



ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

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

By

Panayiotis Eliades

DYNAMIC MICROSERVICE PLACEMENT IN KUBERNETES IN THE CLOUD

Committee:

Professor Petrakis Euripides (Supervisor)

Professor Lagoudakis Michail

Associate Professor Samoladas Vasileios

Chania, February 2023

Abstract

Microservices-based architectures become a very popular method to design cloud-based containerized applications. Kubernetes is a container orchestration system for automating software deployment, scaling, and management. The Kubernetes cluster consists of nodes (VMs) and the microservices are placed in pods into the nodes. The default kubernetes scheduler selects an optimal node to run newly created or unscheduled pods, but it does not adapt to workload changes, so the default placement is static. Microservices (Pods) should be placed in nodes, so that they are minimizing the infrastructure cost by minimizing the egress traffic (traffic between nodes), the number of nodes and also the response time. In modern applications due to high workloads, services need to run in multiple instances to keep up with high traffic. In Kubernetes a Horizontal Pod Autoscaler (HPA) automatically updates a workload (Service), with the aim of automatically scaling the workload to match demand, but if starting from a sub-optimal placement (using the default Kubernetes Scheduler) this solution can be sub-optimal overall. In this work we aim to improve the initial placement of application services in Pods. The problem is handled as a graph clustering one, to minimize the applications cost and the latency. Graph clustering needs to be fuzzy, so it will allow pods to belong in more than one node. In this thesis two microservice-based applications were deployed in the Kubernetes of the GCP (Google Cloud Platform) to test the service placement solution, iXen an IoT application and Google's OnlineBoutique an e-shop application. The experimental results reveal that the fuzzy placement solution can reduce the total response time of the application and the total monetary cost compared to default kubernetes scheduler and with or without the use of Kubernetes horizontal pod autoscaler.

Table of Contents

List of Figures	4
List of Tables	6
Introduction	7
1 Background and Related Work	10
1.1 Related Work on Service Placement	10
1.2 Related work on Clustering	11
1.3 Related Algorithms	12
2 Infrastructure and Tools	18
2.1 Kubernetes	18
2.1.1 Kubernetes Components	19
2.1.2 Scheduling Process	20
2.1.3 Services	21
2.1.4 Horizontal Pod Autoscaler	22
2.2 Service Mesh and Istio	25
2.3 Metrics Tools	26
2.3.1 Prometheus	26
2.3.2 Kiali	27
2.4 Application Stressing Tool	28
3 Cluster Architecture and Implementation	29
3.1 System Architecture	29
3.2 Microservices Performance-Affinity Metrics	32
3.3 Application Graph	34
3.4 Benchmark Algorithms	35
3.4.1 Heuristic First-Fit	35
3.4.2 Bisecting K-Means (BKM)	36
3.4.3 Fuzzy Partitioning Algorithm	37
3.5 Automated Placement Algorithm	39
3.6 Benchmark Applications	41
3.6.1 Google Online Boutique e-Shop	42
3.6.2 iXen	44

4	Experimental Results	46
4.1	Infrastructure	46
4.2	Benchmark Application Stressing	48
4.2.1	Google Online Boutique e-Shop Stress Testing	48
4.2.2	iXen Stress Testing	48
4.2.3	Horizontal Pod Autoscaler Configurations	49
4.3	Placement Strategies	50
4.4	Infrastructure Cost and Cost Function	52
4.5	Experiemntal Results	54
4.5.1	Execution Time	54
4.5.2	Number of Hosts	55
4.5.3	Egress Traffic	57
4.5.4	Total Monthly Infrastructure Cost	58
4.5.5	Average Response Time	60
4.5.6	Response Time for the 90%ile of Requests	62
	General Conclusion and Future Work	64
	References	66

List of Figures

2.1	Kubernetes Components [14]	19
2.2	Service Architecture clusterIP	22
2.3	Service Architecture Load Balancer	23
2.4	Horizontal Pod Autoscaler	24
2.5	Istio envoy proxy	25
2.6	Istio Architecture	26
2.7	Locust User Interface [19]	28
3.1	System Architecture in Google Cloud Platform	30
3.2	Node's Architecture	31
3.3	Kiali Graph for e-Shop	34
3.4	Kiali Graph for iXen	35
3.5	e-Shop Architecture	43
3.6	iXen Architecture	45
4.1	Traffic between services for Default Online Boutique e-Shop Placement	51
4.2	Traffic between services for MODSOFT-HP Online Boutique e-Shop Placement and HPA disabled	51
4.3	Traffic between services for MODSOFT-HP Online Boutique e-Shop Placement and HPA enabled	52
4.4	Execution Time for Online Boutique e-Shop	55
4.5	Execution Time for iXen	55
4.6	Number of utilized Hosts for Online Boutique e-Shop with 150 users	56
4.7	Number of utilized Hosts for Online Boutique e-Shop with 300 users	56
4.8	Number of utilized Hosts for iXen	57
4.9	Hourly requested MBs for Online Boutique e-Shop with 150 users	57
4.10	Hourly requested MBs for Online Boutique e-Shop with 300 users	58
4.11	Hourly requested MBs for iXen	58
4.12	Estimated Monthly Cluster cost for Online Boutique e-Shop with 150 users	59
4.13	Estimated Monthly Cluster cost for Online Boutique e-Shop with 300 users	59
4.14	Estimated Monthly Cluster cost for iXen	60
4.15	Average Response Time for Online Boutique e-Shop with 150 users	61
4.16	Average Response Time for Online Boutique e-Shop with 300 users	61
4.17	Average Response Time for iXen	62
4.18	Response Time for the 90%ile of Requests for Online Boutique e-Shop with 150 users	62

4.19 Response Time for the 90%ile of Requests for Online Boutique e-Shop with 300 users	63
4.20 Response Time for the 90%ile of Requests for iXen	63

List of Tables

4.1	Cluster Characteristics	47
4.2	Node Pool Characteristics	47
4.3	Stress Testing Requests for Online Boutique e-Shop 150 users	48
4.4	Stress Testing Requests for Online Boutique e-Shop 300 users	49
4.5	Stress Testing Requests for iXen 10 users	49
4.6	GCP pricing for e2-standard machine type	52

Introduction

Problem Definition

Nowadays, modern web applications tend to change from monolithic architectures to microservice-based architectures. Monolithic architectures can be faster to develop and deploy than an application that uses microservices, however monolithic applications suffer from lack of scalability and even for a minor change, the entire application must be redeployed. In the microservices approach, each element can be scaled independently and the entire process can be more cost and time effective than with monoliths. Also, on-premises infrastructures switch to microservices-based architectures on the cloud to reduce the total cost and make it easier to maintain and migrate applications. The continuous evolution of microservices-based applications brings the need for technologies that create, orchestrate, maintain, observe, schedule, and manage microservices such as containerization technologies, like Kubernetes. Kubernetes enables a lightweight packing of services and eases the deployment of applications across different types of infrastructures and manages microservices-based applications.

It is important to know how the consumer is billed in the cloud. Cloud providers charge on the number of reserved virtual machines, the resources of the machines (CPU, Memory, Disk and Operating System) and the traffic between the virtual machines. The advantages of high availability, vertical and horizontal scalability that Kubernetes offers can easily result in high costs. High traffic between nodes placed in different regions and additional virtual machines that are not necessary for the applications' requirements can increase infrastructure costs significantly.

Configuring a Kubernetes cluster is not a simple task; the scheduler needs to place the application services with high communication in the same node (VM) to optimize the deployment of the application in the cloud and minimize the cost and response time. This problem is referred to as the service placement problem (SP) and has raised concerns about the optimal placement solution of an application's services into a Cloud infrastructure's

virtual machines. Controlling which VM each service is hosted on reduces the overall number of VMs required for the application as well as total egress traffic, minimizing the total monetary costs of the infrastructure.

Scope of the Thesis

The scope of Thesis is to reduce the nodes needed to place our application microservices , the egress traffic between the nodes (VMs) and the response time of the application requests by solving the Service Placement problem using a fuzzing partitioning technique and also analyze the effect of Kubernetes Horizontal Pod Autoscaler (HPA) on that technique. We can partition the application services using graph partitioning method by modeling the application as a graph that represents services as graph-nodes and traffic between services as edges. These algorithm aim to create the best possible partitions, allowing services with high traffic rates to coexist in the same partition in our application graph. The amount of traffic transferred between the VMs in the infrastructure, or "egress traffic," is decreased when these services are placed in the same partition. In contrast to other placement algorithms which are flat, a fuzzy placement enable placing several instances of particular services in various nodes (VMs). The workload of each service replica is decreased when more than one instance of the service (replicas) is deployed in the cluster by distributing the incoming requests across the replicas, as a result, requests are processed more quickly, leading to quicker application response times.

Although the fuzzy placement requires more pods to host the application services due to the fact that more than one instance can be placed in different nodes, the bin packing method we apply to the graph partitioning result distributes the services in VMs based on the partitioning results, the traffic between services, and the available VM resources. As a result, the use of VM resources is optimized, and the number of VMs required to host the application could be decreased. This reducing the monetary cost of the infrastructure without raising the application response time significantly.

In addition, in this Thesis the placement solution was automated unlike other works where Placement is done manually by the users. When the algorithm outputs the final service placement, the services migrate and/or replicate on the appropriate Nodes automatically without requiring any user intervention.

Contributions

Below is a summary of the present work's contributions.

- Introduce a service placement algorithm that automates the service (Pod) migration between the nodes
- Compare the fuzzy service placement to the default Kubernetes service placement and other flat service placements
- Compare the impact of HPA on service placement strategies

Thesis Structure

This Thesis is organized in 4 main chapters including the Introduction. In chapter 1 the Thesis background and related work is presented including the background work related to the service placement problem, and the related algorithms used in our solution. The theoretical background of Kubernetes environment and the features of Kubernetes like Horizontal Pod Autoscaler are presented in Chapter 2 of this thesis as well as the tools to produce the application graph and stress the applications in our experiments. Chapter 3 includes the architecture of the Kubernetes cluster, the affinity metrics for the communication evaluation and the Horizontal Pod Autoscaler. Furthermore, this chapter presents the proposed placement strategy, and the algorithm that is automating the Service Placement solution. Also, this Chapter includes a presentation of the applications used in our experiments. The experiments conducted to evaluate the effectiveness of the suggested placement method with and without the use of Kubernetes Horizontal Pod Autoscaler compared with the default Kubernetes scheduler, a flat placement solution and a Heuristic single-step processing algorithm proposed by related works are presented in the thesis final chapter 4.

Chapter 1

Background and Related Work

This chapter's first section will include some background information and a review of previous research in the area of service placement problem. The definition of flat and fuzzy clustering and an introduction to relevant research and algorithms that were modified and applied in this Thesis to provide our solution to the service placement problem are presented in the second section of this chapter.

1.1 Related Work on Service Placement

In “Improving microservice-based applications with runtime placement adaptation” [1] authors suggest a heuristic strategy to move services with higher affinity (high traffic communication rates) in order to optimize service placement in the current infrastructure. The suggested algorithm, a Heuristic First-Fit version algorithm, computes a new placement for the application services by accessing the cluster Nodes and Pods' resource utilization as well as the cluster's available host machines. In this version of algorithm, the First-Fit reorganizes the microservices in the cluster's available host machines (VMs) so that microservices with high affinity are co-located, while taking into account microservice resource utilization and host resource availability. The proposed algorithm improved the infrastructure monetary cost but, it can significantly increase the response time. Recent research by Aznavouridis, Tsakos, and Petrakis [2] aimed to solve the Service Placement issue with the goal of lowering the infrastructure's overall monetary cost. They used a weighted, directed graph to describe the application, using two distinct affinity metrics that focused on the number of requests made to each service (Requests Per Second) and the amount of data transferred between them (Weighted Bidirectional Affinity). A Kubernetes cluster with a predetermined number of available nodes was used to perform all the experi-

ments, which experiments were performed on real applications. In this work they examined a variety of different flat graph partitioning algorithms that produced graph partitions with the least amount of communication feasible across partitions before passing the partitions to the placement algorithm, which placed the services in the available nodes. Furthermore they also examined a non-clustering algorithm, the Heuristic First-Fit [1] as mentioned before. The graph partitioning algorithms used in this work [2] are flat, and their final placement limits each service to a single instance.

In “Service deployment strategies for efficient execution of composite SaaS applications on cloud platform” Huang and Shen [3] developed a service deployment strategy that decreases application response times but their proposed approach did not attempt to reduce infrastructure cost for the application. They used a graph to represent communication costs and service parallelism in order to model the application, then they applied a minimum k-cut to the modeled graph. On Amazon’s cloud architecture, they conducted experiments for four service deployment methodologies. All the experiments were conducted on a single virtual machine (VM) without the use of Kubernetes.

1.2 Related work on Clustering

The purpose of cluster analysis or clustering is to organize a collection of objects into groups that are more similar (in some ways) to one another than to objects in other groups (clusters). Cluster analysis is a general problem to be solved, not a particular algorithm. Different algorithms that have quite different ideas of what clusters are and how to find them effectively can accomplish it. The aim of optimal clustering is to place items with the greatest degree of similarity in the same clusters and items with the least degree of similarity in different clusters.

An analysis of clusters may be **flat** (hard) or **fuzzy** (soft). A point can either belong to or not belong to a subset according to **flat clustering**, which means that the subsets of a set are mutually exclusive. Flat clustering methods that will be used in this work are Heuristic first fit, and bisecting K means (Algorithms in section 1.3). On the other hand a clustering method known as **fuzzy clustering** assigns one data or object to one or more clusters. In this work we are using MODSOFT which is a fuzzy clustering method (Algorithm in section 1.3).

Cannon, Dave, and Bezdek, in 1986 [4], extended the work of Dunn [5] on the Fuzzy c-Means (FCM) algorithm, it can be used if the objects of interest are represented as points in a multi-dimensional space and is the most well-known fuzzy clustering algorithm. By attempting to minimize the objective function, the FCM algorithm, which is very similar

to the k-means algorithm, generates a matrix w_{ij} that indicates the extent to which element x_i belongs to cluster c_j .

Fuzzy Partitioning

In the graph partitioning problem, which is a modified clustering problem, nodes are divided into graph partitions, which are smaller groupings of nodes. We can make the analysis of a graph simpler by dividing it into smaller partitions. Flat (or hard) partitioning and fuzzy (or soft) partitioning are the two types of graph partitioning. Producing flat partitions of a graph means that each node belongs to exactly one partition. Fuzzy partitioning, on the other hand, encodes ambiguities in node-to-partition assignments since each node is awarded a membership grade to each fuzzy partition.

The FCM algorithm [4] expanded by Yan and Hsiao [6] to address the issue of graph bisection. A graph is partitioned into two divisions in the graph bisection problem so that the weights of the edges connecting these two partitions are optimized. Their approach provides a feasible fuzzy graph partition of the vertex set V by iterative optimization of the objective function $J(U, v)$. The first stage in their algorithm is to define the clustering distance matrix d_{ij} , which is defined as the weight of edge w_{ij} if there is a direct edge between vertex v_i and vertex v_j , or the shortest path between the two vertices if no such edge exists. Then, two arbitrary divisions of the vertices are initialized to create a fuzzy matrix U . The fuzzy U matrix is a $n \times 2$ matrix, where n is the number of nodes and represents the degree of membership of each service to each of the 2 initialized partitions. In each iteration, the objective function is minimized to determine two partition centers. The Fuzzy U matrix is then updated by calculating a degree of membership based on the distance between each vertex and the center of the partition. After the Fuzzy U matrix converges, the iteration ends, and the matrix outputs the degree of memberships for each vertex to each partition. Finally, after sorting the vertices into groups, all vertices will be split into two even subsets with a minimal cut for graph bisection.

1.3 Related Algorithms

Flat Clustering Algorithm

The algorithms that will be utilized during the implementation phase will be described and analyzed briefly in this section.

Heuristic First-Fit

As we mentioned before Sampaio et al. [1] propose a heuristic approach to optimize service placement and minimize the monetary cost of the infrastructure by migrate services with higher affinity on the same node (VM). The Heuristic First Fit (presented in algorithm 1) iterates across affinities in descending order and attempts to co-locate microservices with higher traffic communication rates on the same host machine. In **line 5** for every pair of affinities and for each corresponding pair of microservices m_i, m_j the algorithm in **lines 10-13** tries to assign m_j onto the host of m_i , (H_i) since they are connected by an affinity. If H_i lacks sufficient resources, the algorithm attempts to place m_i on the host of m_j , (H_j) (**lines 14-17**). If neither host has sufficient resources to co-locate m_i and m_j , these microservices will remain at their original hosts. When a microservice is transferred to a new host, it is recorded as moved and cannot be moved again, even if it is linked to another service in the application with lower affinity traffic rates (**lines 19-21**). Finally, a list of movements (Final Placement) with microservice IDs and new locations is constructed. This algorithm does not guarantee that the list of movements generated is optimal for a given cluster of microservices.

Algorithm 1 Heuristic First Fit [1]

```

1: Input: Hosts (H), microservices (m), resources (r)
2: Output: Placement Solution
3: moved  $\leftarrow$  [ ]
4: //Affinities are sorted in decreasing order
5: for every pair of affinities do
6:    $m_i \in H_i$  //  $m_i$  located at host  $H_i$ 
7:    $m_j \in H_j$  //  $m_j$  located at host  $H_j$ 
8:    $m_j \neq m_i, H_j \neq H_i$ 
9:   hasMoved  $\leftarrow$  False
10:  if  $r(m_i) + r(m_j) \leq r(H_i) \wedge m_j \notin \text{moved}$  then
11:     $H_j \leftarrow H_j - m_j$ 
12:     $H_i \leftarrow H_i \cup m_j$ 
13:    hasMoved  $\leftarrow$  True
14:  else if  $r(m_i) + r(m_j) \leq r(H_j) \wedge m_i \notin \text{moved}$  then
15:     $H_i \leftarrow H_i - m_i$ 
16:     $H_j \leftarrow H_j \cup m_i$ 
17:    hasMoved  $\leftarrow$  True
18:  end if
19:  if hasMoved then
20:    moved  $\leftarrow$  moved  $\cup [m_i, m_j]$ 
21:  end if
22: end for

```

Bisecting K-Means

Based on the K-Means algorithm, Bisecting K-Means is a divisive hierarchical clustering algorithm [7]. Given a data set, the algorithm uses the K-Means algorithm to choose the two best fit sub-clusters to divide the data points into. All of the available data points are initially assigned to a single cluster. Sum of Squares Error (SSE) is determined for each iteration in order to quantify the inter-cluster dissimilarity and thereafter choose the next centroids. In **lines 4-12** the proposed sub-cluster is then chosen and split into two new sub-clusters based on the SSE and this procedure is repeated until the algorithm generates K clusters in the desired number, which is determined by the users' input. The algorithm creates a binary clustering hierarchy, however it may not reach the global optimum because of how the initial centroids are chosen. The Bisecting K-Means algorithm is shown in 2.

Algorithm 2 Bisecting K-Means [7]

```

1: Input: Cluster C, number k of desired clusters
2: Output: k Clusters of Application
3:  $i \leftarrow 1$ 
4: while  $i < k$  do
5:   Select a parent cluster, C to split
6:   for fixed number of iterations do
7:     Use K-Means to split C into  $C_1$  and  $C_2$ 
8:     Calculate inter-cluster dissimilarity for  $C_1$  and  $C_2$ 
9:   end for
10:  Select the sub-clusters with highest inter-cluster dissimilarity
11:   $i \leftarrow i + 1$ 
12: end while

```

Fuzzy Clustering Algorithms

Modularity-based Graph Clustering

Hollocou, Bonald, and Lelarge [8] propose in their work a modification of the modularity optimization issue that Newman and Girvan first proposed [9], which it has a large-scale usage in community detection in big networks. A function called modularity measures how well a graph is partitioned and it is described as the difference between the likelihood that two randomly chosen graph nodes will be placed on the same partition as well as the likelihood that two nodes of graph G will be on the same partition. The modularity function introduced by Newman and Girvan [9] is known as hard modularity, whereas the MODSOFT [8] algorithm is relaxation of the modularity maximization problem, results in fuzzy partitions.

Contrary to other proposed relaxation algorithms, the proposed algorithm (Listing 1.1) does not require prior knowledge of the number of potential clusters. It is efficient because

it doesn't need to process the entire network because any particular node's membership information only depends on its immediate neighbors. By updating the t parameter, the algorithm can produce soft or hard partitions.

The weighted average vector \bar{p} is initialized as the weighted degree of each node, and the membership matrix (n by n matrix) is initialized as a one's matrix (\mathbf{I}) during the initialization phase of the algorithm. The membership matrix displays a number between 0 and 1 that represents the probability that each node (matrix row) belongs to the same partition as another node (matrix column).

The following stages can be used to describe one epoch of the MODSOFT algorithm. In **line 1** as an update rule for the membership of each node, a gradient descent step is performed for each node. The algorithm's gradient descent phase is local, as it only uses the neighbors of each node to update the node's membership and attempts to maximize modularity. In **line 2** a projection is performed and finally in **line 3** the weighted average vector \bar{p} is updated. Each epoch represents an algorithm iteration, and at the end of each iteration, the modularity is computed, evaluating the partitions' optimality. The algorithm is repeated until the modularity growth falls below a predetermined threshold. The membership matrix is the final result of the MODSOFT algorithm, with each cell representing the probability of the row-node being in the same partition as the column-row.

Listing 1.1. MODSOFT Algorithm [8]

- **Initialization** : $\mathbf{p} \leftarrow \mathbf{I}$ and $\bar{\mathbf{p}} \leftarrow \mathbf{w}/w$.
 - **One Epoch**: For each node $i \in V$,
 - 1: $\forall k \in \text{supp}_i(\mathbf{p}), \hat{p}_{ik} \leftarrow p_{ik} + t' \sum_{j \sim i} W_{ij} (p_{jk} - \bar{p}_k)$
 - 2: $\mathbf{p}_{i.}^+ \leftarrow \text{project}(\hat{\mathbf{p}}_{i.})$
 - 3: $\bar{\mathbf{p}} \leftarrow \bar{\mathbf{p}} + (w_i/w) (\mathbf{p}_{i.}^+ - \mathbf{p}_{i.})$ and $\mathbf{p}_{i.} \leftarrow \mathbf{p}_{i.}^+$.
-

Placement Algorithms

Bin-Packing Problem

The bin packing problem is an optimization problem, in which items of different sizes must be packed into a finite number of bins or containers, each of a fixed given capacity, in

a way that minimizes the number of bins used [10][11]. The problem is computationally NP-hard, and the equivalent decision problem is NP-complete. Despite its worst-case hardness, advanced algorithms such as the first-fit algorithm, which provides a rapid but often non-optimal solution, can produce ideal solutions to very large instances of the issue. In practice, bin packing happens when items can share space when packed into a bin. In particular, a collection of items may take up less space when packed together than the total of their individual sizes. This variation is known as Virtual Machine (VM) packing because when virtual machines are packed in a server, their overall memory and CPU requirements may drop due to shared pages that only need to be saved once. If items can share space in any way they like, the bin packing problem is difficult to even approach. However, if the space sharing fits into a hierarchy, as in virtual machine memory sharing, the bin packing problem can be easily approximated.

Heuristic Packing

Aznavouridis, Tsakos, and Petrakis [2] proposed a Heuristic Packing algorithm, which is a placement finding algorithm that can be thought of as a multi-dimensional bin packing challenge. The algorithm takes the created partitions as input and attempts to arrange each partition in one of the available host machines (VMs) using two greedy heuristics to find the optimal placement. The Most-Loaded Situation (ml), which is a scalar value of the load between the available resources in the host machine and the requested resources from the currently processing part, and the Traffic Awareness (tf), which is the sum of the traffic rate between the services in the current processing part and the already processed partition services. The Most-Loaded heuristic prioritizes the machines in which services will be placed when Traffic Awareness factors are equal. Heuristic Packing is shown in algorithm 3

Algorithm 3 Heuristic Packing algorithm

```

1: Input: Partition  $P = (S_1, S_2 \dots S_N)$  of application, vectors of available resources on
   each machine  $V = (V_1, V_2 \dots V_M)$ 
2: Output: a placement solution  $X$ 
3: Calculate vectors of resource demands of each part  $a$  ( $D'_1, D'_2 \dots D'_N$ )
4:  $X \leftarrow [x_{ij} = 0]$  for every part and host machine
5: for  $i \leftarrow 1; i \leq N'; i++$  do
6:    $tf \leftarrow 0, ml \leftarrow 0, y \leftarrow 0$ 
7:   for  $j \leftarrow 1; j \leq M; j++$  do
8:     if part  $S_i$  can be packed into machine  $m_j$  then
9:        $tf \leftarrow \sum t_{uv}$ 
10:       $ml \leftarrow \sum_{k=1}^R \frac{d_i^k}{v_j^k}$ 
11:      if  $tf_j \geq tf$  then
12:         $tf \leftarrow tf_j, ml \leftarrow ml_j, y \leftarrow j$ 
13:      else if  $tf_j == tf$  and  $ml_j > ml$  then
14:         $tf \leftarrow tf_j, ml \leftarrow ml_j, y \leftarrow j$ 
15:      end if
16:    end if
17:  end for
18:  if  $y == 0$  then
19:    Return null
20:  else
21:     $V_y \leftarrow V_y - D'_i$ 
22:     $x_{iy} \leftarrow 1$ 
23:  end if
24: end for
25: Return  $X$ 

```

Chapter 2

Infrastructure and Tools

The infrastructure and tools for our suggested solution are presented in the second chapter of this thesis, along with the tools used to test and evaluate the effectiveness of our approach.

2.1 Kubernetes

The microservice architecture is designed so that application functionality can execute in independent, separated containers that are connected through APIs (Application Programming Interfaces). Large, complicated applications can be delivered quickly, often, and reliably thanks to the microservice architecture, with each microservice implementing a particular aspect of the application logic. Containers [12] is used to run each microservice, consequently the program operates efficiently and dependably in various computing environments.

The emergence of architectures based on microservices created a demand for microservice orchestration tools like Kubernetes [13]. Kubernetes is a portable extensible, open-source platform for managing containerized applications. It provides automation's for containerized applications management, deployment and scaling. Kubernetes provides the tools and frameworks to run distributed systems resiliently and handle the behavior and maintenance of containerized services by providing load balancing, storage orchestration, automatic bin packing, security, automated rollouts, self-healing, rollbacks and service discovery. Nonetheless Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system, it provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is necessary.

2.1.1 Kubernetes Components

To create a Kubernetes cluster its necessary first to configure a Node Pool. A Node Pool according to the needs of each application contains the CPU, RAM, storage, and OS specifications that each initialized virtual machine (Node) must meet. The user is responsible to define the number of nodes that the Node Pool must generate. Then the virtual machines (Nodes) will boot and connect to the Kubernetes cluster. Each virtual machine (Node) can hosts multiple Pods (containers) and each pod can host numerous microservices, but is recommended that each pod host only one microservice.

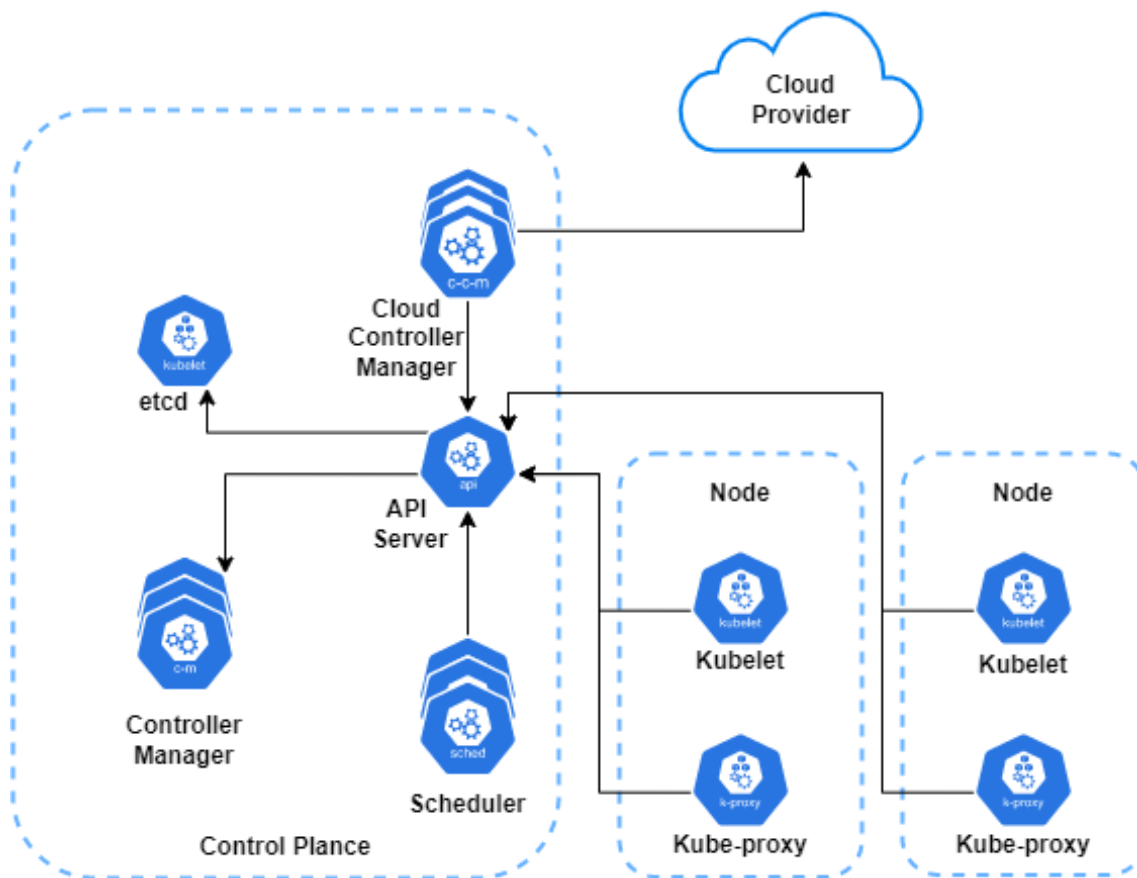


Figure 2.1. Kubernetes Components [14]

A cluster is created when Kubernetes is deployed. A group of worker machines, known as nodes, that run containerized applications make up a Kubernetes cluster. There is at least one worker node in each cluster. The Pods that make up the application workload are hosted on the worker node(s). A kubelet component, which is an agent that ensures that containers are running in Pod and manages the containers that Kubernetes creates, and is operated by each Node. The Kubernetes Service concept is implemented in part via kube-proxy, a network proxy that runs on each node in the cluster and maintains network rules on nodes. These network rules permit network communication to the Pods from network sessions both inside and outside of your cluster.

The control plane manages the worker nodes and the Pods in the cluster and make global decisions about the cluster. The Control Plane (CP) consist of five main components, API server, etcd, controller manager, cloud controller manager and scheduler. The API server exposes the Kubernetes API and it operates as the control plane's front end. Additionally, API server can scale by adding additional instances and balance traffic between those instances. Etcd is a consistent and highly-available key value store used by Kubernetes to store all cluster data. The controller-manager runs all controller processes, although each controller runs as a separate process, they are all compiled into a single binary and operated in a single process to reduce complexity. The cloud-controller-manager is a component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and distinguishes between components that only communicate with your cluster and those that communicate with that cloud platform. The cloud-controller-manager only runs if the cluster is deployed on the cloud, if the cluster is on your own premises the cluster does not have a cloud controller manager.

2.1.2 Scheduling Process

Kube-scheduler ensures that Pods and Nodes are matched so that Kubelet can run them, is one of the most important components. A scheduler keeps track of newly created Pods that are unassigned to any Nodes. For every Pod that the scheduler detects, the scheduler becomes responsible for finding the best Node for that Pod to run on. Individual and group resource requirements, hardware, software, and policy restrictions, affinity and anti-affinity specifications, data locality, inter-workload interference, and other considerations must all be made while making scheduling decisions. The Kubernetes scheduler selects the first unscheduled Pod from his priority queue throughout the scheduling cycle, discovers feasible nodes, and then executes a series of functions to assess each Node. In a procedure known as binding, the scheduler assigns the Pod to the feasible node with the greatest score and notifies the kube-apiserver of his choice. Three main factors are taken into consideration when the scheduler make a decision. First factor is the hardware, where a pod might have mentioned in the deployment file that it needs a certain type of hardware. The second factor is data locality, it places a Pod in a specified availability zone, ensuring faster reads and greater write throughput. The third and last factor is data locality, by co-locating highly communicative pods in a single availability zone.

Additionally you can customise the behavior of the kube-scheduler. A scheduling Profile (configuration file) allows you to configure the different stages of scheduling in the kube-scheduler. A point of extension exposes each stage (an extension point can be the *score* who provides a score to each node). Plugins implement one or more of these extension points to offer scheduling behaviors. The scheduling happens in a series of stages that are

exposed through the extensions points and the plugins implement one or more of these extension points. Two interesting plugins (and why they were not used in this Thesis) is the following :

- *PodTopologySpread* : This can help to achieve high availability as well as efficient resource utilization. Pods are spread across your cluster among failure-domains such as regions, zones, nodes, and other user-defined topology domains. One of the problem of current Kubernetes version is that the Kubernetes Scheduler doesn't ignore Nodes Tainted with NoSchedule when Evaluating. This could be a disaster when you are trying to resolve an incident or do maintenance and rescheduled pods are silently getting stuck in a "pending" state. Also, you don't reduce the cost of the cluster.
- *NodeResourcesBalancedAllocation* : Favors nodes that would obtain a more balanced resource usage if the Pod is scheduled there. We achieve more balanced nodes, but we don't reduce cost. We may reduce latency for a small number of users, then we will need again the Horizontal Pod Autoscaler (HPA).

2.1.3 Services

A kubernetes service is an abstract method of making an application running on a set of Pods available as a network service. Each Kubernetes Pod is connected to its corresponding Service, and the Service is in charge of sending all traffic to the Pod. When created, each Service is assigned a unique IP address. This address is tied to the lifespan of the Service, and will not change while the Service is alive. The Service discovers the IP address of the Pod upon creation or update and exposes a permanent address (user-defined in Service YAML) and a port to enable communication with other services. The types of Kubernetes services include ClusterIP, NodePort, LoadBalancer, and External Name and have the following behaviour :

- **ClusterIP** : This is the default value that is used if you don't specify the type for the Service, it exposes the Service on a cluster-internal IP and makes Service only reachable from within the cluster.
- **NodePort** : Exposes the Service on each Node's IP at port (static port called NodePort) and makes the Service accessible through the external network.
- **LoadBalancer** : Exposes the Service externally using the cloud provider's load balancer.
- **External Name** : Provides a mapping between the Service and the content in the externalName field.

The kube-proxy component is accessible as long as there is at least one healthy node in the cluster and has access to all of the Kubernetes Services communication information. If more than one Pod instance is associated to the Service, Kubernetes Services function as a load balancer for the incoming traffic. In Figure 2.2 we display the Service Architecture with clusterIP, the internal incoming traffic come into the Kubernetes service and load balanced into the replicated pods of the same service. In Figure 2.3 we display the Service Architecture with External Load Balancer, the internal or external incoming traffic come into the Kubernetes service and load balanced into the replicated pods of the same service. The Service will select the best Pod to route each request if there is a replica set of pods, according to the Kubernetes Documentation. In our thesis, we assume that, if such a destination Pod is placed in the same Node as the request source Pod, the Service will choose to redirect the requests there, otherwise the service will choose randomly a different node where the pod exists.

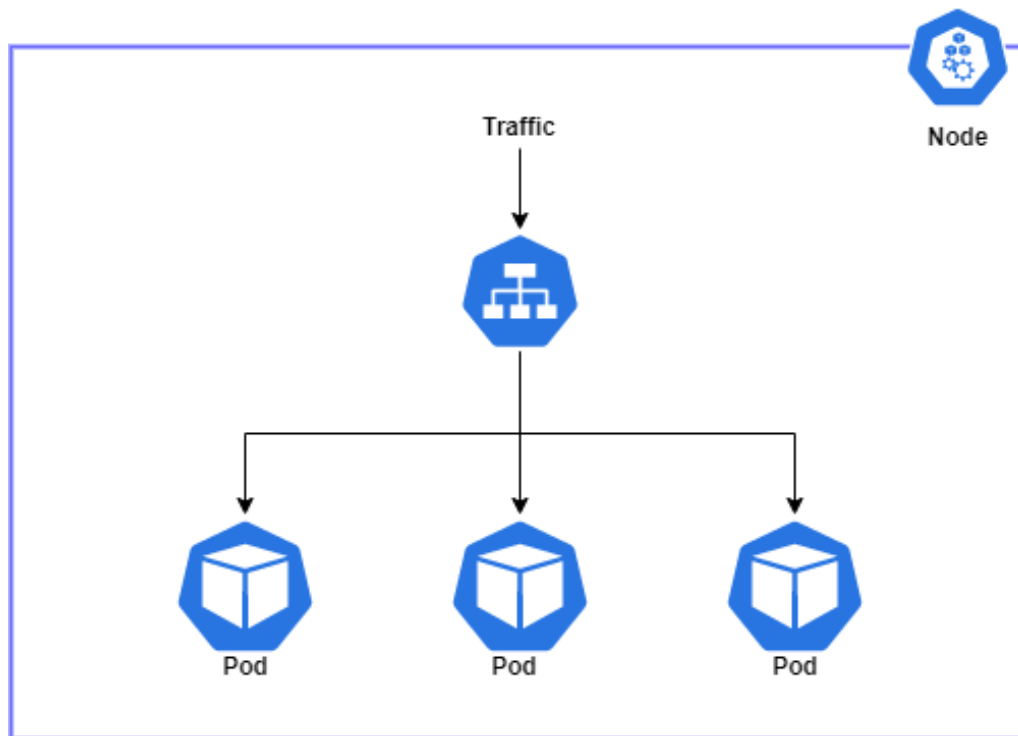


Figure 2.2. Service Architecture clusterIP

2.1.4 Horizontal Pod Autoscaler

In modern applications the network traffic can increase significantly and therefore the performance of the application will decrease and users will face high response times. Kubernetes have the ability to scale pods horizontally. The reaction to an increase in traffic is to deploy more Pods, which is known as horizontal scaling. With the intention of automatically scaling the workload to match demand, a Horizontal Pod Autoscaler (HPA) [15] automatically changes a workload resource such as a Deployment (figure 2.4).

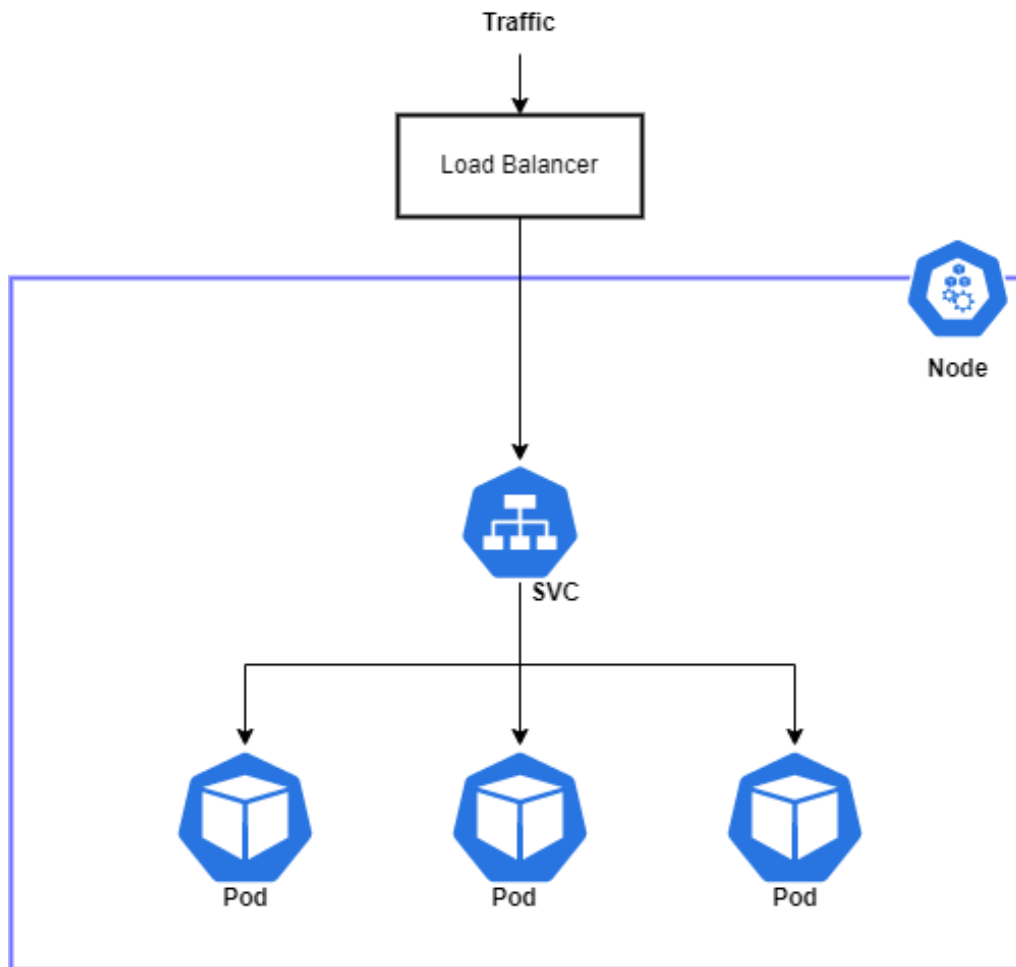


Figure 2.3. Service Architecture with External Load Balancer

As mentioned in documentation [15] Horizontal Pod Autoscaler is implemented as a controller and a Kubernetes API resource, and the resource determines the behavior of the controller. When operating within the Kubernetes control plane, the horizontal pod autoscaling controller regularly modifies the target's desired number of pods (e.g deployment) to match observed metrics like average CPU consumption, average memory utilization, or any other custom metric you provide.

Horizontal pod autoscaling is implemented by Kubernetes as a control loop that runs periodically, but is not a continuous process. The interval is set by the user and the default interval value is 15 seconds. In each period the Horizontal Pod Autoscaler specified metrics definition's are checked against the resource utilization by the controller manager. The controller manager finds the target resource, then selects the pods and gets the metrics from the resource metrics API or from other custom metrics API. There are three types of resource metrics, per-pod resource metrics, per-pod custom metrics, and, object and external metrics. For per-pod resource metrics like CPU the controller retrieves the metrics for each Pod that the Horizontal Pod Autoscaler has selected from the resource metrics API, then the

usage value is determined by the controller as a proportion of the corresponding resource request on each Pod's containers, or if the target raw value has been determined, the raw metric data are used. The controller then calculates a ratio to scale the number of required replicas by taking the mean of the utilization or raw value across all targeted Pods. For per-pod custom metrics the controller works similarly to per-pod resource metrics, except that it works only with raw values. Lastly for object and external metrics it retrieves a single metric that describes the object in query. To create the ratio we describe above, this metric is compared with the desired value.

The HorizontalPodAutoscaler controller functions by dividing the desired metric value by the current metric value as is shown below :

$$desiredReplicas = \lceil currentReplicas * (currentMetricValue / desiredMetricValue) \rceil$$

For instance, if the current metric value of CPU is 100m and the desired value is 50m quantity of replicas will be doubled, because $100/50 == 2$. On the contrary, if the current value is 25m the quantity of replicas will be reduced by half, because $25/50 == 0.5$.

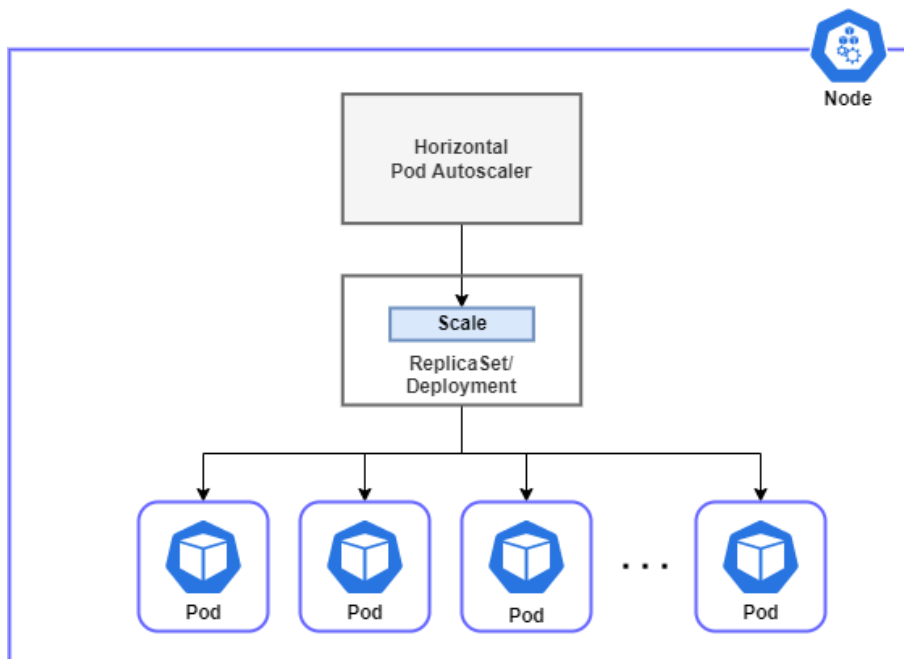


Figure 2.4. Horizontal Pod Autoscaler

2.2 Service Mesh and Istio

Typically, modern applications are designed as distributed collections of microservices, with each collection of microservices carrying out a specific application function. A service mesh is a specific layer of infrastructure for applications that enhances features like observability, traffic management, and communication security. As a Kubernetes-based system expands and becomes more complicated, it could be more difficult to comprehend and control. A service mesh can address requirements like discovery, load balancing, failure recovery, metrics, and monitoring. Both the type of software you use to accomplish this design and the security or network domain that is generated when you employ that software are referred to as "service mesh" [16]. The service mesh's Control Plane injects a Sidecar Proxy service as a third-party application, which performs all of the above-mentioned logic. In figure Figure 2.5 we show the sidecar proxy injected to pod.

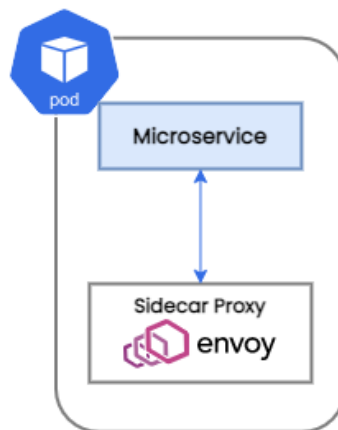


Figure 2.5. Istio envoy proxy

Istio is an open-source service mesh and overlays current distributed applications in a transparent way. Istiod is the Control Plane of Istio. Istiod deploys an Envoy Sidecar Proxy service in each newly formed Pod after automatically identifying new services and endpoints in the cluster. Additionally, Istiod monitors all certificates and sets up secure TLS connection between services and collects metrics from each Pod and exports telemetry data, which may be accessed by a monitoring server like Prometheus. Istio supports application level protocols such as HTTP, which includes HTTP/1.1, HTTP/2, and gRPC, TLS, which includes HTTPS and TCP. In the figure Figure 2.6 the Istio architecture is presented.

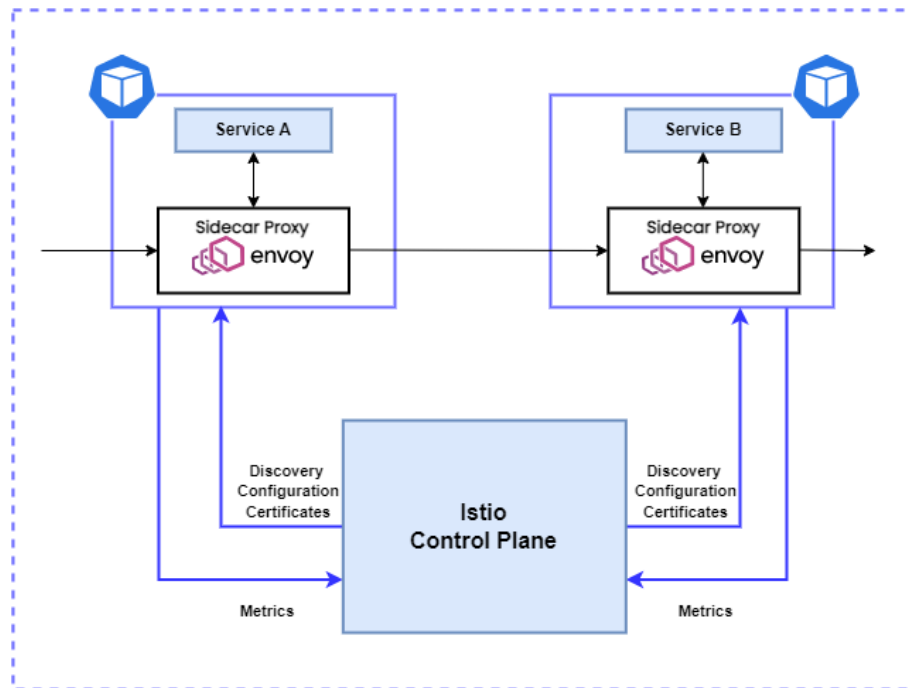


Figure 2.6. Istio Architecture

2.3 Metrics Tools

2.3.1 Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit firstly build at SoundCloud. It continuously checks its targets, finds issues before they even occur, and sends out alerts when a crash does occur. Prometheus collects and stores its metrics as time-series data, i.e., metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels [17]. A target must expose its metrics in the proper format and through the Prometheus endpoints in order for Prometheus to monitor it. Since many services lack native Prometheus support, an additional component called Exporter is required to receive the service's metrics, convert them into a compatible format, and expose the endpoint so that Prometheus may retrieve them.

Prometheus cant pull metrics from the node by-itself. Each VM in the cluster has Node Exporter installed and injected into it so it can fetch Node data. Node exporter is a tool that exports hardware and kernel measurements as well as physical and virtual machine metrics from Linux nodes and they must be configured to listen on a dedicated port. The Node exporter makes it possible to measure the usage of various VM resources, including memory, disk, and CPU.

Prometheus pulls metrics from its targets and stores them in text- based format, and the

metrics can be requested by another tool or by a user. The Prometheus server and more sophisticated visualization tools like Grafana may both display the metrics.

2.3.2 Kiali

Kiali [18] is a management console for an Istio-based service mesh. It offers dashboards, observability, and allows for the operation of a service mesh with powerful configuration and validation features. By modelling traffic topology, it illustrates the service mesh's structure and indicates the mesh's overall health. It retrieves Istio configurations and data that are made available via Prometheus and the Cluster API. Kiali communicates directly with Prometheus and makes use of the information stored there. Visualizing the topology of the service mesh is made effective by the Kiali graph. It shows the communication between the services as well as their individual traffic rates and latencies, making it easier to visually spot problem regions and swiftly narrow down any issues. Since Kiali is implemented as a service, it provides an API that can be used to access the Kiali graph and mesh data.

Kiali is a management console for an Istio-based service mesh. It retrieves Istio data and configurations, which are exposed through Prometheus and the Cluster API. Kiali communicates with Prometheus directly and uses the data stored in Prometheus to figure out the mesh topology, show metrics, calculate health, show possible problems, etc. Kiali provides a powerful way to visualize the topology of the service mesh by creating the Kiali Graph, which displays the services' network communication protocol, their traffic rates, and the latency between them. Kiali is deployed as a service, and it offers an API through which the mesh information and the Kiali graph can be obtained.

The Kiali graph shows the communication between microservices, the graph is weighted and directed. The deployments and the corresponding Kubernetes service components are represented by the graph's nodes, while the graph's edges show the weighted traffic between the microservices. The weight of this traffic is Requests per Second for the HTTP and gRPC communication protocols and Bytes per Second (BPS) for the TCP traffic. Finally for the purpose of visualizing the traffic communication between the microservices in HTTP, gRPC and TCP protocols, Kiali graph supports traffic animation.

2.4 Application Stressing Tool

Locust

Locust [19] is a simple, scriptable, and scalable performance testing tool. Instead of being constrained by a UI or a restrictive domain-specific language, you specify the behavior of your users in standard Python code, and it can be deployed as a microservice even without a UI, applying the predetermined behavior. It provides an application for distributed event-based requests and can support thousands of users at once. It has an intuitive user interface, exports data in several formats, and gives different evaluation metrics, such as average latency, 90ile percentile, median repsonce time, etc.

Locust can be run from insight the cluster and sent requests on application endpoints using the front-end private IP, or can be run locally on a linux machine and performs requests on application endpoints using the front-end public IP.

Locust is used in both stress tests of our application. The default workload consists of 10 users. To have more users we set the numbers of users and the spawn rate, e.g. 300 users with spawn rate 30users/second (figure 2.7). We can choose any number of users we want as long as we don't exceed the CPU and MEMORY capacity of the computer running Locust.

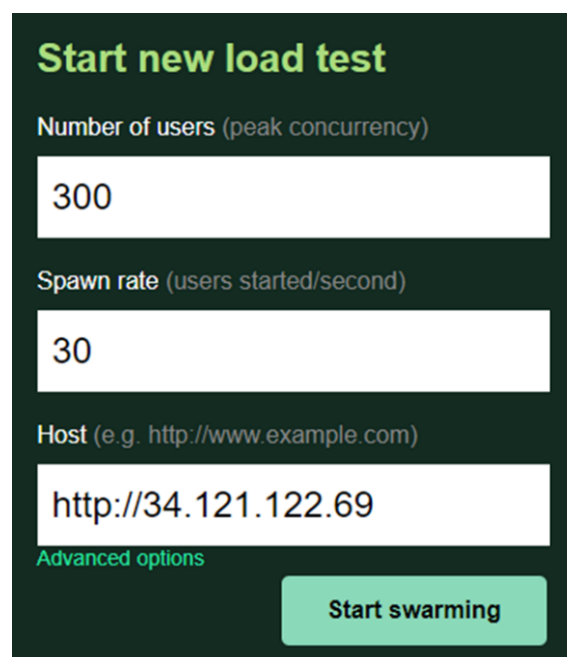
The image shows a web-based user interface for starting a new load test. It has a dark green background. At the top, the text 'Start new load test' is displayed in a light green font. Below this, there are three input fields. The first is labeled 'Number of users (peak concurrency)' and contains the value '300'. The second is labeled 'Spawn rate (users started/second)' and contains the value '30'. The third is labeled 'Host (e.g. http://www.example.com)' and contains the value 'http://34.121.122.69'. Below these fields, the text 'Advanced options' is visible in a small, light green font. At the bottom right, there is a light green button with the text 'Start swarming' in dark green.

Figure 2.7. Locust User Interface [19]

Chapter 3

Cluster Architecture and Implementation

This chapter presents the System architecture that was utilized to achieve the Thesis requirements as well as the affinity metrics that we developed to measure communication between microservices. This Chapter also introduces and analyzes the benchmark applications we used to test and implement our placement techniques, the automated service placement algorithm we implemented, and the configuration of the Kubernetes Horizontal Pod Autoscaler in our System.

3.1 System Architecture

To enable the communication of all of its microservices, each application operating on a Kubernetes Cluster must be initially properly configured. In our Thesis we set up a Kubernetes Cluster figure (3.1) in Google Cloud Platform (GCP). In the cloud, the Google Kubernetes Engine is in charge of managing the Cluster, and the Virtual Machines (Kubernetes Nodes) are located on the Google Compute Engine. The cluster is homogeneous, which means that every Node (VM) in the cluster have the same hardware specification, the same operating system and the same network configurations. As we wish to specify the initial number of Nodes that will be used to host each application and the specific resources for each Node, we have disabled the Kubernetes extensions for controlling Node size and resource allocation known as Horizontal and Vertical auto-scaling. In our work we only enable the kubernetes Horizontal Pod Autoscaler for the experiments.

Each Pod has a specific Service attached to it that enables network connection. It is nec-

essary to configure Pods by injecting Istio Envoy Proxies into each application's Pod in order to integrate the Istio Service Mesh into the application. Istiod is the Istio control plane that is randomly placed in a single Node and is in charge of injecting Envoy Proxies into the newly created Pods and monitoring the status of the current Envoy's. All the network traffic (internal or external) on the cluster is re-directed through the Envoy proxies injected into the Pods. Every newly created Node has a Prometheus Node exporter installed, which is in charge of exporting Node metrics from the Prometheus Server. These metrics can be retrieved and stored in real-time by the Prometheus Server. In a similar vein for the purpose of building the application's graph, Kiali is deployed in a cluster Node and obtains these data from the Prometheus Server.

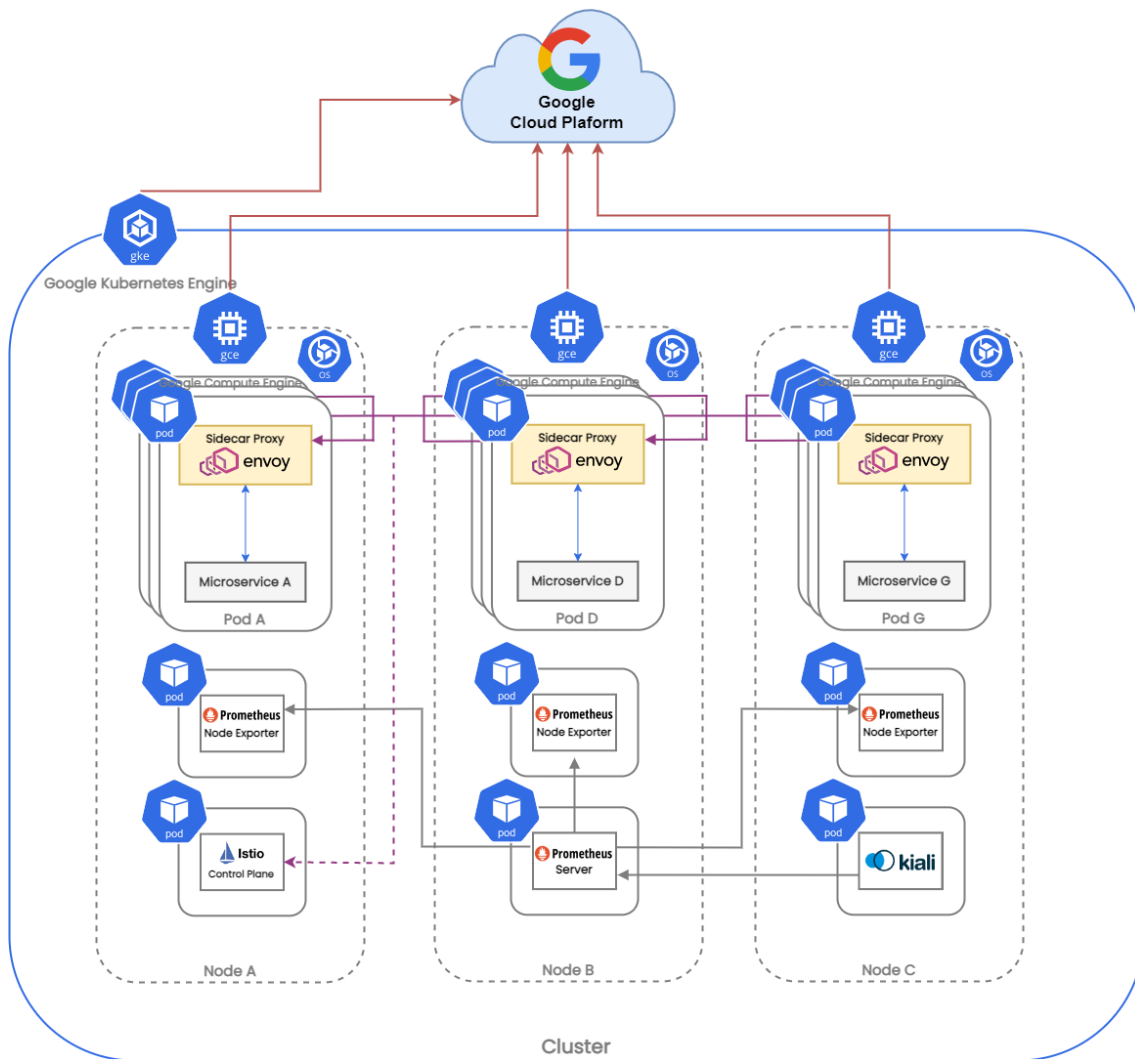


Figure 3.1. System Architecture in Google Cloud Platform

Each Kubernetes cluster has a specific number of Nodes that is user defined, based on the needs of each application. The Node and Pod creation and management are at the control of the Kubernetes cluster. The Envoy Proxies of the Pods in every Node in the cluster communicate via the Istio's Data Plane. Every Pod in a Node contains a microservice as a

3.2 Microservices Performance-Affinity Metrics

We propose two performance metrics in this section that we utilize to build our services graph and which we refer to as affinity metrics. These affinity metrics are incorporated into the application graph as weights.

Requests per Second (RPS)

Requests per Second (RPS) measures the network traffic rates between two microservices. Only the specified traffic rate between services is provided and no additional information regarding message size or number is provided. The Kiali Graph file includes data on the number of requests made per second for each pod for gRPC and TCP traffic over the course of a given period of the application's life. For TCP traffic, Kiali calculates the Bytes Per Second metric because in that case RPS metric is not reliable. In 3.1 we present the formula of RPS affinity metric

$$\text{RPS}_{S_i \rightarrow S_j} = \frac{\text{Sum of Requests from } S_i \text{ to } S_j \text{ in } T_{sec}}{T_{sec}} \quad (3.1)$$

Where,

- S_i is the Source Service
- S_j is the Destination Service
- T_{sec} is the Total Seconds of Measurement

Weighted Bidirectional Affinity (WBA)

Weighted Bidirectional Affinity (WBA) proposed by Adalberto R. Sampaio Jr [20], Julia Rubin, Ivan Beschastnikh, Nelson S. Rosa [[1]]. In order to calculate the affinity metric between two microservices, this metric takes advantage of the size of the messages that were sent in bytes and their overall number. The Weighted Bidirectional Affinity (WBA) formula is presented below.

$$A_{a,b} = w \cdot \frac{m_{a,b}}{m} + (1 - w) \cdot \frac{d_{a,b}}{d} \quad (3.2)$$

- $A_{a,b}$ is the affinity metric between service a and service b
- m is the total number of messages exchanged

- $m_{a,b}$ is the messages exchanged between a and b
- d is the total amount of data exchanged in bytes
- $d_{a,b}$ is the amount of data exchanged in bytes between service a and service b
- w is the weight, such that $\{w \in \mathbb{R} \mid 0 \leq w \leq 1\}$, used to define the significance of each affinity variable (size or count of messages)

The weights (w) is selected according to the importance of the variables, which are calculated to get the total affinity metric between two microservices. The importance of message size and number of messages can be equal to 0.5 because we don't have any strong preference between these two variables.

For example, if we choose $w = 0.2$ then we have a strong preference for the size of the messages, but because we use this metric in different applications that most probably we don't have any prior knowledge about their behavior is preferable to use $w=0.5$ and give equal preference to the size and number of messages.

3.3 Application Graph

Below is a presentation of the Kiali Graph we obtained for e-Shop and iXen applications. Circles are used to symbolize the deployments (Pods), while triangles are used to represent the corresponding Kubernetes Services. Green lines (measured in RPS) reflect HTTP and gRPC traffic between Pods, whereas blue lines (TCP traffic) are shown (and measured in BPS).

Based on data obtained from the Prometheus server and metrics taken from the Kiali API, our application graph is created. We use Kiali to extract the application microservices (Pods) and add a node for each microservice to our graph. The graph's edges are built using the communication edges exported by Kiali, and the edges weights are determined using the chosen affinity metric. For RPS affinity we use information from Kiali, for WBA affinity we use information's from Kiali and metrics extracted from the Prometheus server (information obtained from Istio Service Mesh).

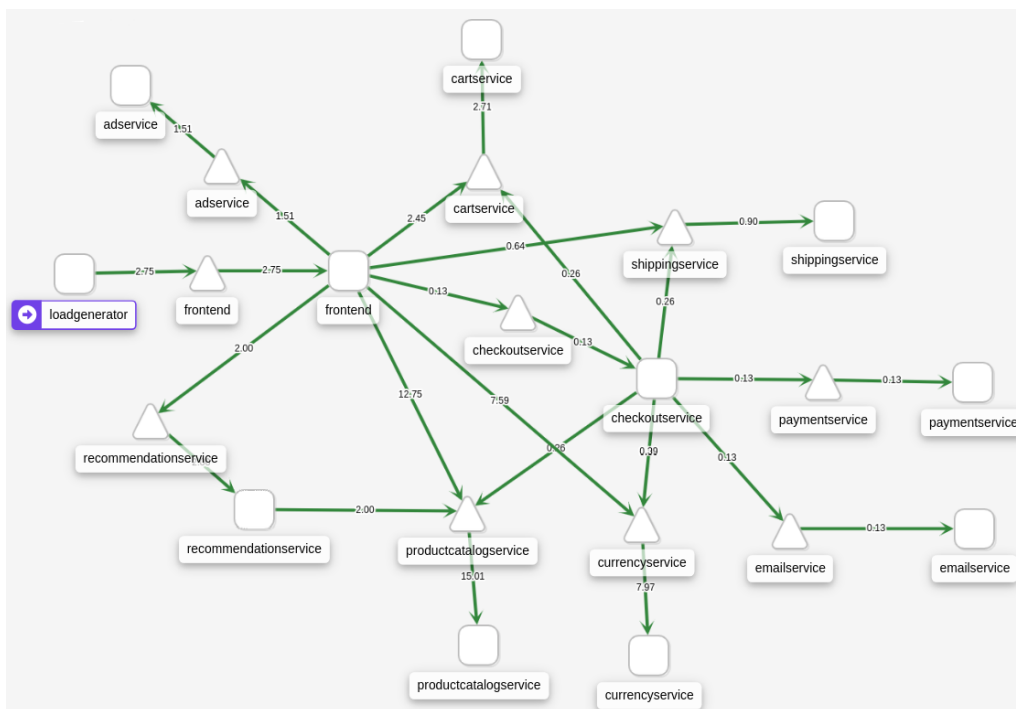


Figure 3.3. Kiali Graph for e-Shop

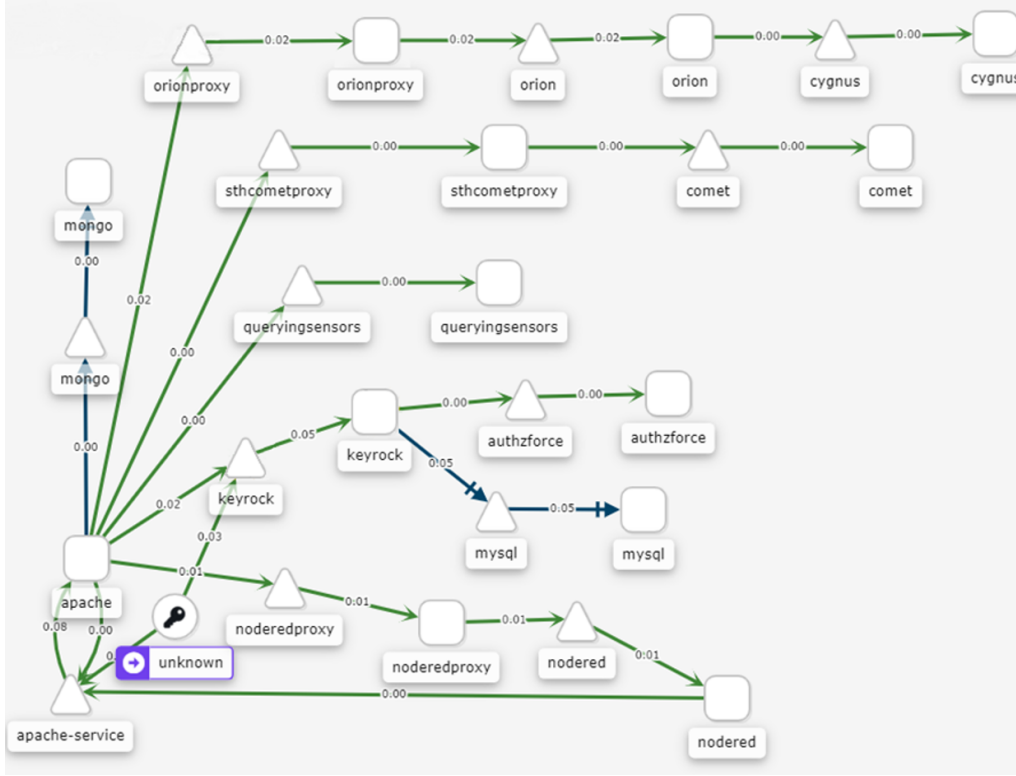


Figure 3.4. Kiali Graph for iXen

3.4 Benchmark Algorithms

3.4.1 Heuristic First-Fit

The Heuristic First Fit (HFF) algorithm is a processing algorithm that takes as input the application's initial service placement and configures it by relocating microservices with high affinity traffic rates on the same host machine. The microservices affinities of the application must be sorted in a descending order, in this manner, microservices with a higher affinity metric (RPS or WBA) are processed first, resulting in the most effective service placement possible. We save the source and destination nodes, their initial hosting VMs, and their resource requests in separate values during each iteration of the algorithm and for each processed affinity edge to make the algorithm execution process easier. If two services belong to the same Node, the algorithm terminates the current iteration and services are marked as migrated in order to remain on the same Node until the algorithmic execution ends. In contrary if microservices belongs at two separate Nodes, the algorithm looks at whether the destination service can move to the source host or if the source service can move to the destination host. Contrary to the other algorithms we use (Bisecting K-means, MODSOFT), Heuristic First Fit algorithm is not a clustering algorithm, but a single-step processing algorithm. The algorithm is only executed once,

and since microservices that have already been validated and tagged as moved cannot be moved for subsequent iterations, thus most executions result in suboptimal placement solutions. Additionally, it cannot be ensured that repeated iterations of the algorithm for the current service placement would result in the creation of an optimal service placement, and in some cases, it may possibly result in a solution with higher egress traffic. In algorithm 4 the HFF is presented, the explanation of the algorithm is presented in chapter 2.

Algorithm 4 Heuristic First Fit [1]

```

1: Input: Hosts (H), microservices (m), resources (r)
2: Output: Placement Solution
3: moved  $\leftarrow$  [ ]
4: //Affinities are sorted in decreasing order
5: for every pair of affinities do
6:    $m_i \in H_i$  //  $m_i$  located at host  $H_i$ 
7:    $m_j \in H_j$  //  $m_j$  located at host  $H_j$ 
8:    $m_j \neq m_i, H_j \neq H_i$ 
9:   hasMoved  $\leftarrow$  False
10:  if  $r(m_i) + r(m_j) \leq r(H_i) \wedge m_j \notin \text{moved}$  then
11:     $H_j \leftarrow H_j - m_j$ 
12:     $H_i \leftarrow H_i \cup m_j$ 
13:    hasMoved  $\leftarrow$  True
14:  else if  $r(m_i) + r(m_j) \leq r(H_j) \wedge m_i \notin \text{moved}$  then
15:     $H_i \leftarrow H_i - m_i$ 
16:     $H_j \leftarrow H_j \cup m_i$ 
17:    hasMoved  $\leftarrow$  True
18:  end if
19:  if hasMoved then
20:    moved  $\leftarrow$  moved  $\cup [m_i, m_j]$ 
21:  end if
22: end for

```

3.4.2 Bisecting K-Means (BKM)

Given a set of microservices, the Bisecting K-Means (BKM) algorithm generates a finite group of services with high affinity traffic. The number of K clusters we used is four ($K=4$) based on the work in [2]. BKM uses the K-Means algorithm and the Sum of Squared Errors (SSE), which is calculated from each cluster point using the centroid point. Bisecting K-Means algorithm primarily depends on the edges of affinities between the microservices. The affinity metric of the microservices can be used to estimate the error in order to satisfy the requirements of the Microservice Architecture and effectively apply the BKM algorithm. In **line 5** algorithm iterates while the Initial partition (P) is smaller than the Number K of desired clusters . In **lines 6-13** two sub-clusters with a greater affinity score should be created by separating microservices with low to no affinity metric and it is preferable that there be no communication edge between these microservices because

the algorithm will choose them as two clusters and stop processing the subsequent affinity edges. In **line 20** if there is no affinity edge or clear preference between them, the remaining microservices in the processed cluster are distributed randomly or in accordance with the affinity metric with the centroids of the chosen subclusters. Additionally, the assignment of microservices to the available cluster centroids is done only based on the affinity score with the cluster centroid, not with the other affinities pairs with the microservices inside the original cluster. Although this version of the BKM algorithm frequently yields suboptimal results it can be improved even more using heuristic packing. In algorithm 5 the BKM is presented.

Algorithm 5 Bisecting K-Means for Microservices Architecture

```

1: Input: Service-based application (S), Initial partition (P), Number K of desired
   clusters,
2:     Service affinities (A)
3: Output: Partition of Services in K Clusters
4:  $P \leftarrow \{S\}$ 
5: while  $\text{size}\{P\} < K$  do
6:   Select a cluster from P with the least sum of service affinities rates in total,  $C_i$ 
7:    $P \leftarrow P - \{C_i\}$ 
8:   Pick and remove two microservices - centroids,  $ms_x$  and  $ms_y$ , from  $C_i$ 
9:   with no or the least affinity rate between them
10:   $C_i \leftarrow C_i - \{ms_x, ms_y\}$ 
11:   $C_x \leftarrow ms_x$ 
12:   $C_y \leftarrow ms_y$ 
13:   $P \leftarrow P \cup \{C_x, C_y\}$ 
14:  for every microservice ( $ms_i$ ) in  $C_i$  do
15:    if  $A(ms_i \rightarrow ms_x) > A(ms_i \rightarrow ms_y)$  then
16:       $C_x \leftarrow C_x \cup \{ms_i\}$ 
17:    else if  $A(ms_i \rightarrow ms_x) < A(ms_i \rightarrow ms_y)$  then
18:       $C_y \leftarrow C_y \cup \{ms_i\}$ 
19:    else
20:      Select and place  $ms_i$  randomly among  $C_x$  and  $C_y$ 
21:    end if
22:  end for
23: end while
24: Return P

```

3.4.3 Fuzzy Partitioning Algorithm

The implementation of the algorithm [21] consists of three procedures, one for efficiently projecting the membership probabilities, one for calculating the modularity of the partitions, and one for initializing and updating the membership matrix p using gradient descent step. First in **lines 5-8** we have the initialization, it compute the graph's overall weight during initialization by adding the weights of all of the graph's edges. The weights of the

edges connecting each node in the graph are then added to determine the node's weighted degree. After that, it adds the weights of all the edges that connect each node in the graph to determine the node's weighted degree. In **lines 11-16** the MODSOFT membership update function is used to update the membership matrix p during the partitioning stage. The likelihood that each service will be in the same partition as another service is represented in the membership matrix p , and the probability distribution is represented by each row of this matrix, which adds up to 1. The projection step and the membership update function are both part of the update membership step, which returns the updated membership matrix p .

The modularity of the suggested partitions is assessed after the membership matrix p has been changed. Using the given modularity function (explained in Chapter 2) from the MODSOFT repository, it is simple to calculate. After the partitioning stage, modularity is computed, and the partitioning procedure is repeated until the rise in modularity drops below a threshold that has been established at 0.01. Calculating our service partitions using iterations of the final membership matrix p is the last step. The membership matrix's rows and columns stand in for the application services, and each cell's value is a number between 0 and 1 that denotes the likelihood that the row-service and column-service will be in the same partition. In **lines 18-24** for each row-service, a partition is constructed, and for each column-service whose value exceeds the threshold of 0.1, we add it to the partition of the row-service. Then we produce the partitions, arrange them according to the number of services in each partition, and then return them. MODSOFT algorithm produces fuzzy partitions, and by modifying the fuzziness parameter as we mentioned in Chapter 2, we can control how fuzzy the produced partitions will be, where more fuzzy partitions mean that there will be more services belonging to more than one partition. Finally, to ensure that the placement solution will fit in the available Nodes of the cluster and that the solution will be as optimal as possible by way of costs and resources, the Heuristic Packing algorithm is used, which produces our fuzzy placement. The implementation of the algorithm is presented in Listing 3.1.

Listing 3.1. Fuzzy Partitioning Algorithm

```

1: Input: Services Graph (G), Graph Nodes (N), Application Services (S),
   Threshold (T), Modularity Threshold (MT), Fuzzyness Parameter (t)
2: Output: Application Partitions (P)
3:
4: # Initialization
5: Calculate the total weight of graph G, total_weight
6: Calculate the weighted degree of each node N, degreenode
7:  $i \leftarrow 0$ , modularity $i$   $\leftarrow 0$ 
8: for node in G do:
    $p_{\text{node}} \leftarrow 1$ 
9:
10: # Partitioning
11: do:
12:    $i \leftarrow i + 1$ 
13:   # Update Membership matrix using MODSOFT [??]
14:    $p \leftarrow \text{update\_membership}(p, t)$ 
15:   modularity $i$   $\leftarrow \text{modularity\_func}(p)$ 
16: while: modularity $i$  - modularity $i-1$   $\leq$  MT
17:
18:  $k \leftarrow 1$ 
19: for service  $i$  in  $p$  do:
20:    $P_k \leftarrow \{S_i\}$ 
21:   for service  $j$  in  $p_i$  do:
22:     if  $p_{ij} > T$  then:
23:        $P_k \leftarrow P_k \cup \{S_j\}$ 
24: Sort  $P$  by partitions with most services , return  $P$ 

```

3.5 Automated Placement Algorithm

In this section we present the algorithm 6 that automates the service placement in the cluster . First is necessary to authenticate to the cluster with a client library (for using the Kubernetes API) and be able to update the deployments for each service. This algorithm uses Application Default Credentials to provide a Service Account key. Also is necessary before running the algorithm to run the default Kubernetes scheduler that will place the microservices (pods) to the nodes of the cluster and have the initial placement. Then, when

a service placement algorithm run and calculate the new host for each service the algorithm automatically update the YAML of each service (deployment) to migrate the service to the new host. In case of the MODSOFT algorithm we can also replicate a service (because MODSOFT is a Fuzzy Partitioning Algorithm). The algorithm for automated service placement can run inside the cluster (for example in a pod running python) or outside of the cluster and connect remotely to authenticate and run the algorithm to migrate/replicate the services (so we don't waste cluster resources). Here we show the steps of the algorithm:

Line 4 First, we get the Application Default credentials, the `project_id` is the Service Accounts, and this may differ to the clusters PROJECT name.

Line 7-9 Will gets the cluster config (configurations) from GCP. The input ClusterCred consist of the project name, cluster name, and the zone of the cluster.

Line 12-13 Will generates and loads a 'kubectl' config necessary for the authentication in the cluster. The kubectl config in Listing 3.2 has the following form where NAME is arbitrary, SERVER is the cluster endpoint and CERT is the cluster certificate. The 'cluster' object from line 8 contains the SERVER and CERT variables needed for kubectl config.

Listing 3.2. Example - Affinity in Deployment YAML

```
NAME="cname" # arbitrary
CONFIG=
apiVersion: v1
kind: Config
clusters:
- name: {NAME}
  cluster:
    certificate-authority-data: {CERT}
    server: https://{SERVER}
contexts:
- name: {NAME}
  context:
    cluster: {NAME}
    user: {NAME}
current-context: {NAME}
users:
- name: {NAME}
  user:
    auth-provider:
```

```
name: gcp
config:
  scopes: https://www.googleapis.com/auth/cloud-platform
```

Line 16 Will creates a cluster instance to be able to run kubectl commands to the cluster and make changes in the deployments.

Line 19-21 First, we get the initial placement of the services in the nodes of the cluster. Then we choose one of the three placement algorithms to execute using the input AlgoName who is the name of the placement algorithm we want to run, they are three options, BKM(K=4) – HP , HFF and MODSOFT - HP. Finally, we execute the selected placement algorithm and get the new placement (finalPlacement) of the services.

Line 23-27 For each service in the finalPlacement we check if the service was on a different host in the initial placement relative to the final placement, if that's true then we migrate the service to the new host. To migrate the service, we change the YAML file of the deployment service using Note Affinities. During the scheduling cycle, the Scheduler checks the Node Affinity specifications which state conditions that a Node must or must not include in order to be a feasible Node for the Pod's deployment. We specified the node affinity as the hostname of the new node that the service should migrate. In case of MODSOFT algorithm we also specified the number of replicas if a microservice have. If the microservice spawn's replicas then we specified the hostnames (because of replicas hostnames can be multiple).

When we migrate a service to a new node in the cluster, Kubernetes will wait until the service status in the new node is RUNNING, after that the service will be terminated in the old node. In that way we don't have any downtime in the application.

3.6 Benchmark Applications

To implement and evaluate the suggested service placement strategies, we use two benchmarking applications. The first application employs the HTTP and gRPC protocols and has 11 microservices, whereas the second application has 15 microservices. In a GKE Cluster (with specified resources which are discussed in more detail in the following chapter and the same infrastructure), both applications are deployed in a single, homogenous environment.

Algorithm 6 Automated Microservice Placement On GCP

```
1: Input : ClusterCred : consist of the project name, cluster name, and the zone of the
   cluster
2: AlgoName : The name of the placement algorithm to run
3:
4: credentials, projectid = getDefaultCredentials()
5:
6: //Get the cluster config from GCP
7: clusterManager = clusterManagerClient(credentials)
8: cluster = clusterManagerGetCluster(ClusterCred, clusterManager)
9: configuration = getClientConfiguration(cluster)
10:
11: //Create's a kubectl config
12: config = createConfig(ClusterCred, cluster)
13: kubeconfig = load(config)
14:
15: //Create Cluster Instance
16: apiInstance = createClusterInstance(configuration)
17:
18: //Service placement logic
19: initialPlacement = getInitialPlacement()
20: SelectedAlgorithm = chooseAlgorithm(AlgoName)
21: finalPlacement = executeAlgorithm(SelectedAlgorithm)
22:
23: for service  $\in$  finalPlacement do
24:   if initialPlacement(service)  $\neq$  finalPlacement(service) then
25:     updateDeploymentYaml(service)
26:   end if
27: end for
```

3.6.1 Google Online Boutique e-Shop

In order to show how to utilize tools like Kubernetes, Istio, and the gRPC protocol, Google uses the Online Boutique eShop [22], a cloud-native microservices example application. It is an 11 microservice web-based e-commerce application that allows users to carry out various e-commerce-related actions. The application is robust, it is implemented and is optimized for use with both the Google Kubernetes Engine and Istio, and it uses five different programming languages and two of the most popular service-to-service communication protocols (HTTP and gRPC). As a result, it is the perfect application to apply our architecture and our placement algorithms. On figure Figure 3.5 the application architecture is displayed. The application architecture for each service is presented below as described in [22].

- **Frontend Service (Go)**: Exposes an HTTP server that serves the website to the web

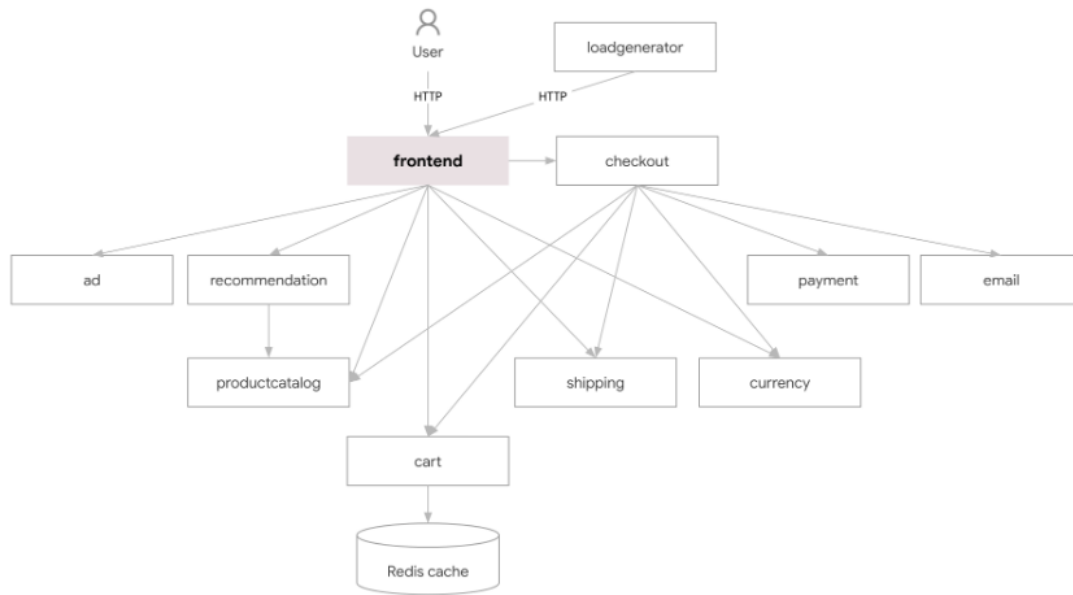


Figure 3.5. e-Shop Architecture [22]

and generates session IDs for all users automatically.

- **Cart Service** (*C#*): Stores and retrieves the items, users place on their shopping cart, in a Redis Database.
- **Product Catalog Service** (*Go*): Provides the list of products (read from a JSON file) and the ability to search and get individual products.
- **Currency Service** (*Node.js*): Fetches real currency values from the European Central Bank and converts one money amount to another currency. It is the highest QPS¹ service.
- **Payment Service** (*Node.js*): Charges the user-provided credit card info (mock) with the payment amount and returns a transaction ID.
- **Shipping Service** (*Go*): Estimates shipping cost based on the shopping cart and ships items to the given address (mock).
- **Email Service** (*Python*): Sends user an order confirmation email (mock).
- **Checkout Service** (*Go*): Retrieves the user cart, prepares the order, and orchestrates the payment, shipping, and email notification.
- **Recommendation Service** (*Python*): Recommends products based on what the user placed in its cart.
- **Ad Service** (*Java*): Provides text ads based on given context words.
- **Load Generator Service** (*Python/ Locust*): Simulates application traffic by continuously sending requests imitating realistic user shopping flows to the frontend service.

¹Queries per Second

3.6.2 iXen

The Technical University of Crete's Intelligence Lab developed the iXen [23] prototype application. It is an IoT (Internet of Things) application based on the service oriented architecture (SOA). For our work, we deployed the application in a GKE Cluster using the Kubernetes configurations that were supplied by Konstantinos Tsakos, who converted the application to a microservice-based architecture with independent microservices that can be deployed in a Google Kubernetes Cluster for orchestration.

Each tier of iXen's 3-tier architecture design model implements distinct logic specifically for its respective target user group. The infrastructure owners and system administrators, who have the ability to install and connect devices in the infrastructure, are included in the first-tier user group. Application developers, who can subscribe to sensors and construct applications using those sensors, are the second-tier user group. Customers who subscribe to developer-made applications make up the final user group. The iXen microservices are shown below on figure Figure 3.6 as they are described in the [23], along with a "load generator" microservice we created by the standards of the corresponding Boutique eShop service.

- **Web Service:** Provides a web interface via which users can interact with the application.
- **Keyrock Identity Management Service:** Provides a REST API for user registration, user rights policies, and uses OAuth2 tokens to authorize users.
- **AuthZForce Service:** XACML format is used to describe the respective user access privileges.
- **PEP Proxy Services:** Providing a security feature for services with a public interface. Each request to the "public" services is being forwarded through its respective PEP Proxy, and only requests from authorized users with service access are forwarded to the service.
- **Querying Sensors Service:** Converts a custom query syntax to mongo queries on the Mongo DB where devices are stored as entities, for the purpose of finding a device based on location, model type, type of measurement, or the unit of the measurement.
- **Orion Context Broker Service:** Publish/Subscribe service collects measurements from devices and makes them available to other services and users based on their subscriptions.
- **Cygnus Service:** Accepts data streams formatted in accordance with the NGSI model and is capable of storing them in a variety of databases.
- **Comet Service:** Reads Orion entities stored in a MongoDB and manages historical

sensor data.

- **Mashup Service:** Responsible for creating developers' applications with the aid of Node-Red, an open-source flow-based programming tool for the Internet of Things (IoT).
- **Load Generator Service:** Written in Python, this service continuously applies distributed requests (defined by the user) on the application's endpoints, simulating realistic user traffic and IoT devices' measurements - updates.

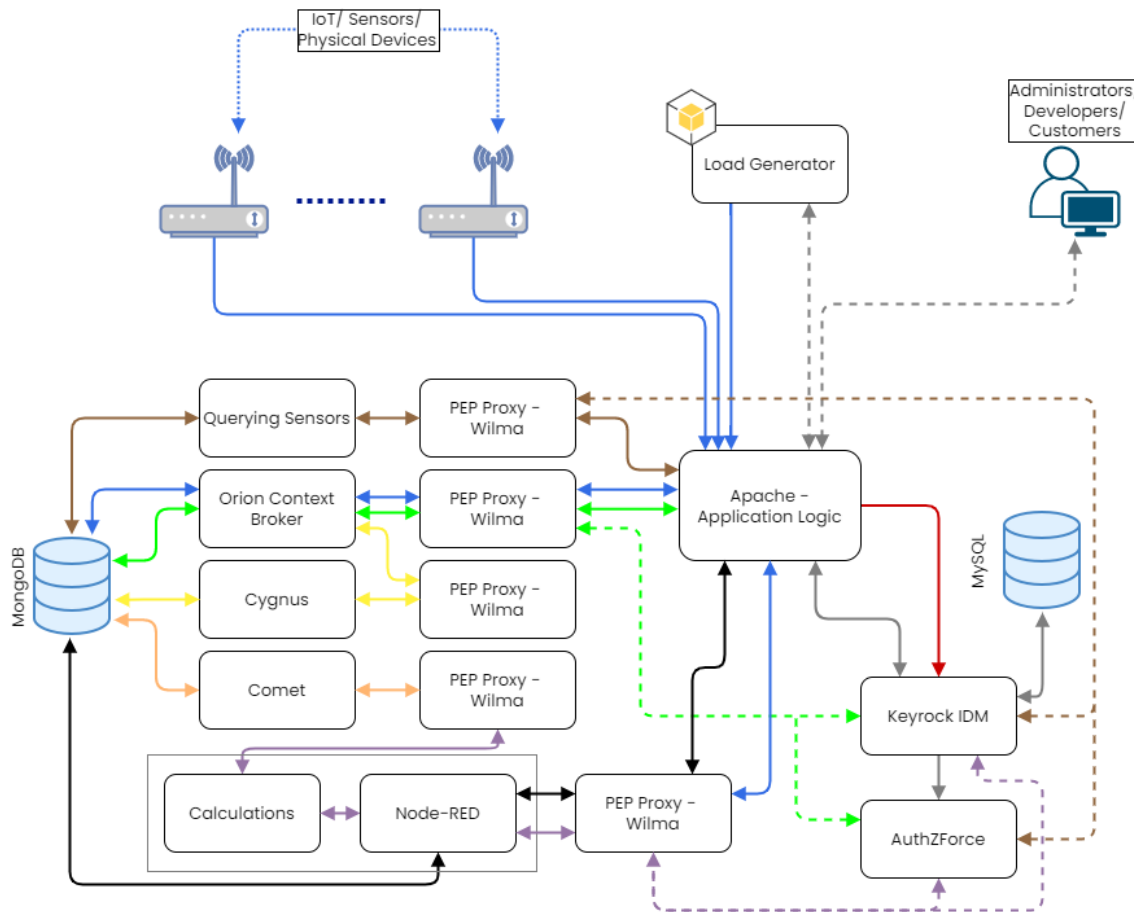


Figure 3.6. iXen Architecture

Chapter 4

Experimental Results

The tests used in our benchmarking applications will be introduced in this chapter, and their results will be analyzed. The infrastructure where the experiments were conducted will also be described, along with its costs. Then, for both applications, iXen and OnlineBoutique eShop, the application stress testing modules, the types of requests, and the distribution of these requests will be examined and presented. Finally, for the default Kubernetes placement and the service placement algorithms the results will be displayed in bar graphs.

4.1 Infrastructure

For the Kubernetes environment, we launch a Kubernetes cluster on the Google Cloud Platform (GCP), specifically the Google Kubernetes Engine (GKE), which will host the required cluster. The cluster was launched in the europe-west3-c region and the GKE version is 1.23.13. The horizontal and vertical **cluster** auto scaling are disabled so that the available Nodes (VMs) and their resource allocation remain the same for the algorithms implementation and experiments. Additionally the external load balancer of the GKE was also disabled to avoid any extra charges. The external load balancer defines how external traffic reaches our microservices and how the traffic is routed to our application. In our experiments it is not necessary because we use load balancer services to load balance traffic into the replicated pods of the same service. Anthos Service Mesh is disabled because in our implementation we use Istio service mesh. Cloud logging and Cloud Monitoring at the cluster were also disabled.

Cluster Attributes	Configuration
Location Type	Zonal
Zone	europe-west3-c
Release Channel	Regular
Cluster Version	1.23.13-gke.900
Horizontal Autoscaling	Disabled
Vertical Autoscaling	Disabled
HTTP Load Balancing	Disabled
Anthos Service Mesh	Disabled
Cloud Logging	Disabled
Cloud Monitoring	Disabled

Table 4.1. Cluster Characteristics

Our kubernetes cluster contains a node pool. The node pool consists of the VMs that will be created as Nodes (VMs) in our cluster. Because our infrastructure is homogeneous, all the virtual machines that are created in the node pool are e2-standard-2 type, this means that each has two virtual CPUs, eight gigabytes of RAM, and a typical boot drive with 40 gigabytes of storage. Furthermore, the node pool, and therefore the Nodes, are located in one zone of our region. The Operating System of each machine is a Container-Optimized OS. The Nodes of the Node Pool are not preemptible, thus we initialize each application by allocating resources or holding a limited number of virtual machines (VMs) on demand.

Node Pool Attributes	Configuration
Machine Type	e2-standard-2
vCPU	2
RAM	8 GB
Zone	europe-west3-c
Image	Container-Optimized OS with containerd
Autoscaling	Disabled
Boot Disk Type	Standard
Boot Disk Size	40 GB

Table 4.2. Node Pool Characteristics

The benchmarking application used to test our placement algorithms in addition to Istio, Kiali, and Prometheus require at least 2 host machines (Nodes) to operate effectively as mentioned in the previous work in [2]. We set up our cluster with 4 Nodes as the upper limit for our experiments and execute our placement algorithms to see if an optimized

placement can make use of fewer machines than our upper limit. Finally, It is considered surplus to host these applications with an initialized volume of Nodes greater than 4.

4.2 Benchmark Application Stressing

This section describes the stress testing procedure used to generate traffic flows for each application. To apply the stressing test for our application we use locust. With locust, we sent distributed requests to simulate a load from concurrent users on our application endpoints. We apply the stressing test for our benchmark applications with the Kubernetes default placement, and with the placement solutions produced by our algorithms and the Horizontal Pod Autoscaler disabled. Another set of stressing is applied with the Horizontal Pod Autoscaler enabled and the aforementioned algorithms to examine also the impact of HPA in the response times for each request and the monetary cost of the cluster. The stressing methods used for both iXen and OnlineBoutique applications will be thoroughly presented in the next subsections.

4.2.1 Google Online Boutique e-Shop Stress Testing

The default load generator for Google Online Boutique e-Shop is simulating 10 users. We altered this service and create two loads, one has 150 concurrent users, while the largest load has 300 concurrent users. For the stressing of 150 concurrent users we apply about 28,000 requests, and for the stressing of 300 concurrent users we apply about 57,000 requests. In tables 4.3 and 4.4 we display the request distribution.

Request	Request Type	# Requests	Distribution
Visit Homepage	GET	1249	4%
Show items in Cart	GET	3808	13%
Add item to Cart	POST	3816	13%
Submit an order	POST	1269	4%
Get a Product	GET	16197	56%
Change Currency	POST	2514	9%
# Total Requests		28853	

Table 4.3. Stress Testing Requests for Online Boutique e-Shop 150 users

4.2.2 iXen Stress Testing

For the Stress Testing of the iXen application, we needed to create a load generator service that would imitate 100 users making about 5.000 requests. We have already configured the

Request	Request Type	# Requests	Distribution
Visit Homepage	GET	2498	4%
Show items in Cart	GET	7616	13%
Add item to Cart	POST	7632	13%
Submit an order	POST	2538	4%
Get a Product	GET	32339	56%
Change Currency	POST	5028	9%
# Total Requests		57651	

Table 4.4. Stress Testing Requests for Online Boutique e-Shop 300 users

sensors and mashup applications, and developers and users can subscribe to them. Each simulated user first logs into the program and receives a cookie, which we keep and utilize for authentication in all subsequent requests he makes. In table 4.5 we display the request distribution.

Request	Request Type	# Requests	Distribution
Visit Homepage	GET	957	17%
Search Available Sensors	POST	632	11%
Subscribe Developer to Sensor	POST	545	10%
Search Applications	POST	639	12%
Search Application Subscriptions	GET	654	12%
Search Subscriptions to Sensors	GET	580	11%
Send Measurement to Sensor	POST	604	11%
Subscribe to Application	POST	264	5%
Deploy a Mashup Application	POST	91	2%
Access Mashup Application	GET	545	10%
Login into the App	POST	100	2%
# Total Requests		5520	

Table 4.5. Stress Testing Requests for iXen 10 users

4.2.3 Horizontal Pod Autoscaler Configurations

To use the HPA we need configure some basic characteristics. It is mandatory to set the minimum and maximum number of pods that the HPA can create, and also the average CPU utilization that the HPA uses to scale a service (HPA controller will increase and decrease the number of replicas to maintain an average CPU utilization across all Pods).

The average CPU utilization is set to *90percent*, the minimum number of pods is set to 1

and the maximum number of pods is set to 10. The maximum number of pods is set to a high number so no service can reach that number of replicas, this was done in order not to limit the HPA and get the right results. If we set a low limit and the HPA was not allowed to produce additional replicas that could possibly give us better results, then we would not have reliable results.

4.3 Placement Strategies

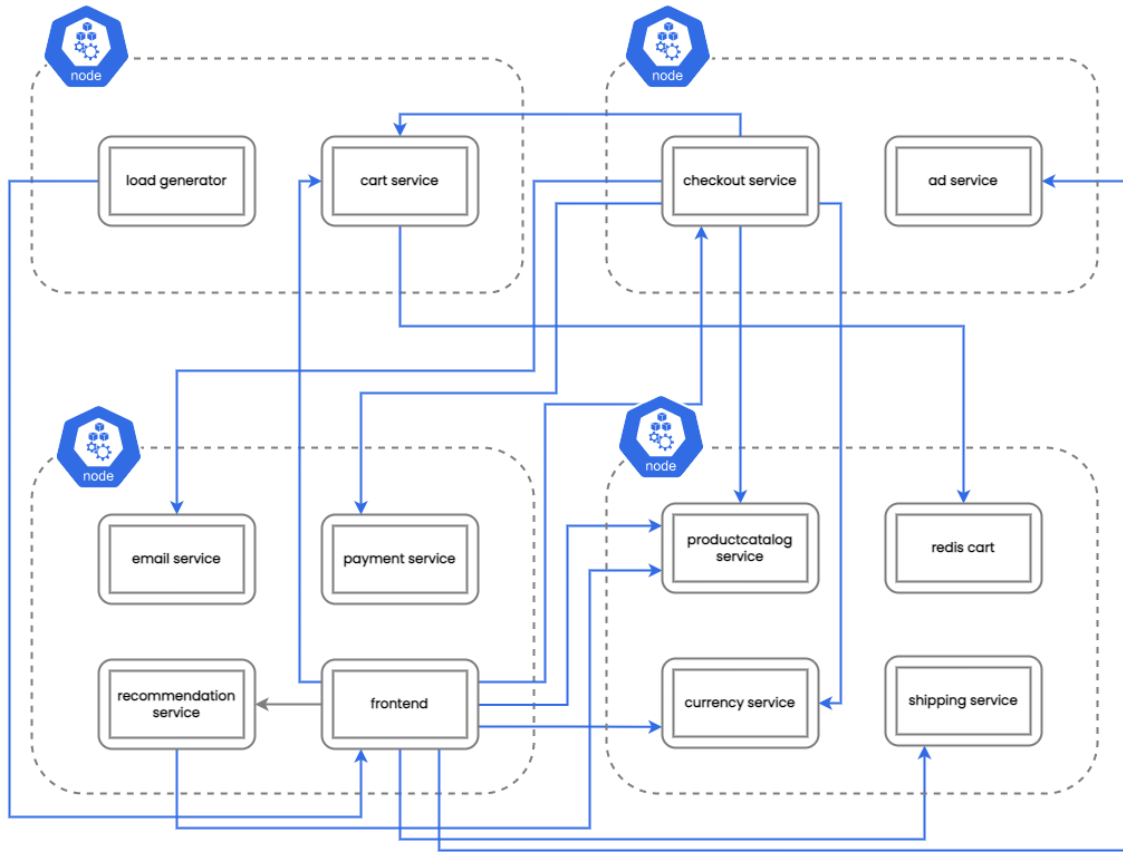
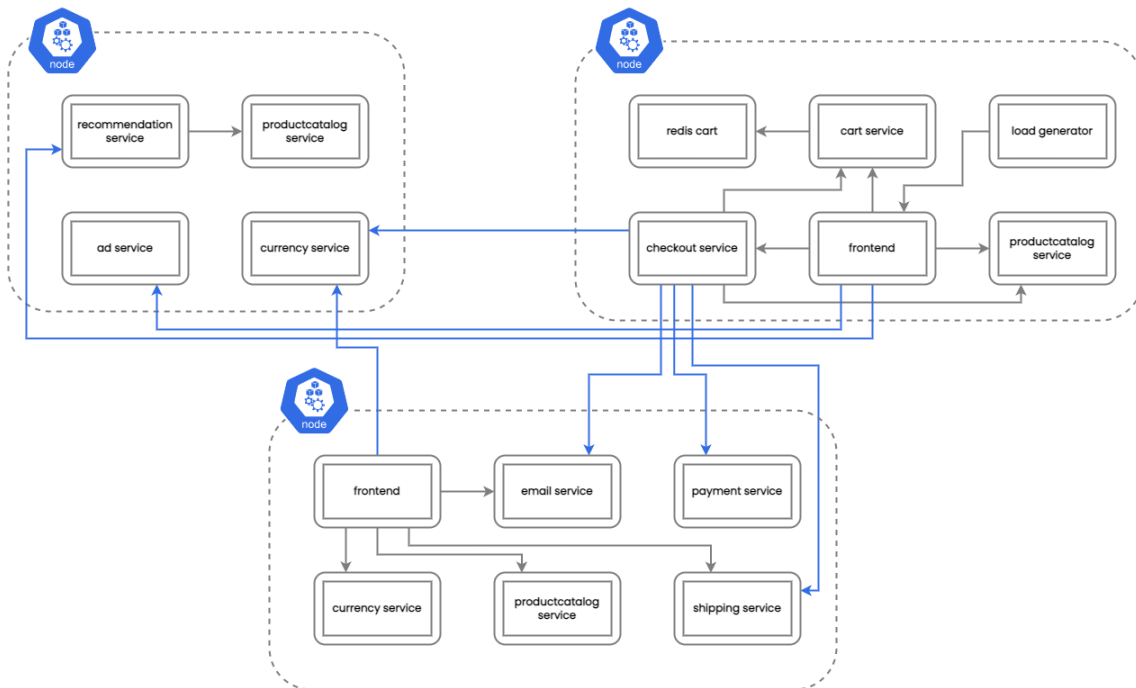
We have tested four different placement strategies. First placement strategy is the default Kubernetes scheduler who performs a default placement. When an application is deployed in Kubernetes without defining the Pod relations or Nodes that each Pod must be placed in (using Node and Pod affinities), the Kubernetes Scheduler produces a placement that is based mostly on available Node resources and the requested resources of each Pod. The second strategy we use is the Heuristic First Fit (HFF) [1] that minimizes the number of Nodes (VMs) and the Egress network traffic. The third method is the Bisecting K Means with the Heuristic Packing proposed in [2]. As we mentioned in chapter 3 we use $K = 4$ as the number of clusters based on the work in [2]. BKM-HP also reduces the total Nodes (VMs) number and egress traffic. The fourth and last placement is the fuzzy partitioning algorithm followed by Heuristic Packing (HP) and will be mentioned as MODSOFT-HP in the following chapters.

The figures below demonstrate the service placement and optimization of the traffic between the pods with the use of the fuzzy clustering with the Horizontal Pod Autoscaler (HPA) enabled and disabled.

In figure 4.1 we display the default service placement of eshop application from the Kubernetes Scheduler. Blue lines represent the egress service-to-service traffic, while gray lines represent the ingress traffic.

In figure 4.2 we display the fuzzy placement with the HPA disabled. The front-end service replicates in a different node, and because the service communicated with 7 more services most of the egress traffic converts to ingress traffic. Additionally, the productcatalog service also replicates in all available nodes to reduce the response time by reducing the overall load of each replica of the service.

In figure 4.3 we display the fuzzy placement with the HPA enabled. A replica of recommendation service and currency service is placed in the upper-left node and a replica of front-end service and currency service is placed on the bottom node.

Figure 4.1. Traffic between services for **Default** Online Boutique e-Shop PlacementFigure 4.2. Traffic between services for **MODSOFT-HP** Online Boutique e-Shop Placement and HPA disabled

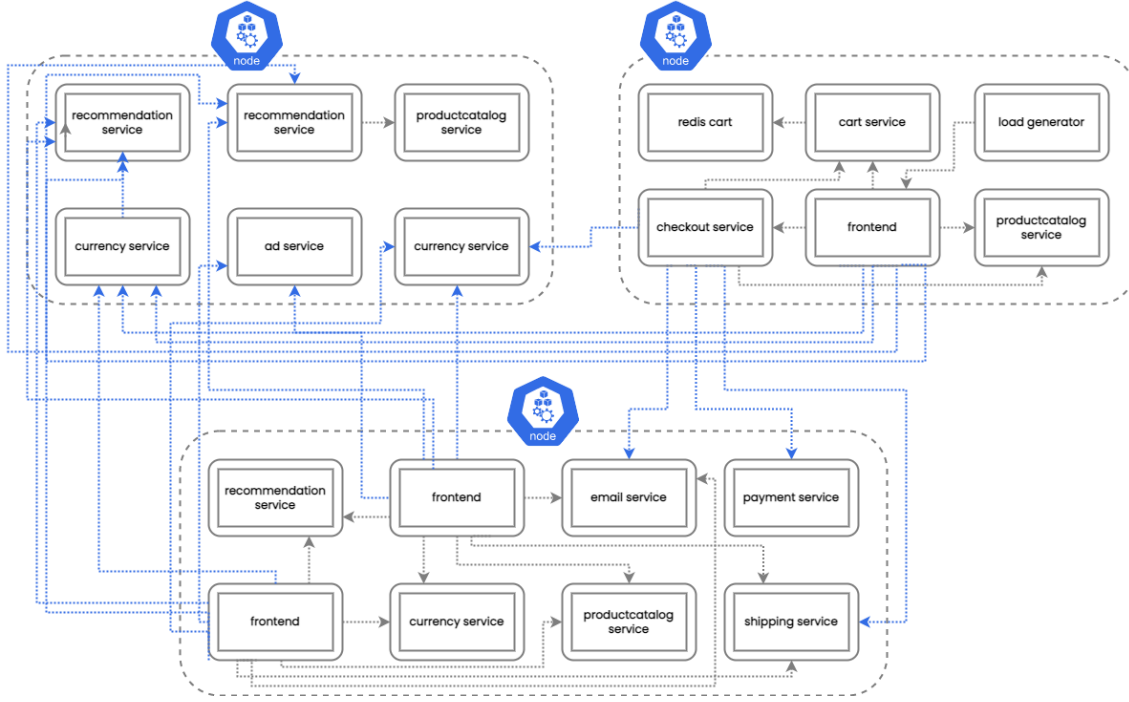


Figure 4.3. Traffic between services for **MODSOFT-HP** Online Boutique e-Shop Placement and HPA enabled

4.4 Infrastructure Cost and Cost Function

This section will display the cost function of our deployed cluster and identify the variables that can change this function. Our Kubernetes cluster run on GCP, and therefore the cost function is based on GCP charges [24]. The allocation of CPU and RAM is the major aspect that GCP charges for, and rates vary by area, for example the price of CPU and RAM in the region of Iowa(us-central1) differs from the region of London(europe-west2). Additionally the storage space is charged by GCP, but is relatively low in our cluster Nodes in relation to CPU, RAM and is regarded negligible. GCP only charges the egress traffic base on the amount of the exchanged bytes. The ingress traffic in not charged. In table 4.6 we present the prices for CPU, RAM and traffic for each VM.

Resource	Cost (USD)
Predefined vCPU	\$0.028103/vCPU/hour
Predefined RAM	\$0.003766/GB/hour
Egress Traffic	\$0.01/GB

Table 4.6. GCP pricing for e2-standard machine type

Each machine's CPU and memory costs are determined by the resources allocated every hour by GCP. The CPU is charged by virtual core per hour and the RAM is charged by

the amount of GB allocated in our virtual machine. As displayed below the total cost of a Node (VM) in our cluster is the sum of the total vCPU used and the amount of allocated RAM.

$$\begin{aligned}
 \text{Cost}_{\text{CPU}} &= 2\text{vCPU} \times \text{vCPU}_{\text{cost}} \times \text{hours} \\
 &= 2 \times 0.028103 \times \text{hours} \\
 &= 0.056206 \times \text{hours}
 \end{aligned} \tag{4.1}$$

$$\begin{aligned}
 \text{Cost}_{\text{RAM}} &= 8\text{GB} \times \text{RAM}_{\text{cost}} \times \text{hours} \\
 &= 8 \times 0.003766 \times \text{hours} \\
 &= 0.030128 \times \text{hours}
 \end{aligned} \tag{4.2}$$

Based on the volume of traffic exchanged between nodes, each cluster's network traffic is charged (egress traffic). The CGP only charged the bytes from the requests and not from the responses. The total cost of egress traffic can be calculated by adding the requested bytes between microservices in different Nodes. The price per GB it depends in the location (region) of each Node. In our work all the Nodes are in the same Google Cloud region and the price is 0.01 dollars per hour. On the contrary, if we have egress from Indonesia or Oceania to any other Google Cloud region the price can be up to 0.15 dollars per hour who is a significant change. Below we present the egress traffic cost function for our cluster.

$$\begin{aligned}
 \text{Cost}_{\text{Traffic}} &= \sum_i^N \sum_j^N t_e(i \rightarrow j) \times \text{cost}_{\text{egress}} \\
 &= \sum_i^N \sum_j^N t_e(i \rightarrow j) \times 0.01
 \end{aligned} \tag{4.3}$$

Where,

$$t_e(i \rightarrow j) = \begin{cases} t(i \rightarrow j), & \text{if } i, j \text{ not in the same node} \\ 0, & \text{in the same node} \end{cases} \tag{4.4}$$

The total cost of the cluster for any number of Nodes can be calculated using the following formula:

$$\begin{aligned}
\text{TotalCost} &= \text{TotalCost}_{\text{CPU}} + \text{TotalCost}_{\text{RAM}} + \text{TotalCost}_{\text{Traffic}} \\
&= n \times (\text{Cost}_{\text{CPU}} + \text{Cost}_{\text{RAM}}) + \text{TotalCost}_{\text{Traffic}} \\
&= n \times (0.086334 \times \text{hours}) + 0.01 \times \text{GB}_{\text{egress}}
\end{aligned} \tag{4.5}$$

Because in this work as we mentioned before we used 4 machines as Nodes, we can calculated the initial placement cost of our cluster using the following function:

$$\text{TotalCost} = 0.345336 \times \text{hours} + 0.01 \times \text{GB}_{\text{egress}} \tag{4.6}$$

4.5 Experiemntal Results

In this section we will analyze the results of fuzzy service placement MODSOFT-HP compare to the default Kubernetes Scheduler, the hard clustering Bisecting K Means with Heuristic Packing service placement (BKM-HP), and the Heuristic First Fit (HFF) service placement algorithm. Additionally the Horizontal Pod Autoscaler (HPA) replicate pods (like fuzzy clustering) if overloaded with requests. We present a set of experiments with the HPA enabled and with the HPA disabled to examine the impact of Horizontal Pod Autoscaler.

For each service placement, we will display the execution time, the number of hosts used, the egress traffic, the infrastructure cost and the average response time of requests. All experiments (except Execution Time) display the results with the HPA enabled and HPA disabled and with RPS and WBA affinities.

4.5.1 Execution Time

An important factor of each placement strategy is the execution time of each algorithm. Execution time, is the time each algorithm needs to produce the final service placement. The graph creation is not part of the execution time. In figures 4.4 and 4.5 we present the execution time of each placement algorithm. As we expected, the lowest execution time is from the HFF algorithm, this is because the HFF is a heuristic algorithm and a single-step processing algorithm. The execution time of MODSOFT-HP is greater than execution time of HFF and BKM-HM. The MODSOFT-HP generates more service replicas and more partitions than the BKM-HP, therefore the higher execution time it is

expected. The RPS affinity in both application is a little faster than the WBA affinity, but the execution for both affinities in both applications as we can see, completes within a few milliseconds and is acceptable. Finally all algorithms run on a local machine with 2.6 GHz processor power, 4 cores, and 16 GB RAM.

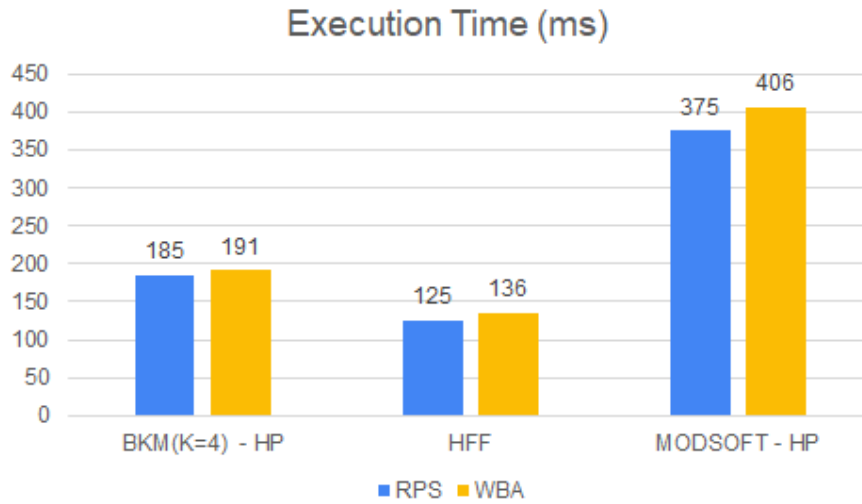


Figure 4.4. Execution Time for Online Boutique e-Shop

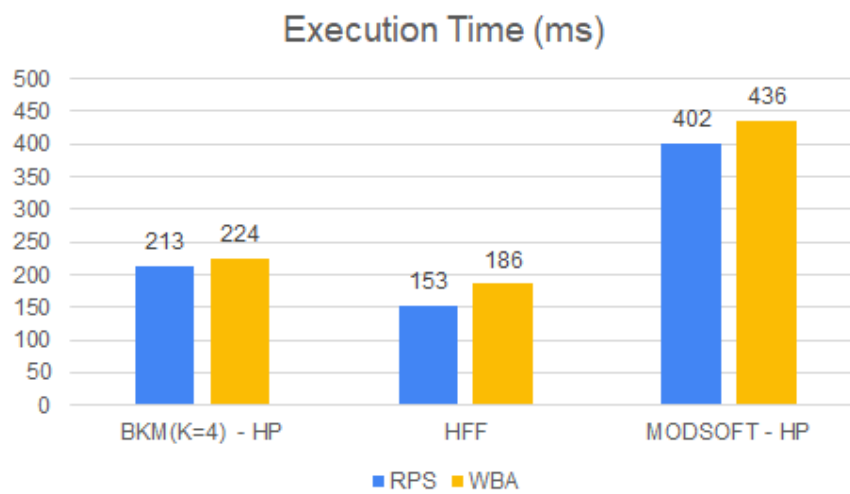


Figure 4.5. Execution Time for iXen

4.5.2 Number of Hosts

In figures 4.6 , 4.7 and 4.8 we display the Node utilization for each service placement. The numbers of utilized Nodes is critical in terms of costs of the infrastructure. The default

Kubernetes scheduler as we mentioned previously is utilizing 4 Nodes. With the Horizontal Pod Autoscaler (HPA) disabled the HFF produced a placement that utilizes 2 Nodes for both affinities for the Online Boutique e-Shop experiments and 3 Nodes for the iXen. BKM-HP utilizes 3 Nodes for iXen but for Online Boutique e-Shop that number can be reduced in just 2 nodes for RPS affinity in the Online Boutique e-Shop with 150 users and during the WBA affinity for Online Boutique e-Shop with 300 users. MODSOFT-HP requires 3 nodes for Online Boutique e-Shop and 4 Nodes for iXen experiments, this is expected because MODSOFT-HP produces more pods due to replication of services. With the Horizontal Pod Autoscaler (HPA) enabled all service placement in iXen utilizes 4 Nodes and 3 nodes in the Online Boutique e-Shop experiments.

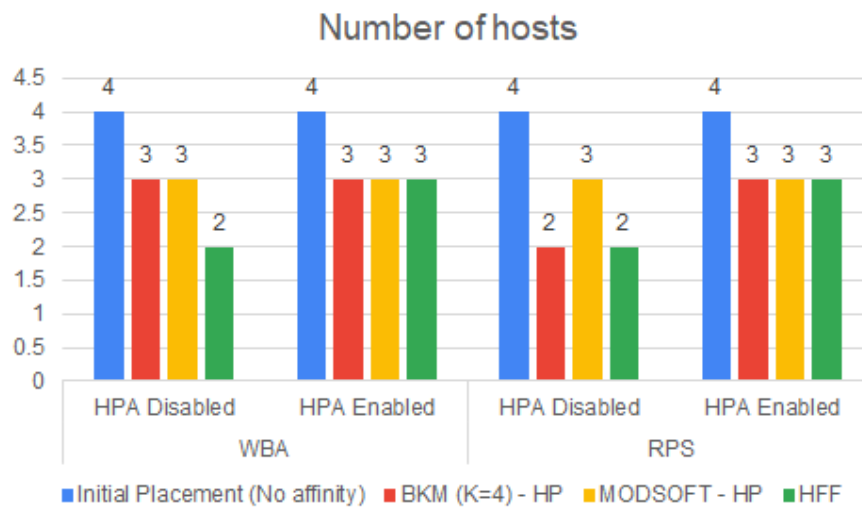


Figure 4.6. Number of utilized Hosts for Online Boutique e-Shop with 150 users

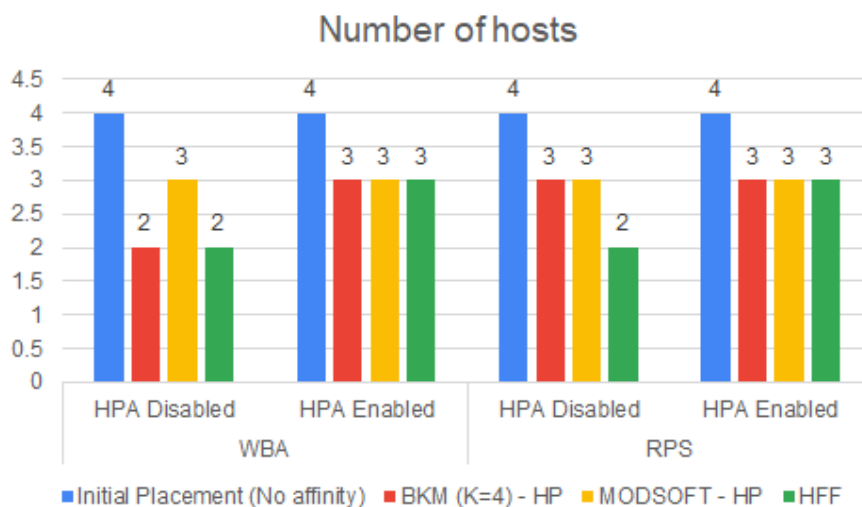


Figure 4.7. Number of utilized Hosts for Online Boutique e-Shop with 300 users

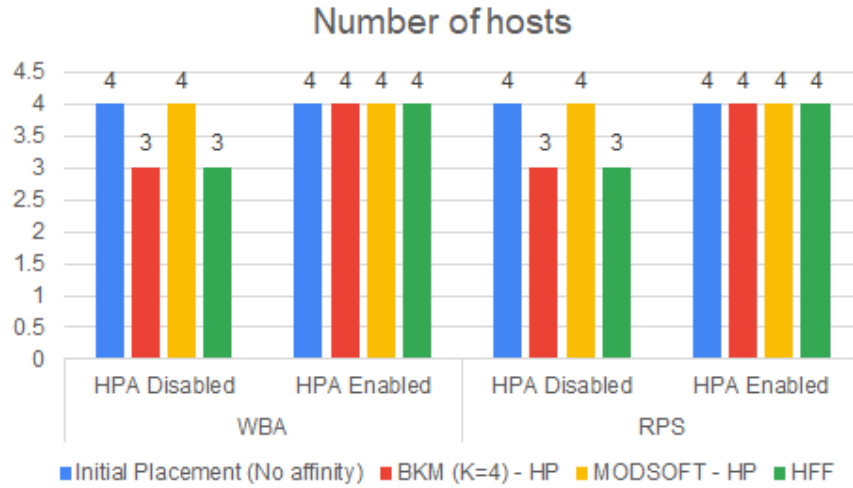


Figure 4.8. Number of utilized Hosts for iXen

4.5.3 Egress Traffic

Figures 4.9, 4.10 and 4.11 present the results for the egress traffic reduction using the service placement algorithms. In general for eShop and iXen applications we achieve a large reduction in requested egress traffic per hour for both affinities (RPS and WBA).

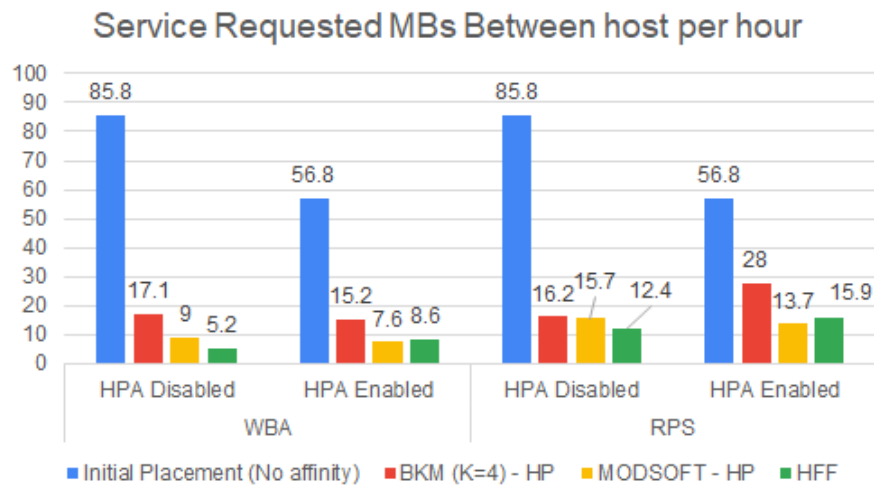


Figure 4.9. Hourly requested MBs for Online Boutique e-Shop with 150 users

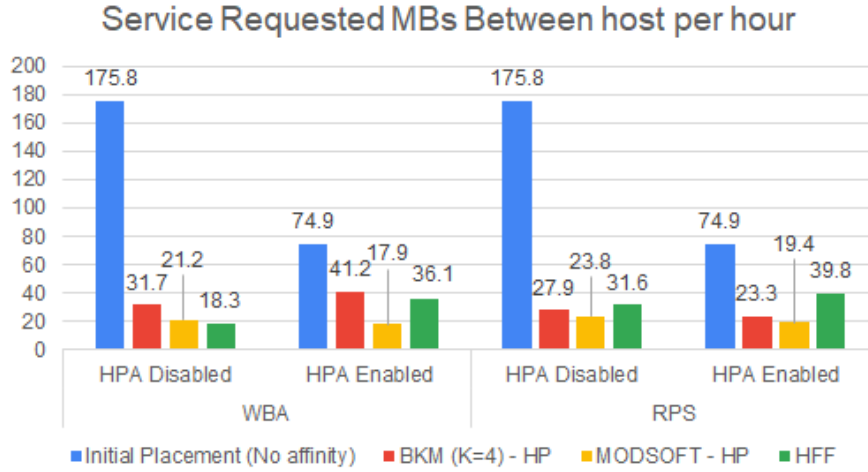


Figure 4.10. Hourly requested MBs for Online Boutique e-Shop with 300 users

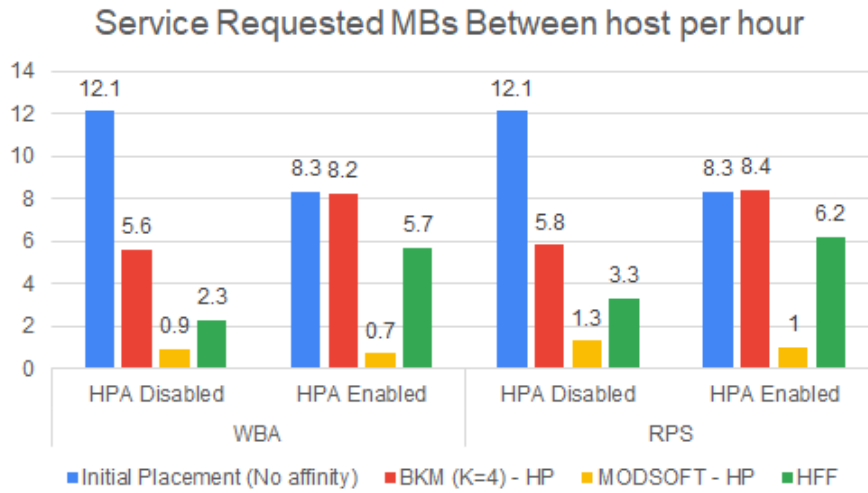


Figure 4.11. Hourly requested MBs for iXen

The HFF algorithm achieve the lowest egress traffic for eshop with 150 users for both affinities and for eshop with 300 users with WBA affinity, this was expected because HFF utilize only 2 Nodes and in most cases is places as many services as he can on the same Node. In iXen, MODSOFT-HP reduce the monthly egress traffic by 92% while in eshop the reduction in monthly egress traffic is around 89%. Additionally, as we can see from the results for both applications, egress traffic was not significantly impacted by HPA.

4.5.4 Total Monthly Infrastructure Cost

In this section we calculated the monthly infrastructure cost for each service placement algorithm using the Equation 4.5. We can see from the Equation that the cost depends

mostly on how many virtual machines are used to host each placement and the type of virtual machines. Since we use homogeneous cloud environment the type of VMs is the same for every placement and this is not considered a comparing variable. Furthermore, again due to homogeneous cloud environment the total cost is also impacted by egress traffic, but not as much as the number of utilized VMs. In case of a heterogeneous cloud environment the total cost of egress traffic can be even 15 times more expensive.

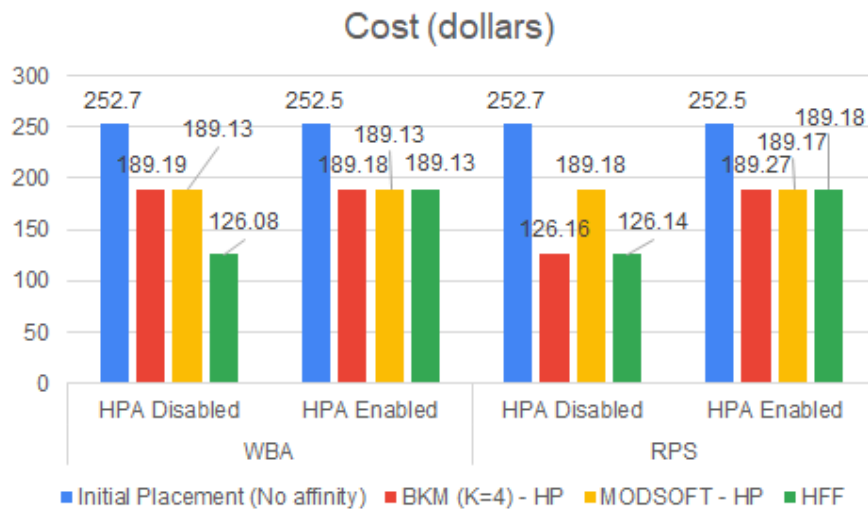


Figure 4.12. Estimated Monthly Cluster cost for Online Boutique e-Shop with 150 users

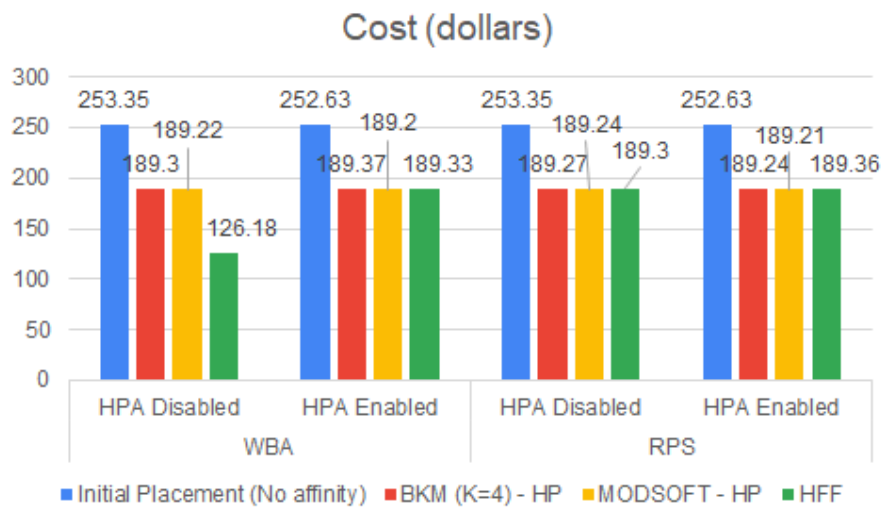


Figure 4.13. Estimated Monthly Cluster cost for Online Boutique e-Shop with 300 users

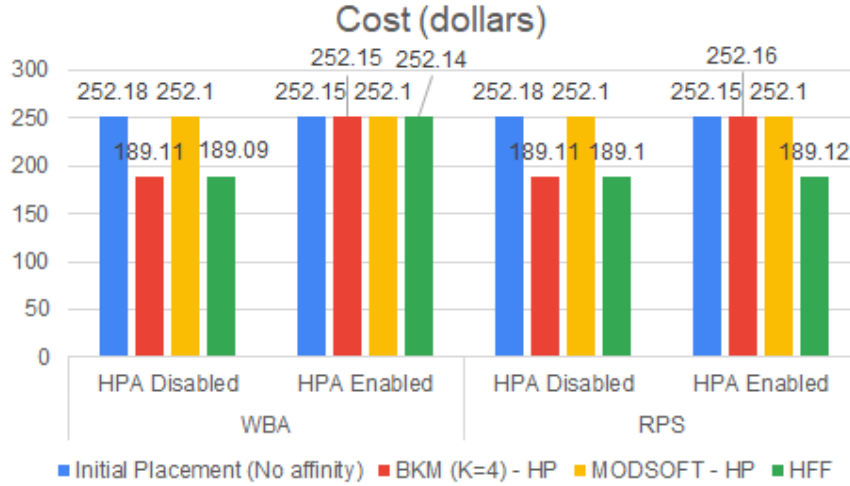


Figure 4.14. Estimated Monthly Cluster cost for iXen

In 4.12, 4.13 and 4.14 we display the total monthly cost for each placement for both affinities and with the HPA enabled and disabled. In eshop BKM-HP and MODSOFT-HP reduce the total cost by at least 25%, while the HFF in some cases can reduce the total cost by 50%. The MODSOFT-HP is unable to decrease the iXen application's monthly cost while the HFF and BKM-HP reduce the total cost by at least 25%. MODSOFT-HP was expected to be unable to reduce the monthly cost because it produces more pods due to replication of services. The impact of HPA in the total cost is insignificant in most cases, we observe only an increase in cost when the algorithm utilizes 2 nodes in eshop and in BKM-HP with WBA affinity in iXen.

4.5.5 Average Response Time

Response time is a critical factor for every application. In figures 4.15, 4.16 and 4.17 we present the average aggregated response times for both application, service affinities, and with HPA enabled and disabled. The three different stress testing we applied with their corresponding distribution is shown in Table 4.3, Table 4.4 and Table 4.5. As we can see MODSOFT-HP significantly decreased the average response time for both applications. That was expected because services that have a heavy load (receives a big volume requests) are replicated in multiple instances, and the incoming requests are load-balanced between the replicated services and requests can be forwarded faster to their targets. This results in an overall reduction in response time.

Additionally, the response time was not reduced by any placement algorithm in e-Shop experiments during the RPS affinity, but with WBA affinity BKM-HP and MODSOFT-HP

both reduce the response time. In case of iXen application MODSOFT-HP reduce the response time by an outstanding 84% using the RPS affinity. In general MODSOFT-HP reduced in almost all cases the response times of the applications, especially with the HPA disabled. When HPA is enabled it creates replicas of services to balance the load, and so in almost all experiments we can see a performance improvement.

Furthermore, the HFF service placement algorithm in both eshop experiments significantly increases the application response time. This happens in both affinity metrics and is due to the fact that HFF algorithm reduced the utilized nodes from 4 to 2. Consequently, the same load of requests is split between 2 nodes (where is the minimum number of required Nodes), resulting in worst response times due to the VM's load.

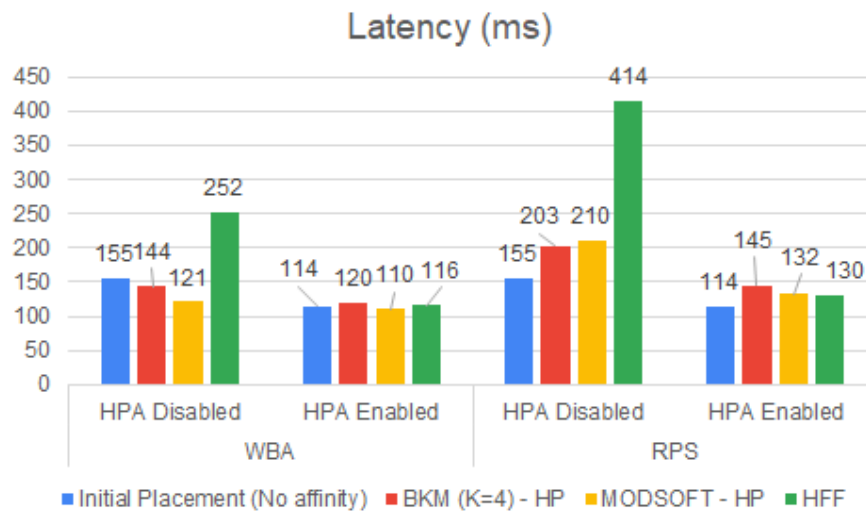


Figure 4.15. Average Response Time for Online Boutique e-Shop with 150 users

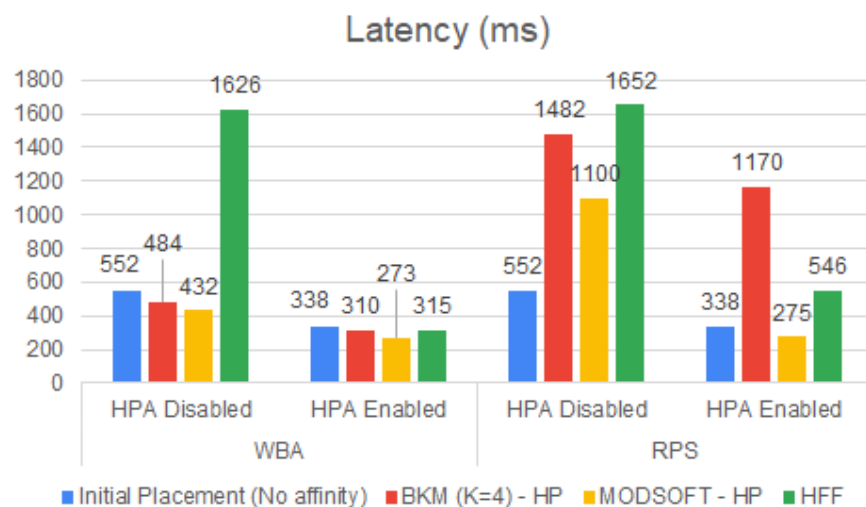


Figure 4.16. Average Response Time for Online Boutique e-Shop with 300 users

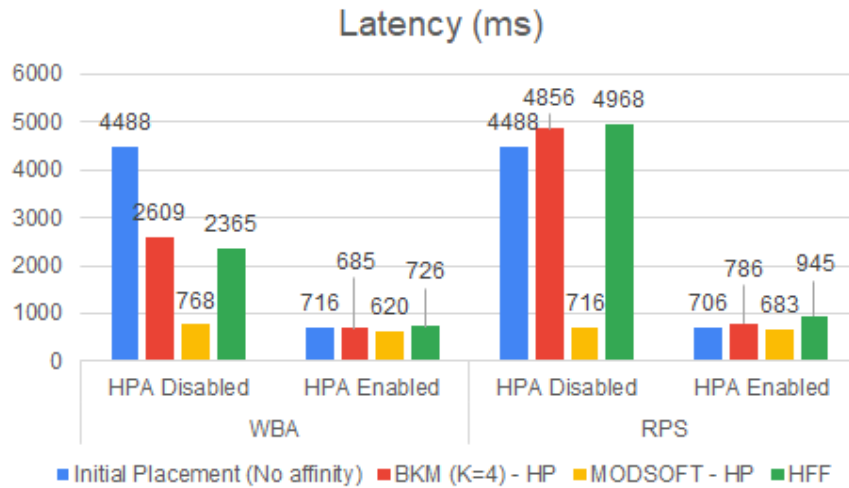


Figure 4.17. Average Response Time for iXen

4.5.6 Response Time for the 90%ile of Requests

In figures 4.18, 4.19 and 4.20 we present the results of the response times for the 90%ile of requests. The 90%ile refers to the 90% of the faster requests. MODSOFT-HP reduces the 90%ile response time for iXen application but not significantly, because as we can see the response times were already reduced. In eshop experiments using WBA affinity, MODSOFT-HP and BKM-HP produce the best results by reducing the response time, while HFF in both affinity metrics produces a significant increase similarly to the average response time experiments.

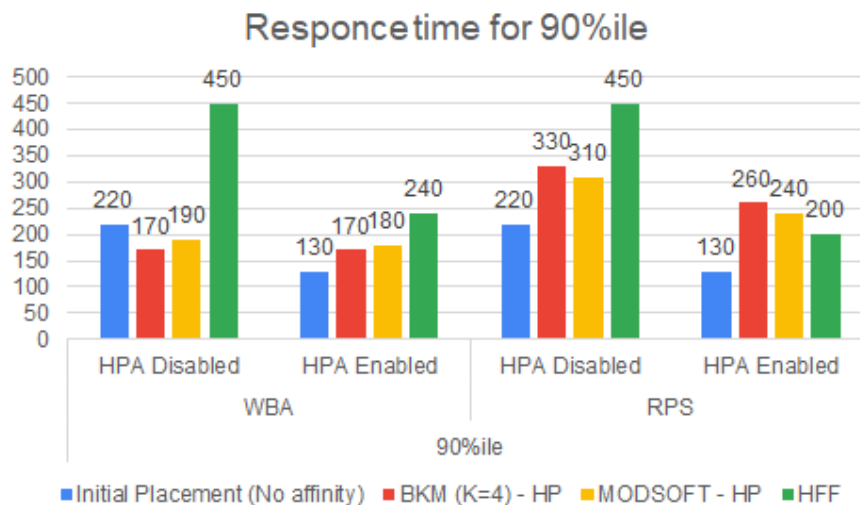


Figure 4.18. Response Time for the 90%ile of Requests for Online Boutique e-Sho with 150 users

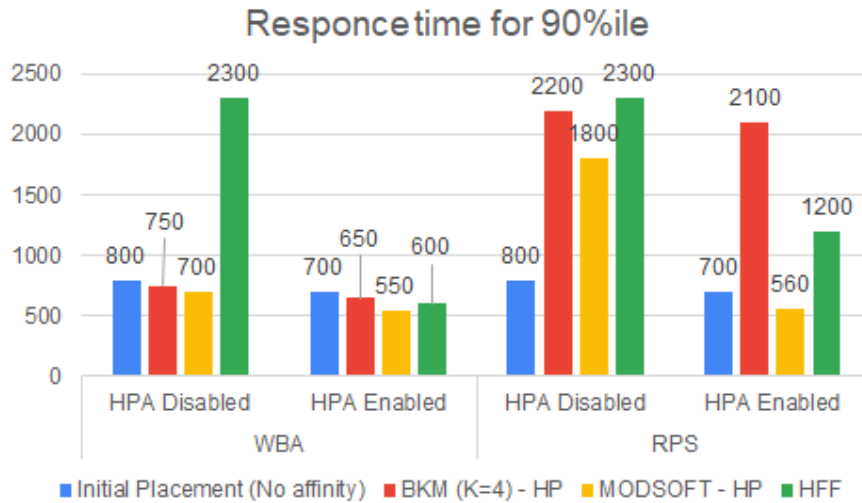


Figure 4.19. Response Time for the 90%ile of Requests for Online Boutique e-Shop with 300 users

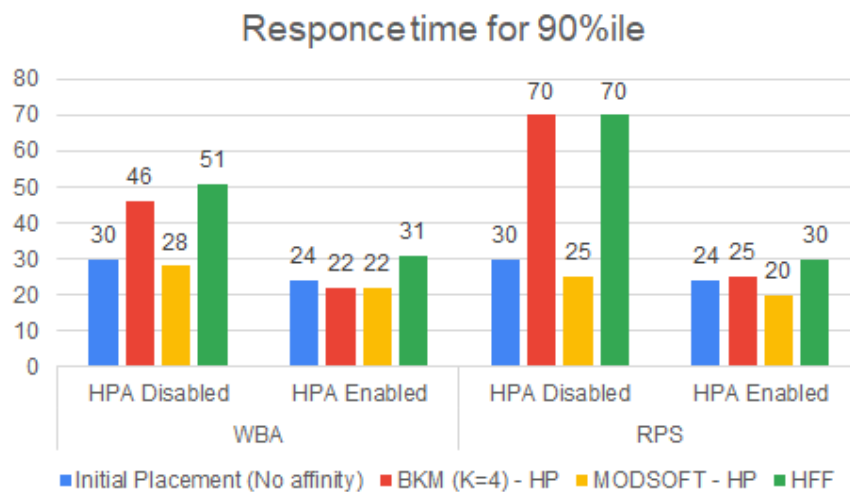


Figure 4.20. Response Time for the 90%ile of Requests for iXen

For iXen in general we can see a significant reduce of of the 90%ile response time in contrast with the average response time. This happens because we leave out the 10% of the requests with a significant slow response time.

Conclusion and Future Work

In this work we examine the service placement problem in a homogeneous Kubernetes Cloud Environment. We aimed to reduce the response time of the application, but also the overall cost of the infrastructure using a fuzzy placement algorithm and the effect of HPA on that algorithm. To reduce the response time of an application you need to create and run multiple instances (replicas) of a service on a cluster to reduce and balance the load of high-utilized services. The service placement problem is addressed as graph-based clustering where services represent the nodes of the graph and the service affinities represent the edges. The two affinity metrics we use to calculate the communications affinities are the Request Per Second (RPS) and Weighted Bidirectional Affinity (WBA). The fuzzy placement uses a heuristic packing method to place the generated partitions into the Nodes of the Cluster, in such a way that network traffic and resource (CPU, RAM) use are optimized and also reduce the infrastructure cost.

Additionally, we create an algorithm to automate the service placement. The algorithm works as an extension of the Kubernetes default scheduler. It is necessary before running the algorithm to run the default Kubernetes scheduler that will place the microservices (Pods) to the nodes of the cluster and have the initial placement. The algorithm for automated service placement can run inside the cluster (for example in a pod running python) or outside of the cluster and connect remotely to authenticate and run the algorithm to migrate/replicate the services (so we don't waste cluster resources). Furthermore with automated service placement algorithm when we migrate a service to a new node in the cluster we don't have any downtime in the application.

To test the service placement algorithm we utilize two applications, the Google's Online Boutique e-Shop and the iXen IoT application. We integrated an Istio Service Mesh into these applications to monitor the Pods and Nodes resources and calculate the performance metrics required to apply the placement strategies.

The experimental results show that MODSOFT-HP can reduce the average response time up

to 80% compare to Default Kubernetes Scheduler, BKM-HP and HFF with the Horizontal Pod Autoscaler (HPA) disabled. If HPA is enabled we can see a response time reduction up to 15%. The effect of the MODSOFT-HP with the HPA disabled is much larger due to the fact that the HPA creates replicas of services at run time and reduces the response time. Additionally, MODSOFT-HP can reduce the infrastructure cost by 25% compared to the default Kubernetes Scheduler. The HFF and BKM-HP placement algorithms can reduce the infrastructure cost up to 50% but have a significant increase in response time. The fuzzy placement can also reduce the egress traffic up to 92% compared to default placement. In our work we use a homogeneous cloud environment and the price impact of egress traffic is not significant, but in a heterogeneous cloud environment the price can even be 15 times more expensive and the impact in the overall infrastructure cost will be significant.

Furthermore, additional work can be implemented in the service placement problem. We propose for future work a dynamic service placement that adapts to the application workloads without downtime and services can migrate to different nodes when the distribution of request changes significantly. Along with our work, this can be implemented using the Automated Service Placement algorithm we created and have zero down-time.

In real-life scenarios a homogeneous environment with VMs with the same CPU and RAM is not always feasible, and in most cases the VMs will vary in resource allocation due to the specific needs of each application. We propose for future work a fuzzy placement in a Heterogeneous Multi-Cluster Cloud environment and/or a Heterogeneous Fog - Edge environment where the egress traffic has a significant impact in the infrastructure cost and the reduction of response time is very important.

Finally, for future work, we propose using latency between services as the affinity metric. Latency metrics will have more impact on VMs placed in different networks, zones, and regions and because response times depend on microservices latency, it could be a more precise affinity metric for optimizing response time.

References

- [1] Adalberto R. Sampaio Jr. Julia Rubin Ivan Beschastnikh Nelson S. Rosa. “Improving microservice-based applications with runtime placement adaptation”. In: *Journal of Internet Services and Applications* (2019).
- [2] Alkiviadis Aznavouridis, Konstantinos Tsakos, and Euripides Petrakis. “Micro-Service Placement Policies for Cost Optimization in Kubernetes”. In: Mar. 2022, pp. 409–420. DOI: 10.1007/978-3-030-99587-4_35.
- [3] Kuo-Chan Huang and Bo-Jun Shen. “Service deployment strategies for efficient execution of composite SaaS applications on cloud platform”. In: *Journal of Systems and Software* 107 (2015), pp. 127–141. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2015.05.050>. URL: <https://www.sciencedirect.com/science/article/pii/S0164121215001156>.
- [4] Robert Cannon, Jitendra Dave, and James Bezdek. “Efficient Implementation of the Fuzzy C-Means Clustering Algorithms”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on PAMI-8* (Apr. 1986), pp. 248–255. DOI: 10.1109/TPAMI.1986.4767778.
- [5] 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>.
- [6] 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](https://doi.org/10.1016/0020-0190(94)00148-0). URL: <https://www.sciencedirect.com/science/article/pii/0020019094001480>.

- [7] Shreya Banerjee, Ankit Choudhary, Somnath Pal. “Empirical Evaluation of K-Means, Bisecting K- Means, Fuzzy C-Means and Genetic K-Means Clustering Algorithms”. In: *ResearchGate* (Dec. 2015).
- [8] 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>.
- [9] 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.
- [10] *Bin Packing Problem*. https://en.wikipedia.org/wiki/Bin_packing_problem. Date inspected: 28-08-2021.
- [11] Kumaraswamy S., Mydhili K. Nair. “Bin packing algorithms for virtual machine placement in cloud computing: a review”. In: *ResearchGate* (Feb. 2019).
- [12] *Use containers to Build, Share and Run your applications*. <https://www.docker.com/resources/what-container/>. Accessed: 2022-06-13.
- [13] *Kubernetes Documentation*. <https://kubernetes.io/docs/home/>. Accessed: 2022-06-13.
- [14] *kubernetes Components*. <https://kubernetes.io/docs/concepts/overview/components/>. Accessed: 2023-01-02.
- [15] *Horizontal Pod Autoscaler*. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. Accessed: 2023-01-02.
- [16] *Istio Documentation*. <https://istio.io/latest/about/service-mesh/>. Accessed: 2022-06-20.
- [17] *Prometheus Documentation*. <https://prometheus.io/docs/introduction/overview/>. Accessed: 2022-06-20.
- [18] *Kiali Architecture*. <https://kiali.io/docs/architecture/architecture/>. Accessed: 2022-06-20.
- [19] *Locust Documentation*. <https://docs.locust.io/en/stable/what-is-locust.html>. Accessed: 2022-06-20.
- [20] Beschastnikh Ivan and Rosa Nelson S. “Improving microservice-based applications with runtime placement adaptation.” In: *Journal of Internet Services and Applications* 10.4 (2019).

- [21] *MODSOFT GitHub Repository*. <https://github.com/ahollocou/modsoft>. Accessed: 2022-05-12.
- [22] *Online Boutique*. <https://github.com/GoogleCloudPlatform/microservices-demo>. Accessed: 2022-05-12.
- [23] Xenofon Koundourakis and Euripides Petrakis. “iXen: context-driven service oriented architecture for the internet of things in the cloud”. In: Mar. 2020, pp. 409–420.
- [24] *GCP Pricing*. <https://cloud.google.com/compute/all-pricing>. Accessed: 2022-06-25.