ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Dynamic Microservice Placement Strategies in Kubernetes

DIPLOMA THESIS

In partial fulfillment of the requirements for the degree of
DIPLOMA IN ELECTRICAL & COMPUTER ENGINEERING
By
Kastrinakis Nikolaos

Committee:
Assistant Professor Nikolaos Giatrakos
Dr. Chrisa Tsinaraki
Professor Petrakis Euripides (Supervisor)

Chania, July 2024

Abstract

As applications are moving to the cloud, microservices-based architectures are becoming more and more popular. For all their advantages, microservices add complexity to application design and bring with them a new set of problems. To combat these issues, Kubernetes was created to orchestrate containerized microservices applications by automating management, scaling and deployment. Applications in Kubernetes are deployed in Clusters, a set of Nodes (VMs) that are managed by a centralized control plane. Microservices are placed in Pods and those Pods in turn into Nodes. The Kubernetes Scheduler is responsible for placing Pods in Nodes using a set of filtering and scoring rules to attempt to optimally place it in the Cluster. The placement of Pods in certain Nodes is a key problem in microservices architectures, as the increase in physical distance between parts of an application could lead to latency issues. Furthermore, the Node-to-Node traffic, also called egress traffic, is charged by cloud providers. Therefore, placing Pods that communicate a lot with each other in the same Node is considered optimal to minimize egress traffic, decrease infrastructure costs and improve application performance via minimizing response times. In previous works, this was done using graph clustering techniques to produce an optimal placement for the microservices of the application. While the results were stellar, they would not translate well in a practical setting, where workloads shift and scaling of Pods occurs. In this work, we will attempt to address the problem with a more heuristic approach. We will alter the functionality of the Default Scheduler, by adding our own filtering and more importantly a communication aware scoring method. Our aim is to improve all aforementioned areas with a method that more aligns with microservices architecture and Kubernetes design. One application was deployed in the Google Cloud Platform (GCP) using Kubernetes to perform our experiments. The results show that our communication-aware placement is far better in improving application performance, infrastructure cost and egress traffic compared to the Default Scheduler. Furthermore, while it performs slightly worse than MODSOFT-HP, a fuzzy graph clustering method, further experiments with more users and scaling mechanisms that are connected with our method, could prove it to be a better and more practical method overall.

Table of Contents

List of Figures

List of Tables

Introduction

Problem Definition

The architecture of older applications used to be monolithic and on-premises. Modern applications are trending towards microservices-based architectures on the cloud. This approach offers easier to maintain, update, migrate and scale applications. The rise of these architectures has created the need for tools to orchestrate and manage microservices. Kubernetes is one such tool. Designed by Google, it provides control over the scheduling cycle and the scaling and scalability of microservices. Furthermore, it offers the means to schedule them in multiple VMs, as well as load balancing and networking between the VMs.

The benefits of cloud migration come at a higher cost of infrastructure and increased physical distance between different operations of the same application. Cloud providers charge their users based on the resources their application consumes in terms of CPU and RAM, as well as network traffic between VMs. Requests that require multiple microservices to receive a response can have increased latency if the microservices are placed in different VMs. Therefore, inefficient placement of microservices and overutilization of resources can severely increase infrastructure cost and application latency.

Placing microservices with high communication in the same VMs reduces traffic between VMs, also referred to as egress traffic, which in turn decreases cost of infrastructure and application latency. Therefore, it is a solution to both arising problems. Optimizing the deployment of microservices in a cloud environment is known as the service placement problem (SP). The problem can be solved by altering the scheduling process and can even reduce the number of VMs the application requires, further reducing the cost.

Scope of the Thesis

We will approach the solution to the service placement problem by altering the scoring method of the default scheduler with the purpose of reducing the number of nodes the application requires, the egress traffic between them and the latency of the requests in it. A scoring plugin is applied by altering the scoring in the scheduling cycle, changing the decision of the scheduler. The scoring is done based on the traffic between the microservice and each node. The node with the highest score is where the service will be placed.

Via scoring nodes with services that have high communication with the service we want to place, we place services with high traffic in the same nodes. Therefore, we

reduce egress traffic, which in turn reduces the response time of requests, as more of them will be processed in the same machine.

We also consider each node's available resources and the demands of each microservice before the scoring, to ensure we utilize the resources fully. As a consequence we optimize resource utilization and can host the application using less VMs, greatly reducing infrastructure cost, without impacting its performance.

Thesis Structure

This Thesis is arranged in five chapters, this introduction and four more. The first one presents an overview of related works, which is split into several sections. Section one introduces works on the service placement problem in cloud computing and section two is a summary of works on the service placement problem in fog edge computing, which was also studied in the course of this thesis for inspiration. The last section presents the work on which our proposed solution is based on. In the second chapter we dive into the Kubernetes environment and the features we use, as well as other tools used for our implementation. Chapter three describes in detail the architecture of our cluster, its nodes and their resources, which is constructed in the Google Cloud Platform. Furthermore, it also elaborates on our implementation, the algorithm we use, how we manipulate deployment files and the scheduling cycle to place the services and the applications we use in our later experiments. In the final chapter the experimental results, as well as the experiments conducted, are presented. The experiments aim to test the validity and effectiveness of the proposed solution, using comparisons with the default Kubernetes Scheduler and older work on the subject.

Chapter 1

Background and Related Work

There have been many approaches into solving the service placement problem, leading to multiple different categorizations of the proposed solutions based upon their nature and method. The greatest divide lies in the environment in which the solution is proposed, that being either a centralized cloud environment, or a distributed cloud/fog/edge environment.

This chapter will provide an overview of the field relating to the service placement problem in two parts. First, a review of related works on the problem in cloud computing will be presented, followed by a second part with works on the problem in fog/edge computing.

1.1 Related Work on Service Placement in the Cloud

One category of solutions of the service placement problem come in the form of clustering methods. As clustering methods we refer to methods that transpose the Service Placement problem into a graph clustering/partitioning one, where usually the services represent the nodes of the graph and the edges of the graph are a form of communication between them. The clustering algorithm then forms partitions of the graph, which represent Nodes in the cluster, aimed at minimizing the sum of the weight of the edges, therefore minimizing egress traffic in the cluster.

Aznavouridis, Tsakos and Petrakis [1] proposed a solution for the Service Placement problem aimed at reducing the cost of infrastructure. To achieve this, they modeled the application as a weighted and directed graph, where the microservices were the vertices or nodes of the graph, represented by the microservices and the weight of each edge was one of two affinity metrics representing the communication between them. They implemented their methods in a Kubernetes cluster with a predefined number of Kubernetes Nodes. They used several flat graph partitioning algorithms that produced partitions of the graph with as little weight, which in their case was the communication of services, as possible between them. They then placed the services of each partition in a different node of the cluster.

Clustering methods are centralized in nature, making them very popular and effective in cloud environments, where there is usually a centralized agent with knowledge of the whole system and in control of all placement decisions. Clustering methods or more accurately graph partitioning methods are split into two types, flat and fuzzy partitioning. So far, we have referred to flat methods. Fuzzy partitioning or fuzzy clustering, as we will refer to it, is where each service can belong to multiple partitions. This distinction is very important, as although more services are deployed due to duplicates, it allows for much lower egress traffic and response times, because highly demanded services are placed in multiple Nodes.

In 1994, Yan and Hsiao [2] expanded the FCM algorithm [3] to solve the graph bisection problem. The graph bisection problem is a graph partitioning problem where a graph is to be divided into two partitions with the aim of optimizing the sum of the weights of the edges joining the two partitions.

In 2019, Hollocou, Bonald, and Lelarge [5] proposed the “Modularity-based Sparse Soft Graph Clustering”, which is a relaxation of the modularity optimization problem introduced by Newman and Girvan [6]. The MODSOFT [5] algorithm produces fuzzy partitions and does not require prior knowledge of the available clusters, unlike other proposed relaxation algorithms. Furthermore, the algorithm includes a parameter “t”, which can be “calibrated” to produce “softer” or “harder” partitions.

More recently, Skevakis and Petrakis [7] attempted to solve the Service Placement problem aimed at reducing the response time of application requests. They approached the problem as a fuzzy clustering one and used the MODSOFT algorithm, as well as the heuristic packing algorithm proposed by Aznavouridis, Tsakos and Petrakis [1].

1.2 Related Work on Service Placement in the Fog

Even though this work is in the cloud, it draws many elements from heuristic methods used in fog-edge environments. In their work, Prountzos and Petrakis [8] propose distributed heuristic methods in such an environment, aimed at reducing response time of applications dynamically according to workload changes. They implemented their strategies in a multi-cluster Kubernetes environment, where each cluster contained a single Node and made independent decisions about its placement. The decisions were made by using one of four different algorithms. Each algorithm placed microservices on the Node based on a metric, while at the same time made ejection decisions based on the same or a different metric. The first two, namely “lfu” (least frequently used) and “rlsd” (response latency-based service deployment) , place the microservices on the Node that have the highest request rate and lowest response time respectively, while ejecting the lowest request rate and highest response time ones. The other two used the same placement mechanism, but chose to eject the highest RAM usage services from a Node that lacked resources instead.

Lera, Guerrero and Juiz [9] propose a multi-criteria analysis to determine the placement of services. Specifically, they use the Electre III method, which is part of a family of outranking multi-criteria decision aiding (MCDA) methods, and ranks alternatives based on multiple criteria. They consider the latency between nodes, the number of devices a request travels (hop count), the energy consumption in watts of the deployment of a service, the cost of deploying a service on a device and the overhead for deploying a service in a node (deployment penalty). The problem is the method and criteria are tailored to solve the placement problem in a simulation and are not suited for a real world analysis.

Ascigil, Phan, Sourlas and Psaras [10] use several uncoordinated resource allocation strategies, involving admission, scheduling and placement mechanisms in order to satisfy

the maximum number of user requests in terms of their latency constraints, arguing for the simplicity of such methods. They draw inspiration from the problem of resource storage allocation and apply principles of cache management in their solutions. More specifically, the admission and scheduling of incoming service requests is performed using earliest **deadline first** (EDF) and **First-In-First-Out** (FIFO) methods. For the service placement policy they use four ranking policies, namely **strictest deadline first** (SDF), **least frequently used** (LFU), **hybrid** and **least recently used** (LRU). Each node performs placement/replacement procedures independently. The downside with their methods is that they are tested using simulations and although they use real-world data, it is imperative to test methods for such resource-intensive and time-constrained environments in the real world.

The success of the heuristic approach used in Prountzos and Petrakis' work, as well as its creativity, inspired us to consider such methods in a cloud environment. While there are stark differences between the two environments, the most important being the information we have of the system, we believe that there are definite advantages to a heuristic approach in the solution of the Service Placement problem. To be more precise, in a cloud environment we have complete knowledge of the system and all the services in the applications deployed in it, whereas in a fog environment, the architecture is typically split in three or more levels, namely the edge, the fog and the cloud, where each one has knowledge only of its own level.

We mention above that a heuristic approach may have advantages over an algorithmic one. One very important one is the ability to dynamically and independently place and eject microservices from Nodes. In a fully algorithmic approach, it is imperative to consider the whole of the system and its state before a decision about it is made. Then the algorithm makes a decision about the whole system, rearranging multiple services at the same time. This creates periods of downtime and is too slow a method in a dynamic system. With a heuristic method, we can make informed decisions about placement and ejection of single microservices dynamically, as the workload changes. Coupling this with a distributed approach, where multiple of these decisions are made simultaneously, we can reasonably conclude that a heuristic method is far better at responding to dynamic changes in workload relative to an algorithmic one. Furthermore, this also applies to scaling of existing services by adding and removing replicas.

In accordance to this, we have chosen to try a heuristic approach on a cloud environment by utilizing a communication aware scoring algorithm to place the services.

Chapter 2

Infrastructure and Tools

In the second chapter of the Thesis, we will present the infrastructure where the proposed solution was implemented and the tools that were utilized for implementing it and evaluating its efficacy.

2.1 Kubernetes

To understand Kubernetes it is important to know of the microservices architecture and how it came about. As application deployment has moved in the cloud, application requirements are steadily rising, causing the cost of infrastructure to increase as well as multiple other problems. The microservices architecture solved these issues by splitting complex applications in smaller pieces called microservices. Each microservice implements a distinct part of the application logic of a larger application and is deployed as a single unit called a container. Containers are a lightweight virtual environment. Via this splitting of application logic, the microservices architecture achieves optimized resource usage, reliability and scalability for an application.

Kubernetes [11] is a Google developed open-source orchestration tool for microservices-based architectures. It automates various tasks such as microservice deployment and scaling, network connectivity, load balancing and security. Furthermore, it provides tools for debugging via service discovery and allows for configuration on multiple of its features.

2.1.1 Infrastructure and Components

Kubernetes employs various abstractions to create a complete environment. The highest level of these abstractions is called a cluster and it is the sum of all the virtual machines that the application uses – usually belonging to the same server – that are connected in a network. The cluster is composed of Nodes, the Kubernetes term for a VM. Nodes are configured in groups called Node Pools, which specify the resources each VM will possess, as well as the number of Nodes to be deployed. All Nodes in a Node Pool have the exact same resource allocation and OS specifications.

There are two types of nodes in a cluster, a Master Node, where the Control Plane resides, and one or more Worker Nodes, where applications are deployed. To deploy a workload, Kubernetes employs an abstraction called a Pod. Pods are computing units that most often deploy a single container. They have storage and a single network IP. Networking in Kubernetes is done using abstractions called Services, not to be confused with application services, also known as microservices. Services reference a specific deployment, usually a microservice of an application, and manage network traffic, as well as load balancing between multiple instances of that deployment that are represented as Pods.

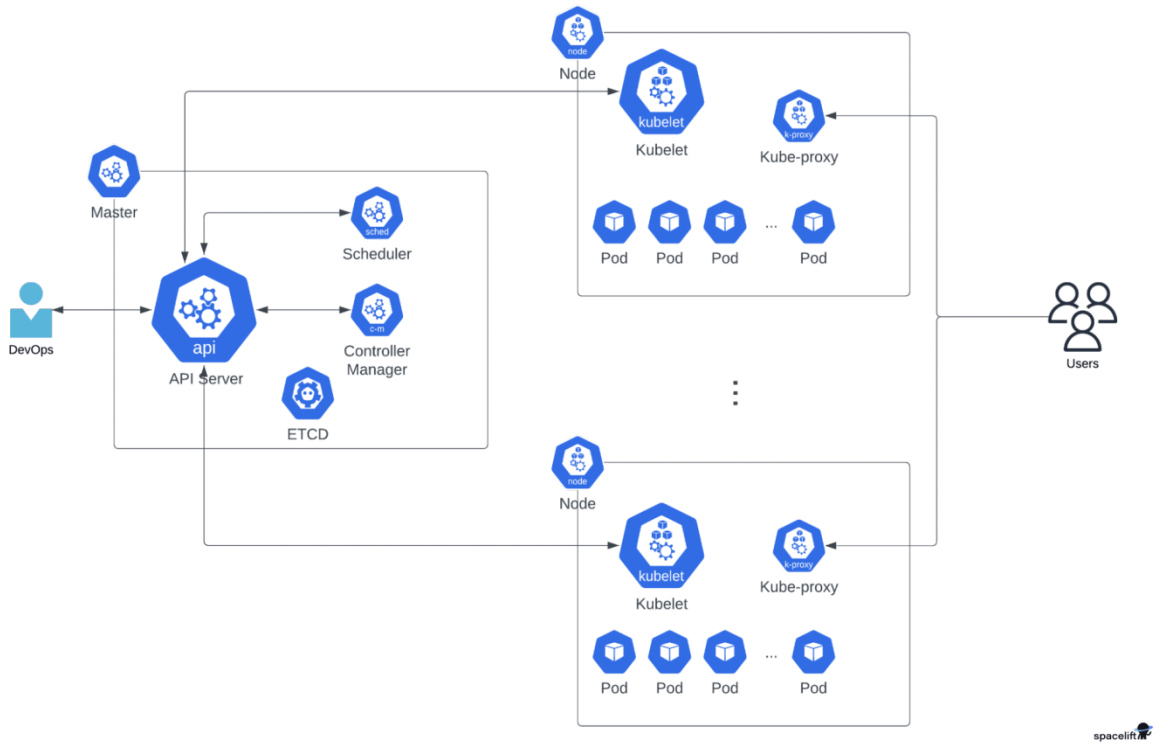


Figure 2.1 Kubernetes Architecture and Components

Besides all the abstractions, Nodes have various components that perform vital tasks for the cluster. Worker Nodes use the **kubelet** to ensure Pods are “healthy” and to manage the containers created. Furthermore, the **kube-proxy** manages network communication for Pods to communicate with integrity throughout all the Nodes in the cluster.

The Control Plane, which resides in the Master Node, is composed of five components. The **kube-apiserver** that exposes the Kubernetes API, the **etcd** that stores all cluster data in key-value pairs, the **kube-controller-manager** that manages all controller processes and the **cloud-controller-manager** that connects the cluster to the cloud provider’s API. Finally, the **kube-scheduler**, which is an integral part of this thesis and we will talk about in detail in the next chapter.

2.1.2 Scheduling Process

The **kube-scheduler’s** function is to assign newly created Pods into Nodes inside the cluster. The scheduling process, or more accurately scheduling cycle is done in two phases: the filtering phase and the scoring phase. During the filtering phase, the scheduler rules out any Node that doesn’t meet certain criteria that the Pod requires to function, such as available resources, or criteria set by the developer. In the scoring phase, the filtered Nodes will be scored based on several factors, such as affinities and anti-affinities, to determine the most fitting Node for the Pod. After all Nodes have been evaluated the scheduler will bind the Pod to the Node. This process is called binding and is internal to the system and not relevant to this Thesis.

The scheduling cycle is designed to be extensible, providing various means for developers to alter it according to their needs. By using a combination of nodeSelectors and Node labels, affinities and anti-affinities and taints and tolerations developers can control how the scheduler assigns Pods to Nodes. In this Thesis, we will show how we can extend the scoring of the base scheduler by running our own calculations and manipulating deployment configuration files to assign Pods according to our strategy.

2.1.3 Services and Network Traffic

Pods are assigned an IP address inside the cluster's network upon creation, but Pods may restart and a microservice can have multiple instances of itself on multiple Pods. Therefore, the IP addresses of Pods are dynamic. To maintain network integrity, Kubernetes has created a resource called **Service.** Kubernetes Services represent a single application service in the cluster's network. They bind with all Pods of the application service and forward traffic to them, as well as load balance between multiple instances of them. To achieve that, each Pod reveals its IP address upon creation to its Service, while the Service exposes a single permanent address for all other services in the network.

Kubernetes Services have four different types for a multitude of purposes. The most common type is the ClusterIP, which exposes a cluster-internal IP for the service to communicate with the other services in the cluster. The NodePort type exposes the service on all of the cluster's Node IP addresses in a specific port to allow external communication. The LoadBalancer type exposes the service to external communication without providing load balancing. It is useful to provide your own load balancing component. Lastly, the ExternalName type maps the service's traffic to a custom field specified in the configuration file, called the externalName.

In the cloud, traffic is defined as ingress when it is the inbound traffic of a network and egress when it is the outbound traffic of the network. In this work, **ingress traffic** is defined as the traffic between two services whose Pods are located in the same Node (in-Node traffic) and **egress traffic** as the traffic between Pods of different Nodes. This is an important distinction for this work as we will later see that reducing **egress traffic** defined in this way directly affects one of the key goals of this thesis.

2.2 Service Mesh and Istio

A Kubernetes Architecture introduces a multitude of problems not prevalent in previous monolithic designs. Developers have to consider the security of the network, implement ways to monitor the Pods and extract quantitative and qualitative metrics such as communication rates, successful or failed requests and much more. To create such a system a whole team would be required in addition to developers working on the application itself. Therefore, an additional layer of software called a **Service Mesh** was created.

A **Service Mesh** [12] is a layer of infrastructure dedicated to add features to microservices architectures such as observability, traffic management and network security. To achieve

this seamlessly, Service Meshes use small containerized applications called sidecar proxies or simply proxies and “inject” them into every microservice’s Pod. The proxies are responsible for monitoring and extracting data from each microservice and send all the information to a dedicated control plane, which handles all the logic.

In this thesis, **Istio** [13] is the Service Mesh used for the implementation, which is an open-source Service Mesh. Istio’s control plane is called **Istiod**. During configuration a developer can choose which services Istio will monitor by labeling namespaces. Istiod automatically detects services and Pods deployed in the labeled namespaces and deploys a proxy in each one. Furthermore, it can enable secure communication between the cluster’s services through various means, which is beyond the scope of this thesis. Lastly, it gathers telemetry data from the proxies deployed in each Pod, which can be exported to various monitoring tools, such as Prometheus and Kiali.

2.3 Metrics Tools

2.3.1 Prometheus

Prometheus [14] is an open-source monitoring and alerting system for cloud environments. It provides a time series database, multiple metric types, automatic data scraping, a query language, custom alerts and many modes of dashboard and graphing support. All data is stored as time series, meaning each metric has a stream of time-stamped values. Similar to Istio, Prometheus has to be configured to pull metrics from a service. More specifically, the **data retrieval worker** component in the Prometheus server sends HTTP requests to the targets and the data is then stored in the **TSDB (time-series database)**. Lastly, the **HTTP server** is responsible for retrieving the data stored in the **TSDB** [16].

Another vital component in the Prometheus architecture are the **exporters**. Exporters are intermediaries between a third-party system and Prometheus. They are responsible for extracting metrics from the target system, transforming them into a data format Prometheus can understand and exposing an endpoint for the Prometheus Server to read them.

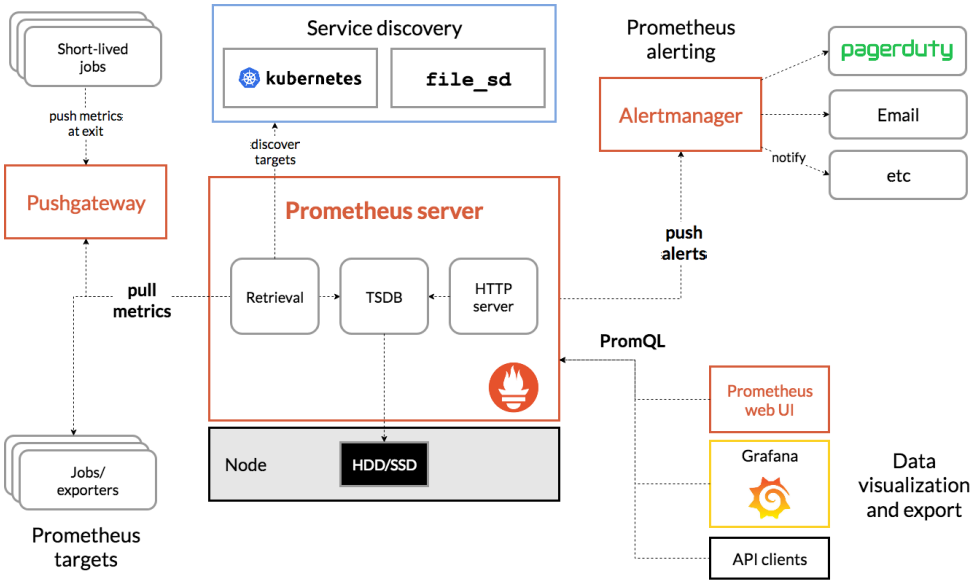


Figure 2.2. Prometheus Architecture

There are four main metric types [15]: counter, gauge, histogram and summary. Counter is a cumulative metric, representing the sum of a metric's values. The gauge is a metric that increases or decreases and usually represents the current value of the metric. A histogram samples values and sums them in configurable buckets and also provides the sum of all values. Lastly, a summary is similar to a histogram, while also providing a configurable quantile of the value.

To gain access to Prometheus' metrics, a request has to be sent to the Prometheus API server, using a PromQL query. PromQL is a query language designed specifically for Prometheus metrics. The request can be done using API or directly via the Prometheus Dashboard. The metrics can then be visualized on the Dashboard or using another compatible tool such as Grafana or Kiali or Jaeger, each providing their own dashboards and use cases.

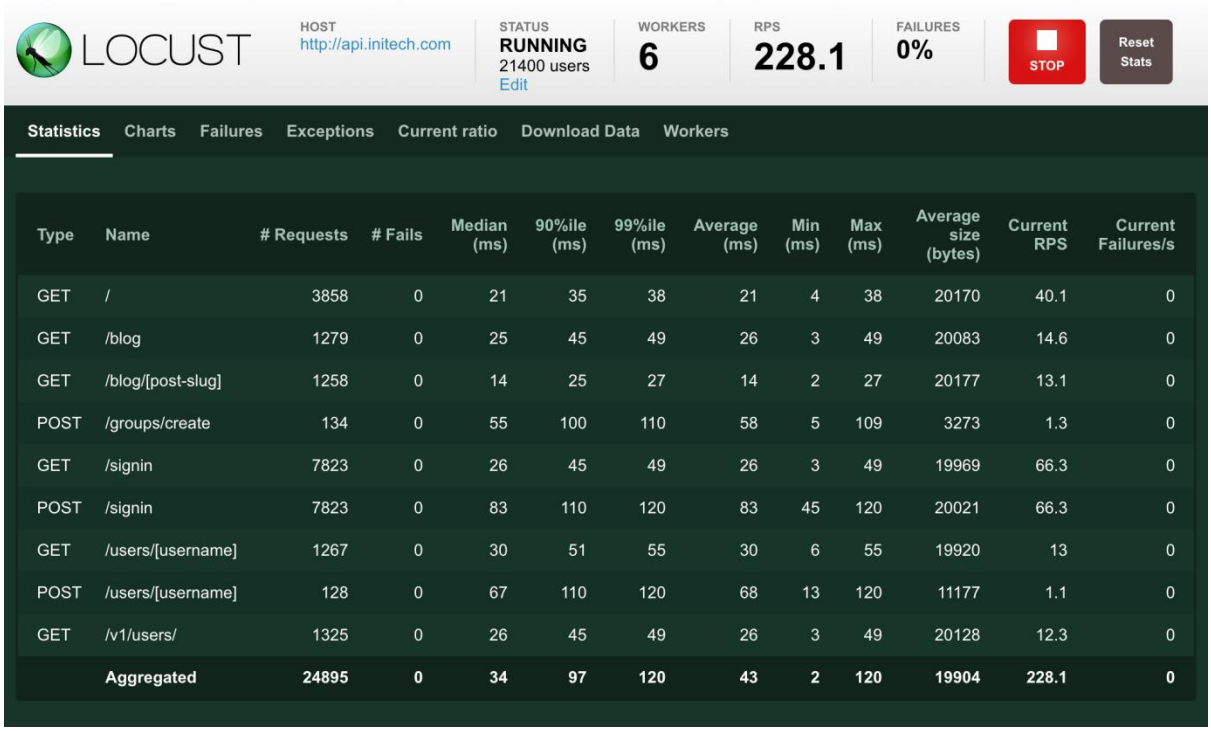
We use Prometheus to gain access to metrics such as communication rates between services dynamically and utilize them for an educated service placement decision.

2.4 Load Testing Tool

Locust

Locust [17] is an open-source, developer-friendly load testing tool. It provides the ability to define user behavior in simple Python code, which makes it easily pluggable in any architecture and expandable in any way the developer needs using other Python libraries if necessary. Furthermore, it is event-based making a single locust process able to handle thousands of concurrent users. It also includes a web based UI, where developers can see all requests in each endpoint and various metrics about them in real time, as well as having the ability to export these results in several formats.

We use locust in our experiments as a workload to stress both applications from our local host. We used Python scripts to define user behavior and entered the web UI to gather important metrics for our experimental results. More detail is given in Chapter 4.



Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	3858	0	21	35	38	21	4	38	20170	40.1	0
GET	/blog	1279	0	25	45	49	26	3	49	20083	14.6	0
GET	/blog/[post-slug]	1258	0	14	25	27	14	2	27	20177	13.1	0
POST	/groups/create	134	0	55	100	110	58	5	109	3273	1.3	0
GET	/signin	7823	0	26	45	49	26	3	49	19969	66.3	0
POST	/signin	7823	0	83	110	120	83	45	120	20021	66.3	0
GET	/users/[username]	1267	0	30	51	55	30	6	55	19920	13	0
POST	/users/[username]	128	0	67	110	120	68	13	120	11177	1.1	0
GET	/v1/users/	1325	0	26	45	49	26	3	49	20128	12.3	0
	Aggregated	**24895**	**0**	**34**	**97**	**120**	**43**	**2**	**120**	**19904**	**228.1**	**0**

Figure 2.3. Locust Web UI

Chapter 3

Cluster Architecture and Implementation

This Chapter's objective is to present the architecture of the Cluster used for the implementation of the experiments and give an in depth look into the implementation itself. More specifically, affinity metrics considered, the scheduler's initialization, the algorithm performing the placement itself, as well as the applications used in our experiments to test the efficacy of our strategy will all be presented.

The Service Placement problem in a cloud or fog/edge environment is the task of ascertaining the optimal placement locations for the service replicas that constitute a cloud or Internet of Things (IoT) application within a centralized or distributed computing architecture respectively.

The "Communication Aware Scheduling of Microservices-based Applications on Kubernetes Clusters" [18] was proposed in 2022 by Marchese and Tomarchio. They propose an extension of the Kubernetes base scheduler by using a custom scoring plugin to extend the scoring phase. In detail, for each microservice that is scheduled, each node that has passed the filtering phase is scored based on microservice-to-microservice communication affinities and node-to-node latency. The aim of the work was that of minimizing network distance between critical communication channels. Two communication affinities were used to identify them, one represented as a static contribution to the score and one as a dynamic one. The static contribution was determined by the protocol the communication channel utilizes favoring synchronous protocols. They give a sample hierarchy of the types of communication channels, where higher priority is ascribed to synchronous routes and is associated with a more critical channel.

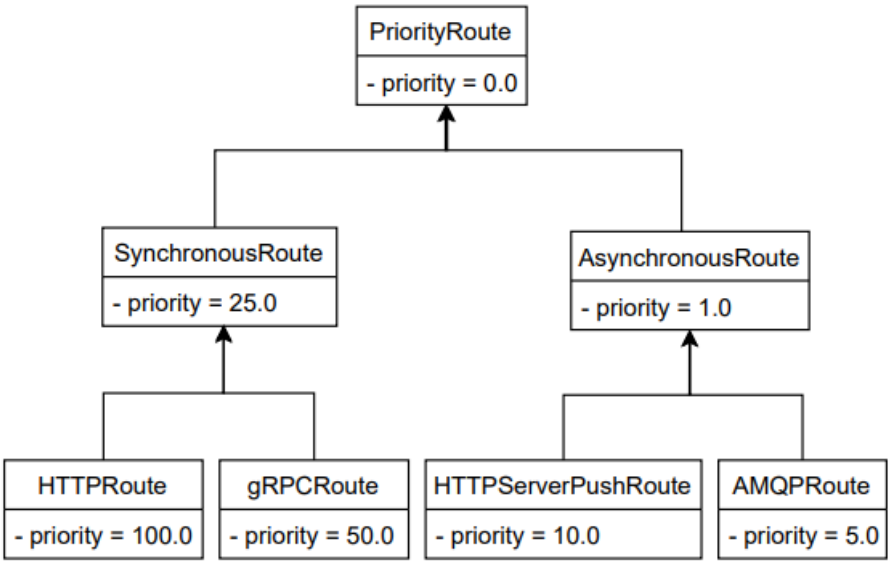


Figure 3.1. Communication Protocols Hierarchy

The dynamic contribution was proportional to the exchanged traffic of that channel at the time, measured as the mean value up to that time. Finally, for the latency, once the affinities between all microservices deployed to that Node and the microservice to be scheduled are added, if the affinities do not belong to the Node being scored, then they divided that score by the latency between the current Node and the Node being scored. The total score of each Node and for each microservice is calculated by the contributions of all Nodes in the cluster and each Node's contribution is divided by its latency to the Node being scored.

Inputs: svc, node, channels, nodeLatencies, clusterNodes
Outputs: score

```
1.  score <- 0
2.  For cNode in clusterNodes:
3.      pScore <- 0
4.      node_pods <- getNodeServices(cNode)
5.      For pod in node_pods:
6.          static_contr <- weight_a * channels[pod, svc].priority
7.          dynamic_contr <- weight_b * channels[pod, svc].traffic
8.          pScore <- pScore + static_contr + dynamic_contr
9.      End For
10.     If cNode == node:
11.         score <- score + weight_c * pScore
12.     Else:
13.         weight_delta <- 1/nodeLatencies[node, cNode]
14.         score <- score + weight_delta * pScore
15.     End If
16. End For
17. Return score
```

Figure 3.2. Node Scoring Algorithm

We modified the scoring algorithm according to the needs of our environment. More details are presented in 3.4. We chose this work for the simplicity of its solution, as well as its heuristic nature and we believe that it can provide the groundwork for dynamic and distributed solutions in the future. Lastly, although they proposed it, Marchese and Tomarchio do not present an implementation of their work, nor include any quantitative results in their experiments.

3.1 Cluster Architecture

For our experiments, we set up a Kubernetes Cluster in the Google Cloud Platform (GCP). The cluster is controlled by the Google Kubernetes Engine (GKE), while its nodes are Virtual Machines (VMs) that belong to the Google Compute Engine (GCE). The cluster contains four identical nodes in terms of hardware and software that belong to a Node Pool. We have disabled all autoscaling and load balancing features for the sake of integrity on our experiments.

In tangent with the cluster we use Istio as our service mesh. The Istio Control Plane (istiod) is deployed in a random node and proxies are configured to run on the default namespace, which is where we deploy our benchmark applications. We also enable all telemetry addons using istioctl, those being Prometheus, Kiali, Grafana and Jaeger.

Every Kubernetes Node has a specific amount of resources designated by the Node Pool, and therefore can host a certain number of Pods, depending on the requirements of their Deployments. Kubernetes Deployments are configured in yaml files, which contain their resource requirements, their container image and more. For every Deployment, there is also a respective Kubernetes Service that handles all traffic for it (2.1.3). Each Pod contains a single microservice, represented as a Deployment, while Istio also injects an Envoy Sidecar Proxy.

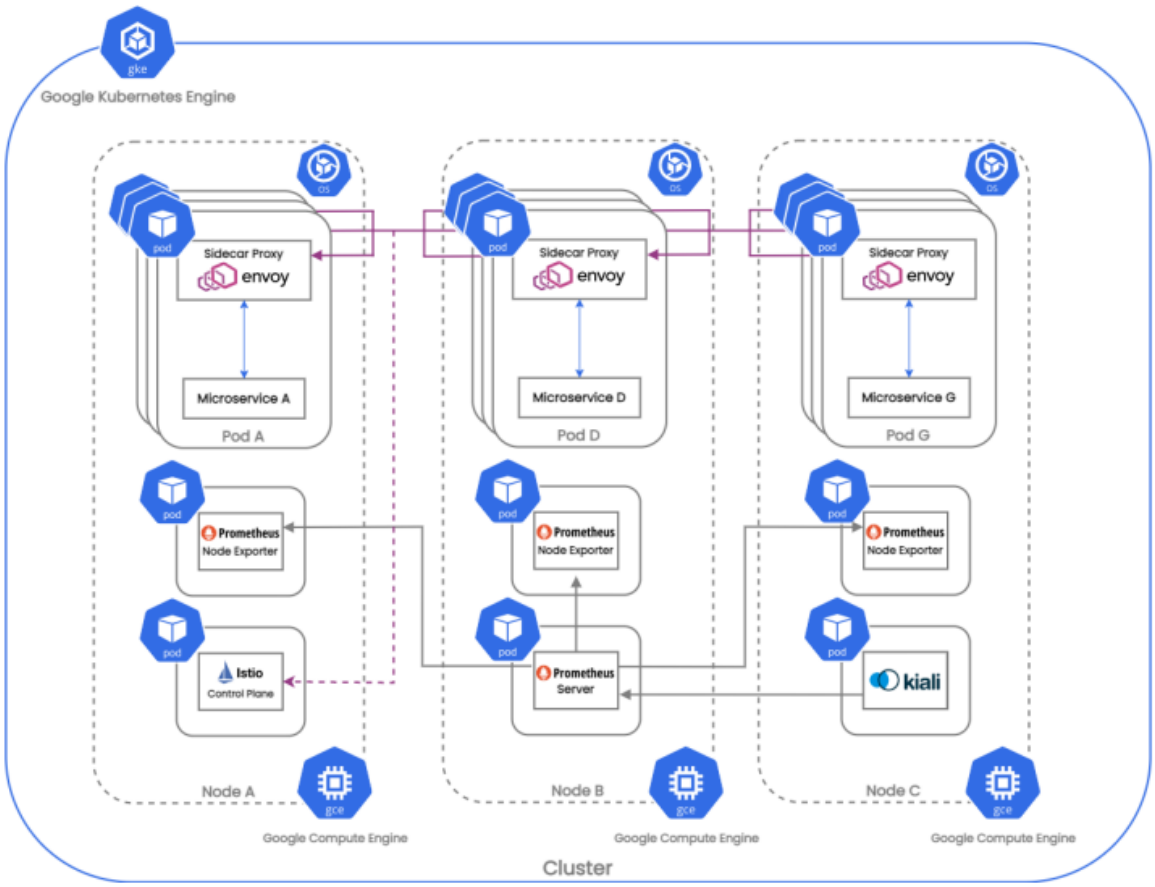


Figure 3.3. Cluster Architecture in GCP with Istio and Prometheus

The proxies are tasked with forwarding all of the Pod's traffic as well as scraping data and sending them to Istiod (Istio Control Plane). More precisely, the proxies forward incoming requests to the Pod to its microservice via the service's Kubernetes Service, while it forwards outgoing requests to the target Pod to be processed by its respective proxy.

External traffic is allowed through the "frontend" services of each application. The Kubernetes Services for these are of NodePort type. We use these to send requests using our load testing tool (Locust) to stress the applications and perform the experiments.

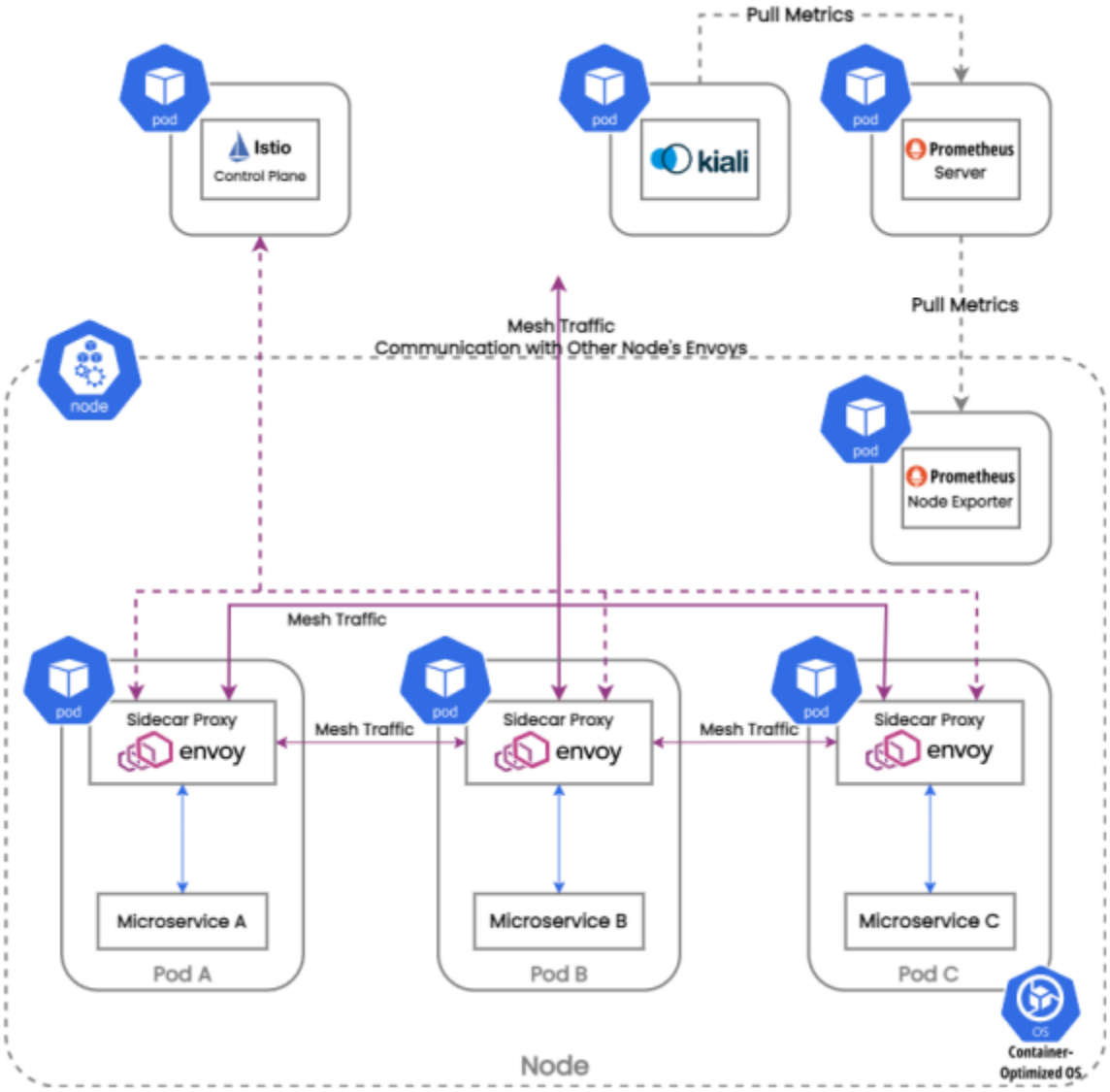


Figure 3.4. Node Architecture in GCP with Istio and Prometheus

3.2 Communication Criteria

Our communication-aware scheduler considers several criteria of communication when making a placement decision about a microservice. This section aims to present each criteria evaluated in the scoring process.

Requests per Second (RPS)

One of the metrics considered is the Requests per Second (RPS), which is the total traffic in requests between two services per second. We acquire this metric via the Prometheus API. Prometheus calculates the metric by summing all requests over a specified time period of a few seconds and dividing by that period. Effectively, the RPS value is the mean of total requests made over a time period x. Furthermore, Prometheus provides the RPS metric for requests from one service to another. Therefore, as we want to calculate all traffic incoming and outgoing for two services we perform two of these requests per pair of microservices.

$$RPS(S_i, S_j, x) = RPS_{S_i \to S_j}(x) + RPS_{S_j \to S_i}(x)$$

$$Where\ S_i\ and\ S_j\ are\ two\ arbitrary\ services\ i\ and\ j, x\ is\ the\ time\ in\ seconds\ and:$$

$$RPS_{S_i \to S_j}(x) = \frac{\sum_x Requests\ from\ S_i\ to\ S_j}{x}$$

Where:

- S_i is the Source service
- S_j is the Destination service
- x is the Total Seconds of Measurement

Lastly, it is important to mention that the RPS metric is accurate and therefore only used for HTTP and gRPC protocols and we use a different metric for TCP protocol communication, which we will go over later in this section.

Bytes per Second (BPS)

Very similar to the RPS metric, we use the BPS metric in order to account for TCP protocol communication. As we will show later in 3.4, both protocols are considered by the scoring algorithm, depending on which protocol the two services use to communicate.

Communication Protocol

The last criteria considered is the Communication Protocol the two services use. We acquire it from the same PromQL queries we use to get the RPS and BPS metrics. The queries are

done to the Prometheus API and it returns the communication protocol, the RPS or BPS depending on the query, as well as several other key:value pairs in JSON format. We use the communication protocol as a weight to the overall traffic between the two services, favoring HTTP communication over gRPC and gRPC over TCP. We will go into more detail when we analyze the scoring algorithm in section 3.4.

3.3 Scheduler Initialization

In this section we will go in depth into the initial state of the system before applying the scheduler and why that is important, as well as the process the scheduler follows before scoring.

3.3.1 Initial Deployment

Our scheduler uses already deployed microservices to determine the score of each node and therefore to make a placement decision for each service that has not been deployed. Therefore, the initial placement of the application's microservices can affect the final placement. In order to maintain integrity in our experiments we only deployed the essential microservices of each application, those being their "frontend" microservice and any service requiring a database connection. Furthermore, we placed the "frontend" service in a different node from the "database" services. The rest of the services are placed one by one by our scheduler. For each service the scheduler considers all the previously deployed services.

3.3.2 Node Filtering

Our scheduler takes all non-deployed microservices and scores each node to decide which one is more suitable for that microservice to be placed. Unfortunately, Nodes do not have an infinite amount of resources, therefore there will be times the Nodes will not have the resources to host the microservice, even if it is the best fit. To get around this issue, a filtering phase occurs before the scoring phase, where the Nodes that do not have enough available resources to fit the new microservice are not accounted for. The filtering is performed by reading the deployment yaml file, which includes the resources the deployment requires to function. Then, the available resources of each node are calculated by reading their maximum available resources and applying a percentage multiplier that can be configured when creating the scheduler, which indicates how much of its resources we want to allow the Node to use. This is important as fully utilizing all of a Node's RAM and CPU could slow down the processes running on that Node and therefore increase latency or even cause Pods or the Node to shut down and restart. Finally, from that number the resources the services deployed in the Node require are subtracted.

$$RAM_{available} > RAM_{requested}$$

$$RAM_{available} = a * RAM_{max} - \sum_{deployed} RAM_{utilized}$$

$$CPU_{available} > CPU_{requested}$$

$$CPU_{available} = a * CPU_{max} - \sum_{deployed} CPU_{utilized}$$

Where ***a*** is the percentage modifier.

The maximum resources available in the Node as well as the microservices that are deployed in them and the resources they utilize are all requested by the Kubernetes API from the scheduler.

3.4 Scoring Algorithm

The Communication-aware Scheduler is based on the proposed scheduler of [ref]. Our version is based on a cloud environment instead of the original's fog-edge one, so we removed some elements of their proposed algorithm. The scheduler's full functionality as well as its subroutines are listed below in pseudocode. During initialization, the scheduler connects with the Kubernetes cluster and Prometheus, as they both provide vital information for the function of the Scheduler. Then, the Scheduler acquires all deployed Kubernetes Services and Nodes in the cluster and performs a series of nested loops. First, it loops until there are no Services left to deploy and then for each Service it filters the Nodes for scoring. Lastly, it scores the filtered Nodes and places the Service to the one with the highest score.

Algorithm 1: Communication-Aware main process

```
1.  Read kubectl configuration
2.  Establish Prometheus connection
3.  cluster_nodes <- getNodes
4.  services <- getServices
5.  While services:
6.    For svc in services:
7.      filtered_nodes <- filterNodes(svc, cluster_nodes)
8.      top_score <- 0
9.      top_node <- None
10.     For node in filtered_nodes:
11.       score <- scoreNode(svc, node)
12.       If score > top_score:
13.         top_node <- node
14.         top_score <- score
15.       End If
16.       If top_node is not None and top_score != 0:
17.         services.remove(svc)
18.         Deploy(svc, top_node)
19.       End If
20.     End For
21.   End For
```

Figure 3.5. Communication-Aware main process

Scoring

To calculate the score of a Node, the scheduler first acquires all deployed Services of that Node via the Kubernetes API. Then, for each Service it adds a static contribution and a dynamic contribution. The static contribution's value is based upon the communication protocol between the two services. The dynamic contribution is the traffic between the two services at the given moment, measured in RPS or BPS (see 3.2). The information for both is obtained via the Prometheus API using specific queries.

Algorithm 2: Node Scoring Algorithm

Inputs: svc, node
Outputs: score

```
1.  score <- 0
2.  services <- getNodeServices(node)
3.  For service in services:
4.      channel <- getChannel(svc, service)
5.      channel_prio <- getPriority(channel.get(request_protocol))
6.      static_contr <- weight_a * channel_prio
7.      channel_traffic <- channel.get(requests)
8.      dynamic_contr <- weight_b * channel_traffic
9.      score <- score + static_contr + dynamic_contr
10. End For
11. Return score
```

Figure 3.6. Node Scoring Algorithm

The original algorithm proposed an extra step in the scoring process. For each Node that was scored the above contribution was calculated for all Nodes and summed, but if the Node was not the one being scored, its total contribution was divided by the latency between it and the Node being scored. That way, in a fog-edge environment a Service would consider more than one layer of infrastructure. In our case, in a cloud environment, the Nodes have practically no distance between them, performing that extra step would make the scheduler's function obsolete. More specifically, as the latency between Nodes is zero, the scheduler would count all traffic between the Service being placed and all deployed Services in all Nodes, effectively not differentiating between any Node.

3.5 Deployment

When deploying a Service, what's really happening in the background is we're asking the default Kubernetes scheduler via Kubernetes API to schedule the Service. In order to schedule a Service, the scheduler needs a configuration file written in yaml, called a deployment file. Deployment files contain vital information about the specifications of the Service, the image it contains, its required resources and more. Our scheduler has access to all the deployment files of each application in order to pass them to the default scheduler when deploying a Service. The files also contain specifications about where the scheduler should place them. Our scheduler alters those specifications according to the decision it has made, so that it will force the default scheduler to place them in the chosen Node.

There are multiple ways to do this, including soft and hard placement specifications. In our work, we used nodeSelector, which is a hard placement decision maker, in order to test our method in a vacuum. To be more precise, a soft method, such as nodeAffinity, uses weights to place Pods in the Node of the highest weight. The problem is that the default scheduler has its own scoring methods that could interfere with our scheduler's decision.

During the Kubernetes cluster's creation, we label each Node. Our scheduler reads these labels via the Kubernetes API and when it makes a placement decision, it adds that label as a nodeSelector on the Deployment file of the Service it is about to place. The default scheduler then has to match the created Pod's nodeSelector with the Node's label, forcing it to be placed in the Node chosen by our scheduler. Below we show an example of a nodeSelector on a Deployment yaml file.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
      annotations:
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
    spec:
      nodeSelector:
        name: edge1
```

Figure 3.7. Example of nodeSelector in a deployment YAML file

3.6 Benchmark Application

In this section we will go into detail about the application we use in our experiments. We apply our scheduler on Google's Online Boutique e-Shop to test the efficacy of our proposed placement strategy. It contains a total of 11 microservices, while using the HTTP and gRPC communication protocols. The infrastructure for the experiments is explained in full in the next chapter.

3.6.1 Google's Online Boutique e-Shop

Google's Online Boutique e-Shop is a demo application created by Google to showcase and experiment with cloud native technologies such as Kubernetes, Anthos Service Mesh (Istio), gRPC protocol and more. Furthermore, it allows developers to familiarize themselves with microservice architectures. It can be easily set up in any Kubernetes cluster.

The application contains 12 microservices, of which we use 11, that all communicate using the gRPC protocol. Each microservice performs a different and distinct function, following standard microservices architecture. More detail about the architecture and the services, as well as all required yaml files and instructions are included in the official GitHub page [ref]. Below we will present the microservices and architecture of the application:

- **Frontend Service** (Go): Exposes an HTTP server to serve the website. Does not require singup/login and generates session IDs for all users automatically.
- **Cart Service** (C#): Stores the items in the user's shopping cart in Redis and retrieves it.
- **Redis Cart**: Redis database. Cart Service stores its data in this in-cluster Redis database.
- **Product Catalog Service** (Go): Provides the list of products from a JSON file and the ability to search products and get individual products.
- **Currency Service** (Node.js): Converts one money amount to another currency. User real values fetched from European Central Bank. It's the highest QPS service.
- **Payment Service** (Node.js): Charges the given credit card info (mock) with the given amount and returns a transaction ID.
- **Shipping Service** (Go): Gives shipping cost estimates based on the shopping cart. Ships items to the given address (mock).
- **Email Service** (Python): Sends users an order confirmation email (mock).
- **Checkout Service** (Go): Retrieves user's cart, prepares orders and orchestrates the payment, shipping and email notification.
- **Recommendation Service** (Python): Recommends other products based on what;s given in the cart.
- **Ad Service** (Java): Provides text ads based on given context words.
- **Load Generator** (Python): Continuously sends requests imitating realistic user shopping flows to the frontend.

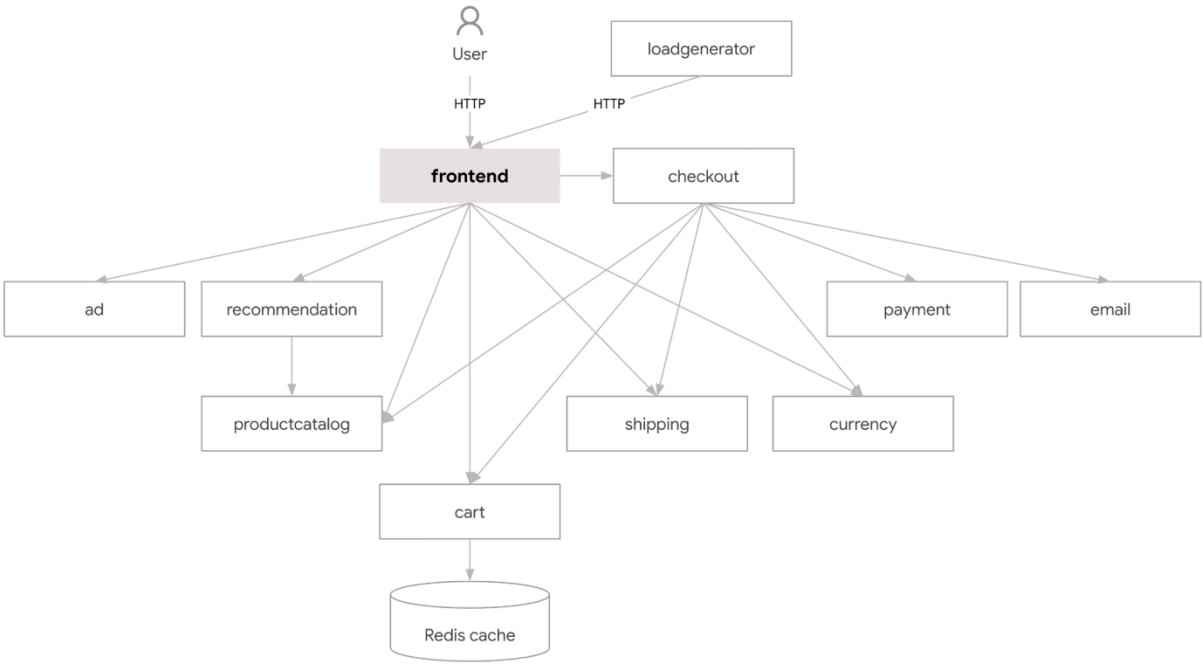


Figure 3.8. Google's Online Boutique e-Shop architecture

To control the workload with which we stress our application and by extension uphold the integrity of our experiments we forgo the deployment of the load generator service, instead using our own load stressing tool (see 2.4).

Chapter 4

Experimental Results

In this Chapter we will go in depth on the experiments performed and analyze their results, as well as the system infrastructure and its cost. More specifically, we will show the specifications and cost of our infrastructure, the workload applied to our benchmark application, the service placement strategy used and the results of the experiments. This work has two objectives, to reduce the cost of infrastructure by lowering the necessary nodes for the deployment of an application and reducing the traffic between Nodes, as well as the optimization of application response times by placing high communication services in the same Node. According to these, we will present four measurements as the results of the experiments: Number of Hosts, Egress Traffic, Total Infrastructure Cost and Response Time.

4.1 System Infrastructure

The configuration of the Kubernetes Cluster is done as described in Chapter 2, using Terraform. The Cluster runs on the Google Kubernetes Engine in the 1.27.11 version, which is the latest stable one, in the europe-west1 region. All autoscaling and load balancing features are disabled to maintain stability of resources and the integrity of the experiments. We have also turned off Anthos Service Mesh and Managed Service for Prometheus, in favor of the custom implementation of Istio Service Mesh and Prometheus.

Cluster Attributes	Options
Cluster Version	1.27.11
Horizontal Autoscaling	Disabled
Vertical Autoscaling	Disabled
HTTP Load Balancing	Disabled
Location Type	Zonal
Zone	europe-west1-b
Cloud Logging	Disabled
Cloud Monitoring	Disabled

Table 4.1. Cluster Attributes Configuration

The Cluster contains four Node Pools, each containing a single Node and they are all connected to a single control plane. We chose this approach to control the configuration specifications of each Node individually, although the final environment is homogeneous.

The VMs are type e2-standard-2. More specifically, they contain 2 vCPUs, 8 GB of RAM and a standard boot disk of 40 GB for storage purposes. Finally, they have disabled autoscaling and a Linux-based container-optimized OS image.

Node Pool Attributes	Options
Machine Type	e2-standard-2
CPU	2 vCPU
RAM	8 GB
Boot Disk	40 GB
Image	Container-Optimized OS
Zone	europe-west1-b

Table 4.2. Node Pool Attributes Configuration

4.2 Benchmark Application Stressing

The stress testing for our experiments was performed using the python-based tool Locust, explained in detail in Chapter 2. It was deployed on our host machine and loaded the application endpoints, simulating multiple concurrent users. As the requests were generated, our scheduler calculated the communication between placed and unplaced services, placing each service depending on the current load and traffic between services. After service placement is complete and the application is running smoothly, we run the same stress testing process to extract the response times for each request. Additionally, Prometheus queries are performed to acquire the traffic between services. Lastly, the egress traffic is calculated by summing the traffic between all services belonging to different Nodes.

4.2.1 Google e-Shop Stress Testing

To test Google's online boutique e-shop, we stressed it with the simulated load of 150 concurrent users applying 24.028 requests with a rate of 30.5 RPS. The request distribution among the application's endpoints is shown below in Table 4.3.

Request Description	Type	Distribution	# Requests
Visit Homepage	GET	4%	1.193
Show items in cart	GET	13%	3.177
Get a Product	GET	56%	13.509
Add item to cart	POST	13%	3.025

Submit an order	POST	4%	1.065
Change currency	POST	9%	2.059
Total Requests	-	-	24.028

Table 4.3. Table of request distribution

4.3 Placement Strategy

Three different placement strategies are tested and compared for different reasons. We compare the newly made communication-aware method with the default Kubernetes Scheduler placement as the lowest benchmark. We also compare it to the MODSOFT fuzzy clustering placement strategy, to analyze how it stacks up against an advanced placement strategy. In our strategy, there are variables determining the percentage of resources each Node is allowed to use to place microservices. We set these at 0.9 or 90% to maximize the efficiency of our strategy, without compromising on Node performance.

Below, two figures are presented, each visualizing the final placement of microservices produced by the default Kubernetes Scheduler and our communication-aware placement strategy respectively. We only show the Nodes in which services have been deployed and represent egress traffic between them with blue arrows. To be more precise, both clusters have 4 Nodes, but in our implementation only 3 are required and the last could be removed.

“cloud”	“fog1”	“fog2”	“edge1”
emailservice redis-cart shippingservice	cartservice paymentservice	adservice currencyservice recommendationservice	checkoutservice frontend productcatalogservice

Table 4.4. Default scheduler’s placement of e-Shop

“cloud”	“fog1”	“fog2”	“edge1”
redis-cart cartservice checkoutservice frontend productcatalogservice	recommendationservice productcatalogservice adservice currencyservice	–	frontend emailservice paymentservice currencyservice productcatalogservice shippingservice

Table 4.5. MODSOFT’s placement of e-Shop

“cloud”	“fog1”	“fog2”	“edge1”
checkoutservice emailservice redis-cart shippingservice	paymentservice	–	adservice cartservice currencyservice frontend productcatalogservice recommendationservice

Table 4.6. CAP’s placement of e-Shop

4.4 Infrastructure Cost Function

In this section, the function that calculates the cost of infrastructure for our experiments will be presented. The function follows the GCP pricing documentation [ref]. The most important resources upon which GCP charges are the total CPU and RAM of the machines in the cluster, which vary based on machine type and region. Egress traffic is also charged based on traffic exchanged between different Nodes in the cluster. More factors are considered, such as storage space, but this is considered negligible in our work, since we use applications with very low to no storage requirements. Below is a table showing the price of the resources mentioned according to the machine type we used in our work, which is the e2-standard type.

Resource	Cost (USD)	Cost (USD)
vCPU	$ 0.023993 / vCPU hour	$ 17.51489 / vCPU month
RAM	$ 0.003216 / GB hour	$ 2.34768 / GB month
Egress Traffic	$ 0.01 / GB	$ 0.01 / GB

Table 4.7. Hourly and Monthly Cost of resources and egress traffic in USD

We specifically used the e2-standard-2 machine type, which each contain 2 vCPUs and 8 GBs of memory. The table below illustrates the hourly and monthly cost of each machine (Node).

Resource	Cost Hourly (USD)	Cost Monthly (USD)
vCPU & RAM total	$ 0.073714 / hour / Node	$ 53.81122 / month / Node

Table 4.8. Hourly and Monthly Cost of an e2-standard-2 Node

Network Traffic is charged per cluster, based on the traffic exchanged between Nodes (egress) in GBs. Another factor for the cost of network traffic is the location of each Node. In our work, all Nodes are deployed in the same region, europe-west1, and therefore all traffic

between them is charged at the same rate, as illustrated in Table 4.7. To calculate the total egress traffic we use Prometheus to draw the total requests between services and sum the ones that belong in different Nodes.

Finally, the total cost for the cluster for *n* number of Nodes is given using the formula below:

$$TotalCost(\$) = n * TotalResourceCost_{e2-standard-2} + 0.01 * GB_{egress} \quad (4.1)$$

Where the hourly cost is given by:

$$TotalCost\left(\frac{\$}{hour}\right) = n * 0.073714 + 0.01 * GB_{egress}(hourly) \quad (4.2)$$

And the monthly cost by:

$$TotalCost\left(\frac{\$}{month}\right) = n * 53.81122 + 0.01 * GB_{egress}(monthly) \quad (4.3)$$

Below is a table illustrating the total hourly and monthly cost for 1, 2, 3 and 4 Nodes, where GB refers to GB of egress traffic at that time period:

Number of Nodes (n)	Hourly Total Cost (USD)	Monthly Total Cost (USD)
1	0.073714 + 0.01 x GB	53.81122 + 0.01 x GB
2	0.147428 + 0.01 x GB	107.62244 + 0.01 x GB
3	0.221142 +0.01 x GB	161.43366+ 0.01 x GB
4	0.294856 + 0.01 x GB	215.24488+ 0.01 x GB

Table 4.9. Hourly and Monthly Costs per Node in the cluster

4.5 Results of Placement Strategy

In this section we will present the results of our communication-aware placement strategy, while comparing them with the default Kubernetes scheduler as a baseline comparison and the MODSOFT placement strategy to determine how it weighs up against an advanced placement strategy. More specifically, we will compare the number of hosts required by the placement methods, egress traffic and total infrastructure cost that follows from the previous two metrics.

Lastly and most importantly, a comparison between the response time of the applications will be illustrated according to each placement strategy. We refer to our method as CAP, short for Communication-Aware Placement.

4.5.1 Number of Hosts

As illustrated in Section 4.4, a major factor impacting the Total Infrastructure Cost is the number of utilized Nodes, represented by the *n* in Equation 4.3. In our experiments, we created clusters of four Nodes. Below, we show how many were utilized by each placement method.

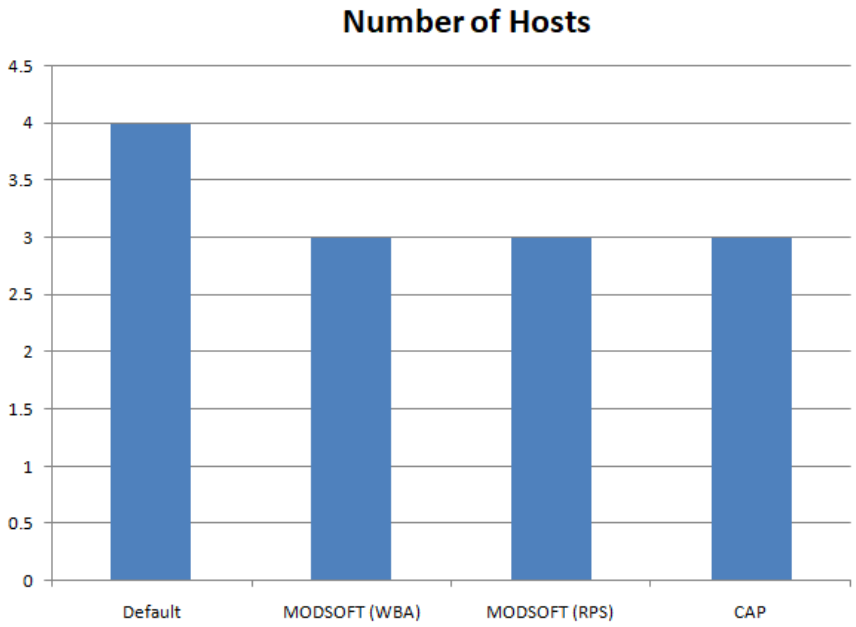


Figure 4.1. Number of utilized Hosts by each method for Online Boutique e-Shop

The CAP method produced a placement utilizing 3 Nodes, which is less than the default placement and ties with the MODSOFT - HP method for both of its affinities. It is worth noting that MODSOFT is a fuzzy clustering method, meaning it produces more replicas of Pods, requiring more resources overall. Although that is the case, as was shown in 4.3, the CAP method's placement only placed two Pods in the third Node, leaving it nearly empty. Therefore, with a slightly less resource intensive application or slightly "larger" Nodes it could produce a placement with less Nodes than MODSOFT, which is of course expected due to the fuzzy nature of the placement method.

4.5.2 Egress Traffic

Egress Traffic as we define it, being the traffic between microservices in different Nodes, is the second factor affecting Total Infrastructure Cost, as shown in section 4.4. In this subsection, the results of our experiments regarding egress traffic reduction will be presented. We utilized Prometheus queries to gather data regarding the traffic between each two microservices and summed the ones where the two Pods belonged in a different Node,

according to the placement shown in 4.3. In the figure below, the total requested MBs between Nodes for each placement is presented. The data is taken in the span of around 20.000 requests, as mentioned in Section 4.2.

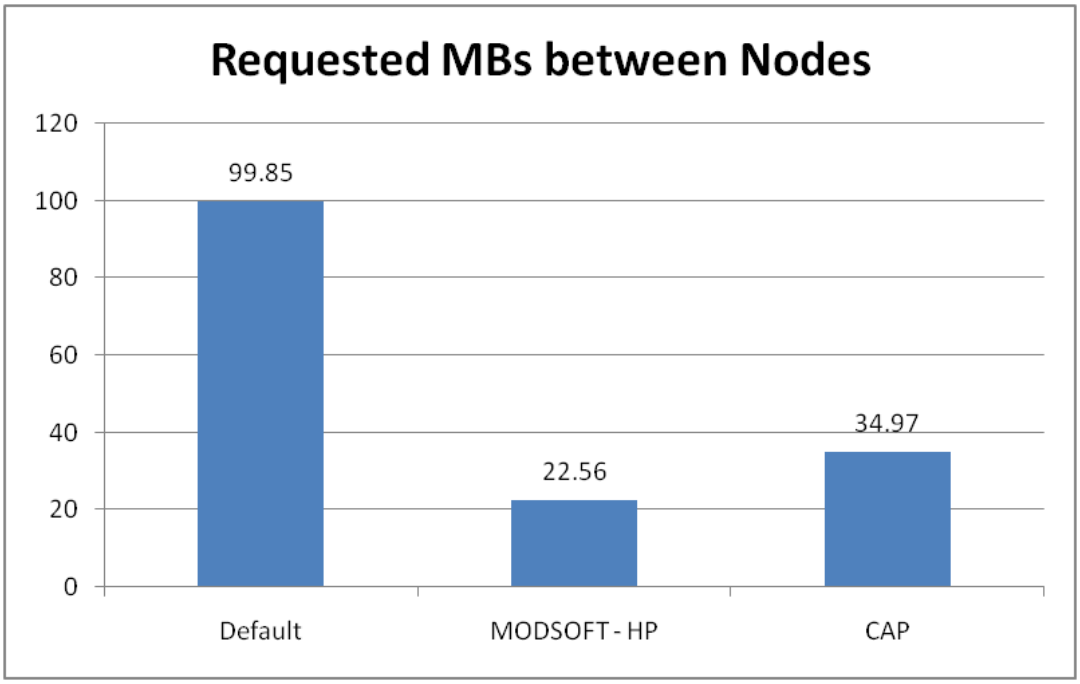


Figure 4.2. Request size in MBs between Nodes in Google's e-Shop

The graph above is useful, but the absolute MBs requested are dependent on multiple factors, such as the length of time of the experiment, the number of users and the activity of each user. Another very similar, but far more useful statistic is that of the percentage (%) reduction of egress traffic each placement achieves. We use the Default placement as a benchmark and the data above as a guide. The results are presented in the figure below.

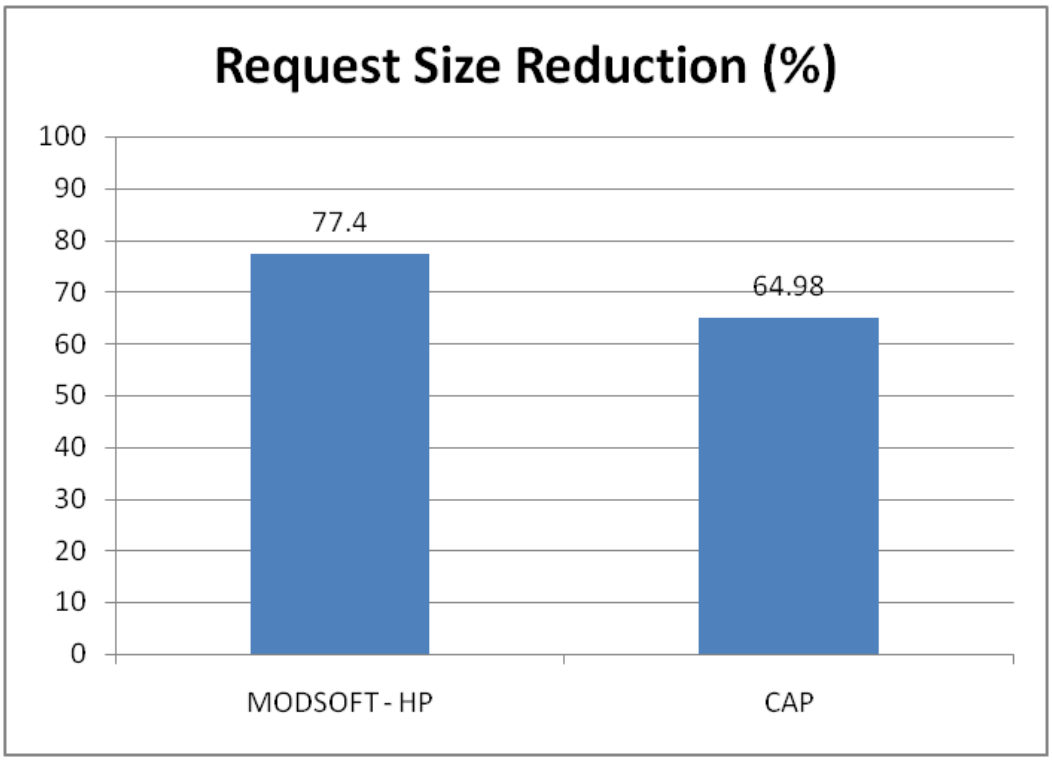


Figure 4.3. Percentage request size reduction between Nodes in Google's e-Shop

As presented above, MODSOFT-HP achieves a staggering 77.4% reduction in egress traffic, while CAP manages a lesser, but still massive 65% reduction in egress traffic. As expected, MODSOFT is superior, since it is a fuzzy method with multiple replicas of highly requested services. Even still, CAP is certainly a competitive method in the realm of egress traffic reduction, which in slightly different circumstances could also be utilized with less Nodes than MODSOFT.

4.5.3 Total Infrastructure Cost

In this subsection we will present the reduction in Infrastructure Cost each placement achieves. We will use the monthly cost equation presented in Section 4.4 (Equation 4.3) to calculate it. It is difficult to compare how much traffic reduction reduces the overall Cost of Infrastructure, as absolute numbers are dependent on the length of the experiment, the number of users and more factors that have to be extrapolated and equated. Therefore, we will approximate the total monthly egress, by projecting the same user activity throughout the whole month.

The experiment mentioned in Section 4.2 lasted approximately 20 minutes, with 150 concurrent users and an average of 30.5 requests per second and resulted in the egress traffic shown in Figure 4.2. Extrapolating for the whole month we get:

$$ExtrapolationFactor = E_f = \frac{minutes_{month}}{minutes_{experiment}} = \frac{30 * 24 * 60}{20} = 2160 \quad (4.4)$$

The final values being:

$$Egress_{Default} = E_f * 99.85MB = 2160 * 99.85MB = 215.68\ GB$$

$$Egress_{MODSOFT} = E_f * 22.56MB = 2160 * 22.56MB = 48.73\ GB$$

$$Egress_{CAP} = E_f * 34.98MB = 2160 * 34.98MB = 75.56\ GB$$

Using the values above and the Equation 4.4 we calculate the Total Infrastructure Cost of each method, as shown below:

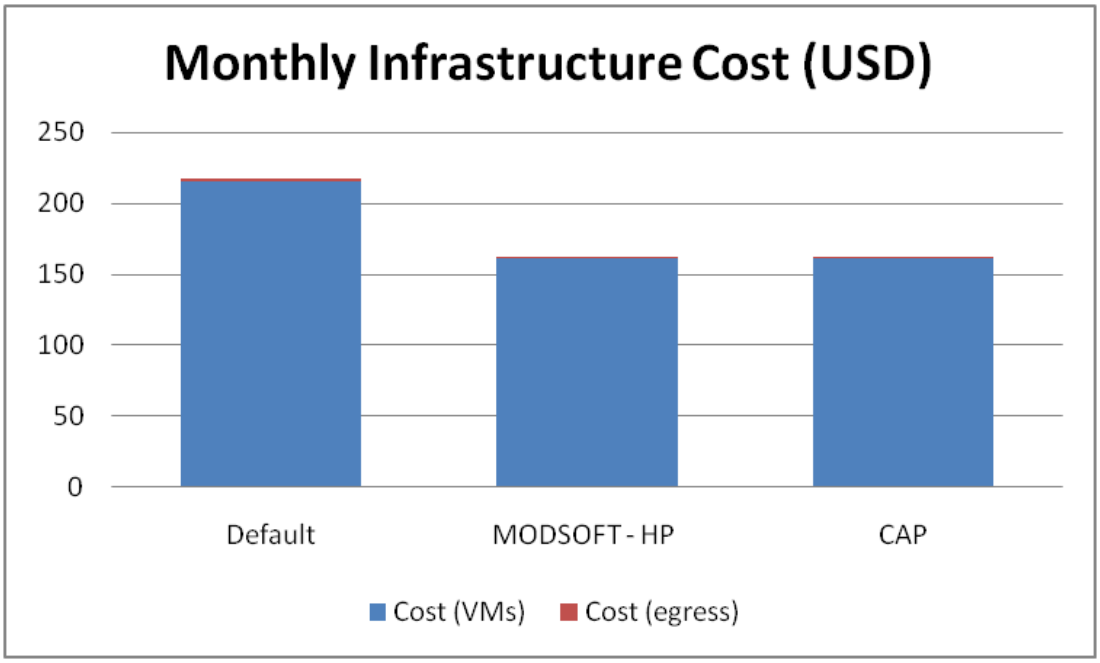


Figure 4.4. Monthly Infrastructure Cost for Google's e-Shop placement in USD

As we can see, the main factor determining the Total Cost of Infrastructure is the number of Nodes used in the placement. Even the most suboptimal method in terms of egress traffic (Default) would require 25 times the users (3750), assuming each user generates the same traffic on average, for the egress cost to be as high as a single Node. Therefore, the MODSOFT - HP and CAP methods are able to reduce the monthly cost by reducing the number of Nodes required in their placement. Since they both reduce the placement to three Nodes and achieve a similar level of egress traffic reduction, which is negligible in terms of cost, they achieve the same monthly cost.

4.5.4 Response Time

In this last subsection of the experimental results, we will present the results of our main goal in this thesis, that of the response times of our CAP method, and how it compares against the Default and MODSOFT - HP placement methods. We measure the response times as the average response time of all requests made during the stress testing of the application,

the distribution of which is shown in Section 4.2, Table 4.3. Below, we present the average response time, the 90%ile response time and the 95%ile of the average request. The 90%ile and 95%ile refer to the 90% and 95% faster requests. We expect that CAP will be strictly better than the Default placement, but it will be slightly worse than MODSOFT-HP. That is because MODSOFT creates multiple replicas of highly demanded services, which helps these services perform better and handle more requests simultaneously.

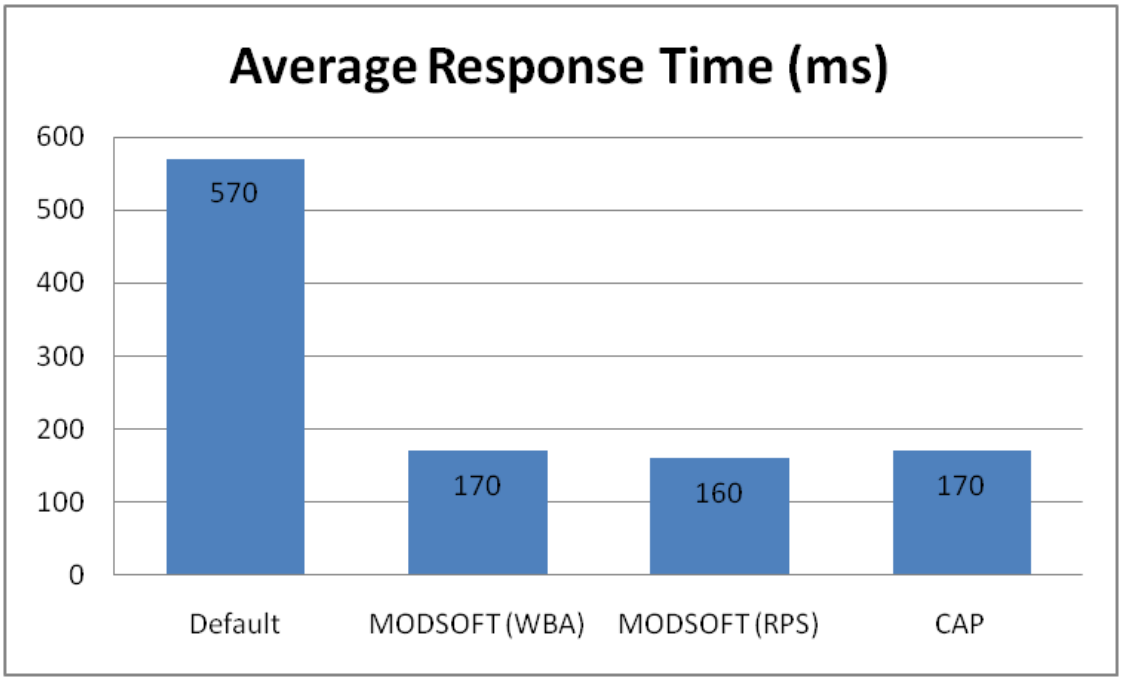


Figure 4.5. Average Response Time for Online Boutique e-Shop

As shown, both CAP and MODSOFT are far better at optimizing the average response time than Default. Contrary to our original prediction, however, the CAP method is as effective as MODSOFT using the WBA affinity for its placement and by an almost insignificant margin worse than MODSOFT using the RPS affinity so far. We still expect MODSOFT to be even better at the 90%ile and 95%ile response times, where the highly stressed services will create a bottleneck in our placement.

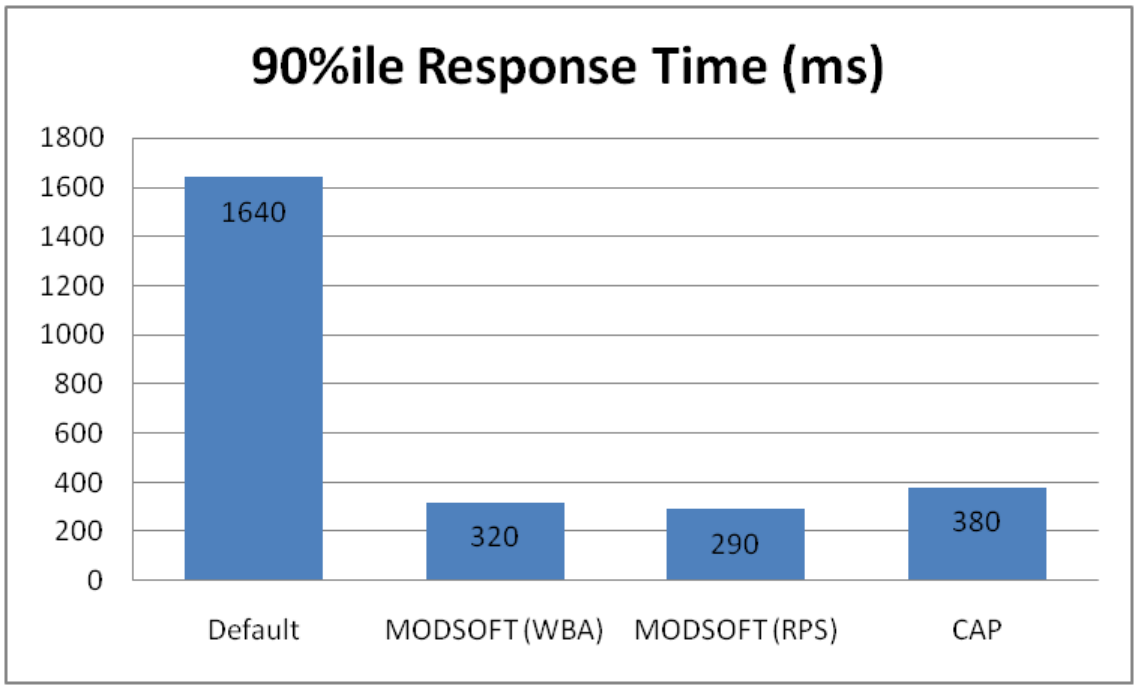


Figure 4.6. 90%ile Response Time for Online Boutique e-Shop

As we predicted, in the 90%ile response times, we can already see that MODSOFT is clearly better than CAP when it comes to managing the workload of highly stressed services. Below, we present the 95%ile response times and then our final conclusions on this topic.

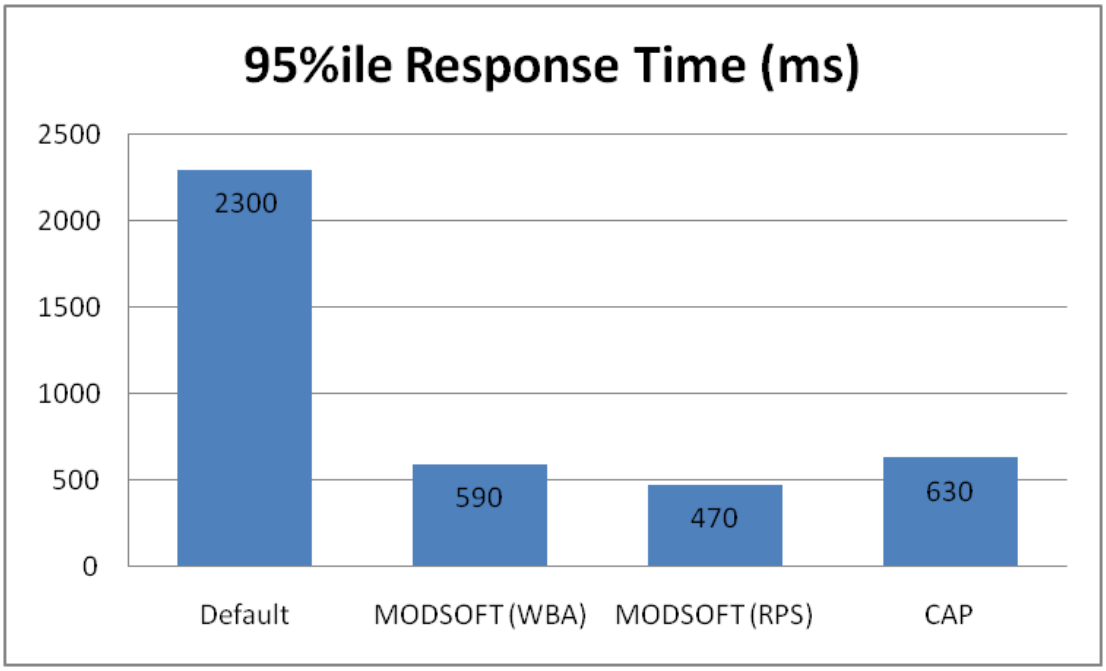


Figure 4.7. 95%ile Response Times for Online Boutique e-Shop

Finally, we can see that the 95%ile is very similar to the 90%ile case. As stated above, we can see that a very important factor in lowering an application's response time is the ability to create multiple replicas for stressed services, and not lowering egress traffic, as it will split the workload and allow it to perform better. Furthermore, it will alleviate bottlenecks that a stressed service might present. To not do so is to not take advantage of one of the most

important strengths in microservices architecture, that being the easy scalability that it provides.

4.5.5 Response Time Relationships

For future work, we performed additional experiments to determine the relationship between the response time of an application depending on both the load placed on it and the load placed on the Nodes. More specifically, we wanted to determine if there is a correlation between the percentage of resources used in the Nodes of a cluster and the response time of an application. For this purpose, we completed a series of additional experiments separately, where we stressed the application in a similar way as before. In each iteration, we used the CAP placement method, but altered the parameters of resources used to allow Nodes to be “filled” at 70% cpu and ram, 80%, 90% and 95%. Furthermore, we did three different iterations for each resource utilization, one with 150 concurrent users, as the original experiments, one with 300 and one with 500. We performed around 10.000 requests each time. We expect to find an increase in response times with more users, as well as with more resources used. Below are the results in three figures, showing the average, the 90%ile and the 95%ile of response times.

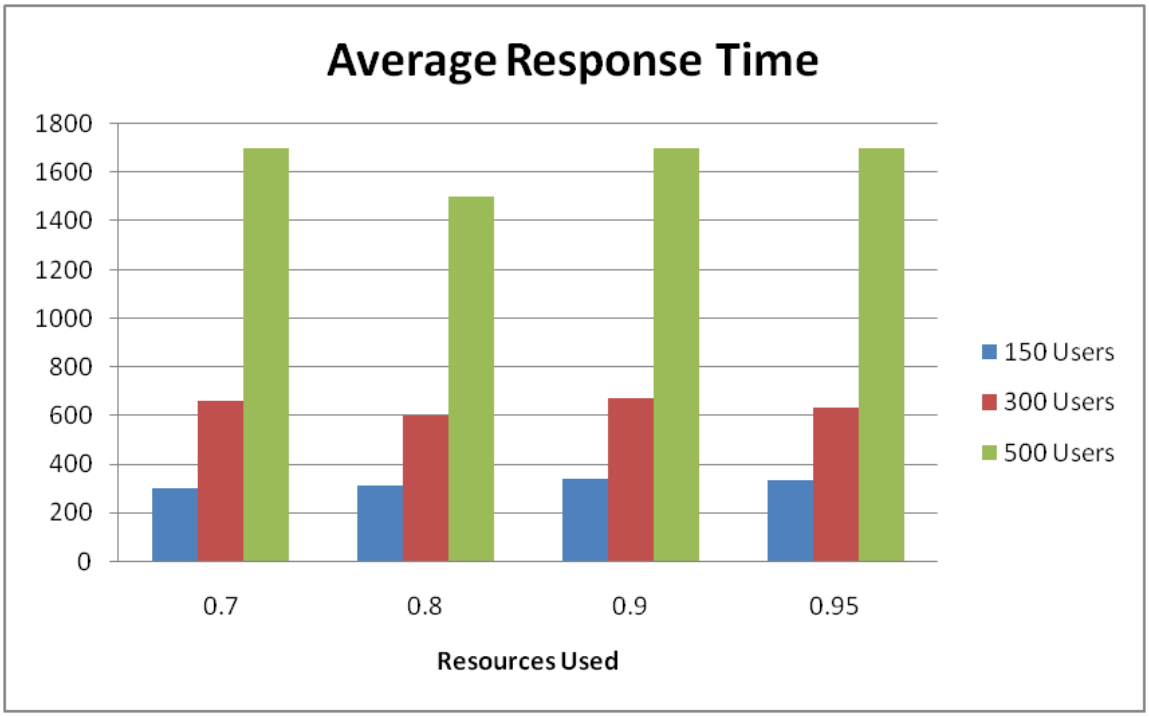


Figure 4.8. Average Response Time of e-Shop by concurrent users & resources utilized

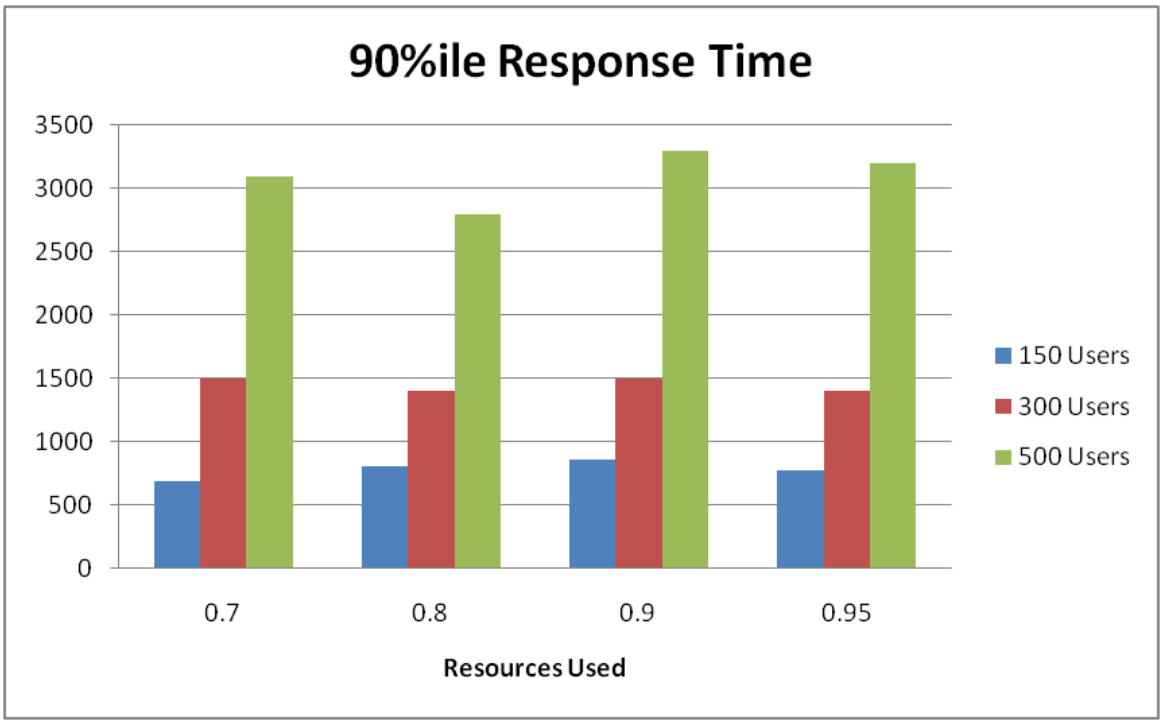


Figure 4.9. 90%ile Response Time of e-Shop by concurrent users & resources utilized

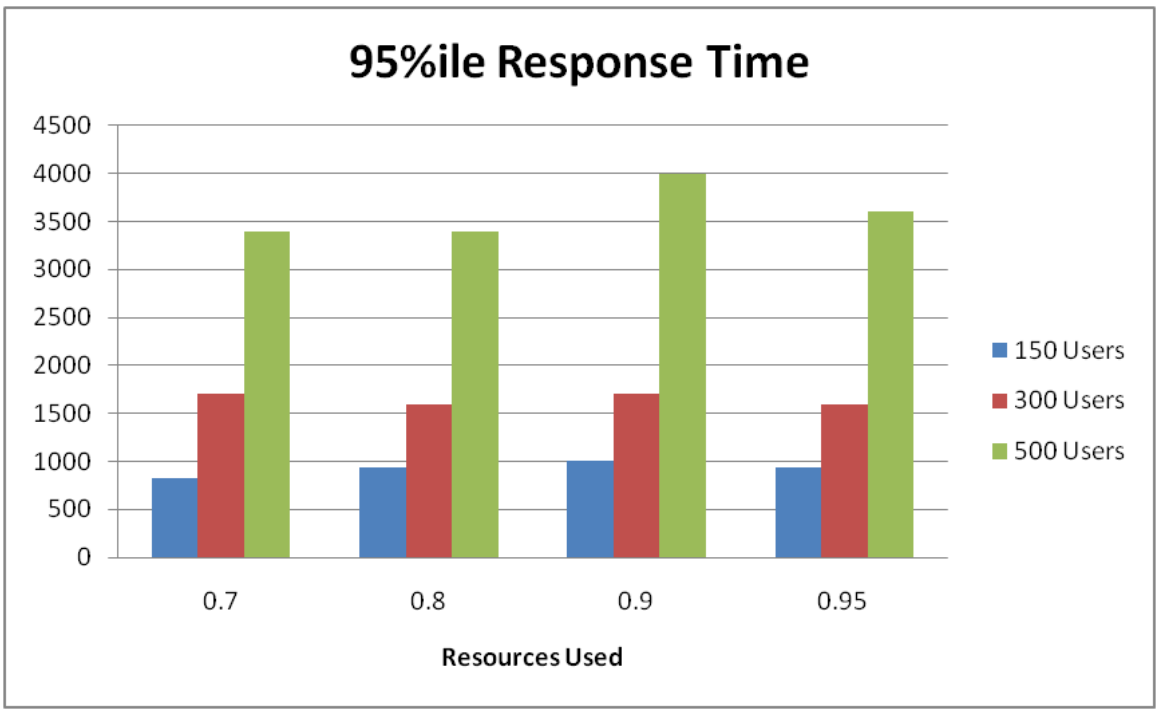


Figure 4.10. 95%ile Response Time of e-Shop by concurrent users & resources utilized

We can see right away that with a greater workload (more concurrent users) the response times skyrocket. On the other hand, the data do not show a statistically significant increase of latency with more utilized Node resources. It is important to note that the total resources used are the same in each case, but simply allocated in less Nodes. From this we can conclude that what affects response times is the ability to allocate additional resources in stressed services. Another important note is that in the experiment only the CPU of the Nodes reached the critical thresholds and not the RAM. It is possible that if the RAM reached these critical levels, the outcome would be different. In the above experiments,

increasing the load placed on the application, significantly increased the response times. Therefore, if we increased the resources used by a stressed service by creating another Pod, we would decrease response times. Lastly, although it is important to not completely “fill” a Node to avoid it crashing, there is no evidence to suggest that maximizing its resources would affect application performance.

Conclusions and Future Work

This work aimed to test the efficacy of a heuristic placement method, our main goal being reducing the response time of applications deployed in the cloud and secondarily reducing the cost of infrastructure. We showed that our heuristic placement method was competitive in the realms of egress traffic reduction, monetary cost reduction and response time reduction with an established fuzzy clustering method. More specifically, CAP achieved the same number of hosts as MODSOFT with three hosts, one less than the Default, as well as the exact same monetary monthly cost of 162 USD, a 25% reduction from the Default. In terms of egress traffic it achieved a significant 65% reduction compared to MODSOFT's even greater 77%. Most importantly, in response time reduction it achieved a 70-77% decrease compared to MODSOFT's 72-82%. It is apparent that MODSOFT having multiple replicas of certain services gives it a significant advantage in these experiments. However, due to its nature, MODSOFT would be limited in a more practical setting, where scaling of services would occur. Firstly, it does not support scaling, as it produces a placement with a maximum of one replica of each service per host. Secondly, if a replica were to be introduced or one to be destroyed, the algorithm would have to be executed again to produce a new optimal placement. In contrast, CAP is designed as an extension to the Default Kubernetes Scheduler and can be used in tandem with scaling, as well as other dynamic mechanisms. More specifically, CAP could take newly created replicas and place them optimally in the cluster, without requiring the total restructuring of the cluster. We should note that replicas can still be created when using MODSOFT, but they will be placed by the Default scheduler, without accounting for communication between the other services of the cluster.

We utilized Google's Online Boutique e-Shop as our benchmarking application for our experiments and locust as our stressing tool. We configured Istio as our service mesh and Prometheus as our data scraping and monitoring tools in our cluster, while our scheduler connected with Prometheus to gain access to critical data for the implementation of our strategy via queries in the Prometheus API.

For the response time of applications, by using our experimental results and those of previous studies, we have determined a few factors that are required to be present in a method for its reduction. First, the method has to have the ability to create more replicas of stressed services for better distribution of load and to avoid bottlenecks. Second, since we didn't find a correlation between resource overload of the Nodes and response time, it is important that the method maximizes resource utilization, without fully loading the Nodes to avoid crashes. Third, the method has to have the ability to dynamically react to the changes in the distribution of the load. In this work, we did not focus on changing the distribution of the load, but in such a case, we can determine that the advantage that a method such as MODSOFT has by creating replicas of stressed services would be lost if different services became stressed. Fourth, the method has to create a placement that reduces the Node-to-Node communication, or as we refer to it in this thesis; egress traffic. Although our experiments did show correlation between egress traffic reduction and response time reduction, they are also limited in their environment. In our experiments, all Nodes reside in the same region of the same zone and in the same availability zone, minimizing the physical distance between them and therefore the latency that would be created by sending a request

from one to another. Even in a purely cloud environment, if two Nodes were placed in different regions or availability zones, the results would be more significant, while in a fog/edge cloud environment, this factor would be even greater.

In terms of infrastructure cost, as shown in our last experiment, with a significant increase of load, the response times catapult. Therefore, it is not a stretch to say that egress traffic is insignificant compared to the resources of Nodes in the total cost. This factor is of course dependent on the application and how well it manages stress in its services, but barring some extreme cases, reduction of egress traffic does not affect cost reduction to a sufficient degree. An important observation is the fact that applications respond far better and faster when given sufficient resources, which results in higher infrastructure costs. This leads us to the conclusion that the goals of reducing the cost of infrastructure and reducing the response time of applications are counter productive to one another.

As for the reduction of egress traffic, we can see that it is beneficial to both goals, as it can lead to better response times in the correct environments, as well as reducing the total monetary cost, because it leads to placement with less egress traffic and more importantly less required Nodes.

While we achieved the goals of our thesis, by both reducing the total infrastructure cost and the application response time, we have multiple proposals for future work to expand gaps in our knowledge of the subject. We propose an extension of our method in two ways. First, the extension should allow replication of highly stressed services and optimally place them in the cluster. The scaling mechanism could gather response time data from services and create a replica if it crosses a certain threshold. Then, CAP could place it optimally in the cluster, taking into account communication relationships with other services. In this way, it can take advantage of the scalability of the microservices architecture. Second, it should adapt to workload changes dynamically to remove or add replicas, as well as alter the placement of certain services.

In this work, we performed experiments regarding resource utilization and we mentioned that fully loading a Node could lead to crashes, even though this did not occur in our experiments. Further experimentation is required to more accurately determine optimal thresholds of resource utilization, taking into account the risk of a Node crashing and the cost in downtime this would result in.

Furthermore, the functionality of our strategy can be easily extended to a fog/edge cloud environment, where the reduction in egress traffic is much more impactful in both monetary cost and application response times. Additionally, the extensions proposed above can prove very significant in such a resource-intensive and latency-sensitive environment.

Lastly, most strategies, including our own and most of the ones examined in this work, consider only one criteria when making placement decisions. We propose a multi-criteria solution where two or even all of the major criteria that we have observed, those being communication between services, machine resources and latency between microservices, are considered before making a placement decision.

References

[1] Alkiviadis Aznavouridis, Konstantinos Tsakos, and Euripides Petrakis. “MicroService Placement Policies for Cost Optimization in Kubernetes”. In: Mar. 2022, pp. 409–420. DOI: 10.1007/978-3-030-99587-4_35.

[2] Jin-Tai Yan and Pei-Yung Hsiao. “A fuzzy clustering algorithm for graph bisection”. In: Information Processing Letters 52.5 (1994), pp. 259–263. ISSN: 0020- 0190. DOI: https://doi.org/10.1016/0020- 0190(94)00148- 0. URL: https://www.sciencedirect.com/science/article/pii/ 0020019094001480.

[3] Robert Cannon, Jitendra Dave, and James Bezdek. “Efficient Implementation of the Fuzzy C-Means Clustering Algorithms”. In: *Patterns Analysis and Machine Intelligence, IEEE Transactions on* PAMI-8 (Apr. 1986), pp. 248-255. DOI: 10.1109/TPAMI.1986.4767778

[4] J. C. Dunn. “A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters”. In: Journal of Cybernetics 3.3 (1973), pp. 32– 57. DOI: 10.1080/01969727308546046. eprint: https://doi.org/ 10.1080/01969727308546046. URL: https://doi.org/10.1080/ 01969727308546046.

[5] Alexandre Hollocou, Thomas Bonald, and Marc Lelarge. “Modularity-based Sparse Soft Graph Clustering”. In: Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics. Ed. by Kamalika Chaudhuri and Masashi Sugiyama. Vol. 89. Proceedings of Machine Learning Research. PMLR, Apr. 2019, pp. 323–332. URL: https://proceedings.mlr.press/v89/ hollocou19a.html.

[6] Mark Newman and Michelle Girvan. “Finding and Evaluating Community Structure in Networks”. In: Physical review. E, Statistical, nonlinear, and soft matter physics 69 (Mar. 2004), p. 026113. DOI: 10.1103/PhysRevE.69.026113

[7] Petrakis EGM, Skevakis V, Eliades P, Aznavouridis A, Tsakos K. ModSoft-HP: Fuzzy Microservices Placement in Kubernetes. Electronics. 2024; 13(1):65.

[8] Athanasios Prountzos, “Dynamic micro-service placement in hybrid cloud-fog infrastructures”, Diploma Work, School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece, 2023

[9] I. Lera, C. Guerrero and C. Juiz, "Analyzing the Applicability of a Multi-Criteria Decision Method in Fog Computing Placement Problem," 2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC), Rome, Italy, 2019, pp. 13-20, doi: 10.1109/FMEC.2019.8795361.

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

[11] Kubernetes Documentation. https://kubernetes.io/docs/home/

[12] What is a service mesh? https://aws.amazon.com/what-is/service-mesh/

[13] Istio Documentation. https://istio.io/latest/docs/concepts/

[14] Prometheus Documentation. https://prometheus.io/docs/introduction/overview/

[15] Prometheus Metric Types. https://prometheus.io/docs/concepts/metric_types/

[16] Prometheus Architecture. https://medium.com/@tech_18484/understand-prometheus-architecture-1ab83afd53b8

[17] Locust Documentation. https://docs.locust.io/en/stable/what-is-locust.html

[18] Marchese, Angelo & Tomarchio, Orazio. (2022). Communication Aware Scheduling of Microservices-based Applications on Kubernetes Clusters. 190-198. 10.5220/0011049300003200.

[19] Online Boutique GitHub. https://github.com/GoogleCloudPlatform/microservices-demo

[20] GCP Pricing. https://cloud.google.com/pricing/list?hl=en