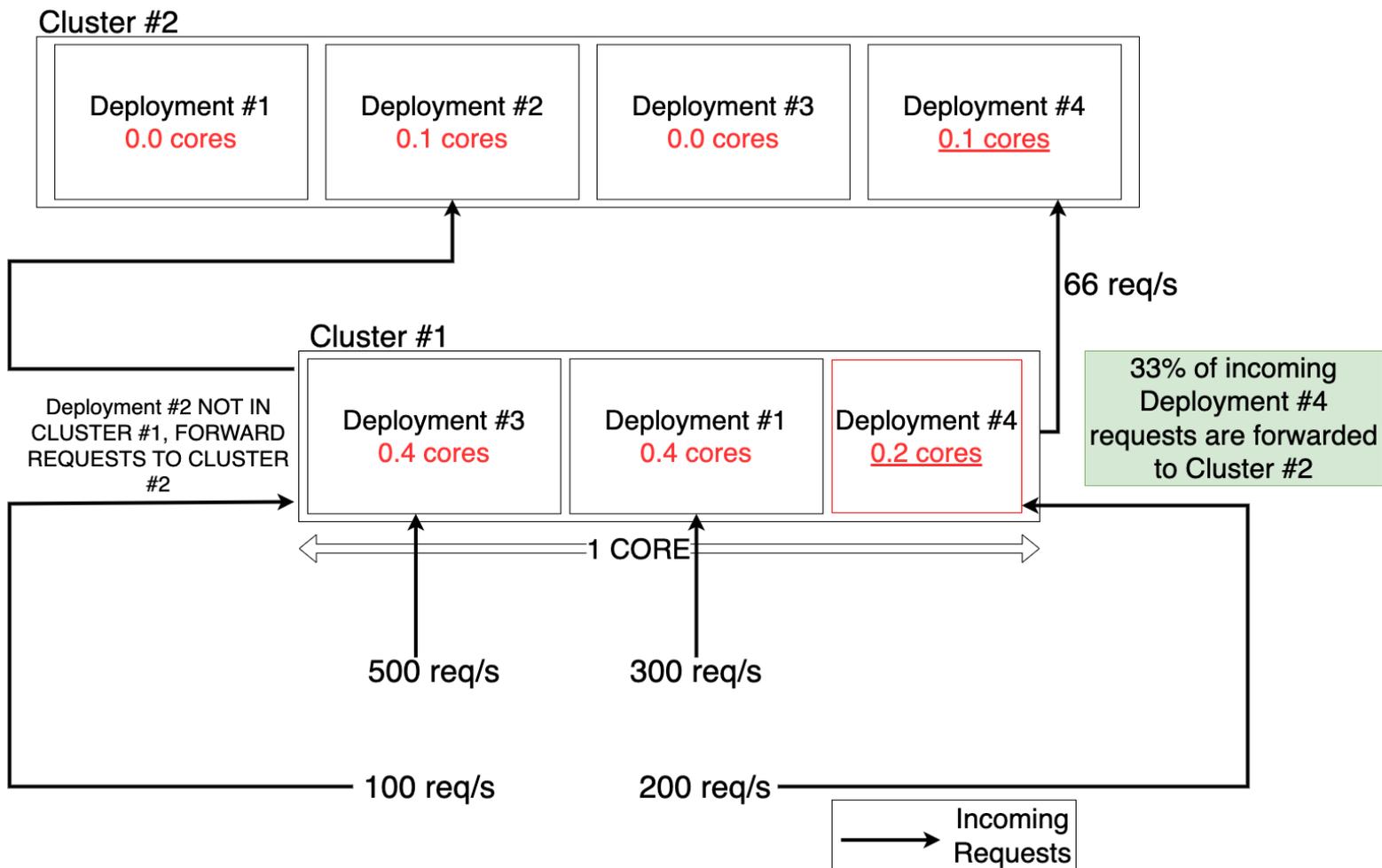
200 req/s

100 req/s

Incoming Requests

Someone can notice that now Deployment #3 and Deployment #4 are deployed in Cluster #1. Requests for Deployment #2 and Deployment #4 are forwarded to Cluster #2 since they are not deployed in Cluster #1. Also we can observe that 0.2 cores in Cluster #1 are not utilized.

Function #1 will observe this and it will try to deploy the next most popular deployment, which is Deployment #2 (200 req/s). Notice that this Deployment is using 0.3 cores, but the Cluster has only 0.2 cores available. Function #1 will initialize the deployment and it will create a traffic split, so it can answer only a part of the requests (the part that is equivalent to 0.2 cores) and it will forward the rest of the traffic to Cluster #2.



Now all Cluster's #1 CPU resources are utilized and they are answering requests.

3.2.4 Algorithms Details

Algorithm is written using the python language. To accomplish the creation and deletion of deployments/services and traffic splits, it is using the official "Kubernetes" python library.[18] In order for this library to achieve this, it is using the Kubernetes REST API. Furthermore in order to run this python program as a pod, it needs to be conteraisted firstly. This requires a docker image that is created from a Dockerfile. The Dockerfile we created for this goal is:

```
FROM python:3
ADD myScheduler.py /
COPY deployments_yamls /deployments_yamls
ADD input_parameters /
ADD traffic_split_template.yaml /
RUN pip install kubernetes
RUN curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/
amd64/kubect1
RUN chmod +x ./kubect1
RUN mv ./kubect1 /usr/local/bin
```

```
RUN curl --proto '=https' --tlsv1.2 -sSfL https://run.linkerd.io/install | sh
ENV PATH "$PATH:/root/.linkerd2/bin"
CMD [ "python", "./myScheduler.py" ]
```

It is starting with the official python 3 image as a base. After this it and then it adds some necessary requirements:

- scheduler code: The python code that will run the described algorithm
- deployment yaml files: A directory with the yaml-formated files that describe the deployments and the services.
- traffic split template files: The code will use this file as template in order to create traffic splits.
- input parameters: The scheduler's default settings. This is a file, following this format:

```
MAXIMUM_PERCENTAGE: 0.95
LOWEST_PERCENTAGE: 0.85
TIME_INTERVAL_COLLECT_REQUESTS: 60
TIME_INTERVAL_TRAFFIC_SPLIT: 20
TRAFFIC_SPLIT_TEMPLATE: traffic_split_template.yaml
namespace: scheduler
```

It defines the range that Cluster's CPU is allowed to run (Maximum and minimum percentages), the replacement period, the time that the scheduler should wait until after it apply a traffic-split, the traffic split template file name and the desired kubernetes namespace that the scheduler should run.

After this, it is installing kubectl (Command Line Interface (CLI) that is another way of communication with the KUBERNETES API) and the Likerd cli.

Finally it sets as an entrypoint the command "python mysScheduler.py", so when the container starts it will execute it.

Having setted up the docker image, the next step is to create a deployment yaml formatted file that describes to kubernetes how to create the pod. It describes from which docker repository should the docker image be pulled (downloaded) and the scheduler's setting as well. If the settings are not existing in the deployment file, the default setting will be applied.

In order for the scheduler to have the permissions to create, watch, delete and edit deployments and traffic splits in the Cluster, ClusterRoles and ClusterRoleBinding should be applied.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: myscheduler-role
rules:
- apiGroups: [ "", "metrics.k8s.io", "apps", "split.smi-spec.io" ]
  resources: [ "pods", "deployments", "trafficsplits", "services" ]
  verbs: [ "get", "watch", "list", "create", "delete", "patch" ]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```

metadata:
  name: scheduler-role-binding
subjects:
- kind: ServiceAccount
  name: default
  namespace: scheduler
roleRef:
  kind: ClusterRole
  name: myscheduler-role
  apiGroup: rbac.authorization.k8s.io

```

By applying this ClusterRole and ClusterRoleBinding to the cluster, every resource that will be deployed at the “scheduler” namespace will have the permissions to get, list, watch, delete, create and patch pods, deployments, services and traffic-splits.

Following stage, after creating the ClusterRole and ClusterRoleBinding yml files and the the docker image of the scheduler to a local docker repository we can apply them to a cluster using this kubernetes cli command:

```
kubectl apply -f {files}
```

3.2.5 Scheduler - Prometheus Communication

By installing linkerd-viz, (an extension of Linkerd service mesh) the cluster has a monitoring application based on Prometheus and Grafana, autoconfigured to collect metrics from linkerd. Linkerd pods is using this Prometheus instance to monitor the various requests among the different services. Since this instance is already setted up and running, “myscheduler” pod is making some APIs call to it in order to receive data like the number of incoming and outgoing requests, response times of the requests and CPU utilization. An examples of the actual API call that myscheduler is using:

```

app_request = {"query":
  "sum(increase(request_total{direction='inbound',app='%s',target_addr!~'0.0.0.0:.*'}[%s])
)" % (service,intervalProm) }

```

This call is asking Prometheus, using PromQL(Prometheus Querying Language) the number of inbound requests for the service “service” for the last “intervalProm” seconds for which the target address is different than “0.0.0.0” and any port.

```

app_request = {"query":
  "increase(request_total{dst_service=~'.*-.*',deployment!~'linkerd.*'}[%s])" %
(intervalProm) }

```

The above call is asking Prometheus the total number of requests which have as destination, a service following this naming convocation: {anything}-{anything}, without including requests coming from services that start with the words “linkerd”. The specific query is used to trace the number of the outbound requests. Outbound requests have as target services with this name convocation: servicename-cluster2. Apart from this it has to make sure that it doesn't include requests that come from linkerd pods to remote clusters linkerd pods, because these pods are not considered part of the workload.

Prometheus instance comes with a service named "prometheus" in the "linkerd-viz" namespace running at the port "9090". By default Kubernetes CoreDNS creates an entry with the same name(prometheus.linkerd-viz), so myscheduler can make its calls to this address:

```
http://prometheus.linkerd-viz:9090
```

In order for the infrastructure's clusters to be able to communicate they should have installed a service mesh technology. In our approach we selected linkerd because:

- it's lighter and smaller: Edge Cluster's resources are limited, so we need to minimize infrastructure's resource consumption and assign more resources to the actual workload.
- it's faster: The main goal of this work is to minimize the response time to the final users. The selected service mesh technology should adapt and help in the achievement of this goal.
- it's safer: Edge computing security is a big concern. Using Linkerd all communication between mesh pods is automatically secured and verified using mutual TLS encryption.

All of the involved Clusters need to run Linkerd. In order for someone to install it first have to install Linkerd's cli using this command:

```
curl --proto '=https' --tlsv1.2 -sSfL  
https://run.linkerd.io/install | sh
```

This will download the binaries which include linkerd's cli.

The next step is to connect to the target cluster through the same computer and run the next command:

```
linkerd install --crds | kubectl apply -f - && linkerd install |  
kubectl apply -f -
```

This command will generate and apply the necessary custom resource definitions and kubernetes resources (deployments, services etc. - (yaml-formated files) on the target Cluster.

So far we have installed linkerd to the Cluster. Following action it is to install the muticluster functionality to the linkerd. Someone can achieve this by running the following command:

```
linkerd multicluster install | kubectl apply -f -
```

Now multicluster functionality is installed. This procedure has to be repeated for all of the infrastructure's Clusters.

Further stage is to connect clusters with each other. To establish a connection between each cluster, certain resources need to be set up in the source cluster. These include the installation of a secret containing a kubeconfig, which grants access to the target cluster's Kubernetes API, a service mirror control for mirroring services, and a custom resource called "Link" for storing the configuration. To establish the linkage between Cluster#1 towards

Cluster#2, the following command would be executed, while the computer that will run the command is connected to Cluster #1

```
linkerd --context=east multicluster link --cluster-name cluster2 |  
kubect1 --context=cluster1 apply -f -
```

By running this, the computer that runs the command will connect to Cluster #2 and will run the first part of the command in order to generate the yaml-files that describe how can another Linkerd cluster connect to Cluster #2. These files will be applied to Cluster #1 which will establish the connection with Cluster #2.

Finally, in order to install the prometheus instance which is pre-configure and provides administrator's and scheduler with the necessary data, like number of requests, CPU consumption and response time a linkerd-viz should be installed (official Linkerd extension), using this command:

```
linkerd viz install | kubect1 apply -f -
```

By default not all of the services are exposed to multicluster functionality. It can be enabled on a specific server by adding this label in the service yaml file:

```
mirror.linkerd.io/exported=true
```

or by running this command:

```
kubect1 label svc foobar mirror.linkerd.io/exported=true
```

This label will allow the source Cluster to explore and mirror the target service, so it can later forward requests to it.

In order to secure the connection between Clusters Linkerd requires a shared trust anchor. With this, communication will be encrypted and verification and identification of the requests will be enabled. Linkerd provides a powerful guide to establish a single trust anchor certificate shared between multiple clusters.

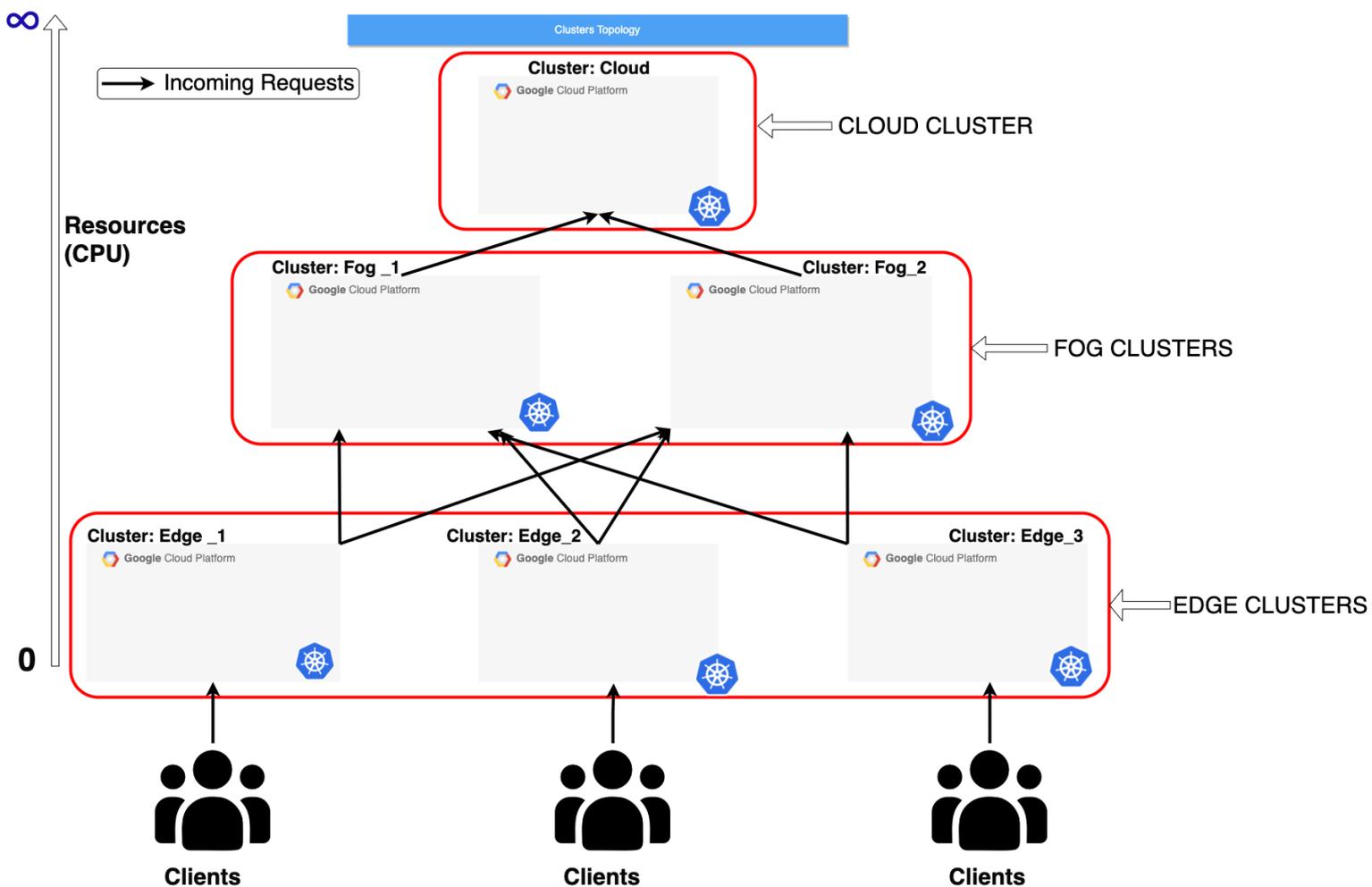
Chapter 4

Implementation

This Chapter will describe the experiments infrastructure set up, the test tools and applications that are going to be used to run a series of test scenarios to benchmark our proposal.

4.1 Architecture

In order to create a cloud architecture that is as near to a real world scenario, we replicated the following structure:



As we can see the first layer consists of the Edge Clusters. Edge clusters are low in terms of resources (CPU, Memory). The advantage of them is that they are physically near the end users, so they have less communication delay with them compared to any other cluster layer. These machines can be represented by any low-resource device, like a router, a raspberry pie, a mini computer etc.

Next layer is made up of the Fog Clusters. These clusters are exactly the next layer of the edge and they are receiving requests from them. Being Fog clusters, their resources are more compared to the below layer, but they are not physically as close to the end users, so the communication delay between them is also more. This layer can be small servers or computers located near by to the end-users.

Finally the last layer is the cloud layer. In this one there is one Cluster which is typically located in a big-data center and has huge resources availability. The drawback is that it is even physically further than the rest of the layers, so the time delay between it and the final users is huge. In our experiments we used Google Cloud Platform (GCP), where we replicated this infrastructure by creating 6 (six) Virtual Machines (VMs) with these characteristics:

Layer name	Cluster's Resources	Zone
CLOUD (X1 cluster)	*N2 series CPU 8 cores 16GB ram	europe-west4-a
FOG(X2 clusters)	*N2 series CPU 4 cores 8GB ram → 2 cores for apps, 2 cores for scheduler and infrastructure software	europe-west4-a
EDGE(X3 clusters)	*N2 series CPU 2 cores 4GB ram → 1 core for apps, 1 core for scheduler and infrastructure software	europe-west4-a

Every of the above VMs is running Ubuntu 20.04 as an operating system. In order for them to be clusters they need to run a kubernetes software. For this purpose, we installed Rancher K3s [19] on them. Rancher K3s is a lightweight version of Kubernetes. We select the specific implementation, because Edge layers are low in terms of resources and K3s does not consume as much of them compared to the Kubernetes vanilla version.

Following this, we installed linkerd, linkerd-multicluster and linkerd-viz as was described previously. Then we linked the clusters so they can replicate the target architecture.

Edge1 → Fog1 & Fog2,

Edge2 → Fog1 & Fog2,

Edge3 → Fog1 & Fog2,

Fog1 → Cloud,

Fog2 → Cloud

Due to the fact that all of the clusters are in the same zone (europe-west4-a), they are physically close to each other (probably in the same building) and they are all connected to the same local network. We end up with this choice, because we would like the communication delay between them to be as close to zero as possible, so we can stimulate the time delay between them artificially. To achieve this, we installed Chaos-Mesh on every of them.

4.2 Choice of tools

4.2.1 Chaos Mesh

Chaos-Mesh[20] is an open source cloud-native Chaos Engineering platform. It provides a range of fault simulation options and possesses significant capacity to coordinate fault scenarios. One of these is the time-delay chaos scenario. It is built based on Kubernetes CRD (Custom Resource Definition). The core components of it are:

- Chaos Daemon: The primary operational component. Its design allows it to interact with network devices, file systems and kernels.
- Chaos Controller Manager: This is the main component, which is responsible for planning and the management of the experiments.
- Chaos Dashboard: A web interface which administrators can use to monitor the experiments.

For our experiments we are interested in the time-delay functionality. Chaos Mesh is using the Linux Traffic Control (tc)[21] utility that gives it the ability to configure the kernel packet scheduler. TC is used by network administrators for rate limitation, traffic priority etc. By editing the tc rules, Chaos Mesh can delay packets sent to or from a specific IP address, ports or network interfaces. In order to create a traffic delay for incoming traffic of a kubernetes service, a yaml-formatted file following this template should be apply to the cluster:

```
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: network-delay
spec:
  action: delay # the specific chaos action to inject
  mode: one # the mode to run chaos action; supported modes are
  one/all/fixed/fixed-percent/random-max-percent
  selector: # pods where to inject chaos actions
    namespaces:
      - linkerd-multicluster
    labelSelectors:
      'app': 'linkerd-gateway' # the label of the pod for chaos injection
  delay:
    latency: '20ms'
```

In the above file, we have to specify in which kubernetes namespace is the service we want to delay, the label of the specific service and finally the delay-latency in milliseconds.

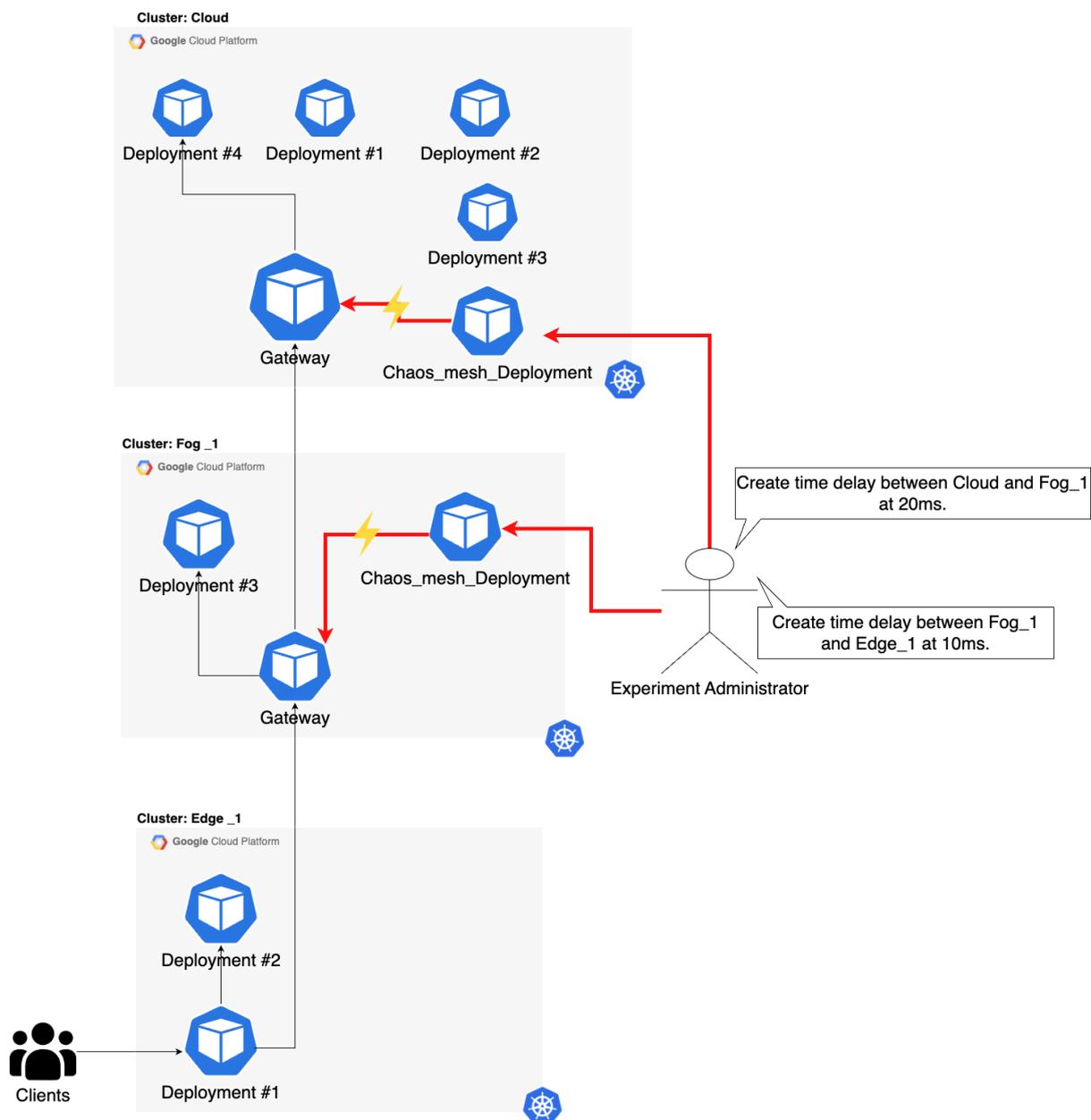
In our case we want to create a specific time-delay between the connection of the clusters. Service “linkerd-gateway” is the service that every cluster with linkerd-multicluster functionality installed has. It is the service which is in charge of receiving the incoming requests from the source Clusters. By creating a delay for this service, we are creating a delay controlled between the communication of the clusters.

Example

To demonstrate how the time-delay simulation works, let's suppose that we have an architecture of three (3) clusters, Edge1, Fog1 and Cloud and they are connected with each other like this:

Edge1 → Fog1 → Cloud

If the administrator wants to create a delay of 10 ms between Fog1 and Edge1, he has to create a yaml file similar to the one described above and apply it to the cluster Fog1. After this Chaos Controller Manage will pick this file up and will give instructions to Chaos Daemon to edit the Linux tc[] rules in kernel level, so it will create delay to the incoming network packages that have as target the IP of the described kubernetes Service. Exactly the same procedure should be followed if we want a time-delay between Fog1 and Cloud clusters, but this time the yaml file should be applied to the Cloud Cluster:



In our experiments, we applied the delay on the linkerd Gateway services, which are in charge of accepting the incoming requests from a remote Cluster. Notice that in any given time, administrators of the experiments can recall their decision for time-delay creation by deleting the applied NetworkChaos yaml file.

4.2.2 DNS Server

We also created a local DNS server using Google's "Cloud DNS". Architecture's cluster is using this DNS Server to communicate with each other, without having to mention a local IP address which would change after every VM restart. During the linkerd-multicluster link we used DNS entries, because if the local IP address of the VMs were changing after a reboot Linkerd wouldn't be able to establish connection with the target Cluster. Moreover this DNS server was useful for the pulling of docker images from the docker registry.

 **Filter** Filter record sets

<input type="checkbox"/>	DNS name 	Type	TTL (seconds)	Routing policy
<input type="checkbox"/>	cloud.thesis.com.	A	300	Default
<input type="checkbox"/>	edge1.thesis.com.	A	300	Default
<input type="checkbox"/>	edge2.thesis.com.	A	300	Default
<input type="checkbox"/>	edge3.thesis.com.	A	300	Default
<input type="checkbox"/>	middle1.alivecheck.thesis.com.	A	300	Default
<input type="checkbox"/>	middle1.thesis.com.	A	300	Default
<input type="checkbox"/>	middle2.alivecheck.thesis.com.	A	300	Default
<input type="checkbox"/>	middle2.thesis.com.	A	300	Default
<input type="checkbox"/>	registry-instance.thesis.com.	A	300	Default
<input type="checkbox"/>	thesis.com.	SOA	21600	Default
<input type="checkbox"/>	thesis.com.	NS	21600	Default

4.2.3 The tooling-cluster

The aim of our work is demonstrate how we can improve the response time to the final users by placing the pods dynamically among the existing infrastructure. Applications a lot of times require access to databases to retrieve, create, edit or delete application data. Since in our approach it is very highly possible for the same service to exist in multiple clusters, this would result in a significant issue if every pod had its own database initialized as a container, because we would not have a single source of truth. To deal with this problem we created another cluster naming registry-instance cluster. We installed the same software as the other clusters on it and we linked it to all of the clusters, but we did not include any time-delay

between these connections. On this cluster we created two kubernetes deployments/services:

- mongoDB
- mySQL
- Redis

These three databases are going to be used by all of the rest services who require a database, so they can have a single source of truth.

Apart from application data, there was a need for a common docker-registry from which the services can download the application's docker images so they can run their pods. For this purpose we installed the Docker software on the registry-instance cluster and we deployed a docker registry as a docker-container in which we pushed all application docker images. With this procedure all of the clusters can have access to the same docker registry and download exactly the same images,if needed.

4.2.4 Ambassador Ingress controller

Ambassador is a Kubernetes-native API Gateway built on Envoy Proxy. Managed entirely via Kubernetes Custom Resource Definitions, Ambassador provides powerful capabilities for traffic management, authentication, and observability.[22] In this work, we used the Ambassador Ingress Controller, in order to accept requests outside of the Kuberntes environment, (from the host machine) and forward them to the frontend kubernetes services.

4.2.5 Users-Workload

For the purpose of the workload, Locust[23] tool was used for all of the experiments. Locust is a scriptable performance testing tool. Performance testers can write the behavior of the users in Python code. it can mimic at a very good level user's behaviors enabling the assessors to test multiple scenarios and workloads. The primary way to control it is through the Locust UI, but due to its scriptable nature someone can just predefine the experiment duration and the workload and save the results in csv files. In our experiments we are going to use three (3) instances of Locust; every instance is mimicking a different workload, amount and type of requests. Controlling all of them instances at the same time through different WEB UI instances would require perfect time synchronization, so due to this, we predefined all the workload behavior and experiments duration time in python files and the results were saved automatically in csv files. Using these files we were able to monitor and observe data like the response times percentiles, the successful and failed ratios of the requests.

4.3 Controller Cluster & Experiment Cost

Finally we created another extra cluster which acts as the controller cluster. It has access to all of the rest of the clusters and through this administrator can connect to them in order to install software and run,manage and observe the experiments. On the same machines Locusts instances are running as well.

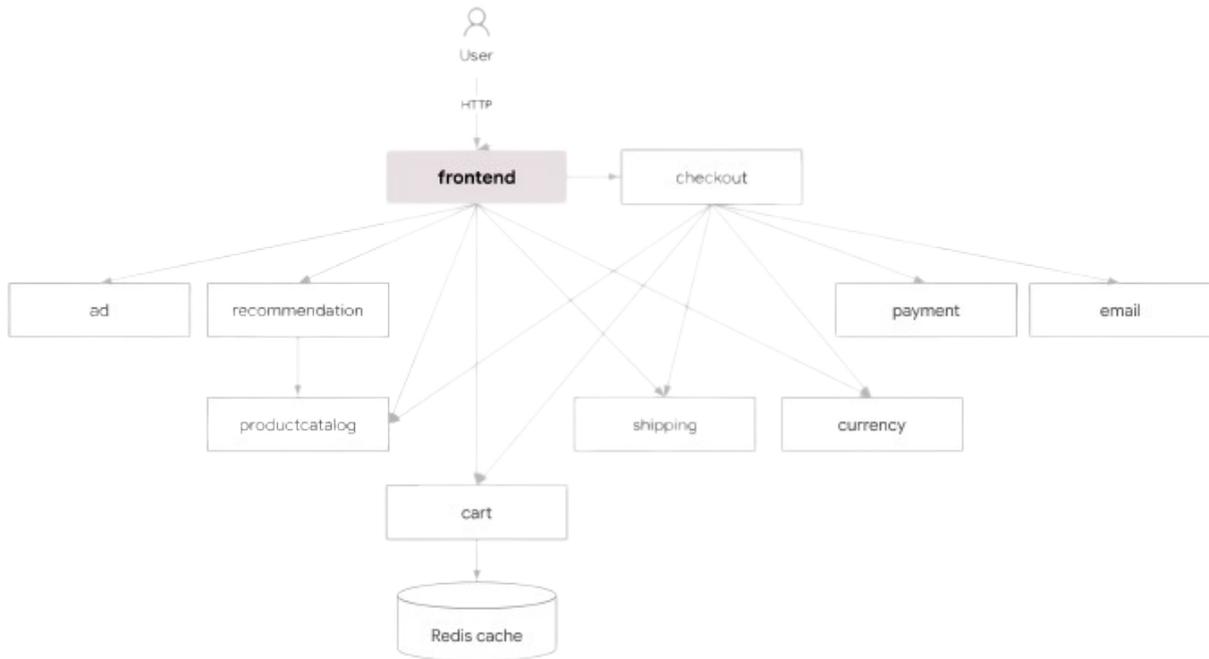
To keep the cost of the experiment low, we assigned a public IP address only to this Cluster. Since It is connected to the same local network as the rest of the Clusters, we avoided assigning public IP addresses to the rest of the Clusters. Furthermore to keep the cost even lower, we decided to select "preemptives VMs".[24] Preemptive VMs are normal VMs provided by the cloud provider (Google) with a significant price discount (at the time ~ 60,91%) compared to the price of standard VMs. However, they can stop at any given time, if the provider needs to reclaim the computer capacity for allocation to other VMs. During our experiments we didn't experience any preemption issue.

4.4 Benchmark Applications

For the experiment phase we used two cloud microservices applications. Both of them consist of various microservices which communicate with each other through various protocols, mainly HTTP. In order to stimulate more accurate low resources cluster's behavior, we eliminated every resource limitation that these microservices may have in the pod level.

4.4.1 Google's Online Boutique demo application

This application is composed of 11 microservices.[25] It's a web-based e-shop where users can view products, add them to the card and purchase them. It's an open source project publicly available, provided by Google for test/experiments purposes. Communication from the users towards the application is conducted with HTTP and communication between microservice through gRPC. Linkerd service mesh is capable of handling both of the protocols.



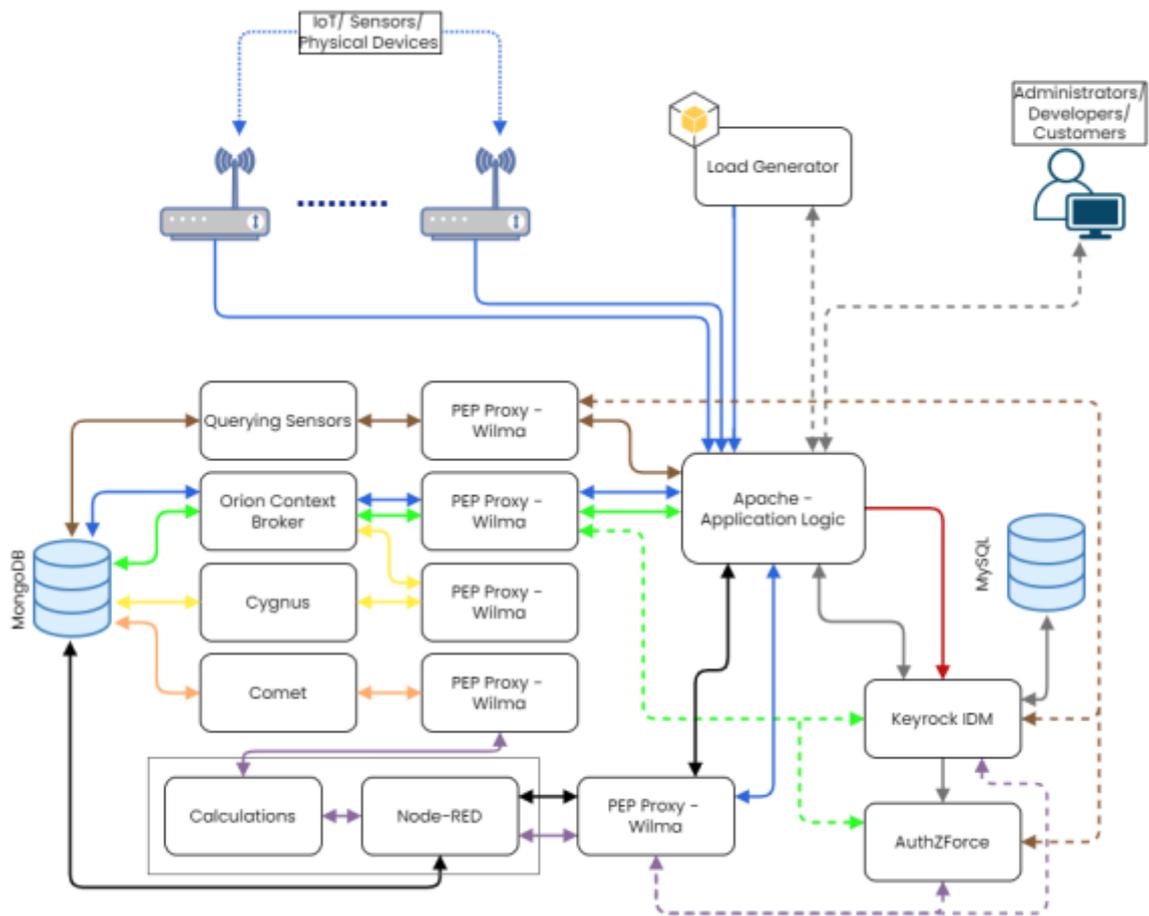
Load Generator microservice was removed, because requests in the experiment are generated from external sources (clients) outside the cluster.

Redis cache is servicing the role of database for this application. Due to this reason this microservice will not be scheduled by the algorithm during our experiments and will be deployed manually in the “tool-ing cluster” without any artificial delay.

4.4.2 IXEN

iXen is an application created and designed by the Intelligence Lab of Technical University of Crete[26].It is based on the Service Oriented Architecture. It excels at managing diverse sensors, which send measurements to the iXen cloud platform. The platform uses a publish/subscribe model, letting users easily subscribe to sensor data. This enables flexible data usage in formats that suit their needs. Currently there is an implementation of the application in microservice-oriented architecture that can be deployed and orchestrated by Kubernetes Clusters.

The application is composed of 11 microservices namely: Frontend, Querying Sensors, Mashup Service, Keyrock, AuthzForce, PEP Proxy, Orion Context Broker, Cygnus, Sth-Comet, MongoDB and MySQL.



MongoDB and MySQL instances are serving data saving purposes. For this reason they are only deployed in the "tooling cluster" and no artificial communication delay was added to them.

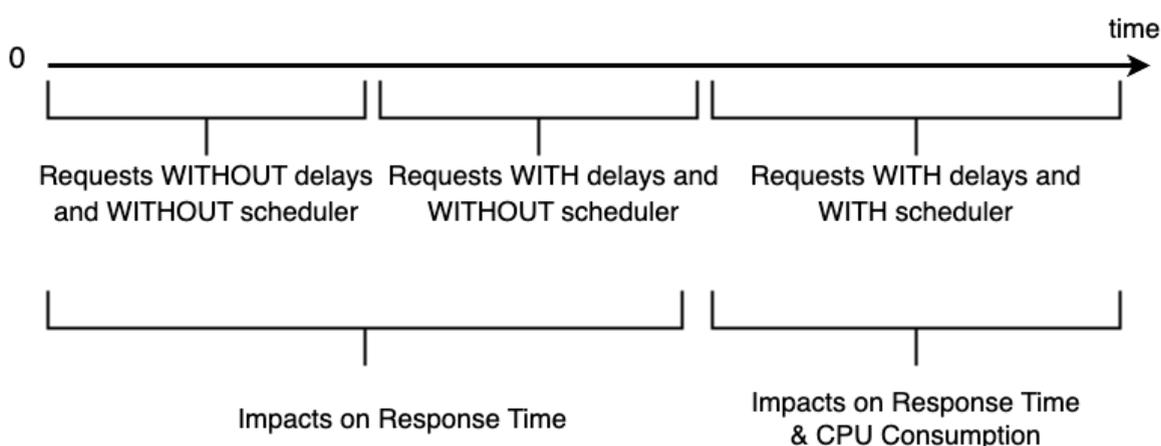
Chapter 5

Experiments & Results

We conducted five experiments. Each of them is simulating a different case scenario. The parameters of them are:

- number of requests
- communication delays between clusters
- workload ratios

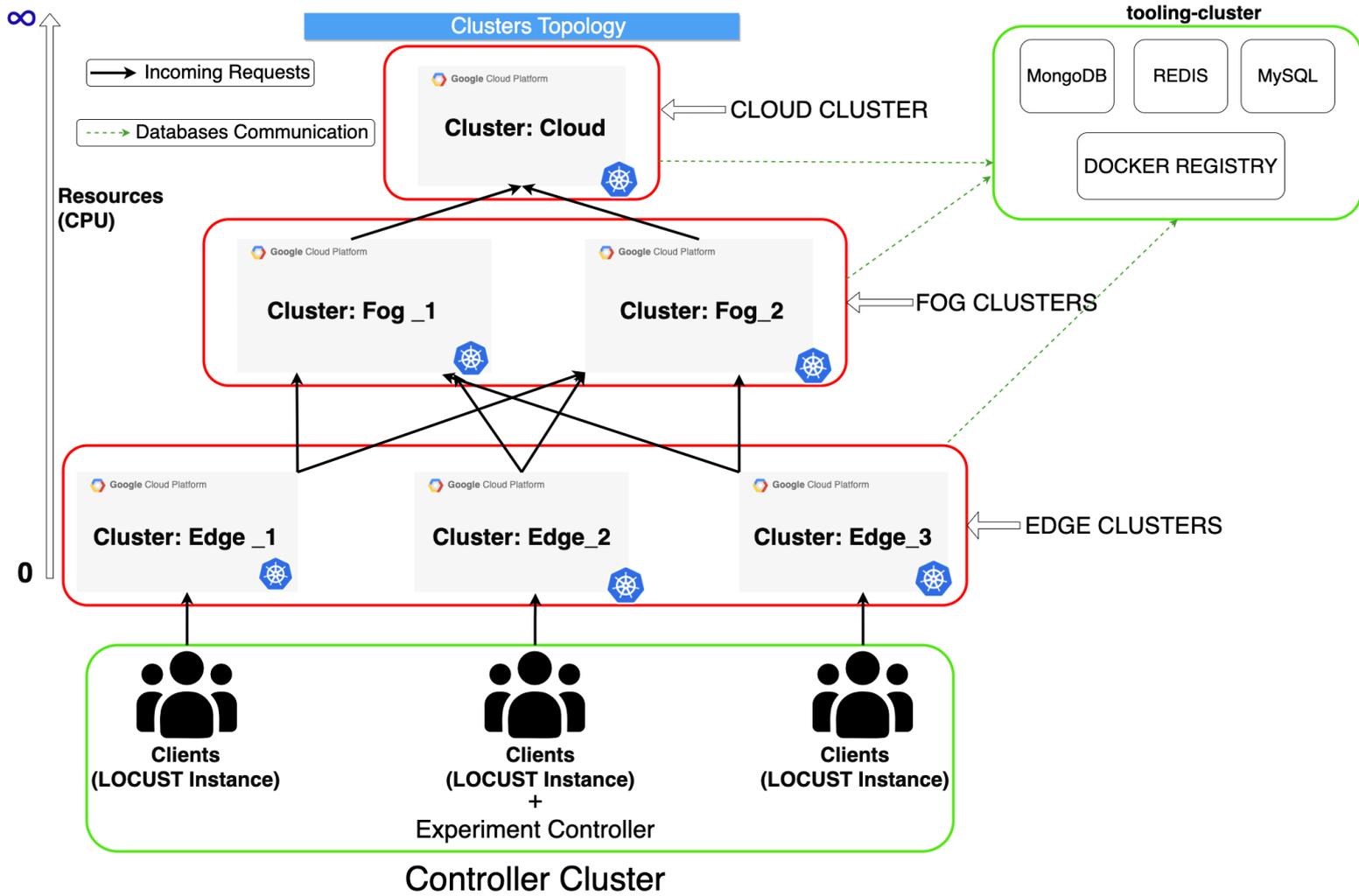
In all of them we are monitoring and observing final response time to the end users and CPU fluctuations of every Cluster in the topology among the experiment time. All of them are following the next linear time frame:



Firstly we start generating requests toward the Edge Clusters. For the first part we are letting clusters answer the incoming requests with the default service placement. Moving on the second part, we are enabling the time-delays between clusters. So far we are not expecting any impacts on the CPU utilization, but in every case we can see the response time to increase while we are entering the second part. Finally we are deploying the scheduler in every cluster which will implement the algorithm and will do changes in the running services of the Clusters. At this point we are expecting to see impacts on the CPU resources.

All of the above parts are conducted automatically by a bash script which is running in the tooling cluster so all of the experiment phases can be synchronized among the various clusters.

In the next graph, we present the finally topology of the experiment:



All of the experiments require an initial static service placement (default placement). We will compare the results of our proposal with this default static placement. The default placement was done manually by taking into consideration the number of requests. E.X.: The services that are receiving the traffic are always placed at the edge Clusters. The default placement is the same for all of the experiments and it can be found in the next tables: (Notice that frontend and apacheservice services are the only point of entrance for the the requests of Eshop and IXEN application respectively. Thus both of them were placed in Edge clusters)

Edge1	Edge2	Edge3
frontend	frontend	frontend
apacheservice	apacheservice	apacheservice

Fog1	Fog2	Cloud (All Services)
paymentservice	paymentservice	cartservice

Fog1	Fog2	Cloud (All Services)
shippingservice	shippingservice	currencyservice
emailservice	emailservice	shippingservice
cartservice	cartservice	emailservice
advertice	advertice	paymentservice
recommendationservice	recommendationservice	recommendationservice
checkoutservice		adservice
currencyservice		checkoutservice
keyrockservice		frontend
		productcatalogservice
		noderedservice
		orionservice
		keyrockservice
		authzforceservice
		cygnusservice
		apacheservice
		queryingsensorservice
		cometservice
		orionproxyservice
		cygnusproxyservice
		noderedproxyservice
		queryingsensorsproxyservice
		sthcometproxyservice

Performance Evaluation

The metrics that were used in order to evaluate our proposal were Requests Per Second (RPS) along with Locust users, Response Time and CPU Consumption.

Requests Per Second refer to the frequency at which clients send requests within a span of one second. In this context, this definition is closely connected with the behavior of users in the Locust software. To elaborate, as outlined earlier, Locust operates based on the specified user actions (including the count and type of requests) that need to be replicated, and subsequently generates and manages these requests accordingly.

Regarding response time, it denotes the time elapsed between a client sending a request and subsequently receiving a response. Due to the fact that the average response latency can sometimes be misleading (it can be significantly skewed by a set of extreme results), we utilized the 95th percentile latency. It represents that the response time of 95% of the requests is equal or lower than that p95 value.

Finally CPU Consumption was measured in "cores". E.X. If a cluster is using 0.7 cores , this means that it is utilizing the 70% of one core of its CPU cores.

5.1 Experiments 1-2-3

In these group of experiments the next parameters used:

Delays: 20ms from the Edge to Fog Clusters, 10ms from the Fog to Cloud Clusters

Workload: 50% of the users are sending requests to the Eshop App and 50% to iXen.

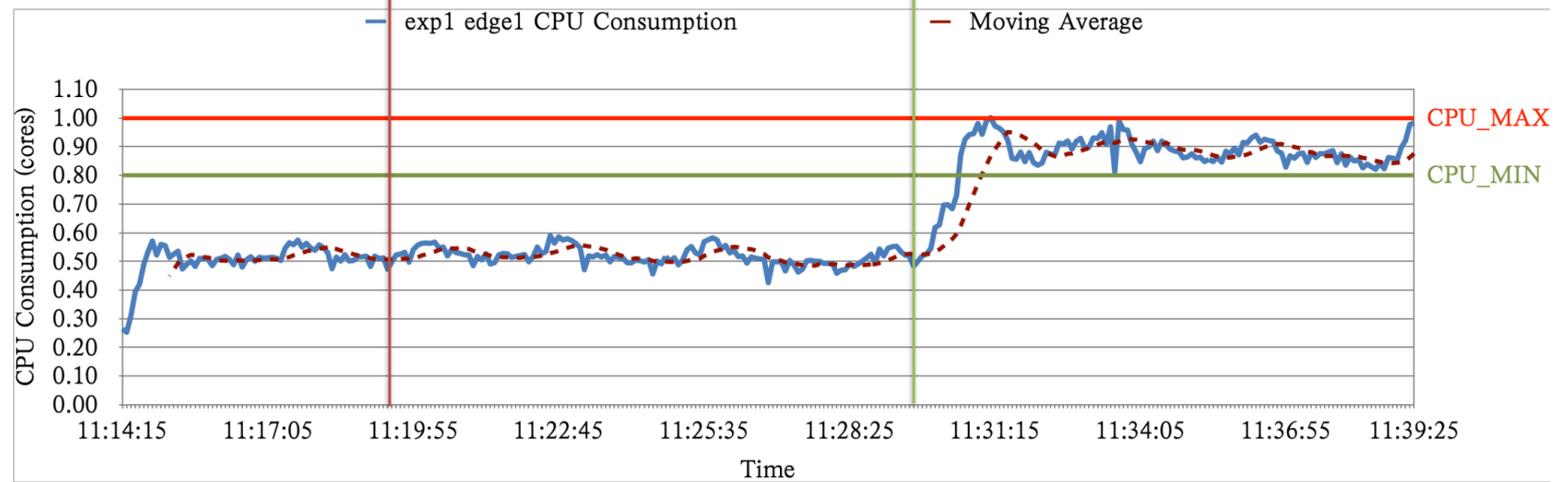
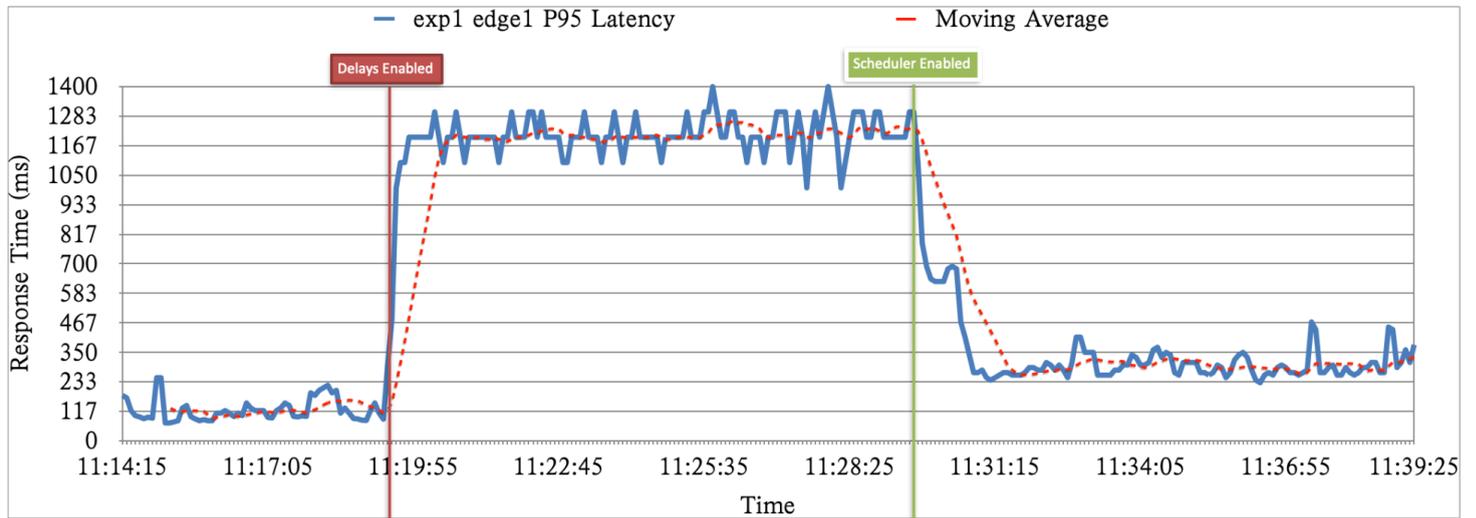
The only parameter change between this group of experiments is the total number of users.

5.1.1 Experiment 1

Users: 100 users/Edge Cluster

Results:

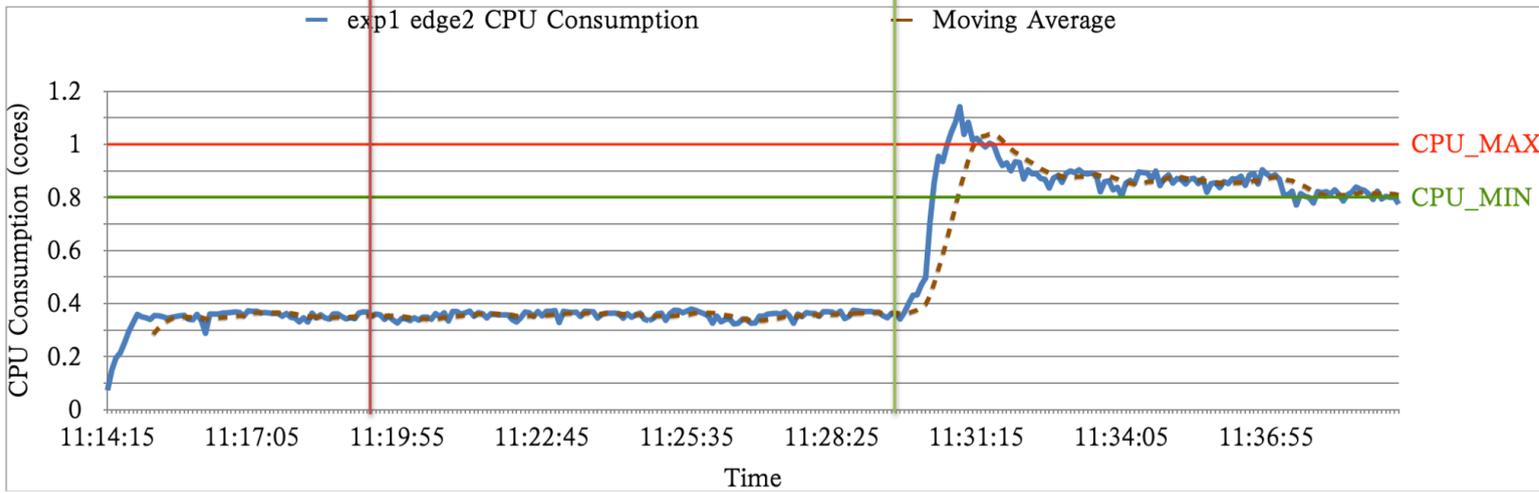
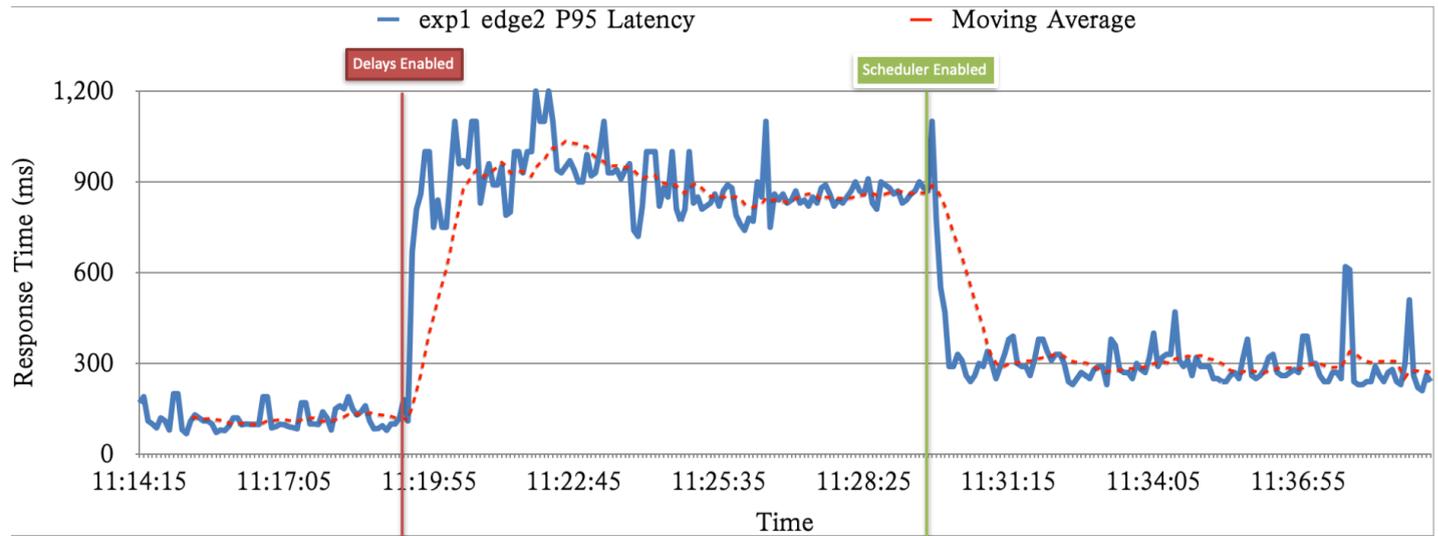
Edge1:



Response Time Average (ms)	
Before Delay Activation	115
After Delay Activation	1170
After Scheduler Stabilization	330
Total Reduction percentage	-71.7%

CPU Consumption Average (cores)	
Before Delay Activation	0.5
After Delay Activation	0.5
After Scheduler Stabilization	0.9
Total Additional percentage	+80%

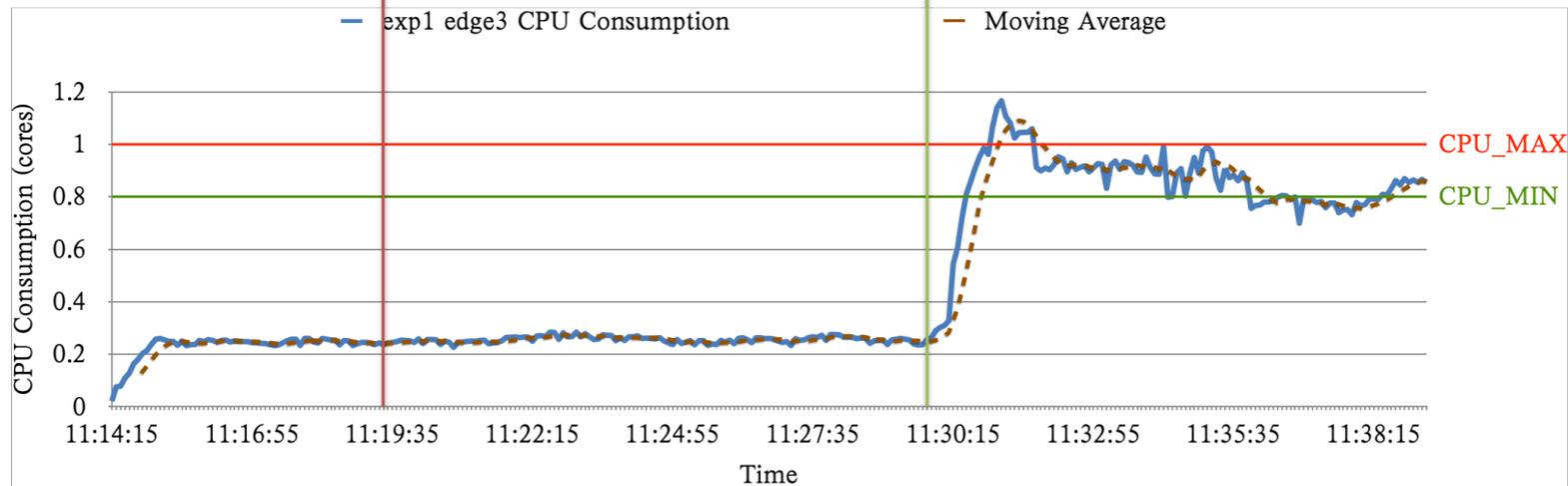
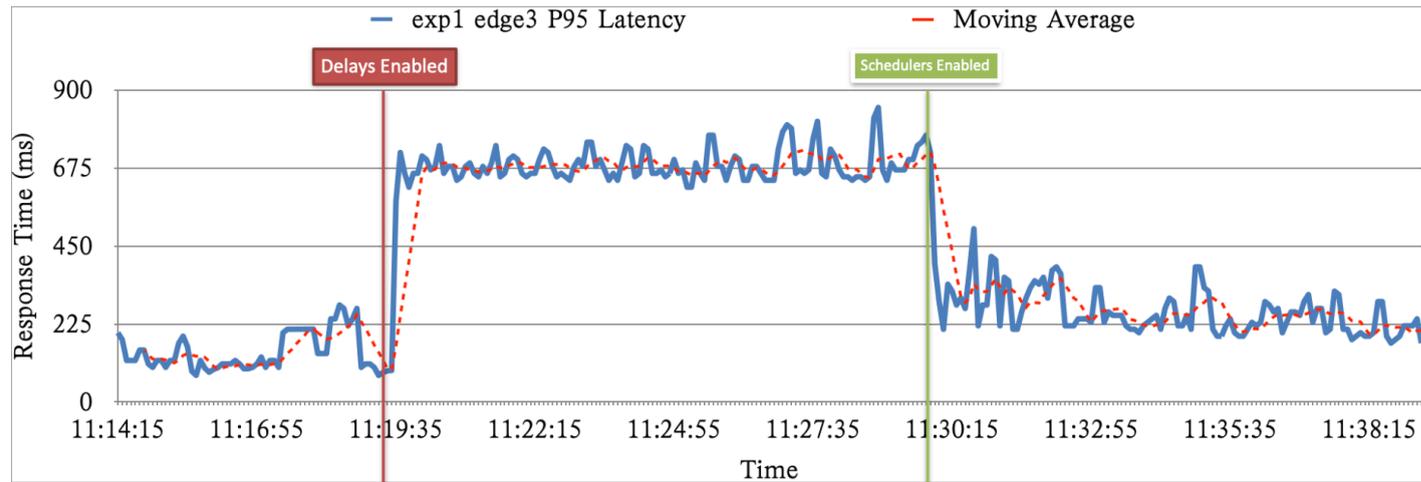
Edge2:



Response Time Average (ms)	
Before Delay Activation	115
After Delay Activation	900
After Scheduler Stabilization	300
Total Reduction percentage	-66.6%

CPU Consumption Average (cores)	
Before Delay Activation	0.35
After Delay Activation	0.35
After Scheduler Stabilization	0.8
Total Additional percentage	+128.5%

Edge3:



Response Time Average (ms)	
Before Delay Activation	115
After Delay Activation	675
After Scheduler Stabilization	230
Total Reduction percentage	-65.9%

CPU Consumption Average (cores)	
Before Delay Activation	0.25
After Delay Activation	0.25
After Scheduler Stabilization	0.9
Total Additional percentage	+260%

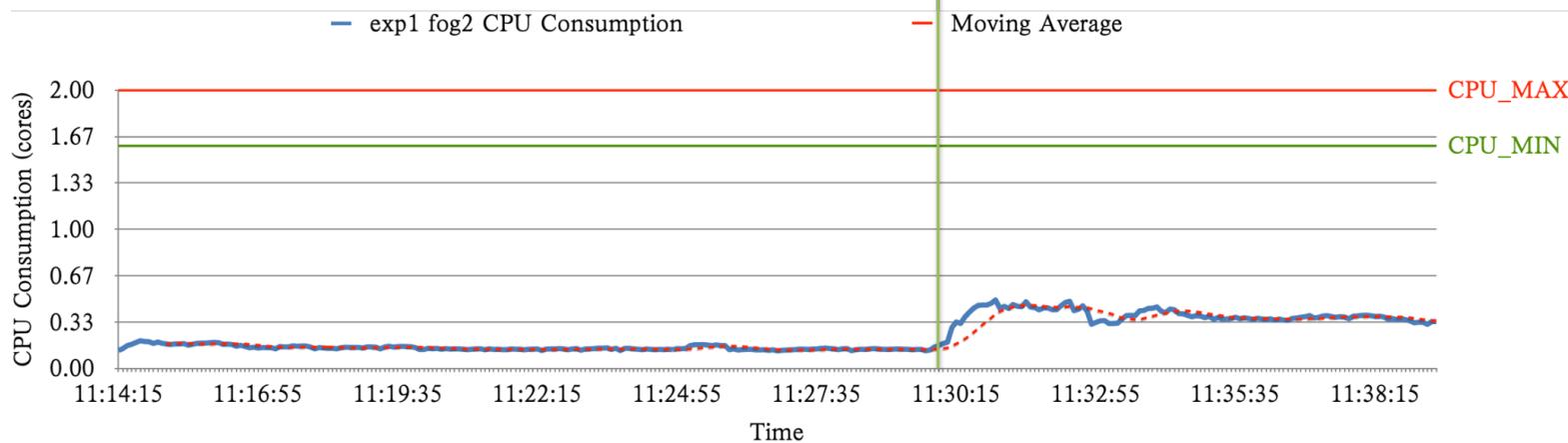
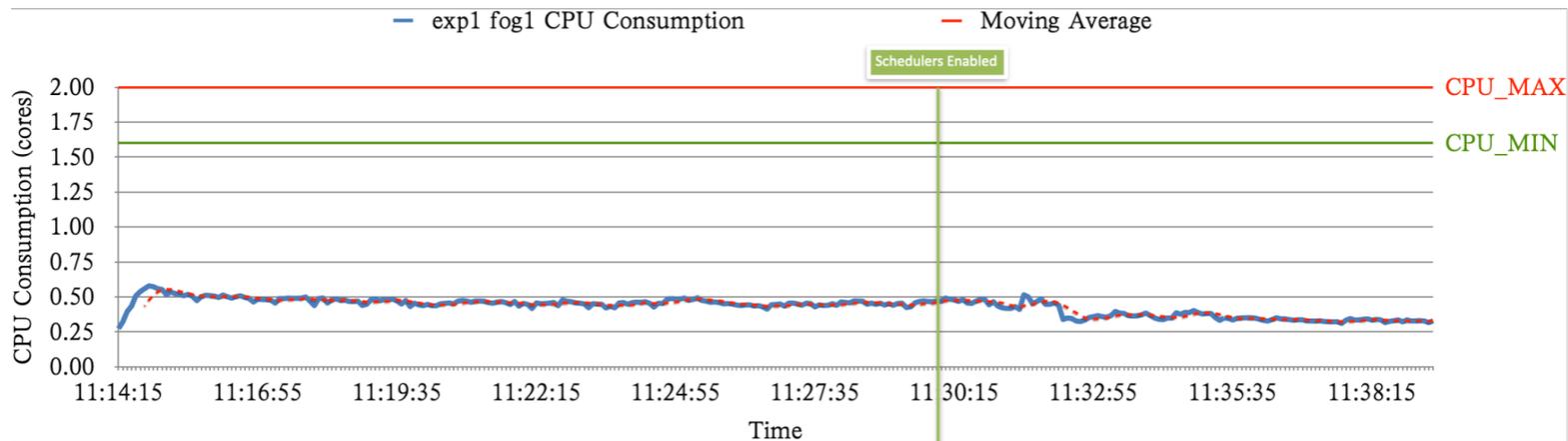
During the experiment's first phase, we let the whole topology run without any communication delay. Moving on to the second phase, we enabled communication delays between clusters, so response time towards final user's increased as well. Finally, transitioning to the final phase of the experiment, we enabled schedulers to all of the clusters. In this particular phase we can notice that the CPU Consumption significantly increased. This happened, due to the fact that schedulers realized that there are CPU cycles which are not in use, so all of them started initializing new services (deployments) in their host Clusters. Because of this, we can notice that the response time decreased as well in all of the three user groups, because now services are deployed physically closer to the source of the requests (less communication delay) and they don't have to be forwarded to a remote cluster (Fog Clusters). Furthermore, someone can notice that schedulers in every case are trying to keep CPU utilization between CPU_MAX and CPU_MIN.

The next table is showing the deployments installed in the edge clusters, after schedulers were stabilized:

Edge1	Edge2	Edge3
frontend	frontend	frontend
apacheservice	apacheservice	apacheservice
productcatalogservice	productcatalogservice	productcatalogservice
cartservice	cartservice	cartservice
currencyservice	currencyservice	currencyservice
keyrockservice	keyrockservice	keyrockservice

Since the edge clusters are identical and they are receiving the same amount and type of requests, we can notice that schedulers deploy the same deployments to all of them.

Fog1 & Fog2:



Fog1 CPU Consumption Average (cores)	
Before Scheduler Activation	0.45
After Schedulers Stabilization	0.35
Total Reduction percentage	-22.2%

Fog2 CPU Consumption Average (cores)	
Before Scheduler Activation	0.15
After Scheduler Stabilization	0.35
Total Additional percentage	+133.3%

On these layers, we can spot two different behaviors of CPU Consumption for both of the clusters. In the Fog1 Cluster, CPU Consumption decreased 22.2% but for the Fog2 it increased 133.3%. In order to understand this behavior we have to observe CPU utilization before and after schedulers stabilization. Final value of CPU Consumption is the same for both of the clusters. The difference between the final ratio variations comes from the fact that they had different initial consumptions. By running the scheduler to the edge clusters, all of them end up having exactly the same amount of services, so they forwarded exactly the same amount

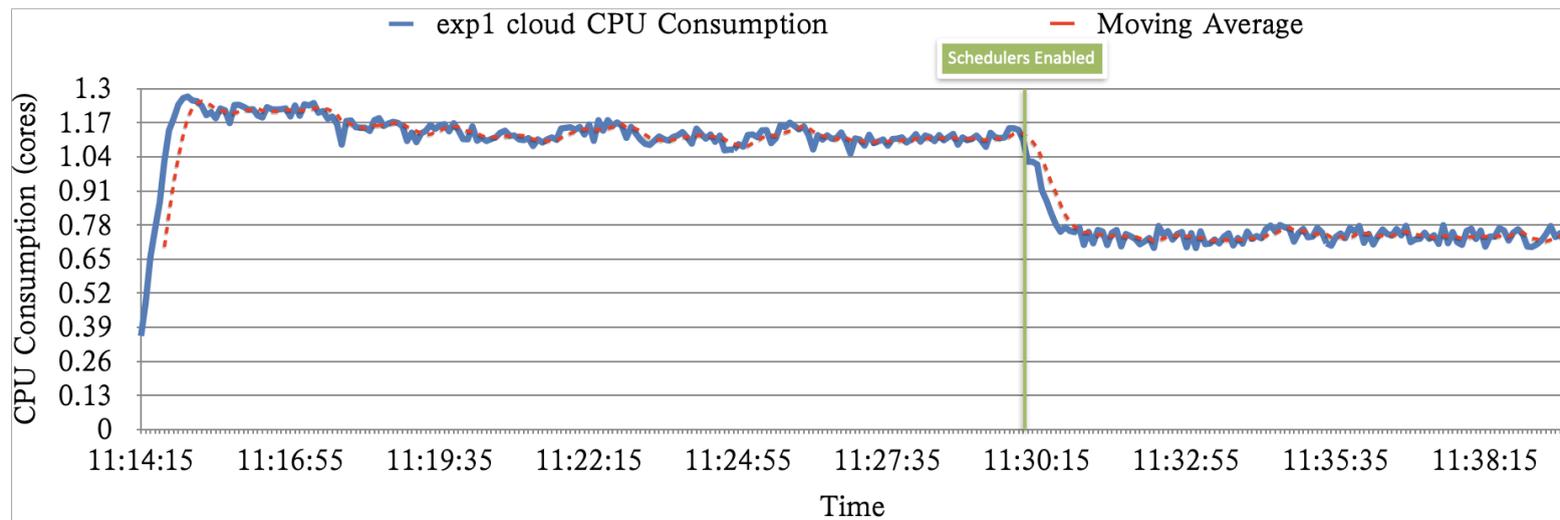
of requests to Fog Clusters. On the other hand, schedulers on the Fog Clusters took into consideration the incoming requests (which after scheduler stabilization are exactly the same) and Cluster's total CPU capacity (which is the same for both of the clusters). Having the same parameters as an input both of the schedulers decided to initialize exactly the same services. That's the reason for the Fog clusters to have exactly the same CPU Consumption. Finally, it's obvious that these clusters don't operate between the scheduler's CPU desired range. This happened because edge clusters are answering the majority of the requests and due to the fact that the total incoming traffic from the users is not enough they don't forward enough requests to the Fog clusters.

Services in Fog Clusters, after schedulers stabilized:

Fog1 (All Services)	Fog2 (All Services)
cartservice	cartservice
currencyservice	currencyservice
shippingservice	shippingservice
emailservice	emailservice
paymentservice	paymentservice
recommendationservice	recommendationservice
adservice	adservice
checkoutservice	checkoutservice
frontend	frontend
productcatalogservice	productcatalogservice
noderedservice	noderedservice
orionservice	orionservice
keyrockservice	keyrockservice
authzforceservice	authzforceservice
cygnusservice	cygnusservice
apacheservice	apacheservice
queryingsensorservice	queryingsensorservice
cometservice	cometservice
orionproxyservice	orionproxyservice
cygnusproxyservice	cygnusproxyservice
noderedproxyservice	noderedproxyservice
queryingsensorsproxyservice	queryingsensorsproxyservice
sthcometproxyservice	sthcometproxyservice

Since both of the Fog clusters had enough CPU capacity, they deployed all of the services. This means that for now and on there is any need to forward requests to Cloud cluster, since they already have all the services installed.

Cloud:



Cloud CPU Consumption Average (cores)	
Before Scheduler Activation	1.1
After Scheduler Stabilized	0.8
Total Reduction percentage	-27.2%

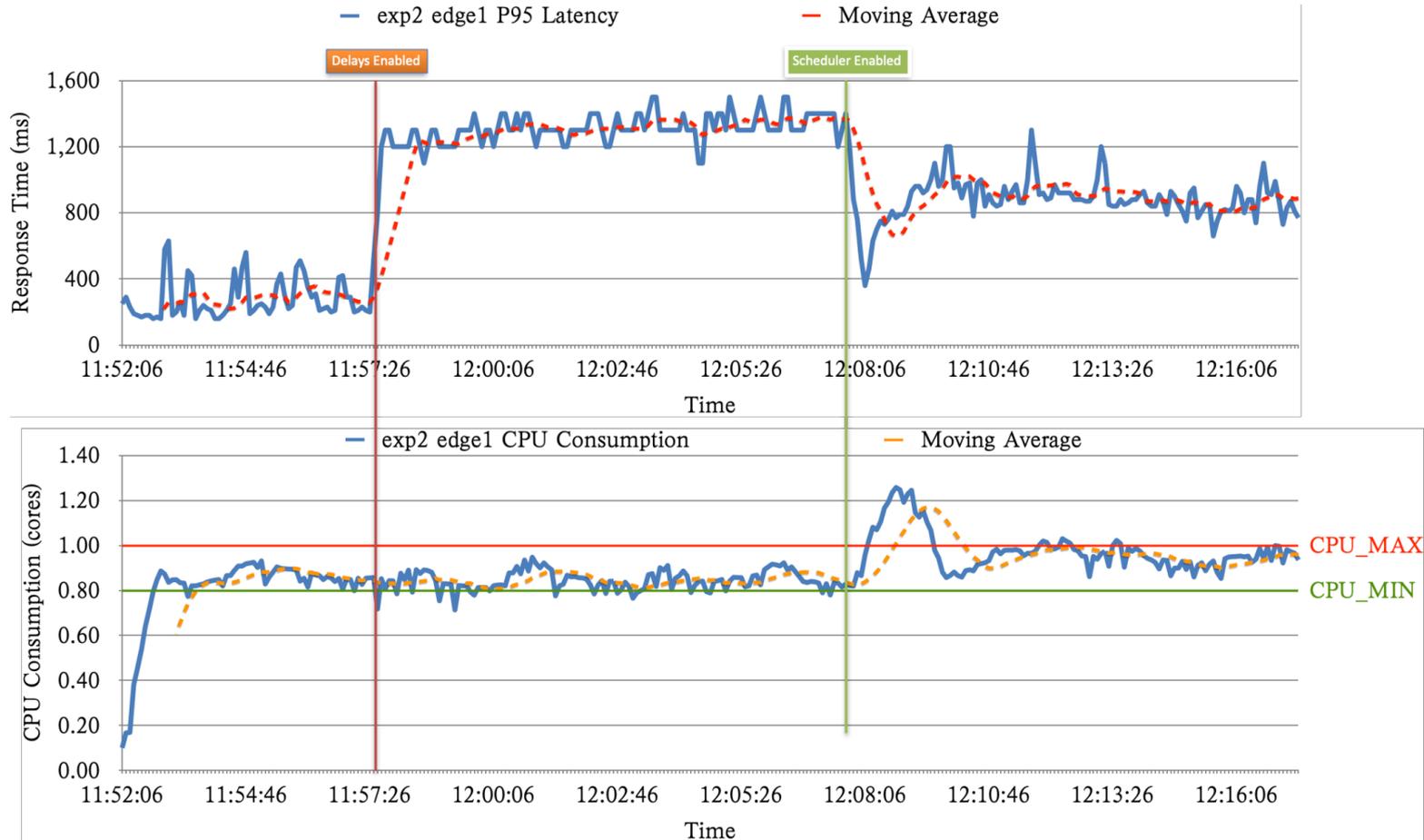
Cloud cluster witnessed a significant drop of CPU Consumption, from 1.1 cores to 0.8 cores. After the scheduler's deployment mainly edge clusters are answering user's requests, so the heavy lifting moved from Cloud towards edge and fog clusters.

5.1.2 Experiment 2

Users: 200 users/Edge Cluster

Results:

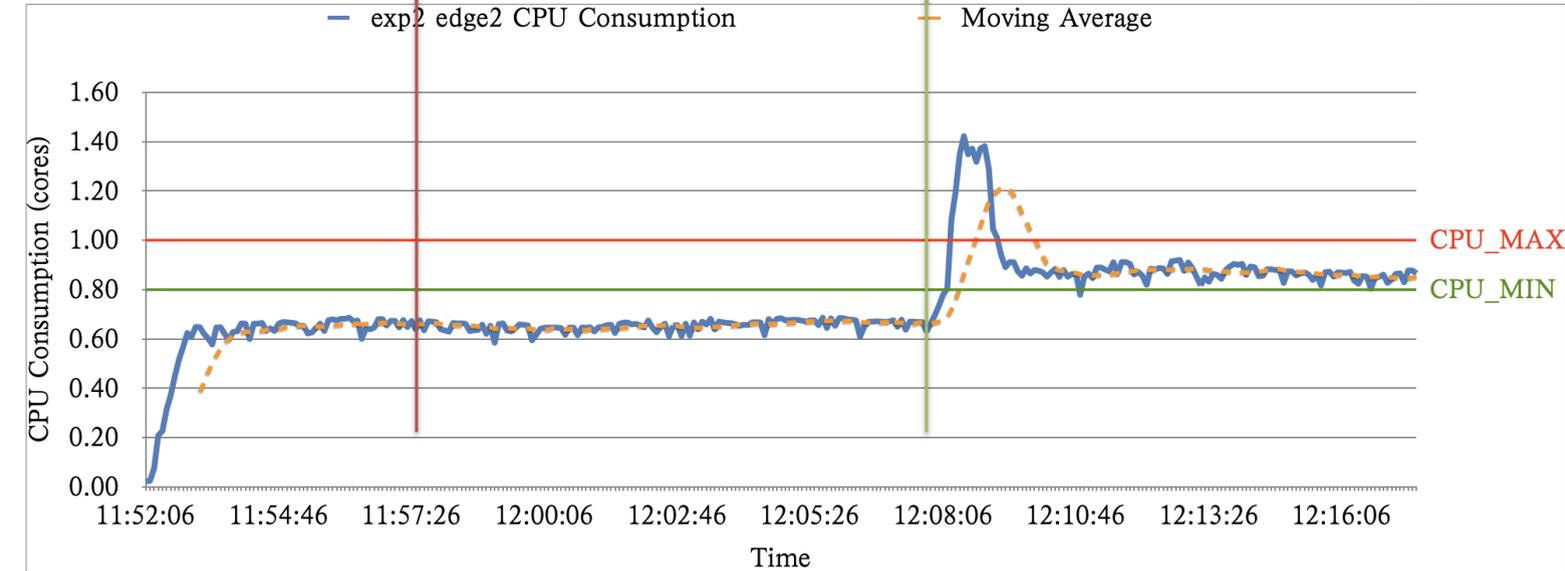
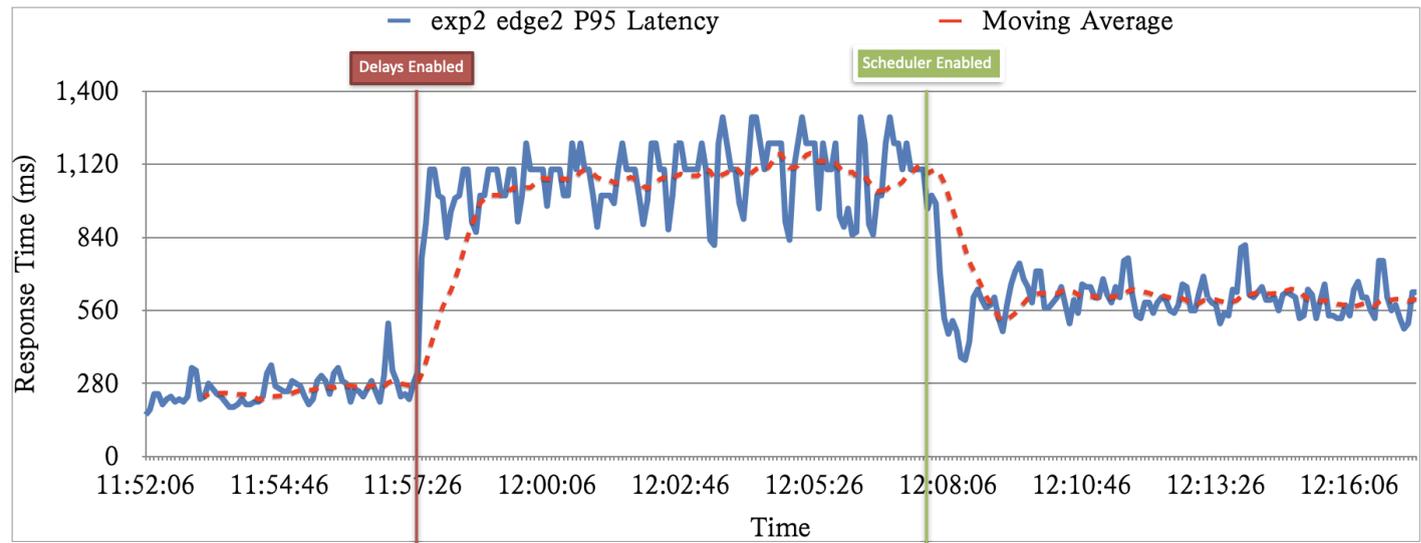
Edge1:



Response Time Average (ms)	
Before Delay Activation	300
After Delay Activation	1300
After Scheduler Stabilization	900
Total Reduction percentage	-30.7%

CPU Consumption Average (cores)	
Before Delay Activation	0.85
After Delay Activation	0.85
After Scheduler Stabilization	0.95
Total Additional percentage	+11.7%

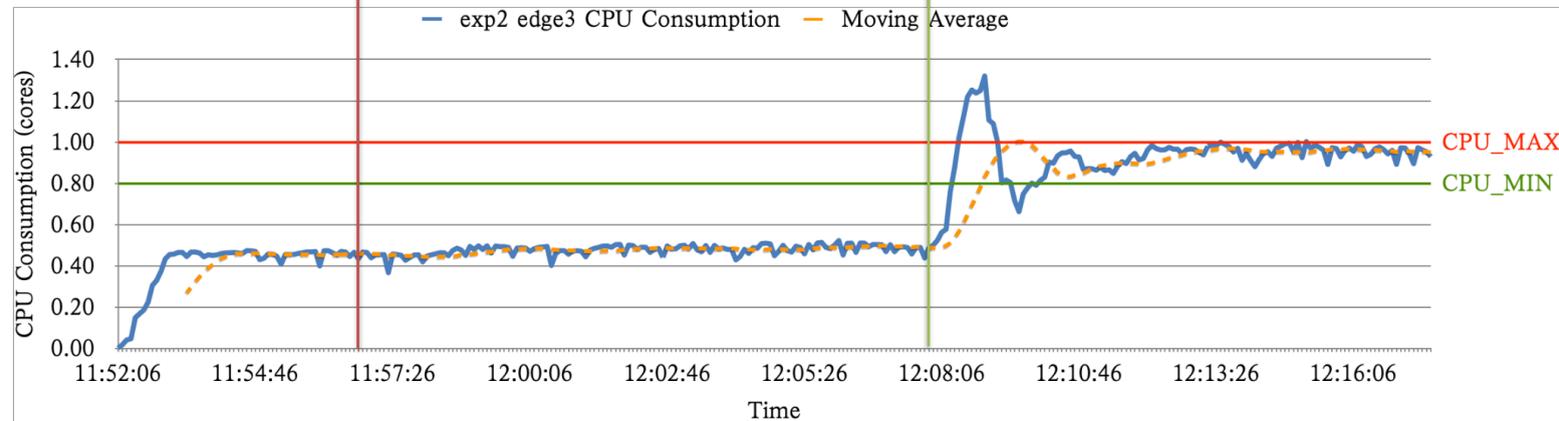
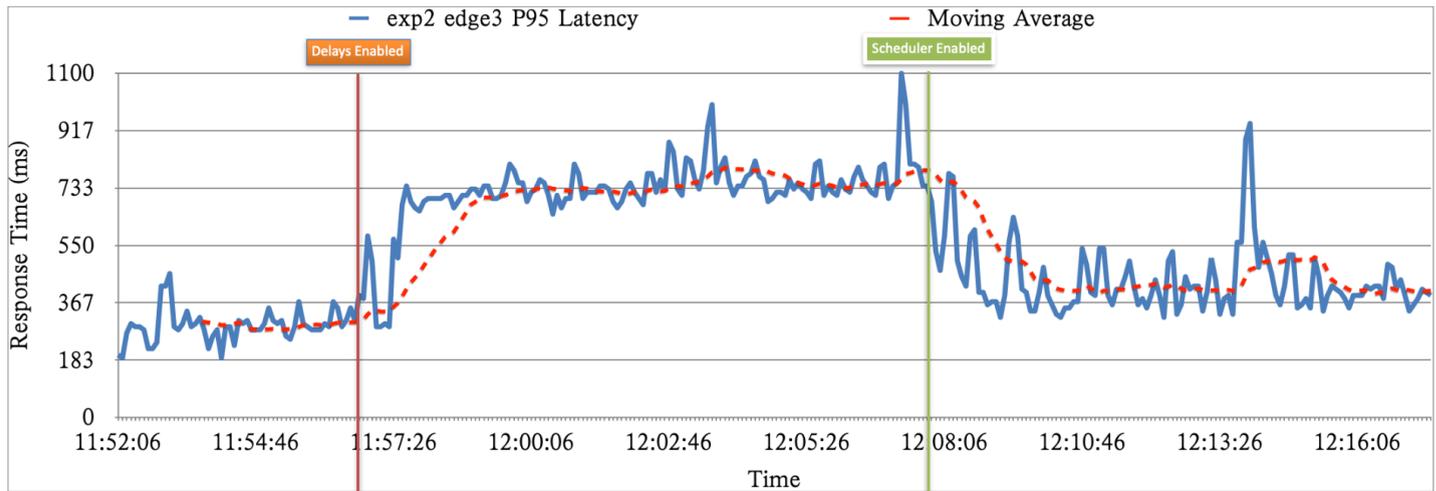
Edge2:



Response Time Average (ms)	
Before Delay Activation	200
After Delay Activation	1050
After Scheduler Stabilization	600
Total Reduction percentage	-42.8%

CPU Consumption Average (cores)	
Before Delay Activation	0.65
After Delay Activation	0.65
After Scheduler Stabilization	0.9
Total Additional percentage	+38.4%

Edge3:



Response Time Average (ms)	
Before Delay Activation	270
After Delay Activation	730
After Scheduler Stabilization	400
Total Reduction percentage	-45.2%

CPU Consumption Average (cores)	
Before Delay Activation	0.5
After Delay Activation	0.5
After Scheduler Stabilization	0.95
Total Additional percentage	+90%

In this experiment we can notice the same behavior as experiment 1. The difference is that now the response time deduction rate and CPU fluctuation rates are both less. This is a result of the user's request increment (double). The total CPU power required to answer these requests is more, but CPU capacity stays the same for all of the topology. Thus, edge clusters have to deploy less services (which this time are using more CPU) in order to reach schedulers desired CPU utilization range and forward more requests towards the upper layer

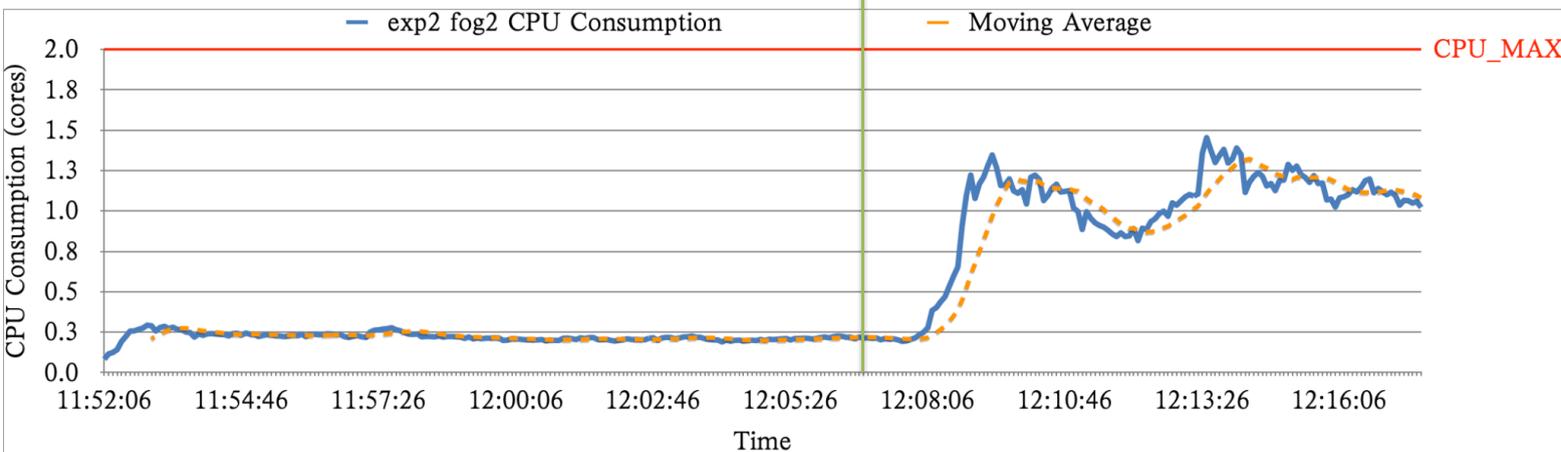
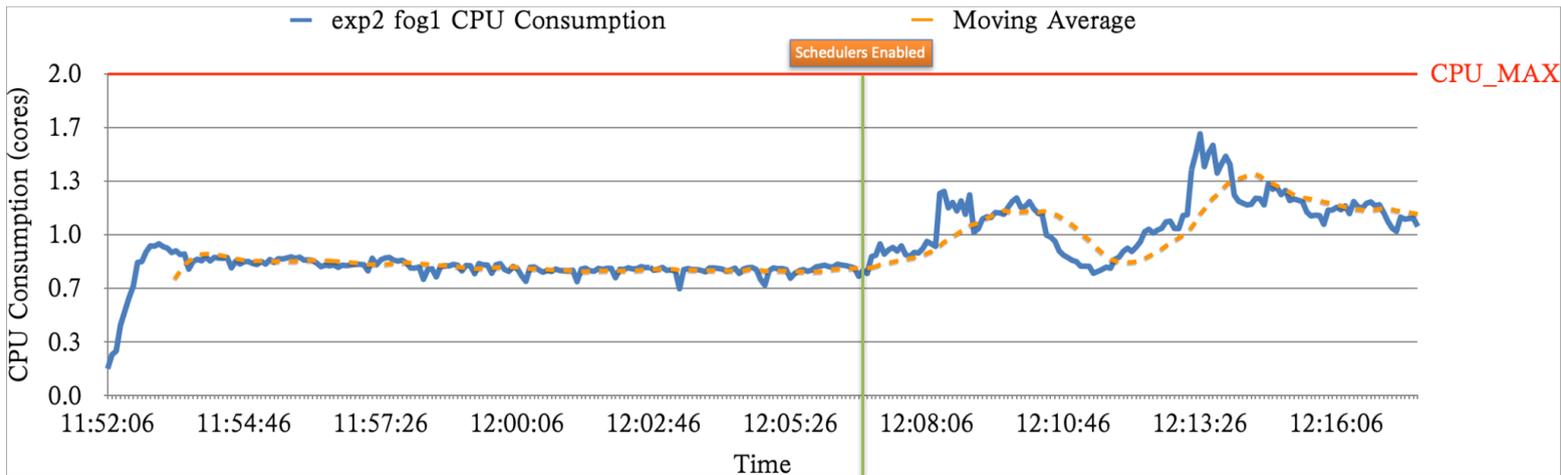
clusters. The “penalty” is the increasement of the response time; More requests are being forwarded to Fog clusters, more communication delay added to the final responses. So response time is still decreased compared to the default placement but not at the same level as the first experiment.

The following table is showing the services installed after schedulers stabilization in the edge clusters:

Edge1	Edge2	Edge3
frontend	frontend	frontend
apacheservice	apacheservice	apacheservice
currencyservice	currencyservice	currencyservice
keyrockservice (5.6%)	keyrockservice (6.4%)	keyrockservice (4.4%)

We can notice that now edge clusters have less services installed compared to the previous experiment(1). This happened because now they are accepting more requests, so the apacheservice and the frontend services require more CPU. Since they are receiving the highest number of requests, schedulers are prioritizing them and choose only to deploy only currencyservice so the cluster can allocate more CPU for the main two services. keyrockservice was initialized as well for all of the three edge clusters, but it is answering only 5.6%, 6.4% and 4.4% of its incoming requests and it is forwarding the rest to the fog clusters.

Fog1 & Fog2:



Fog1 CPU Consumption Average (cores)	
Before Scheduler Activation	0.8
After Schedulers Stabilization	1.2
Total Reduction percentage	+50%

Fog2 CPU Consumption Average (cores)	
Before Scheduler Activation	0.3
After Scheduler Stabilization	1.2
Total Additional percentage	+300%

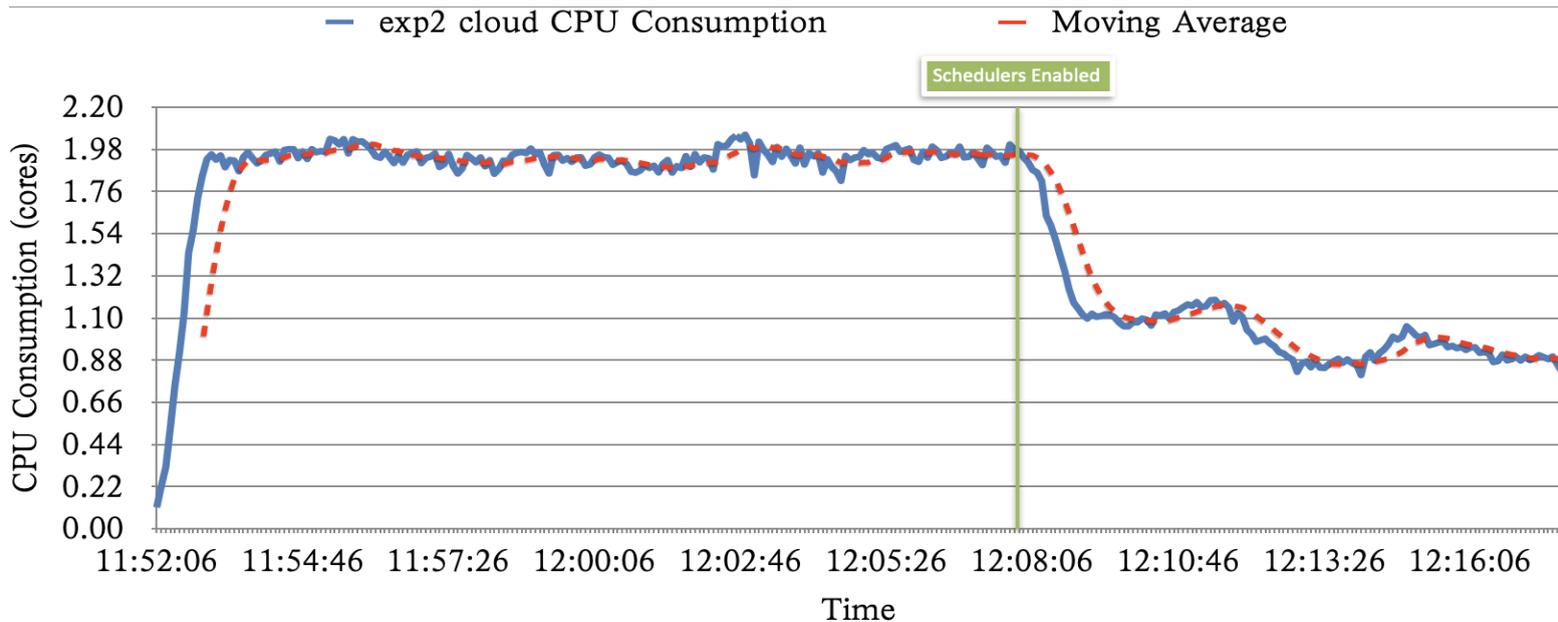
On these layers, again the overall behavior is the same as experiment 1 Fog Cluster's. The difference is spotted to the fact that the final CPU Consumption is more compared to the previous experiment, because now the incoming requests from the Edge Clusters are more.

Table with services for Fog clusters after schedulers stabilization:

Fog1	Fog2
All Services	All Services

Once again, we can notice that all the services were initialized in the Fog clusters, because there was plenty of CPU capacity unexploited.

Cloud:



Cloud CPU Consumption Average (cores)	
Before Scheduler Activation	2
After Scheduler Stabilized	0.85
Total Reduction percentage	-57.5%

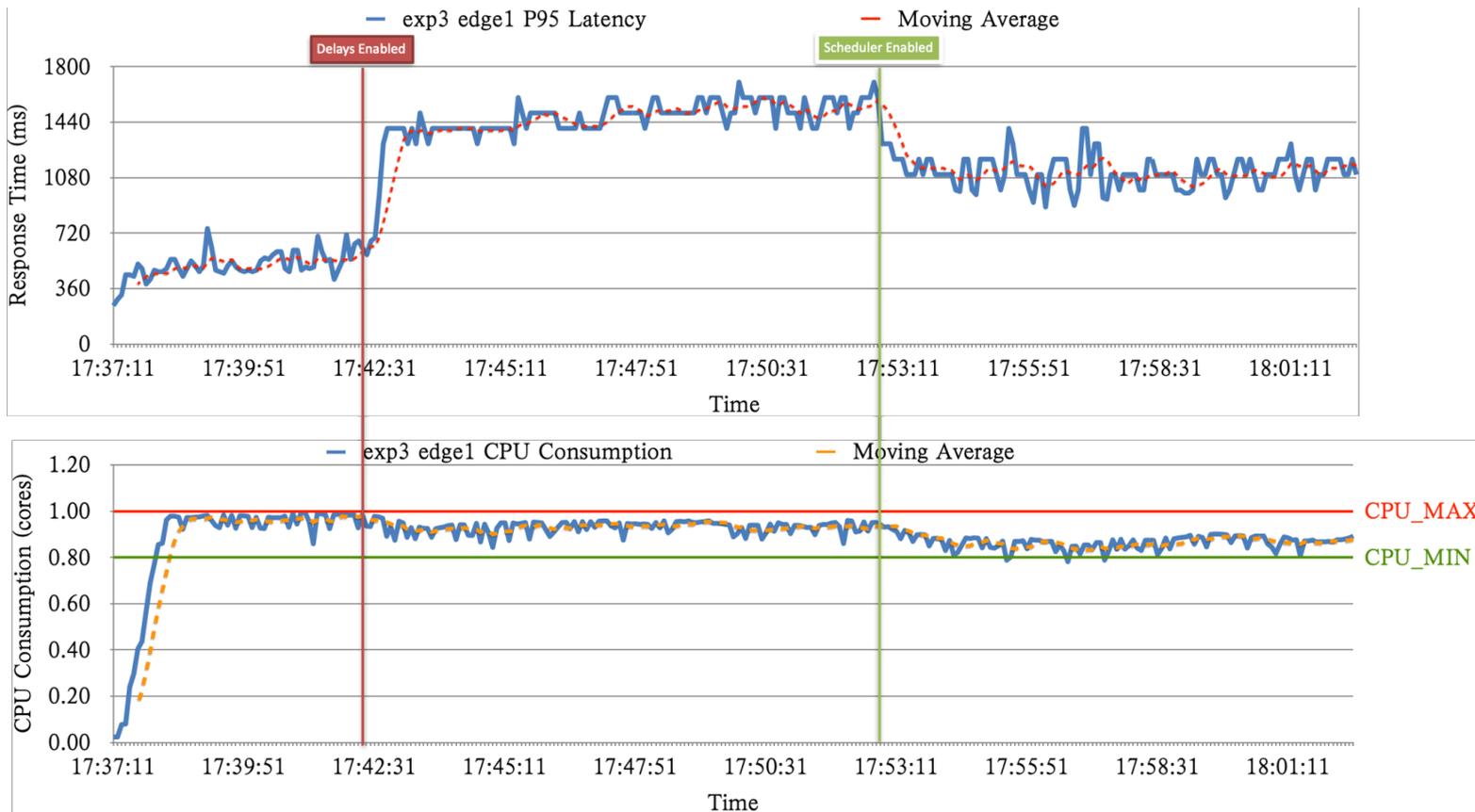
Again, CPU consumption of Cloud Cluster decreased. This time the overall ratio of this decrement is more (57.5) compared to the first experiment (27.2), but the final CPU usage is very close (0.8 and 0.85 cores respectfully)) This happened because Fog layer Clusters were able to absorb the majority of the workload and not forward requests to cloud cluster. Default placement was forwarding more requests to cloud compared to experiment 1 default placement because of were increased, but schedulers running on the Fog and Edge layer were able to move the answering of the workload from the Cloud to the lower layers clusters.

5.1.3 Experiment 3

Users: 300 users/Edge Cluster

Results:

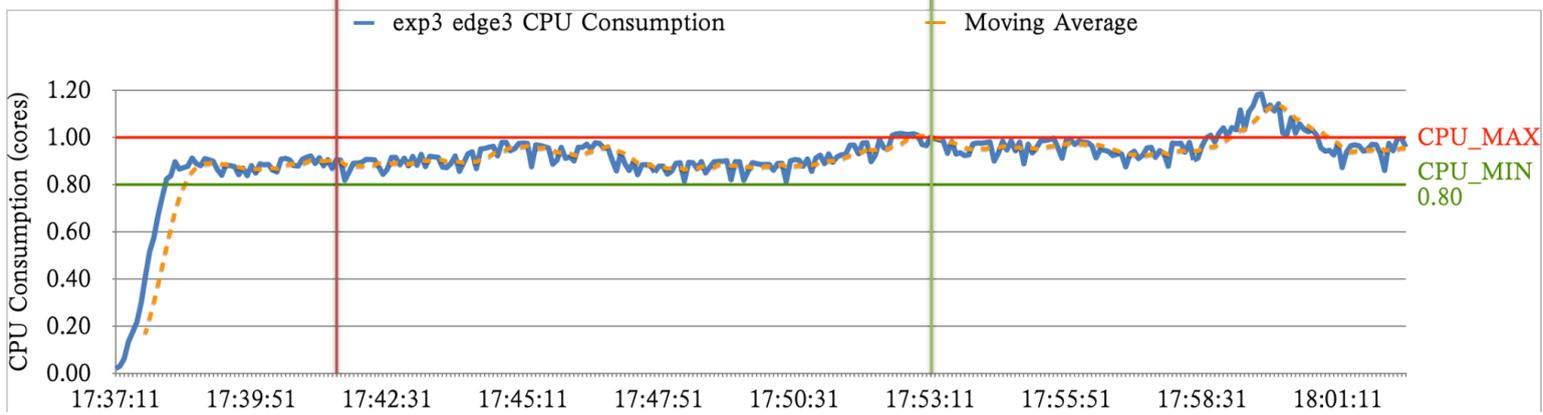
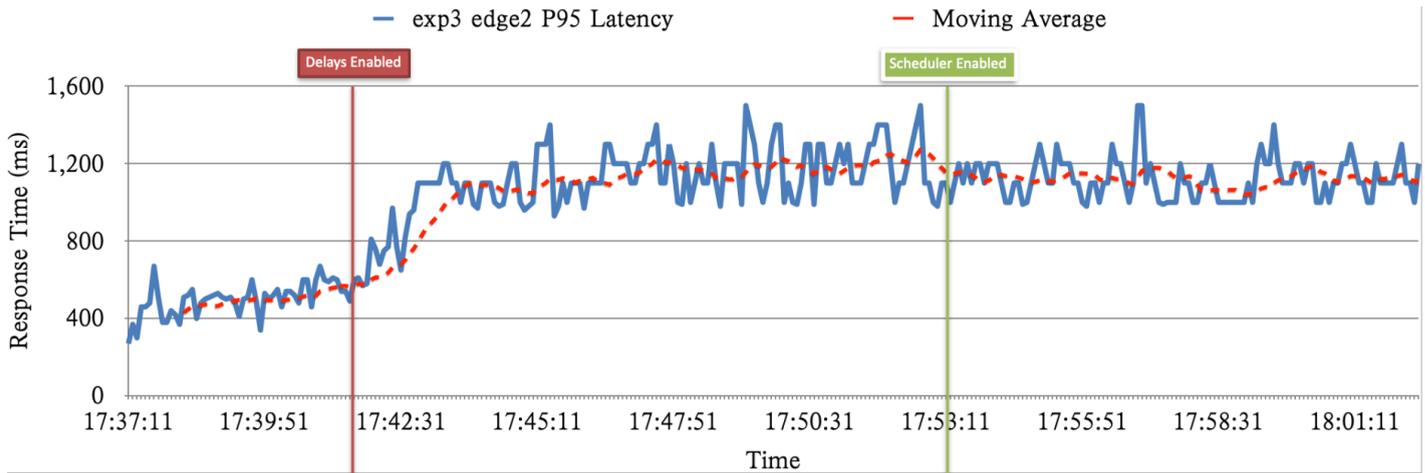
Edge1:



Response Time Average (ms)	
Before Delay Activation	500
After Delay Activation	1460
After Scheduler Stabilization	1100
Total Reduction percentage	-24.6%

CPU Consumption Average (cores)	
Before Delay Activation	0.95
After Delay Activation	0.95
After Scheduler Stabilization	0.9
Total Additional percentage	-5.2%

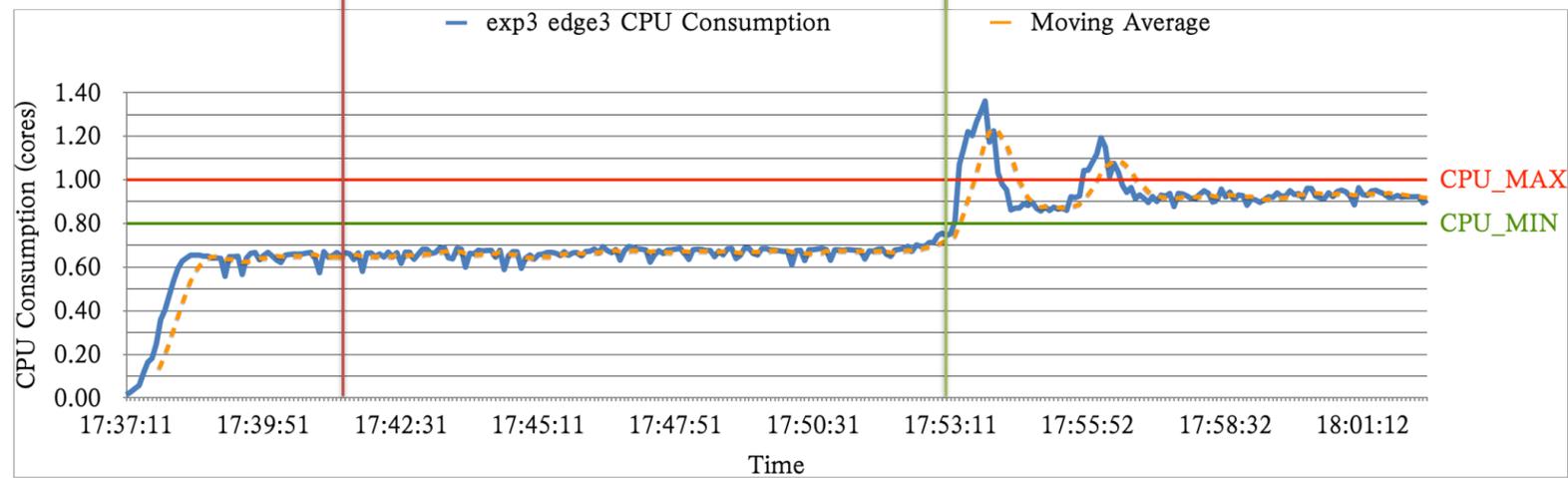
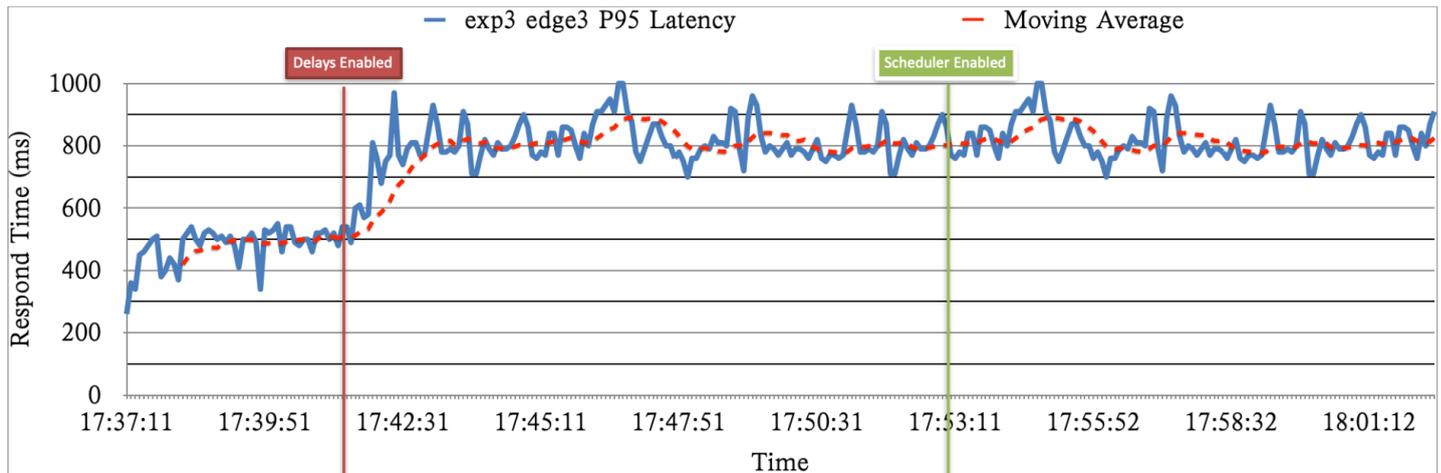
Edge2:



Response Time Average (ms)	
Before Delay Activation	500
After Delay Activation	1150
After Scheduler Stabilization	1100
Total Reduction percentage	-4.3%

CPU Consumption Average (cores)	
Before Delay Activation	0.9
After Delay Activation	0.9
After Scheduler Stabilization	0.95
Total Additional percentage	+5.5%

Edge3:



Response Time Average (ms)	
Before Delay Activation	500
After Delay Activation	800
After Scheduler Stabilization	800
Total Reduction percentage	-0%

CPU Consumption Average (cores)	
Before Delay Activation	0.7
After Delay Activation	0.7
After Scheduler Stabilization	0.9
Total Additional percentage	+28.5%

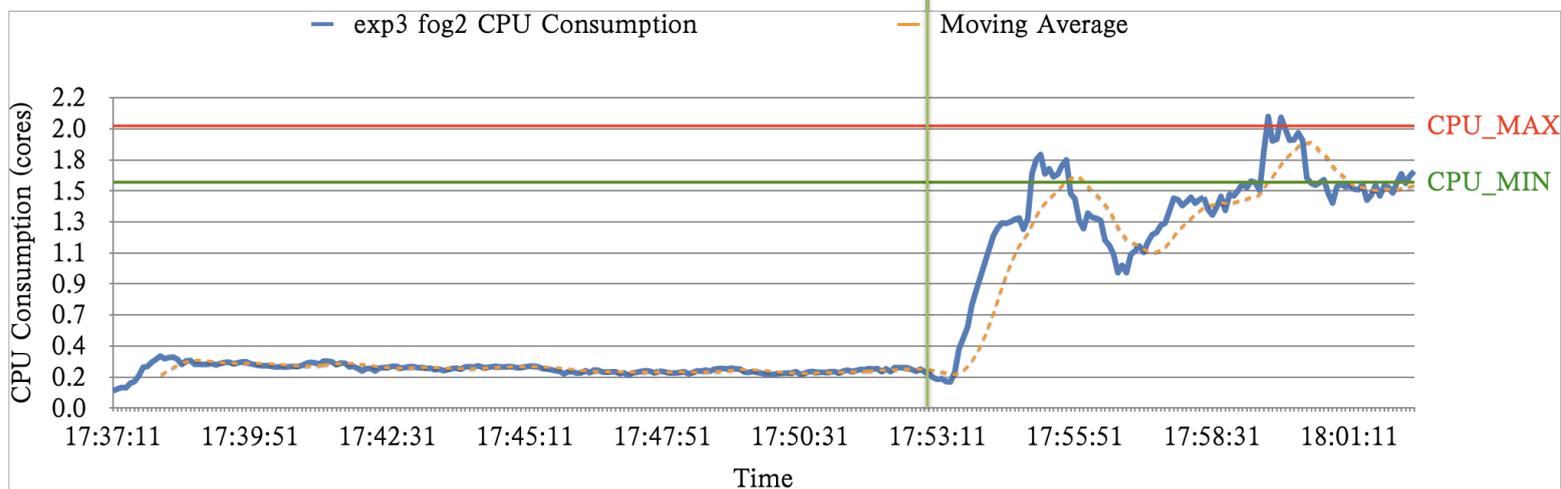
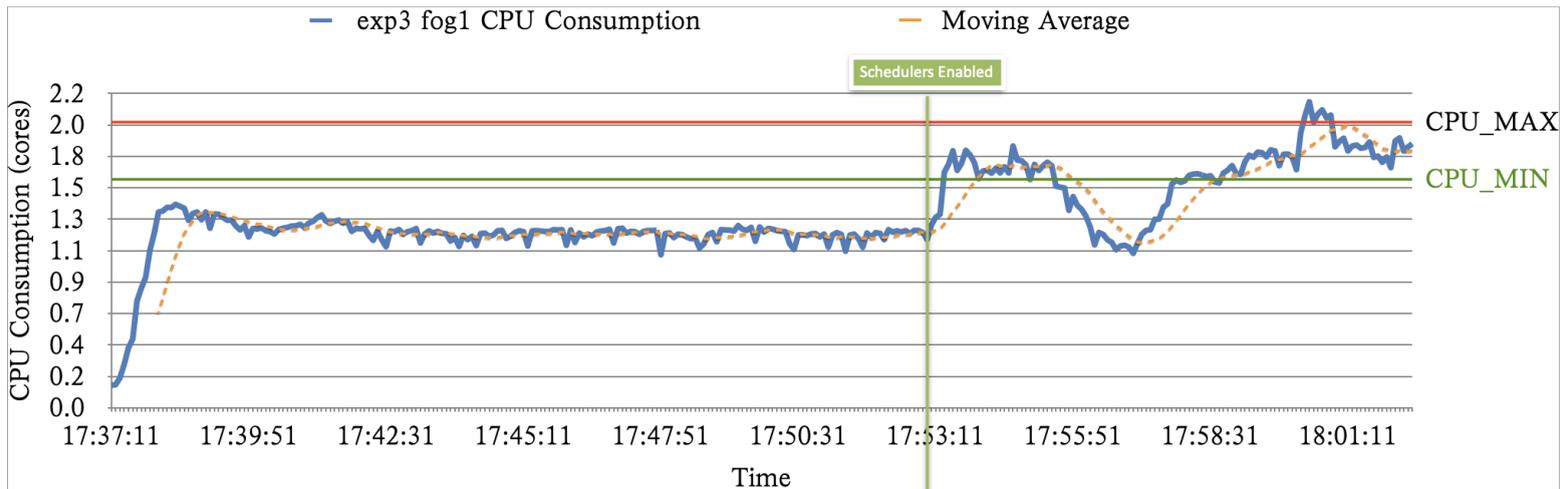
In this experiment we can notice that for clusters edge1 and edge2 response time almost remained the same after the scheduler's deployment. For cluster edge1 we can observe a decrease of 24.6 %.

Table of services after scheduler stabilization:

Edge1	Edge2	Edge3
frontend	frontend	frontend
apacheservice	apacheservice	apacheservice

This time, due to the increased traffic, all the schedulers decided to keep the bare minimum services in the clusters -the services who are the entry point for the requests-. There wasn't any CPU capacity left for any other service.

Fog1 & Fog2:



Fog1 CPU Consumption Average (cores)	
Before Scheduler Activation	1.2
After Scheduler Stabilization	1.8
Total Reduction percentage	50%

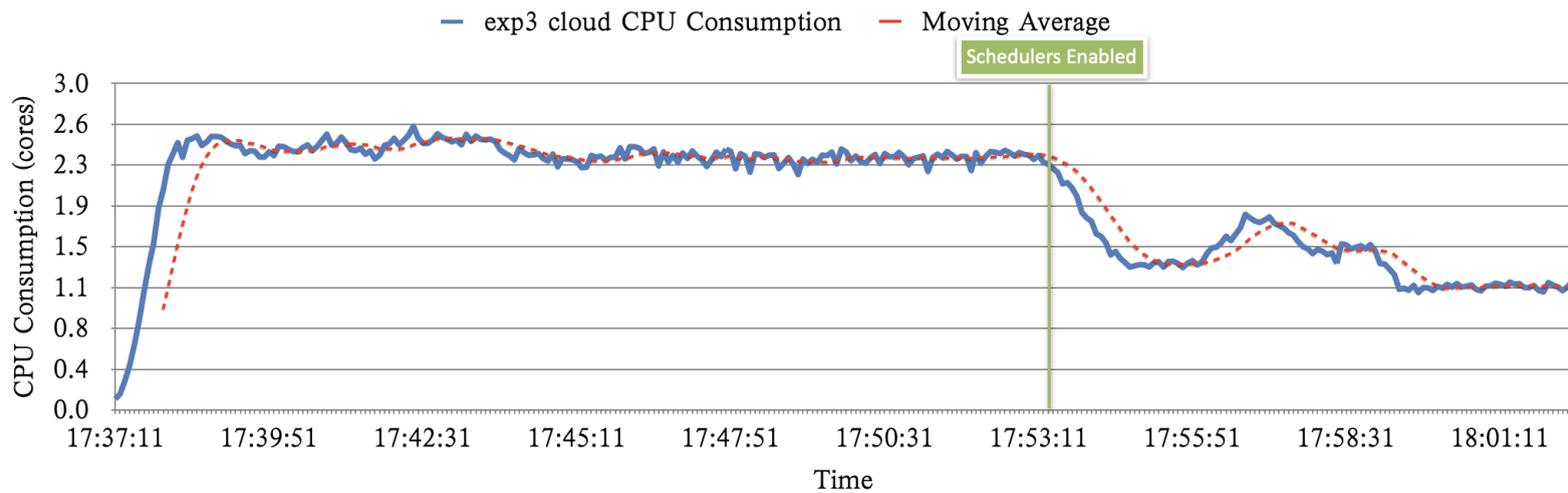
Fog2 CPU Consumption Average (cores)	
Before Scheduler Activation	0.3
After Scheduler Stabilization	1.7
Total Additional percentage	+466.6%

Both of the Fog clusters experienced a CPU usage increment. This is a result from the fact that now Edge clusters are forwarding even more requests to the Fog layer. Another interesting point is the fact that both of the clusters are operating between the scheduler's desired CPU usage range and as we can see from the next graph (Cloud cluster graph) they are forwarding more requests to the cloud cluster, because their compute capacity is maxed out.

Fog1	Fog2
cartservice	cartservice
currencyservice	currencyservice
shippingservice	shippingservice
emailservice	emailservice
paymentservice	paymentservice
recommendationservice	recommendationservice
adservice	adservice
checkoutservice	checkoutservice
productcatalogservice	productcatalogservice
noderedservice	noderedservice
orionservice	orionservice
keyrockservice	keyrockservice
authzforceservice	authzforceservice
cygnusservice	cygnusservice
cometservice	cometservice
orionproxyservice	orionproxyservice
cygnusproxyservice	cygnusproxyservice

On this occasion, the Fog clusters faced a limitation in their CPU capacity, which prevented them from initiating all of the services. As a result incoming requests for these services were redirected to the Cloud cluster.

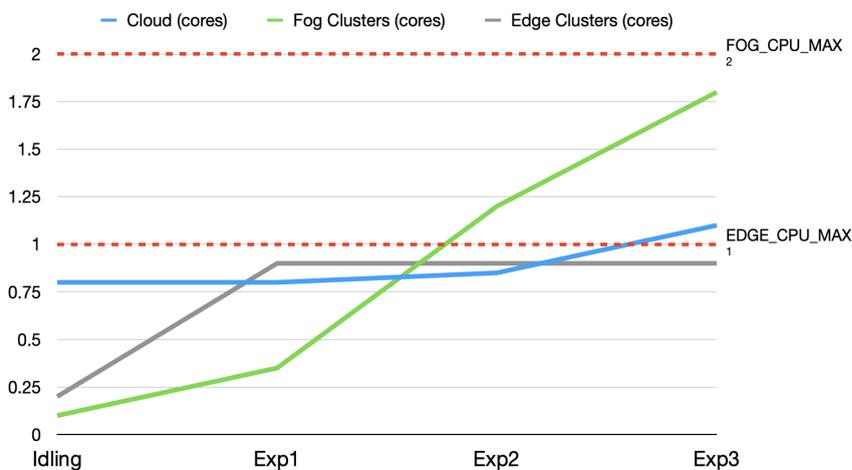
Cloud:



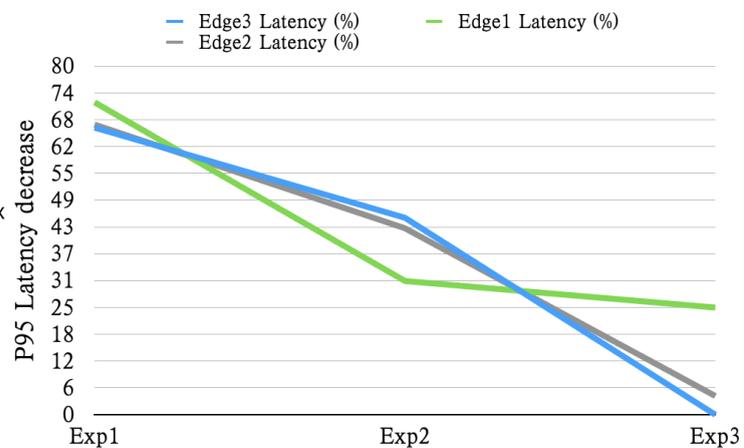
Cloud CPU Consumption Average (cores)	
Before Scheduler Activation	2.5
After Scheduler Stabilized	1.1
Total Reduction percentage	-56%

This time, we can observe again an decrease in the CPU utilization, but now the CPU consumption after scheduler stabilization is 1.1 cores. This means we have an increment compared to the previous experiment (0.85 cores) , because Fog layer started to forward more traffic to the cloud layer.

5.1.4 Experiments 1-2-3 Overview



(Graph1: CPU/experiment/cluster)



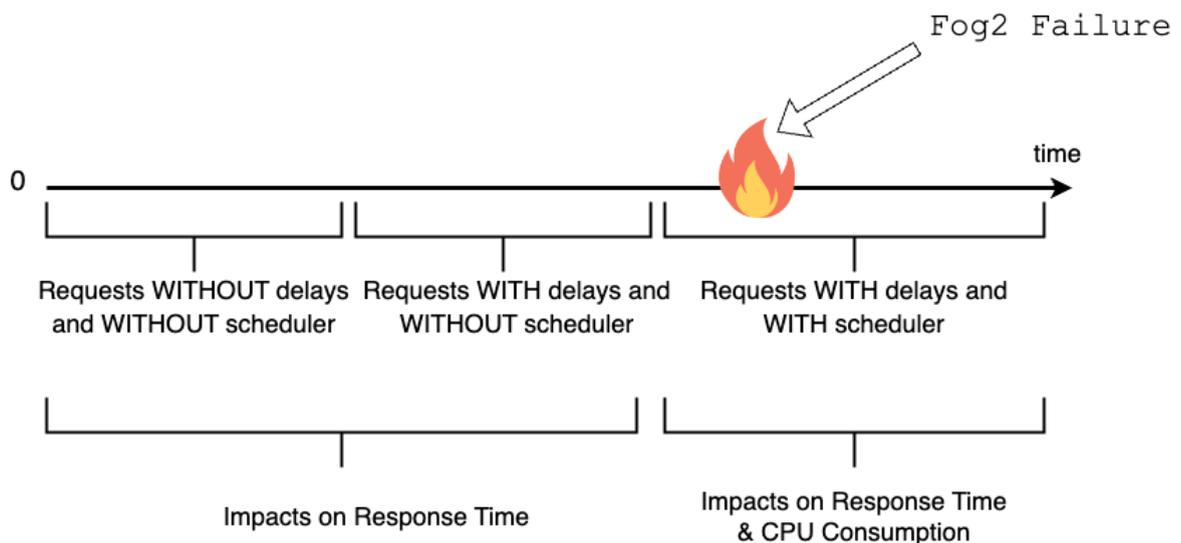
(Graph 2: CPU/experiment/cluster)

In the above graph (1), we can observe the CPU utilization across the three experiments. As we move from Exp1 to Exp3, the number of requests increases. Consequently, the topology needs to allocate more CPU power to accommodate this surge. Initially, from the idle state until the first experiment, the Edge Clusters bear with the answering of workload. However, in Exp#1, the Fog Clusters reach their peak CPU capacity of 1 Core. As a result, in Exp#2 and Exp#3, since the Edge Clusters are already operating at full capacity, the Fog Clusters take on the additional requests. Notably, during Exp3, the Cloud Cluster also starts responding to some requests, as the Fog Clusters approach their maximum computational capability.

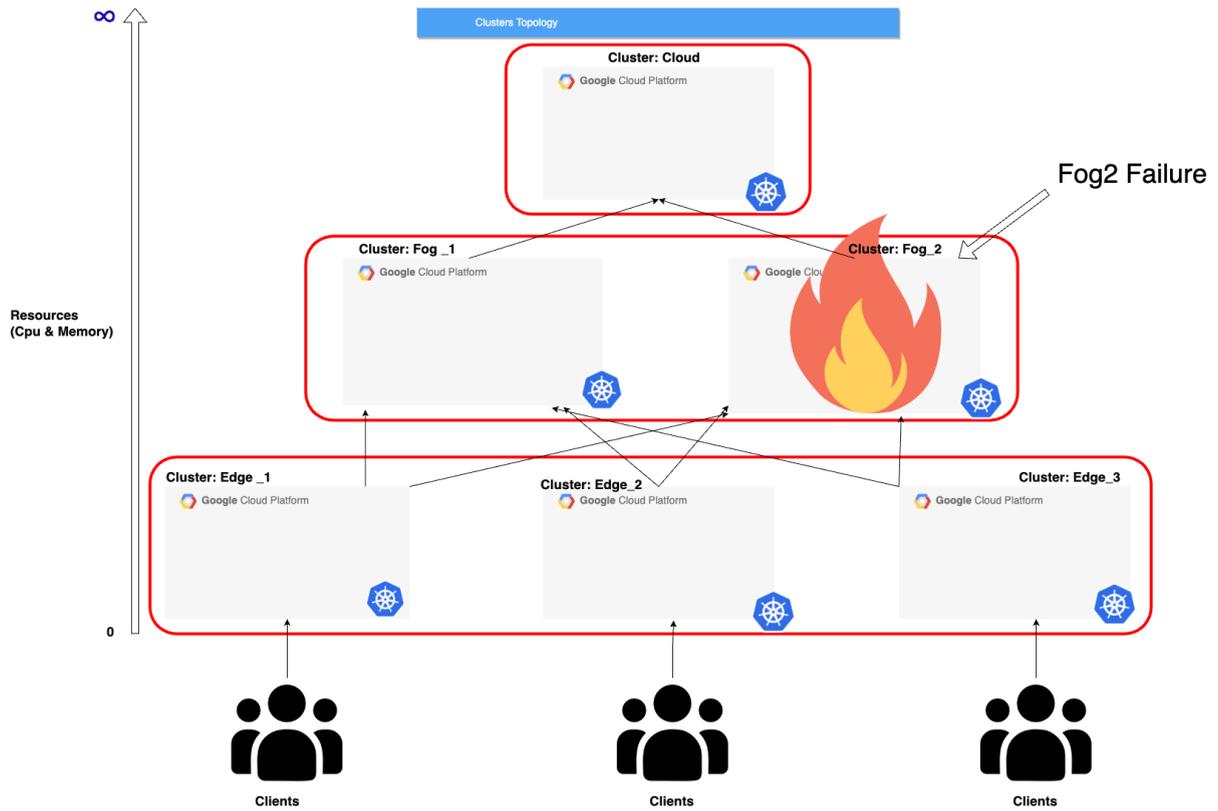
In the second graph (2), we can observe a decline in the delay deduction as a result of schedulers enabling. While the initial experiment showed a more significant decrease, as we progress to the second and third experiments, the average delay reduction becomes less pronounced, eventually approaching zero for Edge1 and Edge2 clusters. As the number of requests increases, there is no notable improvement because the Edge Clusters are already fully utilized. The only potential improvement that can be made is the replacement of less popular services with more popular ones. Nevertheless, in the first experiment, there was a significant amount of untapped CPU power available in the Edge Clusters, which prompted the schedulers to install additional services to utilize it effectively.

5.2 Experiment 4

In this upcoming experiment, we are going to test a challenging scenario. Our intention is to closely mimic a catastrophic situation, presenting an opportunity to test our systems under stress. Initially, the experiment will start just like its three predecessors, following a normal course. However, after the schedulers have been enabled, we will purposefully power off cluster Fog2. By taking this action, we aim to simulate a scenario similar to a real-life catastrophe, such as a physical disaster or a power outage. This will provide us with valuable insights into the resilience and robustness of the whole architecture and how the schedulers are going to react to this catastrophe.



(Time frame for experiment 4)



(Cluster Fog2 Failure Scenario)

In these experiment the next parameters were used:

Users: 200 users/Cluster

Delays: 20ms from the Edge to Fog Clusters, 10ms from the Fog to Cloud Clusters

Workload: 50% of the users are sending requests to the Eshop App and 50% to IXEN.

Due to the fact that the behavior is the same for all the Edge Clusters, we will present and analyze the results for the Edg2 Cluster.

Edge2:



It can be observed that up until the point of Fog2's failure, the architecture exhibited similar behavior to that of Experiment 2. Upon enabling delays, the response time increased, and after enabling the scheduler, the response time towards end users decreased while CPU consumption increased. However, when the Fog2 Cluster began to experience failures, it became evident that requests were no longer being successfully answered (as depicted in graph 3). Simultaneously, the CPU utilization started to decline. This occurred because services in the edge2 cluster were not receiving successful responses, resulting in a lack of CPU usage for processing those responses. After a few seconds, the scheduler recognized the low CPU utilization and attempted to deploy more services, causing the CPU utilization to approach its maximum value. Concurrently, the scheduler noticed that requests from Fog2

were not being answered and initiated the rerouting of requests to the Fog1 Cluster, which was successfully handling and responding to requests. As a result, some requests began to be answered properly.

After some time, all requests were rerouted to Fog1, and they were once again successfully answered. Interestingly, it should be noted that the response time, after the rerouting took place, was slightly higher compared to the period before the failure of Fog2, something that will be analyzed further.

<i>Edge2 CPU Consumption Average (cores)</i>	
Before Delay Activation	0.55
After Delay Activation	0.55
After Scheduler Activation	0.9
During Re-Routing	Peak 1 (MAX)
After Re-Routing	0.9
Total Additional Percentage	0 %

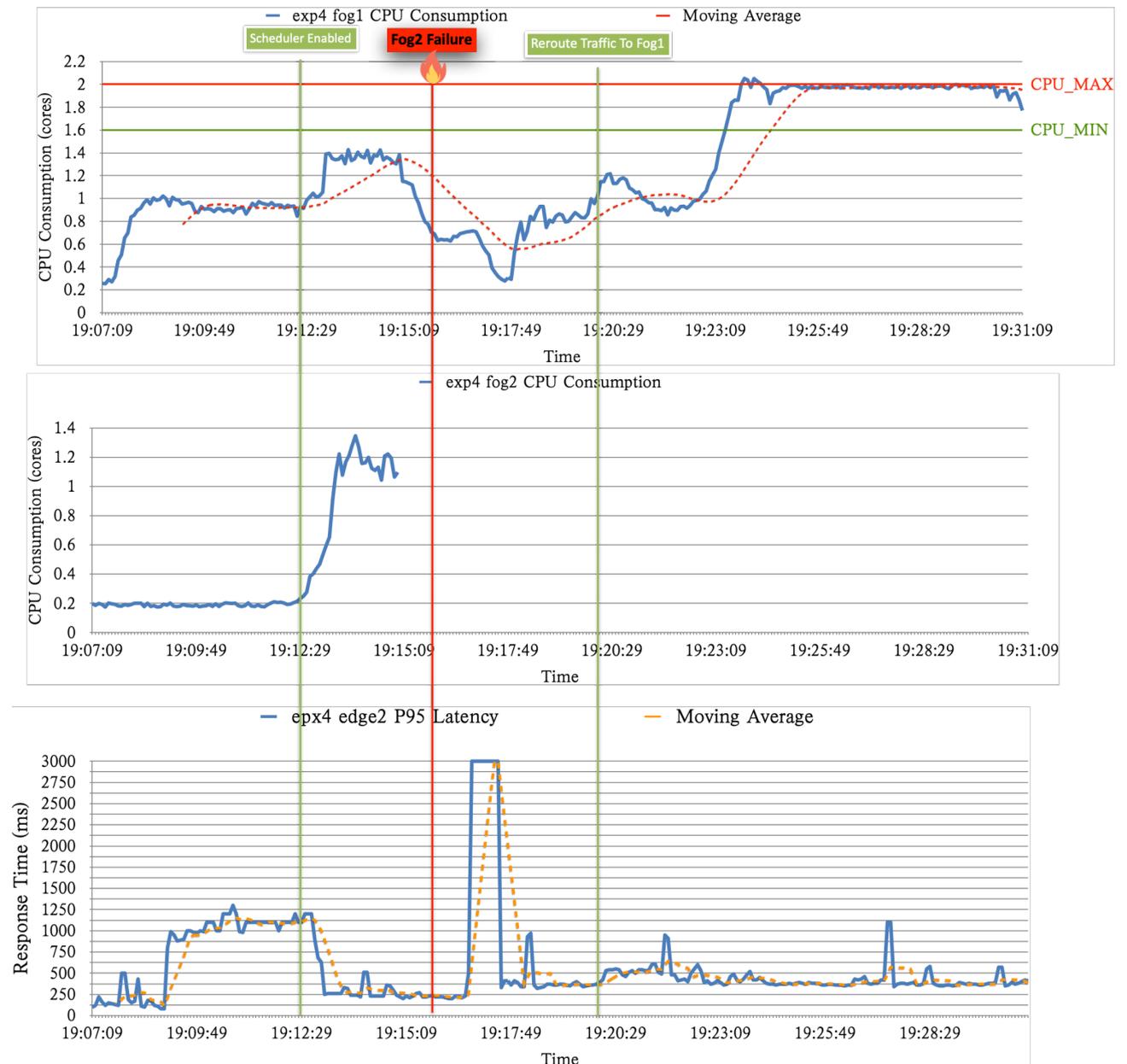
<i>Response Time Average (ms)</i>	
Before Delay Activation	200
After Delay Activation	1120
After Scheduler Activation	250
During Re-Routing	3200 (time-out)
After Re-Routing	380
Total Additional Percentage	+52 %

In the next table we can see the services of initialized in the edge clusters after scheduler stabilization:

Edge1	Edge2	Edge3
frontend	frontend	frontend
apacheservice	apacheservice	apacheservice
currencyservice	currencyservice	currencyservice
keyrockservice (6.2%)	keyrockservice (5.2%)	keyrockservice (4.3%)

Upon observation, it becomes apparent that the services in this experiment are exactly the same as those of experiment 2. This similarity happened, because both of the experiments received an equal quantity and type of requests. Despite the outage experienced by Fog2, its impact on the final services of the edge clusters was insignificant. The only consequence of the outage for edge clusters is that they are currently redirecting their requests solely to fog1, as fog2 remains inactive.

Fog1 & Fog2:



Initially, we can observe that during the failure, the CPU utilization in Fog1 Cluster begins to decrease. This occurs because the Edge Clusters reduce the number of requests they send. This is a consequence of the "chain requests." event. Since the Edge Clusters do not receive responses from Fog2 Cluster, they refrain from generating additional requests based on those successful responses. Subsequently, the scheduler in Fog1 Cluster detects the low CPU utilization and initiates the service of more instances. Eventually, the schedulers in the Edge Clusters successfully re-route all traffic from Fog2 to Fog1. As a result, the CPU utilization in Fog1 reaches its maximum capacity (as it now handles the requests originally intended for Fog2), and the response time for end users increases, because Fog1 schedulers started forwarded some request to Cluster Cloud, because Fog1 reached its maximum CPU capacity.

Fog1 CPU Consumption Average (cores)	
Before Schedulers Activation	0.9
After Scheduler Activation	1.4
After Re-Routing	2 (Max)
Total Additional Percentage	+42.8%

Response Time Average (ms)	
Before Schedulers Activation	1100
After Scheduler Activation	250
After Re-Routing	380
Total Additional Percentage	+52 %

The final result of cluster Fog2 Failure:

- Requests are being answered successfully, but final response time to the end users is increased(+52 %).
- Fog1 CPU Consumption, increased(+42.8%) and reached maximum value (2 cores).
- Because of this, some requests are re-routed, from Fog1 to Cloud, as we are going to see in the cloud cluster analysis.

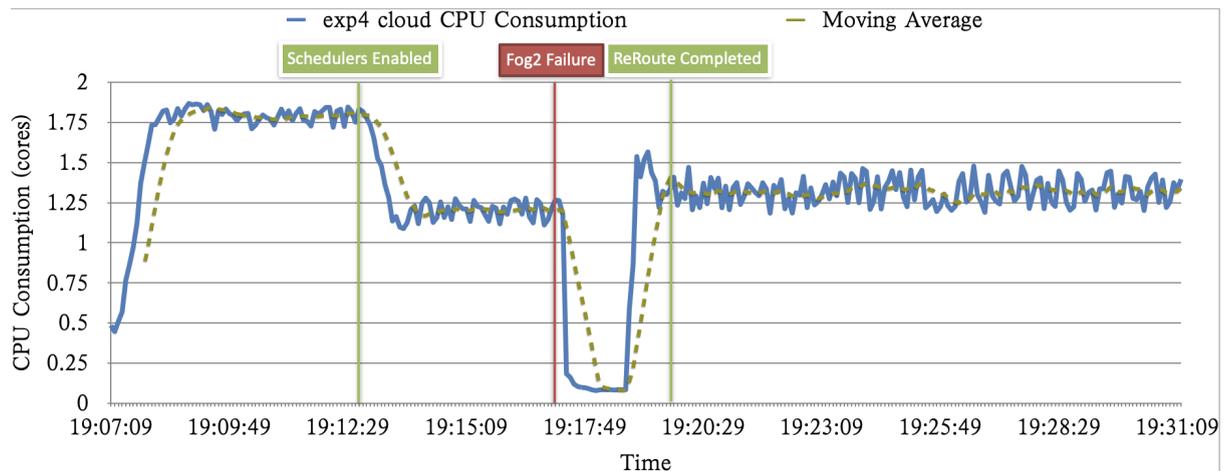
Services in fog1 after scheduler stabilization:¹

Fog1
cartservice
shippingservice
emailservice
paymentservice
recommendationservice
adservice
checkoutservice
productcatalogservice
orionservice
keyrockservice
authzforceservice
cygnuservice
orionproxyservice
cygnusproxyservice

When comparing the services of the fog1 cluster in this experiment to those of experiment 2, a noticeable difference emerges. In the current experiment, fog1 has initialized less services. As previously explained, this reduction occurred due to the fact that fog1 is now dealing with double the amount of requests it previously managed in experiment 2. This increase in workload on fog1 originates from the failure of the fog2 cluster.

¹ There are no services for fog2 cluster, due its failure.

Cloud:



We enabled the schedulers for all Clusters. As anticipated, there was a noticeable decrease in CPU consumption in the Cloud Cluster. Subsequently, the Fog2 Cluster experienced a failure, which resulted in a significant drop in CPU consumption in the Cloud Cluster. This decrease in CPU consumption is a consequence of both Fog2 ceasing to send traffic due to its failure and Fog1 also ceasing to send traffic. Fog1's traffic cessation occurs because the requests it used to send were primarily generated by incoming requests from the Edge Clusters. Since the Edge Clusters are unable to send proper requests due to the failure of Fog2 (as it does not respond), they also stop sending requests to Fog1. After a certain period of time, the schedulers in the Edge Clusters reroute all traffic to Fog1, causing Fog1 to resume sending requests to the Cloud Cluster. However, we can notice that now Cloud's cluster CPU consumption is increased (16.6%) This increase is due to Fog1's scheduler forwarding a portion of the traffic to the Cloud Cluster, as Fog1 Cluster does not have sufficient CPU resources to handle all the requests on its own.

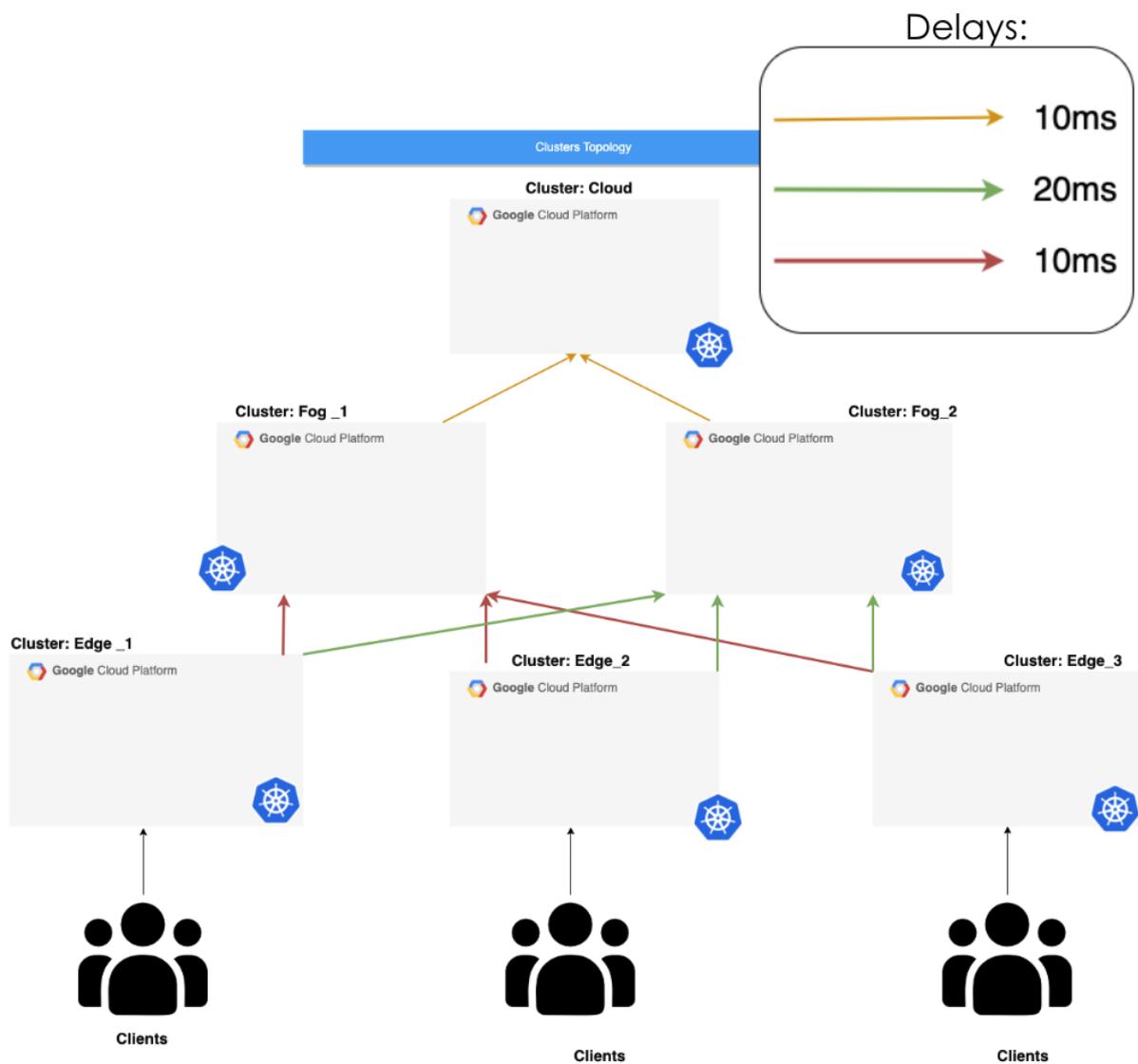
<i>Cloud CPU Consumption Average (cores)</i>	
Before Schedulers Activation	1.8
After Schedulers Activation	1.2
After Re-Routing	1.4
Total Reduction Percentage after Schedulers Activation	-33.3%
Total Additional Percentage after Fog2 Failure	+16.6%

5.3 Experiment 5

This experiment aims to assess the response of the architecture under varying request ratios and time delays between cluster connections. Specifically, we will examine the following request ratios:

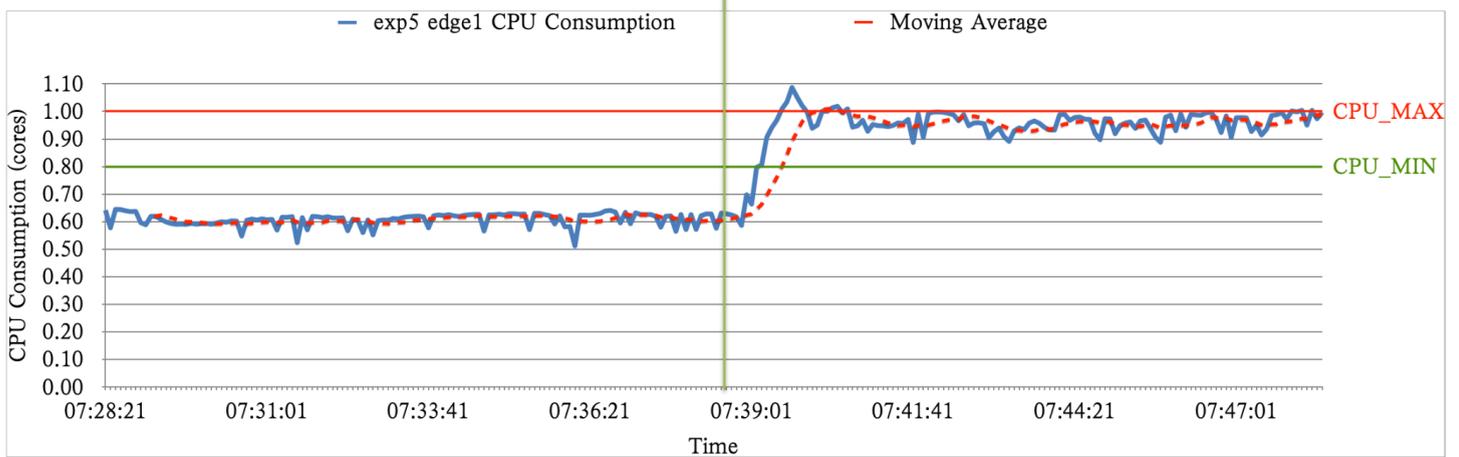
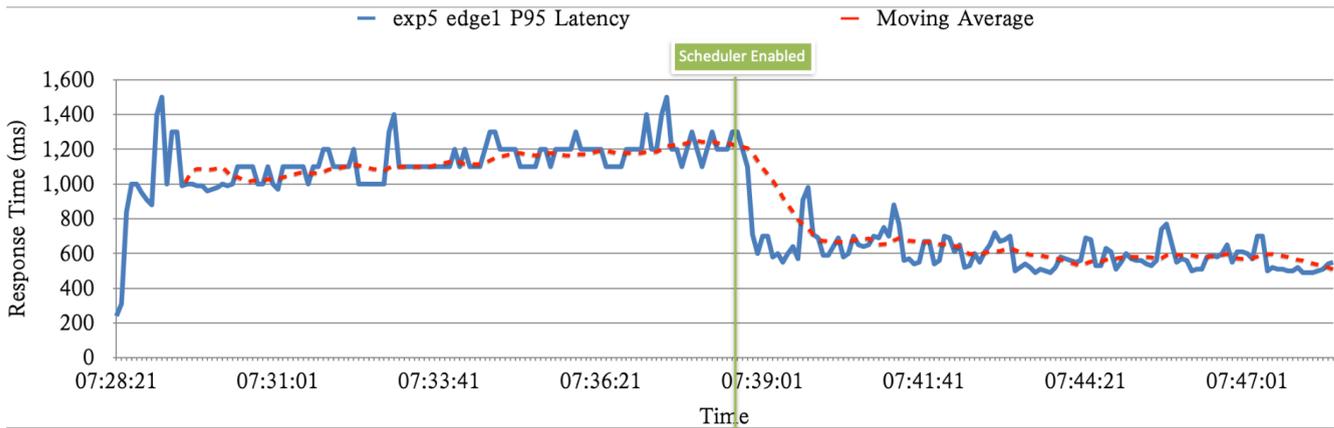
	Edge1	Edge2	Edge3
Eshop	90%	10%	50%
IXEN	10%	90%	50%

and the delays between clusters are described here:



Users: 200 users/Cluster

Edge1:



CPU Consumption Average (cores)	
Before Scheduler Activated	0.6
After Scheduler Stabilization	0.9
Total Additional percentage	+50.0%

Response Time Average (ms)	
Before Scheduler Activated	1100
After Scheduler Stabilization	550
Total Reduction percentage	-50.0%

In this cluster, 90% of the incoming requests are referring to the Eshop application pods and 10% to IXEN pods. Furthermore this cluster is connected with cluster Fog1 and Fog2 but the connections are different as for the time delay concerns. Time delay at the connection with Fog1 is 10ms and at Fog2 connection is 20ms. After schedulers enabled we can see a significant drop to the response time and an increase to the CPU Consumption, nothing different compared to the previous experiments. The interesting points with this experiment are the traffic-splits that the scheduler created and the pods that the scheduler selected to deploy. For all of the requests that needed to be forward to another cluster (Fog1 or Fog2) scheduler selected to forward requests using these ratios:

requests towards Fog1: 66.666%

requests towards Fog2: 33.333%

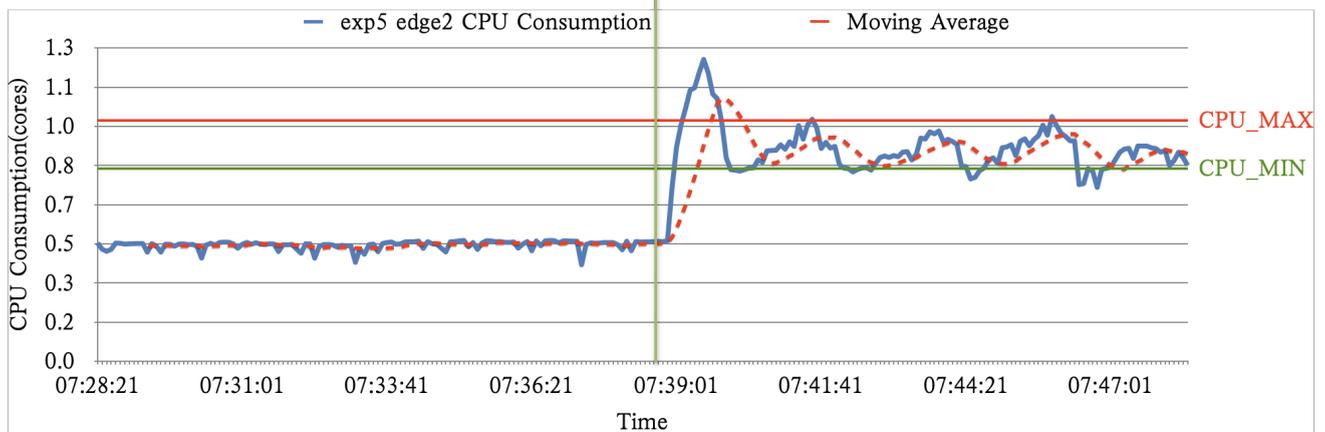
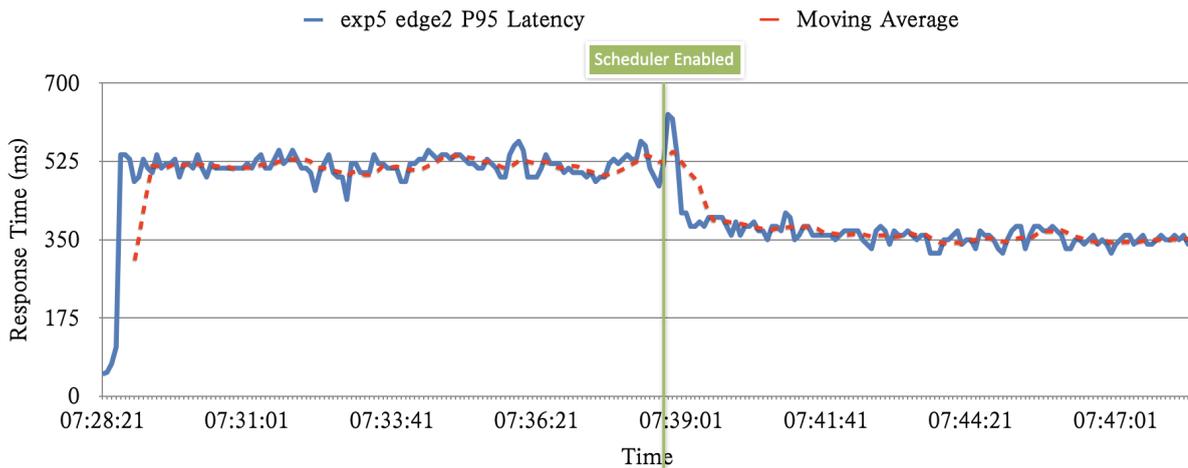
This ratio was computed using the previous described equation. So the Fog1 cluster whose response's time is less is receiving more requests compared to Fog2.

The second interesting point is the services that scheduler selected to deploy/maintain:

Edge1 - Services
apacheservice - IXEN
frontend - ESHOP
currencyservice - ESHOP

Apart from the apachedepoyment and frontend services which should be always present to the Edge cluster because they are the entrance points for the applications, scheduler selected to install currencyservice which is a service that belongs to the ESHOP application. The particular choice was made due to the fact that the majority of the requests created internally to the cluster had to do with the ESHOP application, so its services were more popular among the rest.

Edge2:



Response Time Average (ms)	
Before Scheduler Activated	520
After Scheduler Stabilization	350
Total Reduction percentage	-32.7%

CPU Consumption Average (cores)	
Before Scheduler Activated	0.5
After Scheduler Stabilization	0.9
Total Additional percentage	+80%

In this Cluster someone can notice the same behavior as for the CPU consumption and the response time concerns. The difference with this Cluster is the choice of the service that the scheduler made:

Edge2 - Services
apachservice - IXEN
frontend - ESHOP
keyrockservice - IXEN

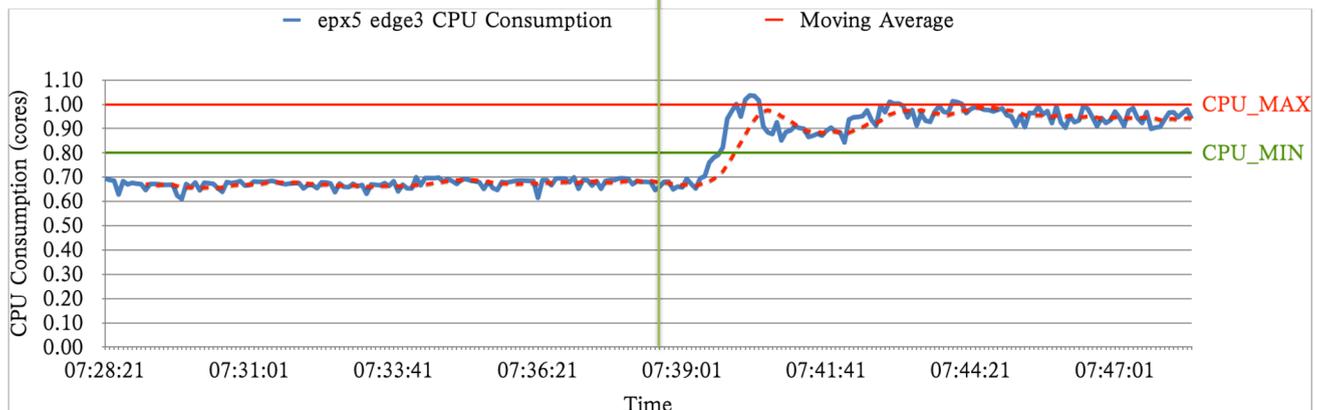
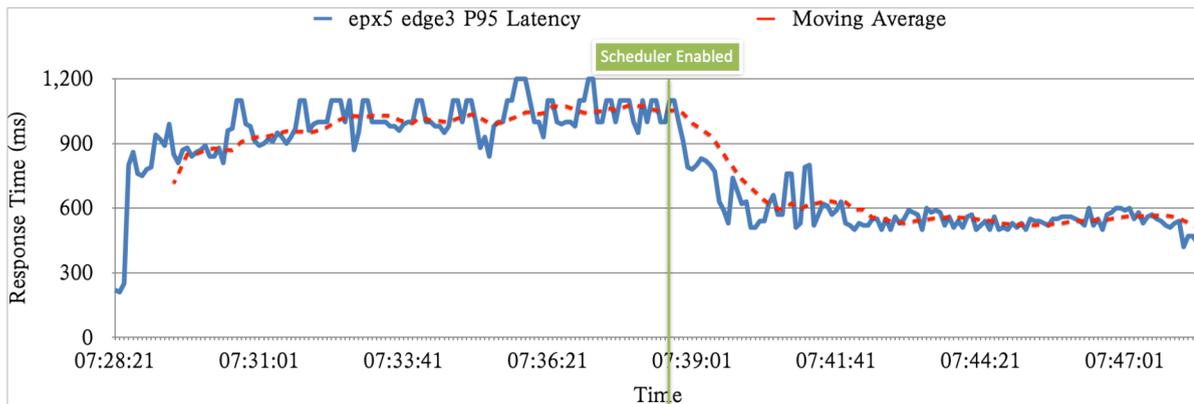
This time, requests for Eshop services were more compared to Eshop, thus the scheduler chose to deploy keyrockservice (IXEN app) to exploit the remaining Cluster CPU capacity. For the traffic-splits, results were the same as Edge1, because time delays between connections were the same:

requests towards Fog1: 66.666%

requests towards Fog2: 33.333%

An interesting point for this Cluster is the oscillation of the CPU consumption between CPU maximum and minimum values after the scheduler was enabled. To be more precise, the scheduler is trying to stabilize the CPU consumption between the desired limits, so when the consumption is not aligned with them it does some minor adjustment to recover it back to the spectrum resulting in this oscillation until it manages to stabilize.

Edge3:



<i>Response Time Average (ms)</i>	
Before Scheduler Activation	1100
After Scheduler Stabilization	550
Total Reduction percentage	-50%

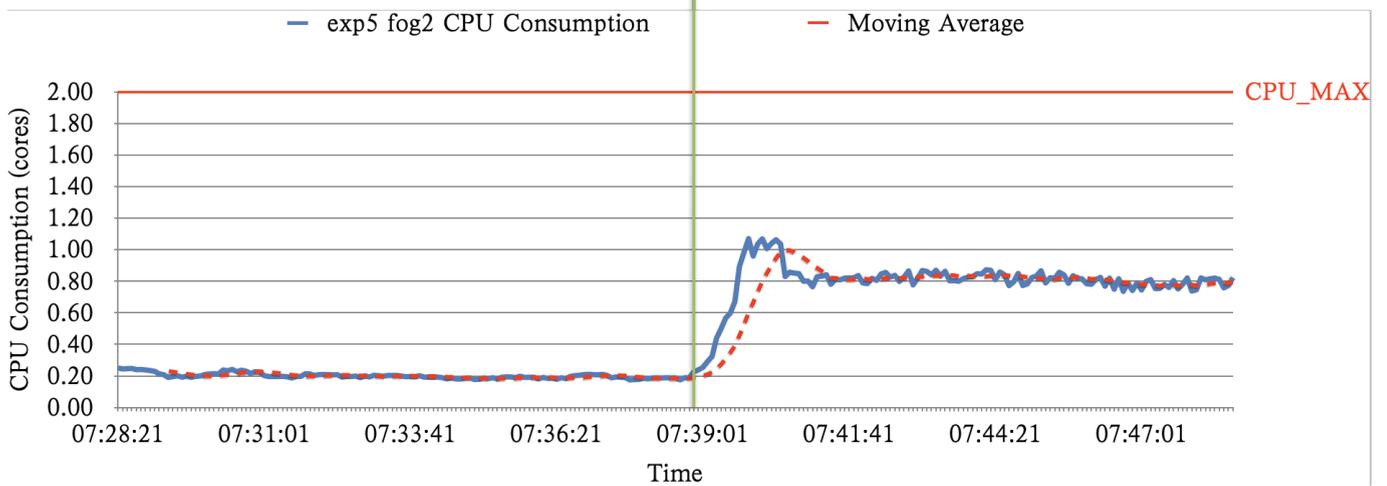
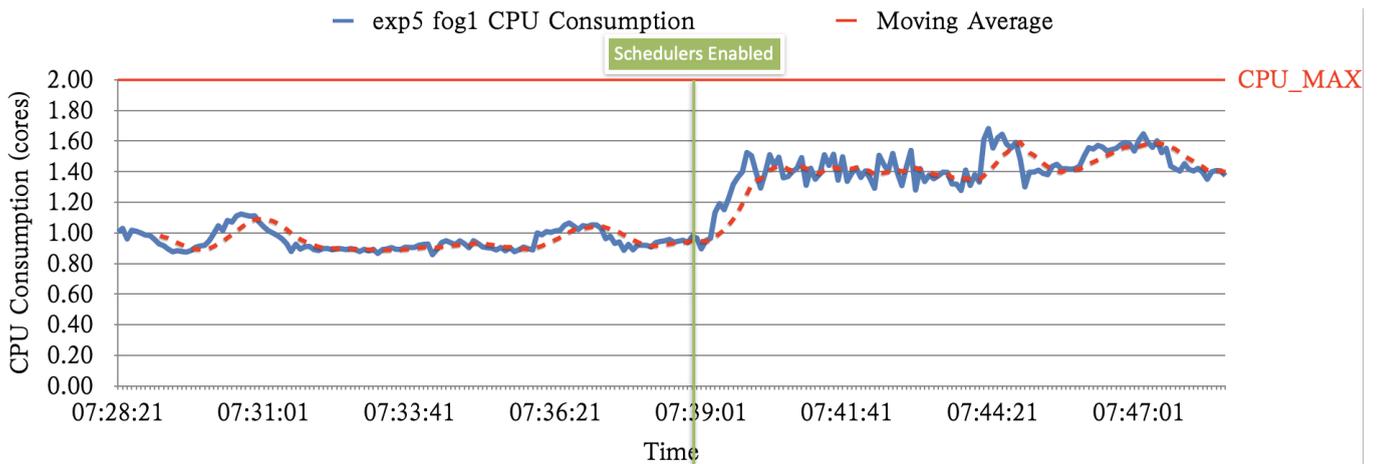
<i>CPU Consumption Average (cores)</i>	
Before Scheduler Activation	0.7
After Scheduler Stabilization	0.9
Total Additional percentage	+28.5%

In this cluster, incoming requests are equally distributed between Eshop and IXEN applications, 50% and 50% respectively. The CPU consumption increased and the response time decreased, following the same trend as the previous clusters of this experiment. In addition the ratio of traffic splits that was applied by the scheduler was the same as well. The services that the scheduler choose to deploy/maintain:

Edge3 - Services
apacheservice - IXEN
frontend - ESHOP
currencyservice - ESHOP

Although requests ratios for the two applications were the same, scheduler chose to deploy/maintain currency service, a service that belongs to Eshop application. This happened due to the fact that frontend service which is the entrance for all of the requests, is generating a lot of requests for currency service compared to the other type of requests. Thus, the scheduler based on the popularity and the frequency of this type of request chose the currencyservice.

Fog1 & Fog2:



Fog1 CPU Consumption Average (cores)	
Before Scheduler Activation	0.9
After Scheduler Stabilization	<u>1.4</u>
Total Additional percentage	+55.6%

Fog2 CPU Consumption Average (cores)	
Before Scheduler Activation	0.2
After Scheduler Stabilization	<u>0.8</u>
Total Additional percentage	+300%

In the edge clusters, we can notice the usual behavior; When schedulers activated the CPU consumption increased. Although in this experiment the increment was unequal. Fog1 average consumption after schedulers enabled was 1.4 cores, but Fog2 scored at 0.8 cores. This is a result of the unequal number of incoming requests. Since the connection of the Fog2 clusters with the edge cluster's are more time-delayed, edge clusters schedulers preferred to send the largest portion of the requests to Fog1.

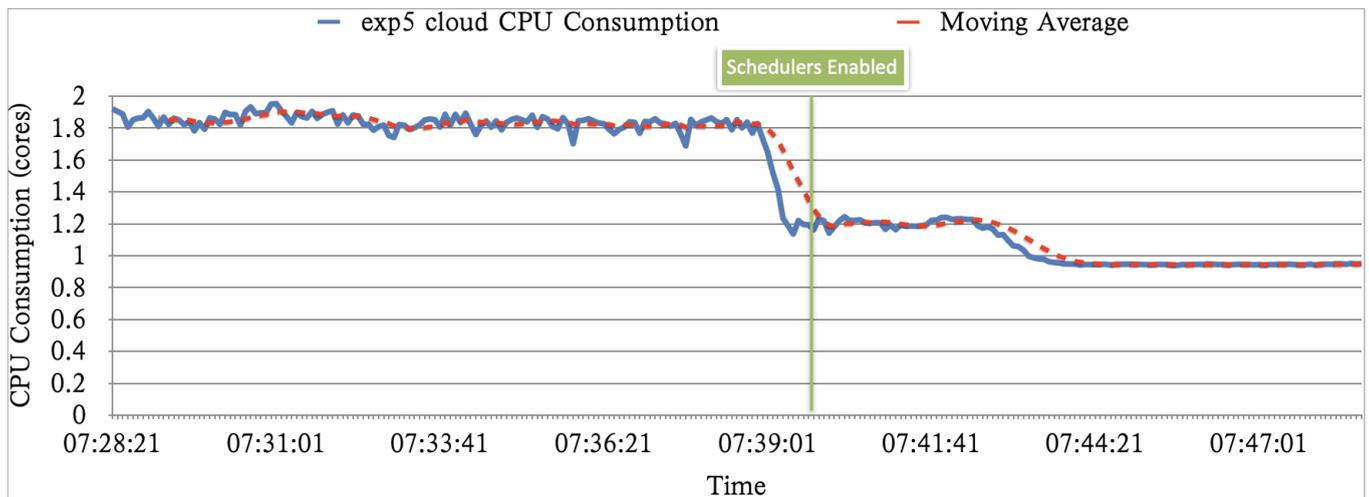
We can see clearer the inequality of the CPU consumption if we compare the above two tables with the respective tables of Experiment 2. There the average consumption after the scheduler's activation was 1.2 cores for both of the Fog Clusters.

Fog clusters services after scheduler stabilization:

Fog1 (All Services)	Fog2 (All Services)
cartservice	cartservice
currencyservice	currencyservice
shippingservice	shippingservice
emailservice	emailservice
paymentservice	paymentservice
recommendationservice	recommendationservice
adservice	adservice
checkoutservice	checkoutservice
frontend	frontend
productcatalogservice	productcatalogservice
noderedservice	noderedservice
orionservice	orionservice
keyrockservice	keyrockservice
authzforceservice	authzforceservice
cygnuservice	cygnuservice
apacheservice	apacheservice
queryingsensorsservice	queryingsensorsservice
cometservice	cometservice
orionproxyservice	orionproxyservice
cygnusproxyservice	cygnusproxyservice
noderedproxyservice	noderedproxyservice
queryingsensorsproxyservice	queryingsensorsproxyservice
sthcometproxyservice	sthcometproxyservice

Once again, as long as there is enough CPU capacity to deal with the incoming requests, all of the services were initialized.

Cloud:



Cloud CPU Consumption Average (cores)	
Before Schedulers Activation	1.9
After Schedulers Activation	0.9
Total Additional Percentage	-52.6%

Cloud CPU consumption didn't experience any unusual or unexpected behavior. Consumption decreased after scheduler activation as it did in all of the previous experiments.

Chapter 6

Conclusions & Future Work

6.1 Conclusions

In this thesis, we present a novel concept for an autonomous decentralized scheduler that operates across the entire cluster topology. This scheduler is designed to intelligently determine which deployments/services should be deployed, deleted, or maintained. Its decision-making process takes into account factors such as the quantity and type of incoming requests, as well as the CPU capacity available in each individual cluster.

The conducted experiments yield several interesting observations:

- Through the first three experiments, it becomes evident that our approach enhances efficiency when the cumulative influx of requests aligns with the CPU capabilities of the initial Clusters Layer (edge Layer).
- When requests are increasing beyond this point, our approach doesn't have any effect, negative or positive on the response time.
- Although, based on the experiment four (4), we can securely conclude that in case of a middle Layer Cluster Failure (Fog layer), our architecture can handle the requests efficiently with minimum down-time.
- Finally, considering the experiment's five (5) results, we can conclude that our approach takes into consideration the type of traffic that the infrastructure is called to answer, adapting accordingly.

6.2 Future Work

An idea for future work is to implement a similar tactic for the static data. In our experiments all of the data was instantly available by a help-cluster without any time-delay. This would require creating an extension of the scheduler so it can choose which data it wants to keep in the local cluster and how often it should update the data with any newer version of it.

Another idea is to extend the scheduler by adding more data to the decision making equation:

- RAM consumption of the services/deployments.
- The history of the reliance of upper level cluster (If there a neighbor cluster experienced a lot of down time, decrease the ratio of the forwarded requests towards this cluster)
- Examine the scenario of forwarding requests not only to an upper layer cluster, but also to a cluster in the same layer.

References

[1] "What is IOT? ":

[https://www.oracle.com/internet-of-things/what-is-iot/#:~:text=The%20Internet%20of%20Things%20\(IoT\)%20describes%20the%20network%20of%20physical, and%20systems%20over%20the%20internet](https://www.oracle.com/internet-of-things/what-is-iot/#:~:text=The%20Internet%20of%20Things%20(IoT)%20describes%20the%20network%20of%20physical, and%20systems%20over%20the%20internet)

[2] Adyson Magalhães Maia; Yacine Ghamri-Doudane; Dario Vieira; Miguel Franklin de Castro, "Dynamic Service Placement and Load Distribution in Edge Computing" in 2020 16th International Conference on Network and Service Management (CNSM), doi: 10.23919/CNSM50824.2020.9269059

[3] Shehenaz Shaik; Sanjeev Baskiyar, "A Scalable Approach to Service Placement in Fog/Cloud Environments" in 2021 IEEE International Performance, Computing, and Communications Conference (IPCCC), doi: 10.1109/IPCCC51483.2021.9679396

[4] Kun Lu; Jianyu Song; Linlin Yang; Guorui Xu; Mingchu Li, "Dynamic Service Placement Algorithm for Partitionable Applications in Mobile Edge Computing" in 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), doi: 10.1109/CCGrid54584.2022.00126

[5] Jianwen Xu; Kaoru Ota; Mianxiong Dong, "Plug-and-Play for Fog: Dynamic Service Placement in Wireless Multimedia Networks" in 2018 IEEE/CIC International Conference on Communications in China (ICCC), doi: 10.1109/ICCCChina.2018.8641090

[6] Onur Ascigil; Truong Khoa Phan; Argyrios G. Tasiopoulos; Vasilis Sourlas; Ioannis Psaras; George Pavlou, "On Uncoordinated Service Placement in Edge-Clouds" in 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), doi: 10.1109/CloudCom.2017.46

[7] "Cloud Computing": <https://www.ibm.com/topics/cloud-computing>

[8] "Fog Computing": <https://www.heavy.ai/technical-glossary/fog-computing>

[9] "Edge Computing":

<https://www.accenture.com/us-en/insights/cloud/edge-computing-index#:~:text=Edge%20computing%20is%20an%20emerging,led%20results%20in%20real%20time>

[10] "Schematic representation of a cloud architecture with cloud, fog and edge layers": https://www.ionos.com/digitalguide/fileadmin/DigitalGuide/Screenshots_2018/cloud-architecture.png

- [11] "Virtualization": <https://www.ibm.com/topics/virtualization>
- [12] "Docker Hub": <https://hub.docker.com/>
- [13] "Container Orchestration":
<https://www.redhat.com/en/topics/containers/what-is-container-orchestration>
- [14] "What is Kubernetes?": <https://kubernetes.io/docs/concepts/overview/>
- [15] "Kubernetes Components":
<https://kubernetes.io/docs/concepts/overview/components/>
- [16] "Linkerd definition": <https://www.techtarget.com/searchitoperations/definition/Linkerd>
- [17] "Linkerd Control Plane schematic":
<https://linkerd.io/images/architecture/control-plane.png>
- [18] "Kubernetes Python Library": <https://github.com/kubernetes-client/python>
- [19] "What is K3s?": <https://docs.k3s.io/>
- [20] "Chaos Mesh Overview": <https://chaos-mesh.org/docs/>
- [21] "Linux Traffic Control (TC)":
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/linux-traffic-control_configuring-and-managing-networking
- [22] "Ambassador":
<https://docs.seldon.io/projects/seldon-core/en/latest/ingress/ambassador.html#:~:text=Ambassador%20is%20a%20Kubernetes%20native.management%2C%20authentication%2C%20and%20observability.>
- [23] "What is Locust?": <https://docs.locust.io/en/stable/what-is-locust.html>
- [24] "Preemptive VMs": <https://cloud.google.com/compute/docs/instances/preemptible>
- [25] "Google Cloud Platform - microservices demo":
<https://github.com/GoogleCloudPlatform/microservices-demo>

[26] X. Koundourakis; E. G. M. Petrakis, "iXen: context-driven service oriented architecture for the internet of things in the cloud" in *Procedia Computer Science* 2020, doi: 10.1016/J.PROCS.2020.03.019.