ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

In partial fulfillment of the requirements for the degree of

DIPLOMA IN ELECTRICAL & COMPUTER ENGINEERING

By

Skevakis Vasileios

FUZZY SERVICE PLACEMENT STRATEGIES IN KUBERNETES

Committee:
Professor Deligiannakis Antonios
Associate Professor Samoladas Vasileios
Professor Petrakis Euripides (Supervisor)

Chania, July 2022

Abstract

The continuous rise of microservices-based architectures in application development provided the need for tools that orchestrate containerized applications deployed in cloud infrastructures, such as Kubernetes. A Kubernetes Cluster hosts a finite number of Nodes (VMs), and the application's services are packed in Pods and placed in the Kubernetes Nodes. The default Kubernetes Service Placement is static and does not adapt to workload changes, so the default placement solutions are sub-optimal. Pods must be placed in their respective Nodes in a way that minimizes Node-to-Node (egress) traffic. That way, the infrastructure cost is optimized, and the response time of the requests is minimized since egress traffic is slower than ingress. Modern application workloads require some high-utilized services to run in more than one instance. In this work, the service placement problem is handled as a graph clustering one, aiming to optimize the application by both cost and response time. The graph clustering needs to be fuzzy, to allow a graph's node (Kubernetes Pod) to belong in more than one partition (Nodes). Two applications were deployed in the Kubernetes infrastructure of the Google Cloud Platform to support our fuzzy service placement solution in real use cases (an e-commerce eShop and an IoT architecture). The experimental results demonstrate that our fuzzy placement solution can significantly reduce both the total requested traffic between Nodes and the response time of the applications' requests compared to the placement using the default Kubernetes Scheduler's method. At the same time, our solution can maintain the monetary cost savings of previous related work on cost-optimization in Kubernetes.

Table of Contents

List of Figures

List of Tables

General Introduction

Problem Definition

Modern application development switches from monolithic architectures and on-premises infrastructures to microservices-based architectures on the cloud to reduce the cost of hardware maintenance and produce easier to maintain, update and migrate applications. The growth of microservices-based applications provides the need for technologies that orchestrate, schedule, and manage microservices, like the Google-designed Kubernetes. Kubernetes provides the means to control the deployment of a microservices-based application and allows the developer to schedule services between multiple hosts (VMs) while handling the communication between the various VMs.

Cloud providers charge cloud consumers based on the number of reserved machines, the resources (CPU, Memory, Disk) allocated for each machine, and the network traffic. Although deploying an application in Kubernetes offers scalability, flexibility, and high availability, it can easily raise application costs. Using more machines than application requirements or over-utilizing traffic between Nodes placed in different regions will significantly increase infrastructure costs.

Controlling the scheduling process of the application services and placing services with high communication rates in the same machines optimizes the deployment of the application in a cloud environment. This problem is referred to as the **service placement problem** (SP). By controlling at which VM each service is deployed, the total number of VMs needed for the application and the total egress traffic is reduced, optimizing the infrastructure's total monetary costs.

Scope of the Thesis

We attempt to solve the service placement problem using a fuzzy partitioning technique that aims to reduce the nodes needed to place our application microservices, the egress traffic between the utilized nodes, and mainly the response time of our application requests. Modeling the application by means of a graph representing services as nodes and traffic between services as node relations allows us to partition the application services using graph partitioning algorithms. These algorithms attempt to produce optimal partitions that, in our application graph, allow services that exchange high rates of traffic to be placed in the same partition. Placing these services in the same partition reduces the traffic exchanged between the infrastructure's VMs, called egress traffic.

Fuzzy placement solutions allow having more than one instance of some services placed in different VMs (Nodes). Having more than one instance of a service (replicas) deployed in the cluster reduces the workload of each service replica by load balancing the incoming requests between the replicas. This results in faster request processing, therefore, faster application response times.

Even though the fuzzy placement requires more Pods to host the application microservices, the bin packing solution we apply to the graph partitioning result places the services in VMs, based on the partitioning results, the traffic rates between services, and the available VM resources. This results in optimizing the VM resources usage and can reduce the number of VMs used to host the application, significantly reducing the infrastructure's monetary cost without compromising the application performance.

Thesis Structure

This Thesis is organized into an introduction Chapter and four main Chapters. Chapter 1 presents our Thesis's background work and is separated into 3 Sections. The first Section introduces the background work related to the service placement problem, while the second section introduces related work on fuzzy clustering techniques and fuzzy partitioning on graphs. The third Section of this Chapter introduces the related work on the algorithms used in our proposed solution. In Chapter 2 of this Thesis, the Kubernetes environment and the features of Kubernetes that we use in this work are introduced, along with the tools used to produce the application graph, implement and evaluate our experiments. Chapter 3 presents the architecture of our cluster, which was deployed in the Google Cloud Platform. This Chapter also presents the implementation of our work, the affinity metrics used to evaluate microservice communication, and the proposed placement strategy. Furthermore,

this Chapter presents the tool implemented for automating the deployment process after the algorithms produce the suggested service placement. The benchmarking applications used in our experiments are also presented in this Chapter. The final Chapter of this Thesis presents the experiments implemented to test the efficiency of the proposed fuzzy placement strategy, comparing it with the default placement decision of the Kubernetes Scheduler and a flat placement strategy proposed by related work for cost-optimization.

Chapter 1

Background and Related Work

The first section of this chapter will provide the background and review the related work on the service placement problem. The second section of this chapter will present the definition of fuzzy clustering and introduce related work and algorithms that were modified and used in this Thesis to provide our solution to the service placement problem.

1.1 Related Work on Service Placement

Huang and Shen [1] proposed a service deployment method that reduces application response times. Their proposed solution did not aim to reduce costs in the application's infrastructure. They modeled the application by means of a graph representing communication costs and services parallelism and applied a minimum k-cut to the modeled graph. They applied experiments for four service deployment methods on amazon's cloud infrastructure and ran all the experiments on only one VM without using Kubernetes.

In recent work, Aznavouridis, Tsakos, and Petrakis [2] attempted to solve the Service Placement problem from the scope of reducing the total monetary cost of the infrastructure. They modeled the application using a weighted and directed graph with two different affinity metrics that focus on the number of requests from one service to another and the bytes exchanged between the services. They implemented the experiments in a Kubernetes cluster with a predefined number of available nodes. They examined several different flat graph partitioning algorithms that produced partitions of the graph with as less communication between partitions as possible and passed the partitions to the placement algorithm, which placed the services in the available nodes. As mentioned before, the graph partitioning algorithms used by *"Micro-Service Placement Policies for Cost Optimization*

in Kubernetes" [2] are flat, and their final placement restricts each service having only one instance.

1.2 Related Work on Fuzzy Clustering

In general, clustering refers to methods of subdividing a set X of items into c subsets called clusters which are pairwise disjoint, nonempty, and their union reproduces X. For optimal clustering, the objective is to have items with the most similarity placed in the same clusters and items with less similarity in separate clusters. A cluster analysis can be *flat* (hard) or *fuzzy* (soft). Flat clustering denotes that the c subsets of set X are mutually exclusive, so a point can either belong or not belong to a subset. On the contrary, fuzzy clustering, as mentioned below, introduces a membership grade for each point to belong to each subset.

In 1965, Zadeh [3] stated that *"the classes of objects encountered in the real physical world do not have precisely defined criteria of membership."*. To represent the above on a data set, Zadeh introduced a membership function that provides each point with a grade of membership to each cluster. The membership function denotes the similarity each point shares with each cluster, and the membership's value is between 0 and 1.

Cannon, Dave, and Bezdek, in 1986 [4], extended the work of Dunn [5] on the Fuzzy c-Means (FCM) algorithm, which is the most-known fuzzy clustering algorithm and can be applied if the objects of interest are represented as points in a multi-dimensional space. The FCM algorithm is very similar to the k-means algorithm, and by aiming to minimize the objective function, it outputs a matrix w_{ij}, which indicates the degree to which element, x_i, belongs to cluster c_j.

Fuzzy Graph Partitioning

The graph partitioning problem is a modified clustering problem where we divide a graph's nodes into smaller groups of nodes, known as graph partitions. By partitioning a graph into smaller partitions, we simplify the graph's analysis process. There are two types of graph partitioning, flat (or hard) partitioning, and fuzzy (or soft) partitioning. Producing flat partitions of a graph means that each node will belong to exactly one partition. On the contrary, fuzzy partitioning encodes uncertainties in node-to-partition assignments because each node is assigned a grade of membership to each fuzzy partition.

Yan and Hsiao [6] expanded the FCM algorithm [4] to solve the graph bisection problem.

The graph bisection problem is a partitioning problem where a graph is divided into two partitions, such that the sum of weights of edges joining these two partitions is optimized. Their algorithm works via iterative optimization of the objective function $J(U, v)$ and produces a feasible fuzzy graph partition of the vertex set V. The first step of their algorithm is to define the clustering distance matrix d_{ij}, which is defined as the weight of edge w_{ij} if a direct edge between vertex v_i and vertex v_j exists, or the shortest path between the two vertices if such an edge does not exist. A fuzzy matrix U is then established by initializing two arbitrary partitions of the vertices. 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. Two partition centers are calculated in each iteration by minimizing the objective function. Then, the Fuzzy U matrix is updated by determining a degree of membership depending on the distance between each vertex and the partition center. The iteration finishes after the Fuzzy U matrix converges, and the matrix produces the degree of memberships for each vertex to each partition. Finally, after sorting the groups of vertices, all the vertices will be separated into two even subsets with a minimum cut for graph bisection. The *"A fuzzy clustering algorithm for graph bisection"* [6] works when graph G is an undirected connected edge-weighted graph but does not produce optimal results when the graph is directed, such as our application's graph.

1.3 Related Algorithms

Modularity-based Graph Clustering

The *"Modularity-based Sparse Soft Graph Clustering"* [7] was proposed in 2019 by Hollocou, Bonald, and Lelarge. It is a relaxation of the modularity optimization problem introduced by Newman and Girvan [8], which is widely used in community detection in large networks. Modularity is a function that evaluates the quality of a graph's partition. It is defined as the difference between the probability of two nodes of graph G being on the same partition and the probability of two random nodes of the graph placed on the same partition. The modularity function proposed by Newman and Girvan [8] can be characterized as hard modularity, and the relaxation of the modularity maximization problem proposed by the MODSOFT [7] algorithm produces fuzzy partitions by introducing a membership matrix $\mathbf{p} \in \mathbb{R}^{n \times n}$.

The proposed algorithm (Listing1.1), unlike other proposed relaxation algorithms, doesn't need prior knowledge of the number of available clusters. It is fast because the membership information of any given node depends only on its direct neighbors and doesn't require processing the whole graph. Most nodes produce membership probabilities equal to zero,

and for efficiency, the algorithm stores only the non-zero values. Finally, by updating the t parameter, the algorithm can produce *"softer"* or *"harder"* partitions.

Listing 1.1. MODSOFT Algorithm [7]

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

During the initialization part of the algorithm, the **membership matrix** $\boldsymbol{p}$ ($n \times n$ matrix) is initialized as a one's matrix ($\boldsymbol{I}$) and the **weighted average vector** $\overline{\boldsymbol{p}}$ is initialized as the weighted degree[1] of each node. The membership matrix shows the probability of each node (matrix row) to belong to the same partition as another node (matrix column), which is a value from 0 to 1. In other words, each row of the matrix, represents the probability distribution for a node to belong to a partition, which sums to 1.

One epoch of the MODSOFT algorithm can be described by the following steps. For each node, a gradient descent step (1) is performed, as an **update rule** for the membership of each node. The gradient descent step of the algorithm is local, as it only uses the neighbors of each node to update the node's membership and aims to maximize the modularity. After the gradient descent step, a projection is performed (2), with a complexity of $O(L_i \log L_i)$ with $L_i = |\,\mathrm{supp}_i(\boldsymbol{p})|$, where $\mathrm{supp}_i(\boldsymbol{p})$ is the union of the support vectors of p_j. Finally, the weighted average vector $\overline{p}$ is updated (3).

Each epoch is an iteration of the algorithm and at the end of each iteration, the modularity is calculated, evaluating the optimality of the partitions. The algorithm is repeated until the increase of modularity falls below a given threshold. The final result of the MODSOFT algorithm is the membership matrix, each cell of the membership matrix representing the probability of the row-node to be in the same partition as the column-row.

[1] Sum of the weights of the node's edges

Heuristic Packing

The Heuristic Packing algorithm, proposed by Aznavouridis, Tsakos, and Petrakis [2], is a placement finding algorithm that can be considered a multi-dimensional bin packing problem. The algorithm takes as an input the generated partitions and attempts to place each partition into the available host machines. The algorithm uses two greedy heuristics to find the optimal placement solution. 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, and 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.

Listing 1.2. Heuristic Packing Algorithm [2]

Input: Partition $\mathrm{P} = (S_1, S_2 \ldots S_N)$ of application, vectors of available resources on each machine $\mathrm{V} = (V_1, V_2 \ldots V_M)$
Output: a placement solution X
Calculate vectors of resource demands of each part a $(D'_1, D'_2 \ldots D'_N)$
$\mathrm{X} \leftarrow [x_{ij} = 0]$ fr every part and host machine
for $\mathrm{i} \leftarrow 1; \mathrm{i} \leq N'; \mathrm{i}++$ **do**
  $\mathrm{tf} \leftarrow 0, \mathrm{ml} \leftarrow 0, \mathrm{y} \leftarrow 0$
  for $\mathrm{j} \leftarrow 1; \mathrm{j} \leq \mathrm{M}; \mathrm{j}++$ **do**
    if part S_i can be packed into machine m_j **then**
      $\mathrm{tf} \leftarrow \sum t_{uv}$
      $\mathrm{ml} \leftarrow \sum_{k=1}^{R} \frac{d_i'^k}{v_j^k}$
      if $tf_j \geq \mathrm{tf}$ **then**
        $\mathrm{tf} \leftarrow tf_j, \mathrm{ml} \leftarrow ml_j, \mathrm{y} \leftarrow \mathrm{j}$
      else if $tf_j = \mathrm{tf}$ and $ml_j > \mathrm{ml}$ **then**
        $\mathrm{tf} \leftarrow tf_j, \mathrm{ml} \leftarrow ml_j, \mathrm{y} \leftarrow \mathrm{j}$
      end if
    end if
  end for
  if $y == 0$ **then**
    Return null
  else
    $V_y \leftarrow V_y - D'_i$
    $x_{iy} \leftarrow 1$
  end if
end for
Return X

Chapter 2

Infrastructure and Tools

The second chapter of this Thesis presents the infrastructure and the tools used for our proposed solution, as well as the tool used to test and evaluate the efficiency of our strategy.

2.1 Kubernetes

The continuous rise in requirements of applications deployed in the cloud resulted in the development of an architectural style that optimizes cloud resource usage. The idea of this architecture is that application logic can run in separate, independent, isolated containers that communicate via APIs[1]. The **microservice architecture** enables the rapid, frequent, and reliable delivery of large, complex applications, with each microservice implementing a specific part of the application logic [9]. Each microservice runs in a container [10], a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.

The rise of microservices-based architectures issued the need for tools that orchestrate microservices, such as **Kubernetes** [11], which was developed by Google. Kubernetes (from the Greek word Kyvernítis - Pilot) is an open-source system for automating containerized applications' deployment, scaling, and management. Kubernetes provides a framework to run distributed systems resiliently by offering **service discovery, load balancing, storage orchestration, automatic bin packing** (automated service placement), **self-healing** of containers, and enhanced **security**.

[1] Application Programming Interfaces

2.1.1 Infrastructure and Components

Upon the initialization of the Kubernetes cluster, a Node Pool is configured, containing the CPU, RAM, storage space, and OS requirements that each initialized virtual machine (Node) must meet according to each application's requirements. The number of Nodes that the Node Pool must spawn is then defined, and the machines boot and connect to the cluster. Each Node runs a container-optimized OS and hosts several containers called Pods. Each Pod can host one or more microservices.

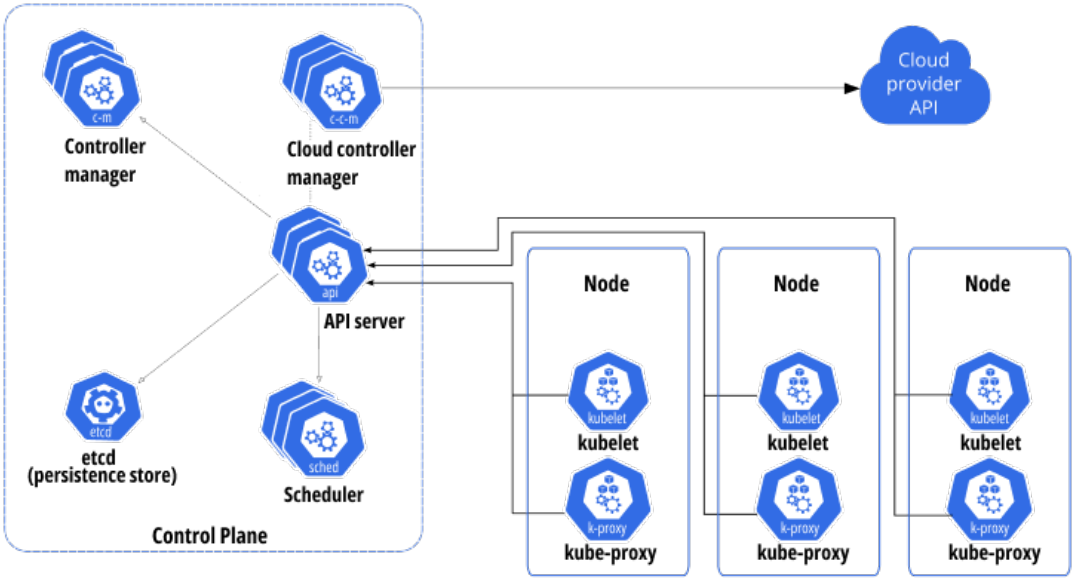


Figure 2.1. Kubernetes Architecture and Components [11]

A Kubernetes cluster (Figure 2.1) includes at least one *Worker Node* and a *Master Node* which host all Kubernetes components. Each Node (Worker Node) runs a **kubelet** component, which is an agent that makes sure that containers are running in a "healthy" Pod and manages the containers that Kubernetes create. The **kube-proxy** component is responsible for maintaining network rules on the Node, which allow the Pods to communicate with other services inside and outside of the cluster.

The Master Node, also referenced as ***Control Plane***, has five main components. The **kube-apiserver** acts as the front end of the control plane and exposes the Kubernetes API. Its design allows horizontal scaling by deploying more instances of itself and balancing traffic between those instances. **Etcd** is a consistent key-value storage used by Kubernetes to store all cluster data. The **kube-controller-manager** runs all controller processes, and the **cloud-controller-manager** is responsible for linking the cluster to the cloud provider's API and exposes the components that interact with the cloud platform. The latter component only runs if the cluster is deployed on the cloud and not in a local environment.

2.1.2 Scheduling Process

One of the most valuable components hosted in the control plane is the **kube-scheduler**, which monitors the cluster and selects which Node will host any newly created Pod. The scheduling process considers available node resources, hardware constraints (defined in Pod's deployment), data locality, and affinity, anti-affinity specifications. During the ***scheduling cycle***, the Kubernetes scheduler picks the first unscheduled Pod from his priority queue, finds feasible[2] nodes, and then runs a set of functions to evaluate each Node. The scheduler assigns the Pod to the highest scoring feasible node and notifies kube-apiserver about his decision in a process called ***binding***

The scheduler's placement decision, as mentioned above, is based on four key factors. The first factor is **dedicated hardware**, where a pod may have specified in the deployment file that it has a specific hardware requirement (e.g., Ensure that a Pod ends up on a machine with an SSD). The second factor is **data locality**, which ensures faster reads and better write throughput by placing a Pod in a specific availability zone. The third is **co-locating** Pods that communicate a lot into the same availability zone, and the fourth is placing the Pods in multiple Nodes to ensure **high availability** and **fault tolerance**.

2.1.3 Services and Network Traffic

Kubernetes uses IP addresses to enable communication between services and components. A Pod is assigned an IP address upon creation, but this IP address is temporary and changes every time the Pod restarts (due to a crash or update). That's why Kubernetes introduced a resource called **Kubernetes Services**. Kubernetes Services are **abstractions** that allow the Pod to use the network to communicate (either internal cluster communication or external network communication). Each Pod bounds with its respective Kubernetes Service, and the Service is responsible for forwarding any traffic to the Pod. The Service discovers the Pod's IP address upon creation or change and exposes a permanent address (user-defined in the Kubernetes Service YAML configuration) and a port so that other services can communicate. A Kubernetes Service can be a *ClusterIP*, *NodePort*, *LoadBalancer* or *External Name* type. The LoadBalancer type exposes the Service to the external network, using the cloud provider's load balancer routes. The NodePort type exposes the Service on each Node's IP at a predefined port. Allowing this port through the cluster's firewall makes the Service accessible through the external network. The ClusterIP service is the most used one and exposes the Service on a cluster-internal IP, so it's accessible to the rest of the cluster's services.

[2]Nodes that meet the scheduling requirements for a Pod

The kube-proxy component has access to all the Kubernetes Services communication information (addresses) and is accessible as long as there is at least one healthy Node in the cluster. Kubernetes Services act as a load balancer for oncoming traffic if multiple Pod instances are attached to the Service. According to the Kubernetes Documentation [11], if there is a replica set of pods, the Service will choose the most optimal Pod to forward each request. In our Thesis, we assume that the Service will decide to forward the requests to a destination Pod deployed in **the same Node** as the request source Pod if such exists.

Figure 2.2 shows an example application deployed in Nodes A and B. The blue box inside the Nodes represents the kube-proxy component which is repeated in all Nodes and contains the configurations of the Kubernetes Services (Dijkstra Service, A* Service, Frontend Service). The Kubernetes Services are network configurations and not instances, and they offer an address in order for a request to be forwarded to their respective Pod (microservice). We use the described example to illustrate the assumption made in the previous paragraph, noticing that the "Frontend" Pod forwards the request to the "Dijkstra" Service, which then forwards the request to the "Dijkstra" Pod, located in the same Node as "Frontend" and not in the "Dijkstra" Pod in Node B. Gray lines in the figure represent ingress traffic, while blue lines represent egress traffic. In our work, if one of the "destination Pod" instances is located in the same Node as the "source Pod", the traffic between these Pods is considered as ***ingress traffic*** (in-Node traffic).

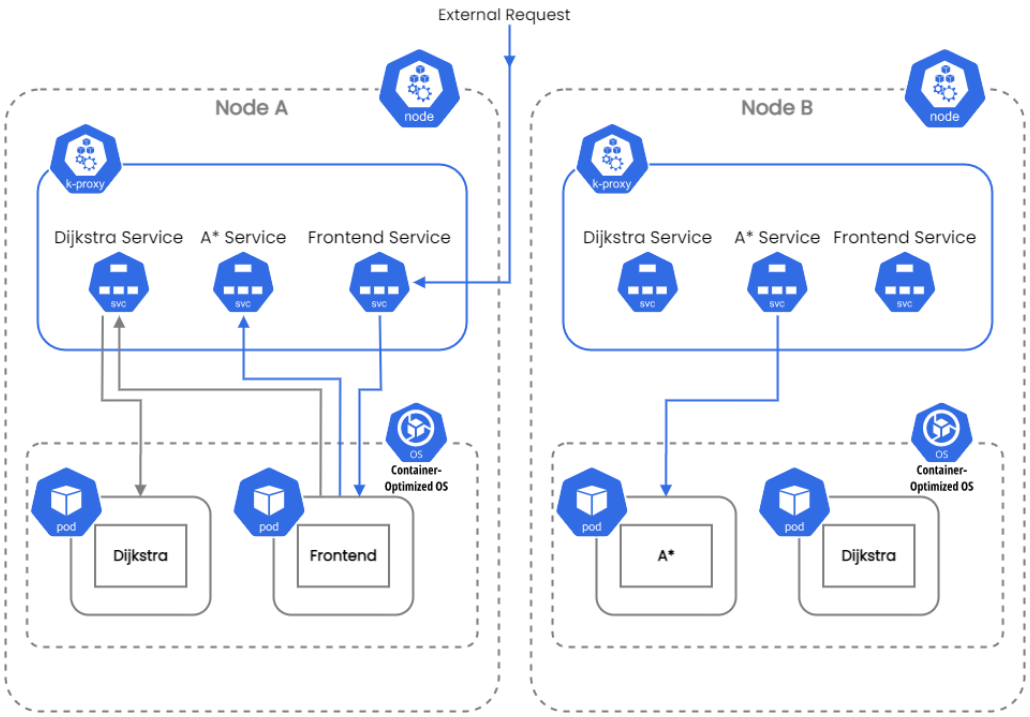
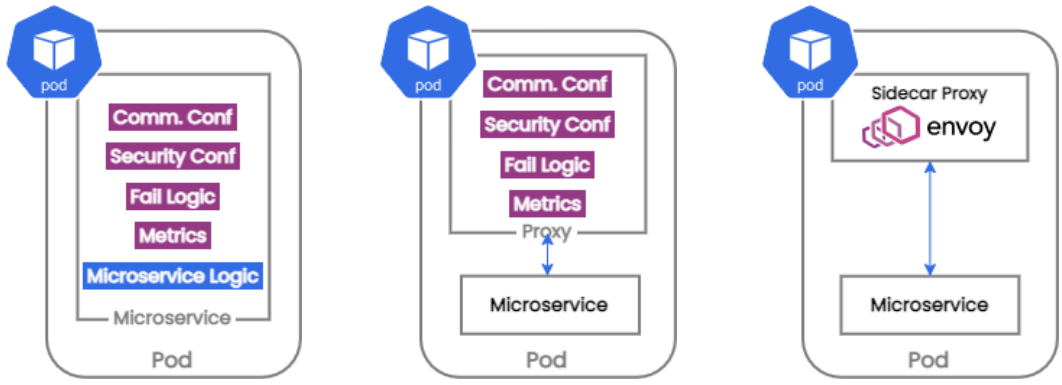


Figure 2.2. Traffic between Pods

2.2 Service Mesh and Istio

Microservices in a Kubernetes Architecture must be configured to have secure and robust communication with each other. The developers of a microservice must implement additional logic to include communication, security configuration, and fail logic. To be able to monitor the Pod, developers must also implement another logic to export metrics such as communication rate or failed requests from the container. All the above consume time and resources, so **Service Mesh** was introduced. A service mesh is a dedicated infrastructure layer for applications that adds capabilities like observability, traffic management, and security of communication [12]. The Control Plane of the Service Mesh injects a Sidecar Proxy service as a third-party application, which handles all the logic mentioned above. If the Figures below we see that without a Service mesh, the developer must implement the application logic (in blue) as well as the additional logic (in purple). Figures (b) and (c) present how the additional logic has been packed in a sidecar proxy, and automatically configured and deployed with the Envoy sidecar proxy.



(a) without Service Mesh (b) with Service Mesh (c) with Istio Service Mesh

Figure 2.3. Pod and Microservice Architecture

Istio is an open-source implementation of a Service Mesh, and Istiod is the Control Plane of Istio. Istiod automatically detects new services and endpoints in the cluster and deploys an Envoy Sidecar Proxy service in each newly created Pod. Istiod also manages all the certificates and configures secure TLS communication between the services. Finally, istiod gathers telemetry data and exports metrics from each Pod, which can be then acquired by a monitoring server, like Prometheus.

2.3 Metrics Tools

2.3.1 Prometheus

Prometheus is an open-source system monitoring and alerting toolkit that constantly monitors its *targets*, identifies problems before they even occur, and provides alerts when a crash does happen. 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. [13]. Prometheus targets can be an entire server, an application, a service, etc. For Prometheus to monitor a target, the target must be providing the correct format for its metrics and exposing them through the Prometheus endpoints. Many services don't have native Prometheus support, so an extra component, called *Exporter*, is needed to read the metrics from the service, transform them into a compatible format and expose the endpoint so Prometheus can pull these metrics.

A Kubernetes Node, which is a Linux Virtual Machine, does not provide native support for Prometheus, so in order to monitor the VM units, we need to add a **Node Exporter**. A Node Exporter is an official Prometheus exporter for capturing all the Linux system-related metrics, which collects all the hardware and Operating System level metrics that are exposed by the kernel. [14]

Prometheus pulls metrics from its targets and stores them in a human-readable, text-based format. The metrics can then be requested by another tool or a user by sending a PromQL[3] query to the Prometheus API Server. The metrics can be visualized either by the Prometheus server or by more advanced visualization tools like Grafana.

2.3.2 Kiali

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 [15]. 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.

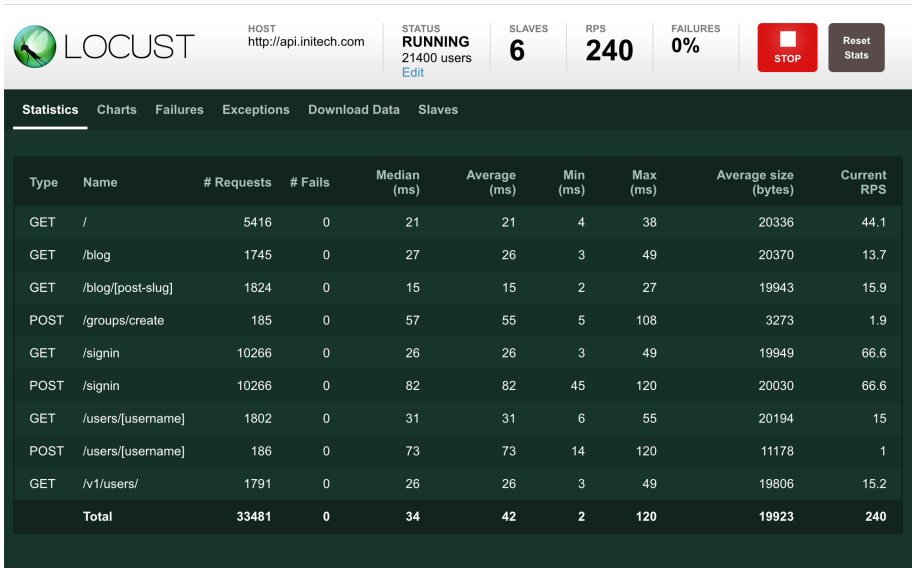
[3]Prometheus Query Language

The Kiali Graph is a weighted, directed graph that shows the communication between micro-services. The graph's nodes represent the deployments (microservices - Pods) and their respective service components (Kubernetes Service) and the graph's edges present the weighted traffic between the micro-services. The weight of this traffic in the Kiali Graph is Requests per Second (RPS) for the HTTP and gRPC communication protocols and Bytes per Second (BPS) for the TCP traffic.

2.4 Load Testing Tool

Locust

Locust is an easy-to-use, scriptable, and scalable performance testing tool [16]. Locust tests behavior is implemented in Python, and it can be deployed as a microservice, applying the predefined behavior, even without the UI. It offers distributed event-based requests application so that a single process can handle thousands of concurrent users. It is well documented, and test scenarios can be easily written in Python using the required libraries. It offers a user-friendly UI, exports results in multiple formats, and provides various evaluation metrics, which we will present in Chapter 4. Locust is deployed as a microservice in both our applications (load generator) and it performs requests on application endpoints with a predefined distribution. Through the Locust UI (Figure 2.4) we can get the analytic results for each of the application endpoints - which is a different application request - and examine the number of requests performed to this endpoint, the number of failed requests and various response time measurements.



Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
GET	/	5416	0	21	21	4	38	20336	44.1
GET	/blog	1745	0	27	26	3	49	20370	13.7
GET	/blog/[post-slug]	1824	0	15	15	2	27	19943	15.9
POST	/groups/create	185	0	57	55	5	108	3273	1.9
GET	/signin	10266	0	26	26	3	49	19949	66.6
POST	/signin	10266	0	82	82	45	120	20030	66.6
GET	/users/[username]	1802	0	31	31	6	55	20194	15
POST	/users/[username]	186	0	73	73	14	120	11178	1
GET	/v1/users/	1791	0	26	26	3	49	19806	15.2
	Total	**33481**	**0**	**34**	**42**	**2**	**120**	**19923**	**240**

Figure 2.4. Locust User Interface [16]

Chapter 3

Cluster Architecture and Implementation

This Chapter introduces the architecture of the Cluster implemented for the requirements of this Thesis, as well as the Affinity Metrics we defined to evaluate communication between the microservices and used as weights on the Kiali generated services graph. The applications used to apply and test our placement strategies are also introduced and analyzed in this Chapter.

3.1 Cluster's Architecture

In the present work, we have configured a Kubernetes Cluster (Figure 3.1), which we deployed in the Google Cloud Platform (GCP). The Google Kubernetes Engine (GKE) is responsible for controlling the Cluster, and the Google Compute Engine (GCE) contains the Virtual Machines (VMs) that Kubernetes uses as the Cluster's Nodes. The Cluster can host a finite number of Nodes, sharing hardware, software, and network requirements with the user-defined Node Pool. Horizontal and Vertical autoscaling (Kubernetes extensions) and all Load Balancing features provided through the GKE are disabled.

The previously described Istio Control Plane (istiod) is randomly deployed in one Node and configured to communicate with the Mesh created by the Envoy Sidecar Proxies to log all network traffic on the Cluster. A Prometheus Node exporter is deployed in every newly created Node and is responsible for exporting Node metrics so that the Prometheus Server, which is deployed randomly amongst the available Nodes, can pull and store these metrics in real-time. Finally, the Kiali service is randomly deployed amongst the available

Nodes based on the scheduler's decision and configured to pull Istio's logged network traffic through the Prometheus Server.

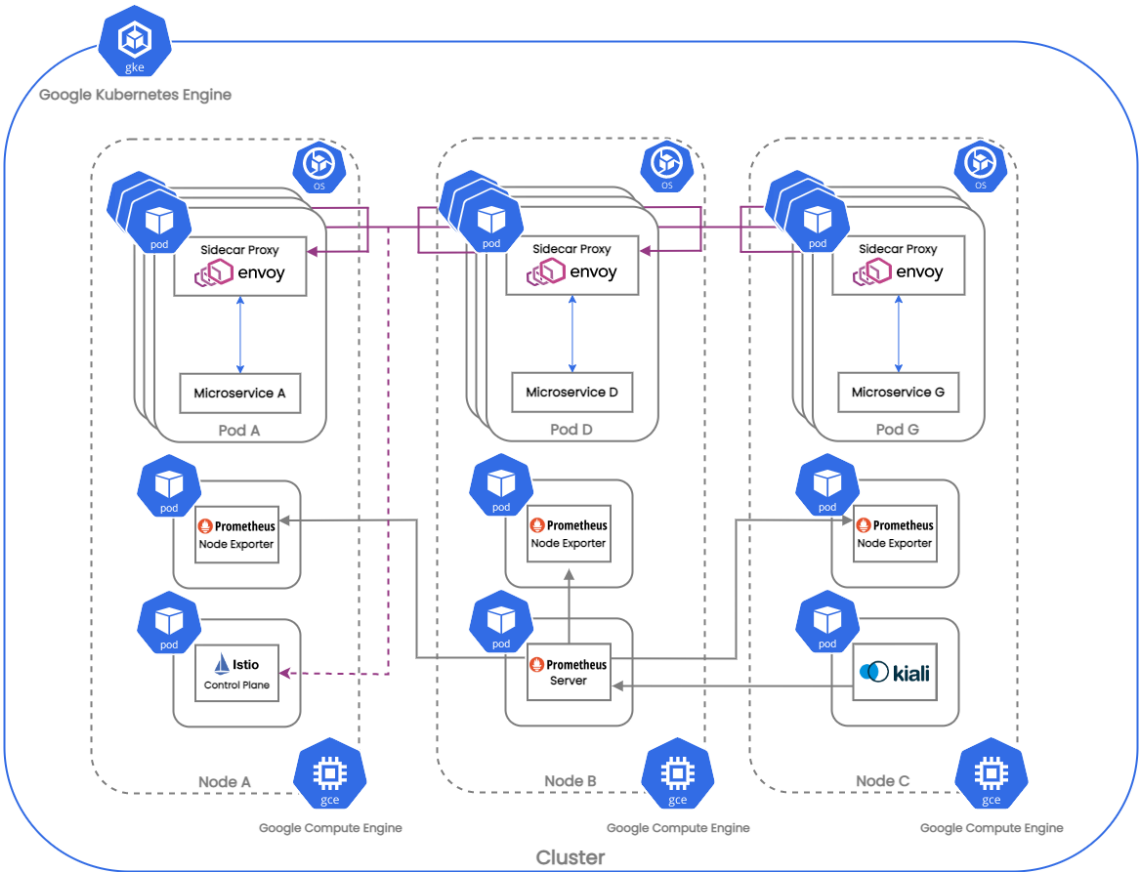


Figure 3.1. Cluster deployed in Google Cloud Platform

Each Kubernetes Node (Figure 3.2) can host a finite number of Pods, depending on the available Node resources. All the traffic within the Pods deployed in the Node is forwarded through the injected Envoy Sidecar Proxies. The Istio Control Plane (istiod) monitors the Service Mesh, which is the communication between the Envoy Sidecar Proxies. Every Pod contains exactly one microservice, described as a Kubernetes Deployment (configured in a YAML file). Each microservice has a respective associate Kubernetes Service (2.1.3) responsible for all microservice communication. Suppose a microservice is deployed as a Replica Set, which means that there will be more than one instance of the microservice in separate Pods. In that case, the associate Kubernetes Service is responsible for forwarding the requests between the instances, as mentioned in Subsection 2.1.3.

Services that communicate with the external network are defined as NodePort type, and any application services that were Load Balancer type are converted to NodePort because we want to avoid external Load Balancers for the integrity of experimental results and to avoid extra infrastructure costs. Firewall rules are defined in the GKE (Cluster manager) so that external traffic is allowed through the defined NodePort Services external ports.

As mentioned above, and as visualized in Figure 2.3, for each created Pod, an Envoy Sidecar Proxy is injected into that Pod. Envoy proxies are responsible for receiving incoming and outgoing requests to the Pod's microservice. If the request is incoming, the proxy forwards it to the microservice through the respective Kubernetes Service. If the request is outgoing, it is sent to the targeted Pod so it can be processed from that Pod's Envoy proxy.

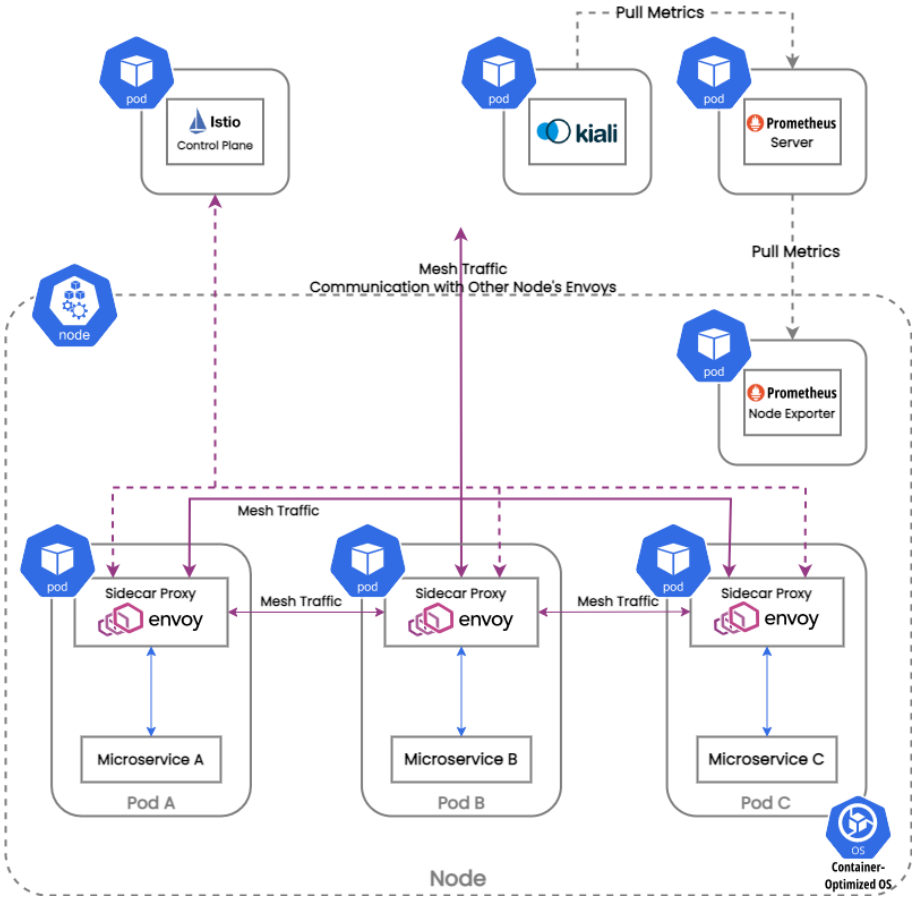


Figure 3.2. Node's Architecture

3.2 Affinity Metrics

Our benchmarking applications are modeled using a graph representing services as nodes and service communication as node relations. In this section, we will identify the similarity measurements (affinity metrics) used to evaluate communication between services and will be added as weights to the application graph.

Requests per Second (RPS)

The first Affinity Metric we suggest in our work is the Requests per Second (RPS) metric which measures the requests forwarded from one service to another per second. The RPS

metric is provided by the Kiali Graph, which is acquired from the Kiali API in JSON format. Kiali calculates the RPS from the formula below for traffic that uses the HTTP or the gRPC protocol and we can state that the RPS Affinity is the *"mean value of requests made from service a to service b in a specific time-frame"*. The RPS metric is not accurate for the TCP connections, as it is substituted by the Bytes per Second measurement (BPS).

$$\text{RPS}_{S_i \to S_j} = \frac{\text{Sum of Requests from } S_i \text{ to } S_j \text{ in } x \text{ seconds}}{x \text{ seconds}} \tag{3.1}$$

Where,

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

Weighted Bidirectional Affinity (WBA)

Introduced in 2019 in the "Improving microservice-based applications with runtime placement adaptation." [17] and adapted in 2022 by Aznavouridis, Tsakos, and Petrakis [2], the Weighted Bidirectional Affinity is a performance metric that exploits the size of the exchanged messages (in bytes) between two microservices, as well as the total number of these messages. Bellow is the formula proposed to calculate the WBA Affinity.

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

- $A_{a,b}$ is the affinity metric between service a and service b
- m is the total number of messages exchanged
- ma, 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 weight factor w is assigned a 0.5 value because both affinity variables (size and message count) are equally significant for our purpose.

3.3 Graph Construction

The Kiali Graph acquired for our benchmarking applications is presented below. The deployments (Pods) are represented using circles and their respective Kubernetes Services are represented using triangles. HTTP and gRPC traffic between Pods is represented by green lines (and measured in RPS), while TCP traffic is represented by blue lines (and measured in BPS).

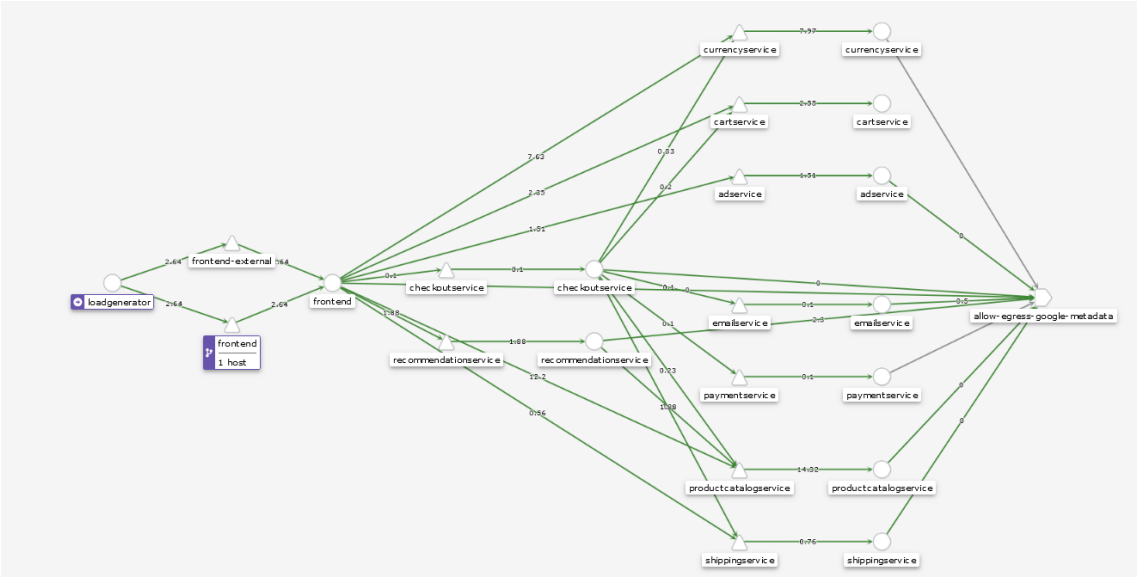


Figure 3.3. Kiali Graph for e-Shop

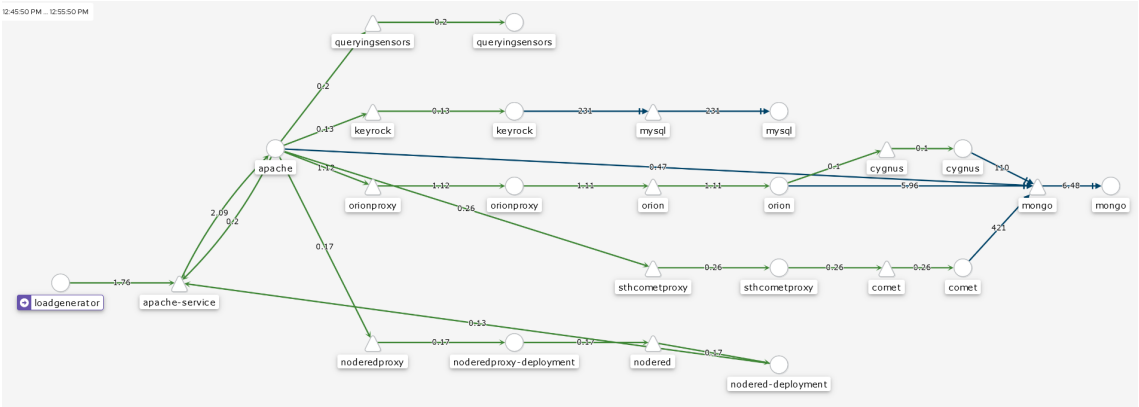


Figure 3.4. Kiali Graph for iXen

Our application graph is constructed based on information provided through the Kiali API, and metrics extracted from the Prometheus server. We extract the application microservices (Pods) through Kiali and create a node on our graph for each microservice. The edges of the graph are created based on the communication edges exported through Kiali and the edges' weights are defined based on the selected Affinity Metric.

3.4 Benchmark Algorithms

3.4.1 Fuzzy Partitioning Algorithm

The "Modularity-based Sparse Soft Graph Clustering" provides a supplementary PDF containing the pseudocode implementation of their proposed algorithm, which can also be found implemented in their GitHub repository [18]. The algorithm implementation consists of 3 subroutines (functions), one for **initializing and updating the membership matrix** p via the gradient descent step 1.3, one for efficiently **projecting the membership probabilities**, and one for **calculating the modularity** of the partitions. We adapted and modified the proposed functions for our work, and our implementation is described in Listing 3.1. During the initialization process, we calculate the total weight of the graph by summing the weights of each of the graph's edges. Then, we calculate the weighted degree of each of the graph's nodes by summing the weights of the edges connected to the node. During the partitioning step, we run the MODSOFT membership update function (gradient descent step) to update the membership matrix p. The membership matrix p contains the services' probability of belonging in the same partition with each other service. Each row of this matrix sums to 1 and represents the probability distribution. The update_-membership step includes both the membership update function and the projection step and it returns the updated membership matrix p.

After the membership matrix p is updated we evaluate the modularity of the proposed partitions. Modularity, as described in Chapter 1, is a quality function that evaluates the partitions. It can easily be calculated using the provided modularity function from the MODSOFT repository. Modularity is calculated after the partitioning step and the partitioning step is repeated until the increase of modularity falls below a threshold, which we have set at 0.01. The final step is to iterate the final membership matrix p and calculate our services partitions. As mentioned before, the membership matrix rows and columns represent the application services, and the value of each cell is a number between 0 and 1 which represents the probability of the row-service being in the same partition as the column-service. We create a partition for each row-service and by using a predefined threshold (which we set at 0.1) we place each column-service whose value is over our threshold, to the partition of the row-service. After the partitions are generated we sort them based on the number of services of each partition and return them.

The MODSOFT Partitioning Algorithm produces fuzzy application partitions P, and by modifying the fuzziness parameter (t), we can control the fuzziness of the produced partitions. A higher t parameter results in less-fuzzy partitions, and by issuing a high

enough value, the partitions produced will be flat and not fuzzy.

Listing 3.1. Fuzzy Partitioning Algorithm

Input: Services Graph (G), Graph Nodes (N), Application Services (S), Threshold (T), Modularity Threshold (MT), Fuzzyness Parameter (t)
Output: Application Partitions (P)

 # Initialization
Calculate the total weight of graph G, total_weight
Calculate the weighted degree of each node N, $\text{degree}_{\text{node}}$
$i \leftarrow 0$, $\text{modularity}_i \leftarrow 0$
for node **in** G **do**:
    $p_{\text{node}} \leftarrow 1$

 # Partitioning
do:
    $i \leftarrow i + 1$
    # Update Membership matrix using MODSOFT [1.3]
    p $\leftarrow$ update_membership(p, t)
    $\text{modularity}_i \leftarrow$ modularity_func(p)
while: modularity_i – $\text{modularity}_{i-1} \leq$ MT

$k \leftarrow 1$
for service i **in** p **do**:
    $P_k \leftarrow \{S_i\}$
    for service j **in** p_i **do**:
        if $p_{ij} >$ T **then**:
            $P_k \leftarrow P_k \bigcup \{S_j\}$
Sort P by partitions with most services , **return** P

3.4.2 Placement Algorithm

Our fuzzy partitioning algorithm effectively partitions the application into fuzzy partitions according to the affinity traffic rates (RPS or WBA) between the microservices. However, the fuzzy partitions produced may not "fit" in the available Nodes, and a post-processing algorithm may optimize their placement in their respective Nodes. The Heuristic Packing

(HP) described in Subsection 1.3, is used as a post-processing algorithm to ensure that the services are optimally placed in the available cluster Nodes by means of traffic rates, CPU utilization, and RAM utilization. The HP algorithm requires the fuzzy partitions, the Node and Pod requested resources, the list of services, and the list of affinities as inputs. It produces a placement solution ensuring that the least resources will be allocated, providing a cost-optimized solution.

As described, our fuzzy placement strategy is separated into three parts. The first part is constructing an application graph using the services list and the services affinities. After the application graph is constructed, we apply the Fuzzy Partitioning Algorithm in the application graph, which efficiently produces the application's fuzzy partitions. To ensure that our placement solution will fit in the available Nodes and that the solution will be optimal by means of costs and allocated resources, we use the Heuristic Packing algorithm, which produces our fuzzy placement.

3.5 Deployment Files Generation

As mentioned before, each microservice is deployed in a Pod using a Kubernetes Deployment. Kubernetes Deployments are configured using a YAML file format, in which the Node Affinity and Node Anti-affinity specifications are configured (2.1.2). During the scheduling cycle, the Scheduler checks the Node Affinity and Anti-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. In the example below, the deployment has specified that the Pod should be deployed in a Node with a hostname which matches "**gke-eshop-cluster-default-pool-066d55f8-z5mr**" and should **not** be deployed in Nodes matching the other three values (**NotIn**), using the **matchExpressions** preference in the **nodeAffinity** specification.

Listing 3.2. Example - Affinity in Deployment YAML

```
affinity :
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - preference:
          matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
            - gke-eshop-cluster-default-pool-066d55f8-z5mr
        weight: 1
```

```
requiredDuringSchedulingIgnoredDuringExecution:
  nodeSelectorTerms:
  - matchExpressions:
    - key: kubernetes.io/hostname
      operator: NotIn
      values:
      - gke-eshop-cluster-default-pool-066d55f8-gr5w
  - matchExpressions:
    - key: kubernetes.io/hostname
      operator: NotIn
      values:
      - gke-eshop-cluster-default-pool-066d55f8-35l1
  - matchExpressions:
    - key: kubernetes.io/hostname
      operator: NotIn
      values:
      - gke-eshop-cluster-default-pool-066d55f8-t4vr
```

By exploiting the above features of the Kubernetes Scheduler, we created a tool that takes as input the hosts list and the placement result in JSON Object format. The tool then parses the placement result while creating a separate deployment YAML file for every microservice, specifying which hostname (or hostnames in case the microservice spawns replicas) the Node must have so it can be a feasible Node to host the service's Pod.

The described tool can be adapted to use Node Labels instead of Node hostnames, as each time the Node Pool creates a Node, the Node is assigned a random hostname. By giving the labels to the Nodes, we can re-apply the generated deployments no matter what the Node's hostnames are. In our work, we used hostnames because all tests in our benchmarking applications were implemented in the same sessions without changing our Cluster's Nodes.

3.6 Benchmark Applications

We utilize two benchmarking applications to apply and test the proposed fuzzy service placement strategies. The first application contains 11 microservices and uses HTTP and gRPC protocols for communication, and the second contains 15 microservices, all communicating via HTTP protocol. Both applications are deployed in a homogeneous environment, in a GKE Cluster with predefined resources, described in the next chapter, and the same infrastructure.

3.6.1 Google Online Boutique e-Shop

The Online Boutique eShop is a cloud-native microservices demo application that Google uses to demonstrate the use of technologies like Kubernetes, Istio, and the gRPC protocol. It is a web-based e-commerce application consisting of 11 microservices, where users can perform multiple e-commerce-related actions [19]. The application is stable, it uses five different coding languages and two of the leading service-to-service communication protocols (HTTP and gRPC), and it is implemented and optimized for use with the Google Kubernetes Engine as well as Istio, so it's an ideal application to apply our architecture and our algorithms. The application's architecture is presented in Figure 3.5, and the microservices of the architecture are represented below as described by the eshop's documentation [19].

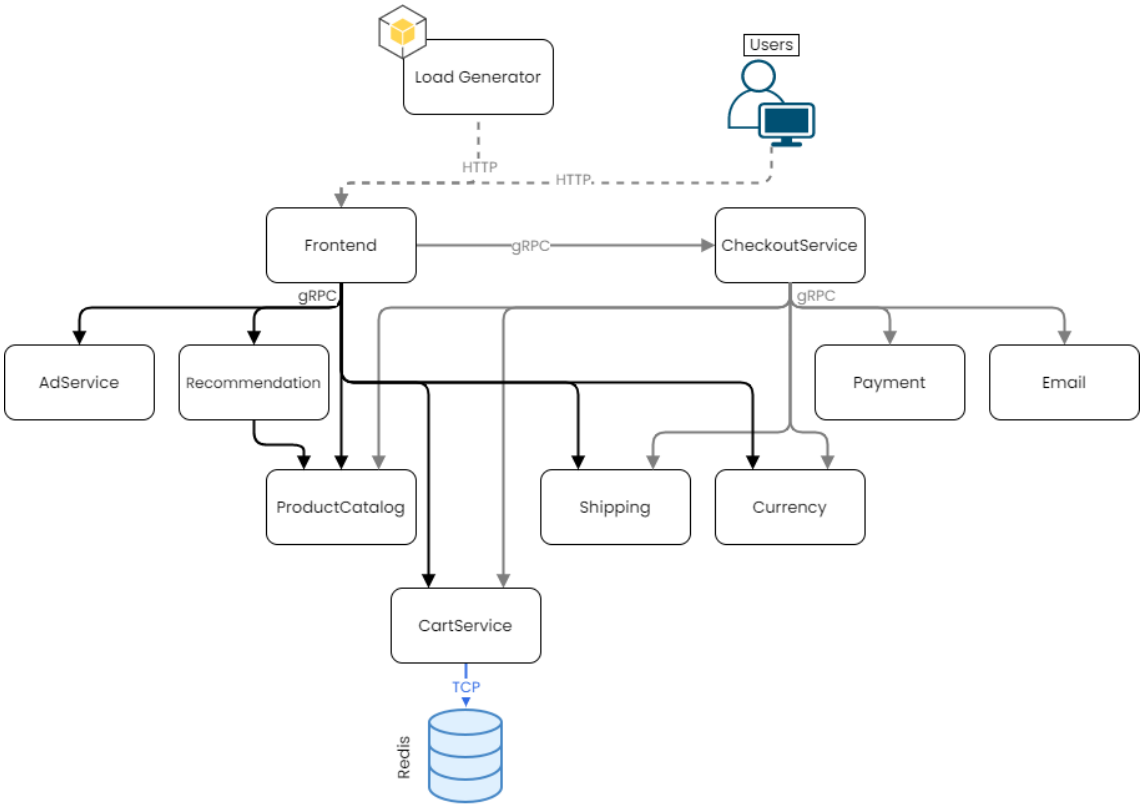


Figure 3.5. e-Shop Architecture

- **Frontend Service** (*Go*): Exposes an HTTP server that serves the website to the web 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 QPS[1] 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.

3.6.2 iXen

iXen is a prototype application developed in the Intelligence Lab of the Technical University of Crete [20]. It is based on the SOA[2] principles for an Internet of Things scenario. Tsakos K. converted the application to a microservice-based architecture with portable, independent microservices that can be deployed in a Kubernetes Cluster for orchestration. For our work, we used the provided Kubernetes configurations and deployed the application in a GKE Cluster.

iXen adopts a 3-tier architecture design model, with each tier (layer) implementing unique logic, for its respective targeted user group. The first-tier user group includes the infrastructure owners and the system administrators, which can install and connect devices in the infrastructure. The second-tier user group is the application developers, which can create subscriptions to sensors and create applications with these sensors. The final user group includes customers who subscribe to developer-created applications. Below we present the microservices of iXen, as they are described in the “iXen: context-driven service oriented architecture for the internet of things in the cloud” [20], along with a "load generator" microservice we implemented by the standards of the respective Boutique eShop service.

- **Web Service**: Provides a web interface so the users can use the application
- **Keyrock Service**: Provides a REST API so that users can register, provides policies

[1] Queries per Second
[2] Service Oriented Architecture

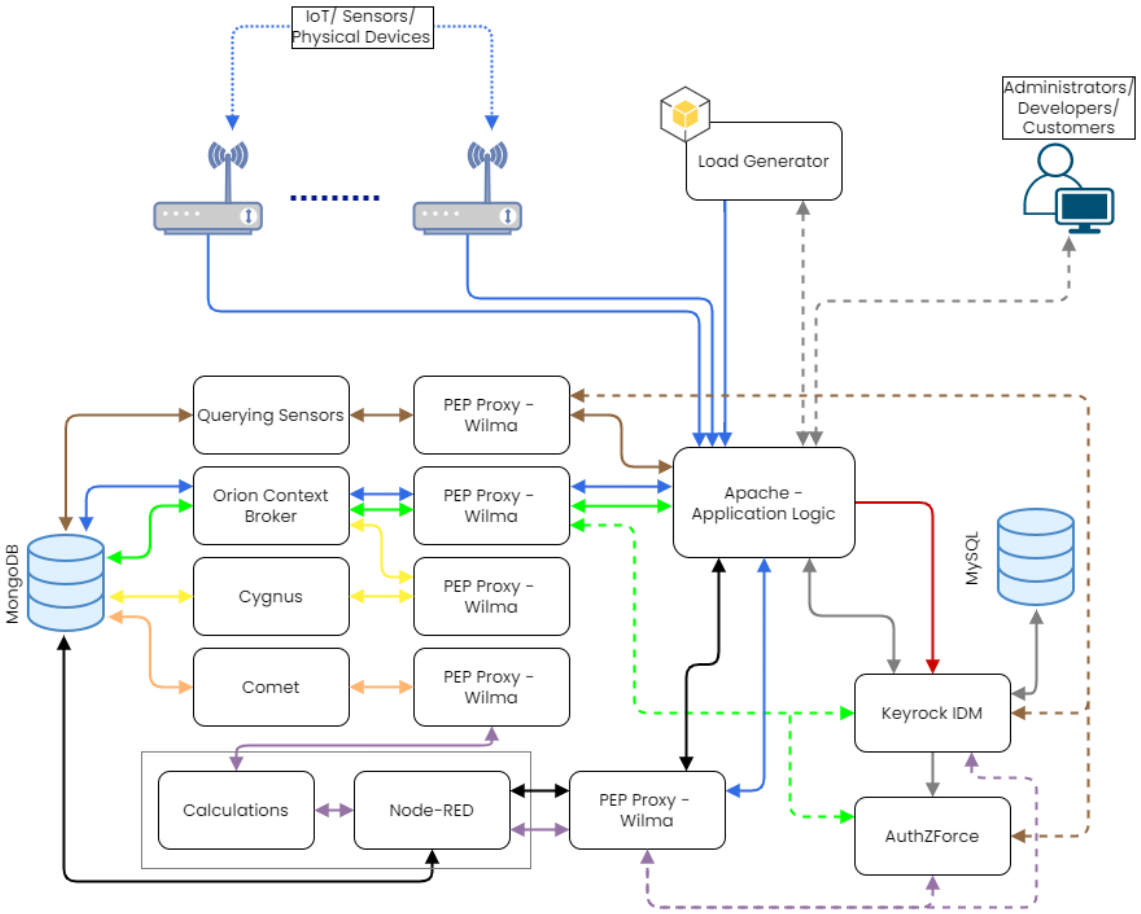


Figure 3.6. iXen Architecture

about user rights, and uses OAuth2 tokens to authorize users.

- **AuthZForce Service**: Describes respective user access rights using XACML format.
- **PEP Proxy Services**: Providing a security mechanism for services offering a public interface. Every request to these "public" services is being forwarded through its respective PEP Proxy, and only requests from authorized users with access to the service are being forwarded to the service.
- **Querying Sensors Service**: Converts a custom query syntax to mongo queries on the Mongo DB, in order to search for a device based on location, model type, type of measurement, or the unit of the measurement.
- **Orion Context Broker Service**: Publish/ Subscribe service that receives measurements from devices and makes this information available to other services and users based on their subscriptions.
- **Cygnus Service**: Accepts data streams compliant formatted with the NGSI model and can store them on multiple types 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 IoT.

- **Load Generator Service**: Written in Python/ Locust, this service continuously applies distributed requests on the application's endpoints, simulating realistic user traffic and IoT devices' measurements - updates.

Chapter 4

Experimental Results

This Chapter will introduce the tests applied in our benchmarking applications and analyze their results. The specifications and the cost of the infrastructure where the experiments took place will also be presented. Our work has two primary objectives, and the results are categorized based on these objectives. The first objective is to produce a fuzzy placement that reduces the total cost of the infrastructure, which will be achieved by reducing the total number of VMs required to host the applications and by reducing the egress traffic of our cluster. The second objective is to optimize the applications' response times, which is a product of optimal service placement but mainly relies on the fuzzy nature of our placement strategy and will be analyzed in the sections below.

4.1 System Infrastructure

We configured the Kubernetes Cluster described in Chapter 2 on the Google Cloud infrastructure. The cluster was deployed in the europe-west3-b region in the latest stable GKE version (1.21.11). Both horizontal and vertical autoscaling are disabled, along with all load balancing features of the GKE, which means that the resources allocated on our VMs will remain the same during the experiments, and our experiments will be "fair". Anthos Service Mesh is disabled because we will implement our Service Mesh, using Istio as described in the previous Chapter. Managed Service for Prometheus is also disabled because it is still in beta on the GKE, and we choose to apply the Prometheus tool in our cluster in a stable and well-documented way.

Cluster Attributes	Configuration
Location Type	Zonal
Zone	europe-west3-b
Release Channel	Regular
Cluster Version	1.21.11-gke.1100
Horizontal Autoscaling	Disabled
Vertical Autoscaling	Disabled
HTTP Load Balancing	Disabled
Managed Service for Prometheus	Disabled
Anthos Service Mesh	Disabled

Table 4.1. Cluster Configuration

A Node Pool is configured for our cluster. The Node Pool contains all the specifications for the VMs that will be spawned as cluster Nodes. The selected machines are e2-standard-2 type, which means that each has 2 vCPUs, 8 GB of RAM, and a standard boot disk with 40 GB available for storage. The virtual machines are located in the same zone as our cluster, and the autoscaling is disabled for the same reasons examined before. The OS of these machines is a Linux-based container-optimized OS that runs flawlessly with containers.

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

Table 4.2. Node Pool Configuration

Previous work [2] has proven that both our benchmarking applications, deployed along with Istio, Kiali, and Prometheus, require at least 2 host machines (Nodes) to run efficiently and stable. For our experiments, we initialize 4 Nodes for our clusters as the upper limit of VMs available in a real-life use case scenario and run the placement algorithms to determine if an optimized placement can use fewer machines than the predefined limit.

4.2 Benchmark Application Stressing

As mentioned in Chapter 2, we applied the stress testing of our applications using the python-based tool called Locust, which was **deployed as a microservice** (load generator) in both applications. We apply distributed requests on our application endpoints to simulate a load by concurrent users. These requests generate a traffic load, through which we calculate the affinity metrics referenced in Chapter 2 and create the application graph. After our algorithms produce the recommended service placement, we deploy it to our cluster and run the same stress testing process to examine the response times for each request and the aggregated response times for all the distributed requests. After acquiring the network traffic between the microservices through the Prometheus queries, we can calculate the egress traffic variation by summing the traffic between services that are not on the same node.

4.2.1 Google Online Boutique e-Shop Stress Testing

Google's online boutique e-Shop has a load generator service simulating 10 users applying distributed requests on the application's endpoints. We modified this service to simulate load from 150 concurrent users using approximately 28.000 requests with a rate of around 35 RPS. The request distribution is indicated in Table 4.3.

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

4.2.2 iXen Stress Testing

For the Stress Testing in the iXen application, we had to implement a load generator service that will simulate 100 users applying approximately 5.000 requests with a rate of around 10 RPS distributed as indicated in Table 4.4. Each simulated user initially logs into the application and gets a cookie which we store and use for authentication in all the requests he performs. We have pre-configured the sensors and mashup applications to which developers and users can subscribe. In our testing process, in addition to simulating

user-performed actions, we simulate a sensor sending random measurements, which the mashup applications then process.

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 4.4. Stress Testing Requests for iXen

4.3 Placement Strategies

For better result comparison, we have tested **three different placement strategies**. The first strategy examined is the default placement decision of the Kubernetes Scheduler. Each time an application is deployed in Kubernetes without specifying the Pod - relations or the Nodes that each Pod must be placed in, the Kubernetes Scheduler produces a placement, which mainly depends on available Node resources and each Pod's requested resources. In our experiments, we use the annotation **"Default Placement"** to represent a default Kubernetes placement. The second strategy we examined is a hard clustering strategy proposed by [2], and it is the Heuristic Packing placement of the Bisecting K Means partitioning algorithm. We used $K = 4$ as the k-value of the BKM algorithm, based on the best results of the related paper (reducing total VM number and egress traffic). We will use the annotation **BKM-HP** to reference this placement. The final placement we tested is our fuzzy service placement solution which is a result of the Heuristic Packing of the fuzzy partitioning algorithm we described in previous Chapters, and it will be mentioned as **MODSOFT-HP**.

In the Figure below, we visualize the default Kubernetes placement and a produced fuzzy placement from our MODSOFT-HP strategy. Blue lines represent the egress service-to-service traffic, while gray lines represent the ingress traffic. These examples show the optimization of the communication between services in the cluster Nodes. The fuzzy

technique used in the MODSOFT-HP placement allows the **frontend** service, which is directly connected to 7 other services, to have a replica service, so almost all communication between its connected services is ingress. The **productcatalog** service, which receives a lot of requests, as seen in Table 4.3, is replicated in all three utilized Nodes to reduce the load of each replica, which leads to faster response times.

It's important to note that as mentioned in Subsection 2.1.3, we assume that if a request requires communication from the source pod to the destination pod, if there is a destination pod instance running on the same node as the source pod, all the request from source to destination will be reserved between these pods and all traffic will be considered ingress.

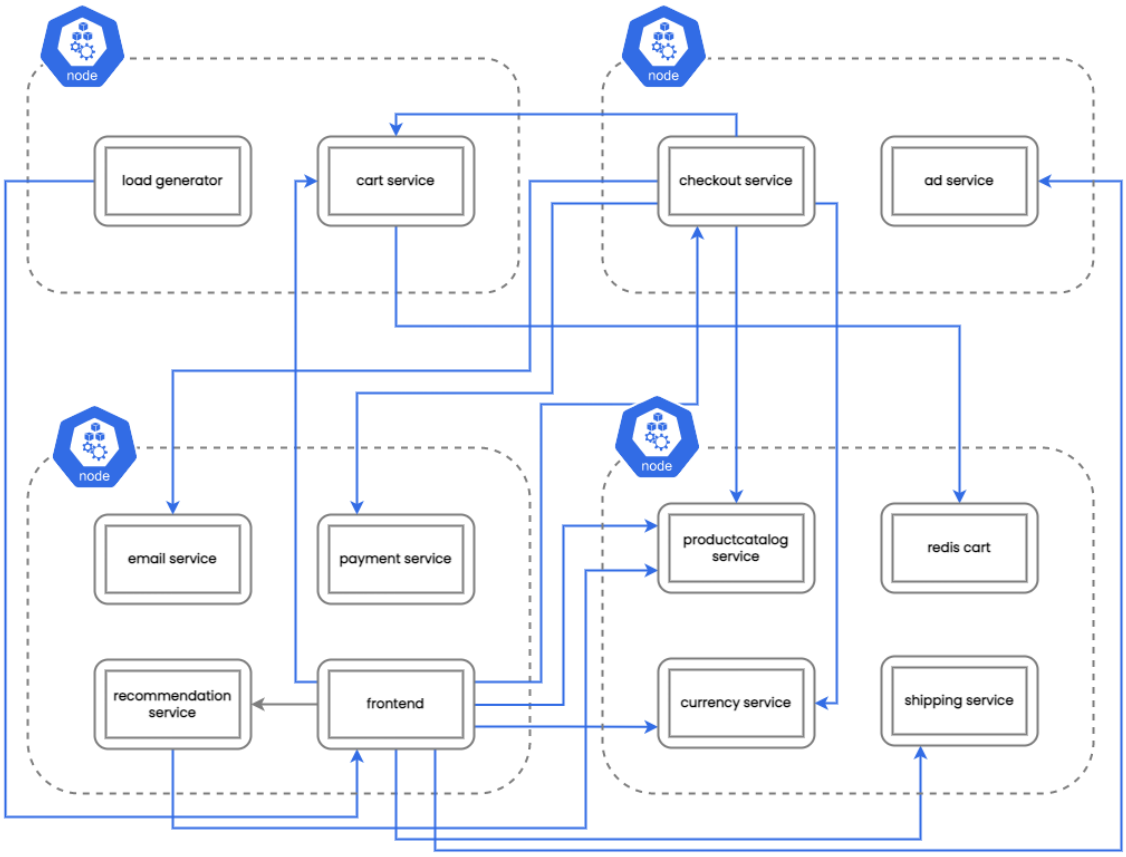


Figure 4.1. Traffic between services for **Default** Online Boutique e-Shop Placement

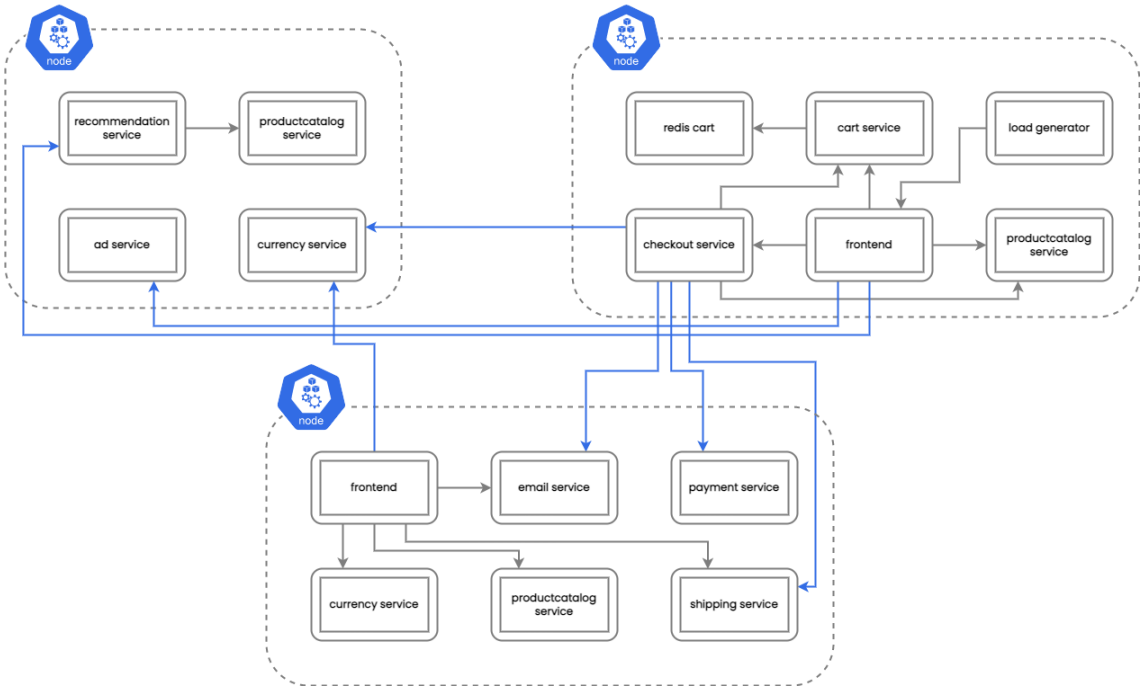


Figure 4.2. Traffic between services for **MODSOFT-HP** Online Boutique e-Shop Placement

4.4 Infrastructure Cost Calculating Function

This section will present the function used to calculate the infrastructure's cost before and after each placement solution. The cost function uses the GCP pricing, according to GCP pricing documentation [21]. The main factors that GCP charges upon are the CPU and RAM allocation, and prices vary per region. Ingress network traffic is not charged, while egress traffic is charged based on the amount of data exchanged between Nodes. Allocated storage space is also charged by GCP, but in our work, the volume of storage space used by the clusters is relatively low; thus, the cost of storage is considered negligible. The table below shows the price per resource for the machine type used in this work.

Resource	Cost (USD)
Predefined vCPU	$0.028103/vCPU/hour
Predefined RAM	$0.003766/GB/hour
Ingress Traffic	$0
Egress Traffic	$0.01/GB

Table 4.5. GCP pricing for e2-standard machine type

GCP charges **CPU** and **Memory Cost** for each machine based on the resources allocated per hour. CPU metric is the virtual cores reserved, while memory metric is the amount (in

GB) of RAM allocated. Below we calculate the function used to calculate CPU and RAM cost **per Node**.

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

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

The **Network Traffic** is charged for each **cluster**, based on the amount of traffic exchanged between Nodes (egress traffic). It can be calculated by summing **requested bytes**[1] between services placed in different Nodes. The location of each Node is also a key factor in calculating egress traffic cost, but in our work, the Nodes are all based on europe-west3 so all egress traffic is charged the same, according to Table 4.5. If i and j services in a Node N, $t(i \to j)$ the requested bytes between services i, j and t_e the egress traffic, the network cost function can be expressed as:

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

Where,

$$t_e(i \to j) = \begin{cases} t(i \to j), & \text{if } i, j \text{ in different Nodes} \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

[1] GCP charges requests, but not responses

The total Cluster cost function for n number of Nodes can be calculated using the formula below:

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

The total cluster cost for the initial placement, which utilizes 4 Nodes is:

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

4.5 Results of each Placement Strategy

This section will examine the results of our **fuzzy service placement strategy** (MODSOFT-HP) and compare them with the **default Kubernetes placement** and the **flat Bisecting K Means/ Heuristic Packing** placement (BKM-HP). We will present the execution time, the number of hosts used, and the egress traffic reduction for each strategy. The monthly infrastructure cost will then be calculated and presented.

Finally, the main objective of this work, the response time of each placement strategy, will be presented and analyzed. The Default placement results are demonstrated in blue color while red and yellow colors demonstrate the execution of the algorithms with WBA and RPS as the application graph weights.

4.5.1 Execution Time

Execution time is calculated after the graph was created and the affinity metrics are produced and it measures the time that the partitioning algorithm and the placement algorithm need to execute and produce the placement. The precise execution time depends on the machine that executes the algorithms, in our work, the algorithms were calculated locally in a machine with 3 GHz processor power, 6 cores, and 16 GB RAM. Execution time will vary in different machine specifications but it is a simple process and can be easily calculated with an acceptable execution time in 1 CPU core. Figures 4.3 and 4.3 present the execution time of each placement. We notice that the execution time for the

fuzzy placement is greater than the BKM-HP. Both the strategies use the Heuristic Packing placement algorithm although the fuzzy strategy produces more partitions and more services, that are given to the HP algorithm. This, in addition to the extra complexity of the MODSOFT algorithm, explains the difference between the execution times. Placement with WBA affinity is calculated a little faster in the e-Shop application because WBA affinities are more accurate and a difference between the pairs of WBA affinities is greater than the respective RPS affinities, meaning that the partitions are more easily "distinct". Both execution times are acceptable since the process is completed within a few milliseconds.

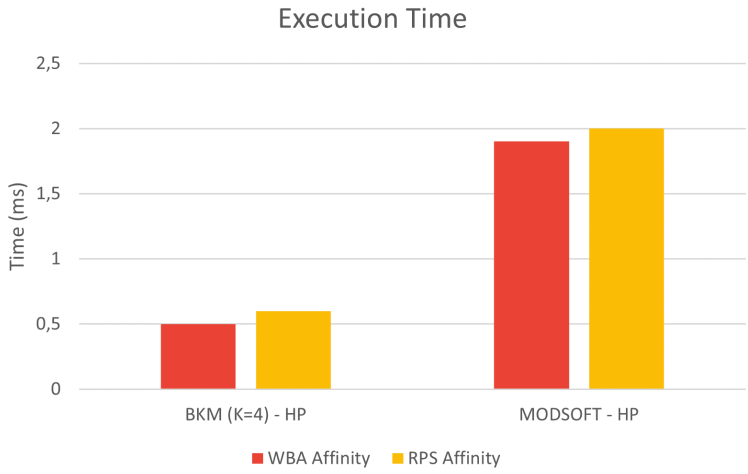


Figure 4.3. Execution Time for Online Boutique e-Shop

Figure 4.4. Execution Time for iXen

4.5.2 Number of Hosts

The Equation 4.5 shows that the most important factor affecting the total cost of the infrastructure is the number of utilized Nodes (n). As mentioned in a previous section, the initial number of available hosts is 4 Nodes. The following figures present the number of hosts utilized for each affinity from each generated placement.

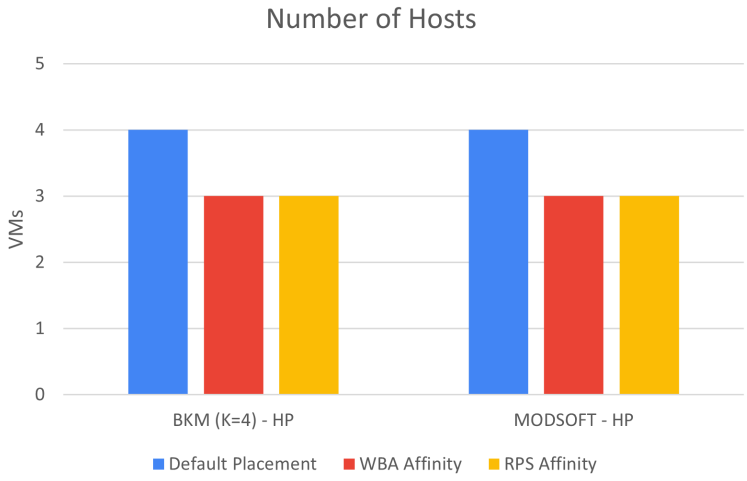


Figure 4.5. Number of utilized Hosts for Online Boutique e-Shop

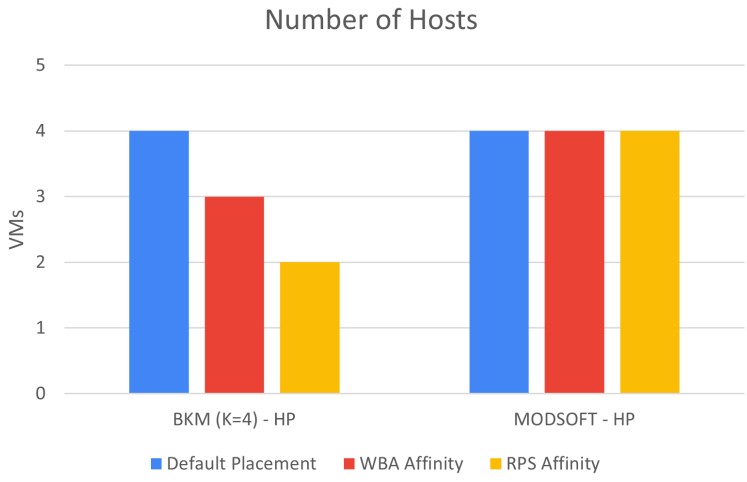


Figure 4.6. Number of utilized Hosts for iXen

The Bisecting K Means combined with the Heuristic Packing placement (BKM - HP) has produced a placement that utilizes 3 Nodes for both affinities for the Online Boutique e-Shop. The same number of Nodes can host our fuzzy placement strategy (MODSOFT - HP), even though more Pods are spawned in total in the fuzzy placement. For the iXen application, we notice that even though BKM - HP can fit the application's Pods in just

2 Nodes (which is the application's minimum requirement), during the RPS Affinity experiments, the MODSOFT - HP placement requires 4 Nodes to host the placement. We expected that the BKM algorithm would produce a placement with fewer hosts than the original because of its efficiency, proved in related work [2]. It is also easy to understand why the MODSOFT - HP placement will usually require more hosts than the hard clustering methods because the fuzzy placement will always produce more Pods than a hard placement.

4.5.3 Egress Traffic

This section presents our experiments' results regarding optimizing the egress traffic on our cluster. Using Prometheus Queries, we can find the requested bytes between any two Pods (services) inside our cluster for any period. To calculate the egress traffic, we measure only the requested bytes between services on different Nodes, as mentioned in Subsection 2.1.3.

Figures 4.7 and 4.8, present the requested MBs between services in different Nodes per hour. We notice that the fuzzy placement, especially using the WBA Affinity, reduces significantly (at around 1/10nth) the requested egress traffic per hour for both the applications using the WBA Affinity.

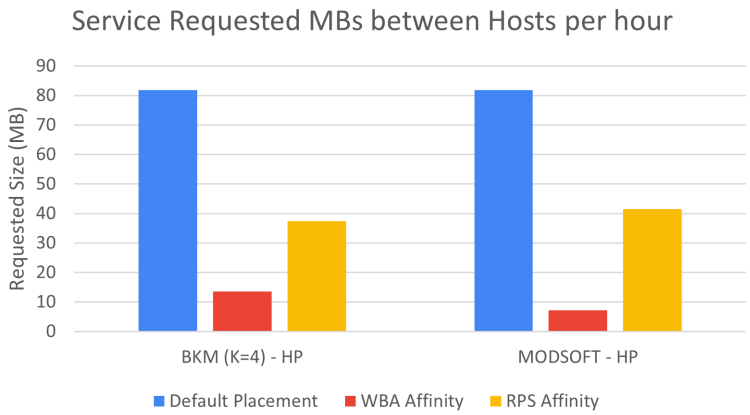


Figure 4.7. Hourly requested MBs for Online Boutique e-Shop

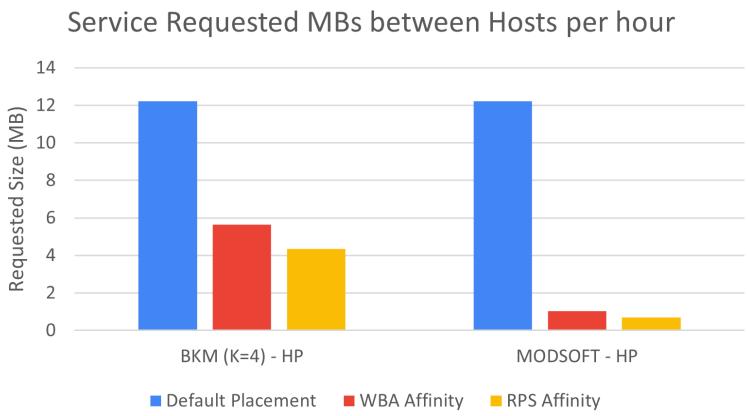


Figure 4.8. Hourly requested MBs for iXen

In figures 4.9 and 4.10 we present the egress traffic reduction per month. This traffic reduction is calculated from the reduction in the requested MBs per hour, projecting that to an one-month period. It is worth mentioning that for the eShop application, BKM-HP achieves around 83% reduction in monthly egress traffic, while for iXen, the MODSOFT-HP achieves an incredible 91% reduction in monthly egress traffic.

We notice that the egress traffic reduction is greater using the WBA Affinity for the eShop application, while reduction is almost the same for both Affinities for iXen. This something we expected, because the size of each message for the eShop application is much greater than the iXen application, so the Affinity that exploits the size of the messages is expected to achieves partitioning with greater traffic reduction.

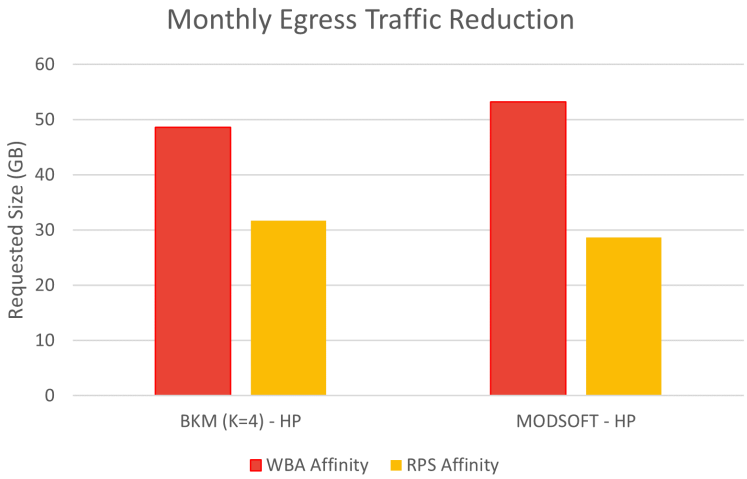


Figure 4.9. Monthly Egress Traffic Reduction for Online Boutique e-Shop

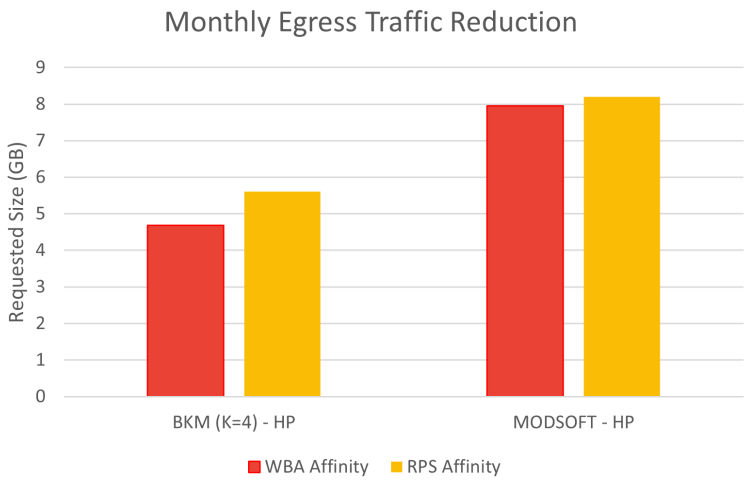


Figure 4.10. Monthly Egress Traffic Reduction for iXen

4.5.4 Total Infrastructure Cost

Using the Equation 4.5 we can calculate a projected monthly cost for each examined placement. Since the *hours* variable is the same while comparing placements per month, the cost function resides mainly on the number of VMs used to host each placement. The egress traffic GBs also affect the total cost but not as much. The figures below present the projected monthly cost for each placement using our two Affinity metrics. We expect that the cost difference between the BKM-HP and the MODSOFT-HP placement for the eShop application will be negligible because both placements utilize the same number of hosts. In contrast, the cost difference is expected to be noticeable for the iXen application, where BKM-HP manages to fit the placement in fewer hosts.

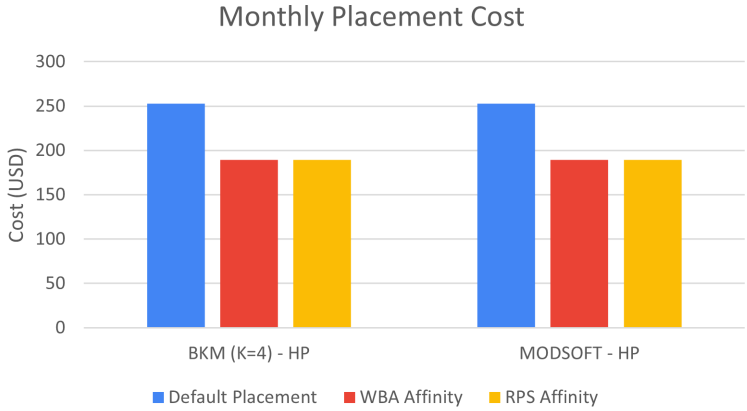


Figure 4.11. Estimated Monthly Cluster cost for Online Boutique e-Shop

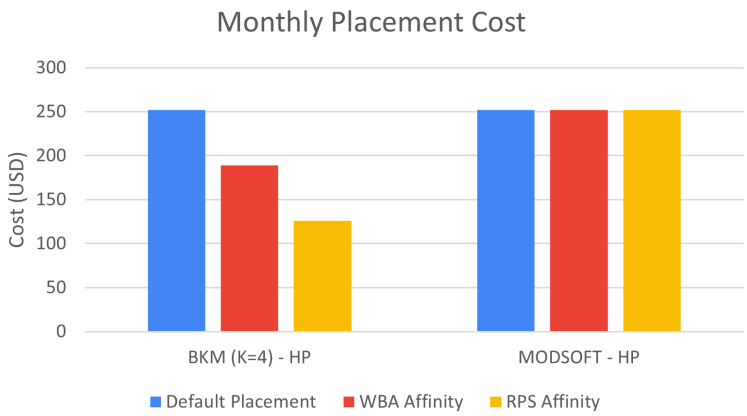


Figure 4.12. Estimated Monthly Cluster cost for iXen

As expected, the monthly placement cost is reduced for the BKM-HP and the MODSOFT-HP placement strategies for the e-Shop application. The MODSOFT-HP doesn't manage to reduce the monthly cost for the iXen application. In contrast, BKM-HP tends to reduce the monthly cost by half using the RPS Affinity because it uses only two Nodes, according to Figure 4.6. We can also notice that the egress traffic reduction does not affect the total cost of the placement strategy since the cost for egress traffic is much less than the cost for a running Node.

4.5.5 Response Time

In the current section, we will examine the response times of each placement strategy. The response times presented below are the aggregated response times of all the different requests we applied during the stress testing of the applications, the distribution of whom is shown in Table 4.3 and Table 4.4. The figures below present the average response time, the 90%ile response time, and the the 95%ile response time. The 90%ile and the 95%ile refer to the 90% and the 95% of the faster requests respectively. We expect that the fuzzy placement will have a noticeable impact on the response time of the applications' requests since, in the fuzzy placement, some high-traffic services run in more than one instance; hence the traffic is served between the instances and the requests are proccessed faster, while each instance's load is reduced.

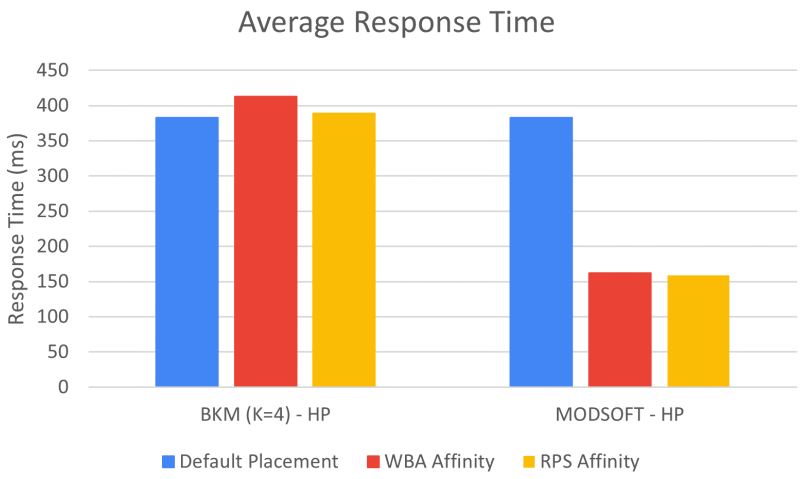


Figure 4.13. Average Response Time for Online Boutique e-Shop

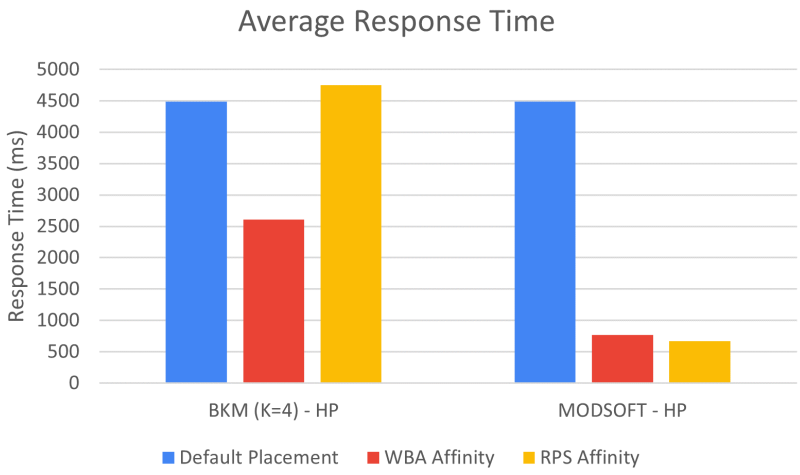


Figure 4.14. Average Response Time for iXen

We can easily see that we were right in our original assumption, and the average response time is significantly reduced in both applications using the fuzzy placement methods. This is due to the fact that essential services, such as the frontend service on each application that receives most of the requests, are replicated in more than one instance. The requests load is then served between the instances, requests are forwarded faster to their respective targets, and their response is significantly faster in overall.

Our experiments show that the BKM-HP placement, even though it produces an egress-optimized placement, does not optimize the response time of the benchmarking applications. The only test that showed the BKM optimized the response time was when using the WBA Affinity for the iXen application, but our fuzzy placement reduced the response time by an extra 70%.

It is worth mentioning that the difference in the response time of the BKM-HP placement between the two affinity metrics is not a result of the affinity metrics, but a result of the total VMs hosting the placement. As seen in the hosts section, the RPS Affinity metric produced a placement requiring just 2 VMs, which is the minimum requirement of the application. This means that the same load of requests are distributed between 3 nodes in the WBA placement and 2 nodes in the RPS placement, so the VM's load results in worst response times in the latter.

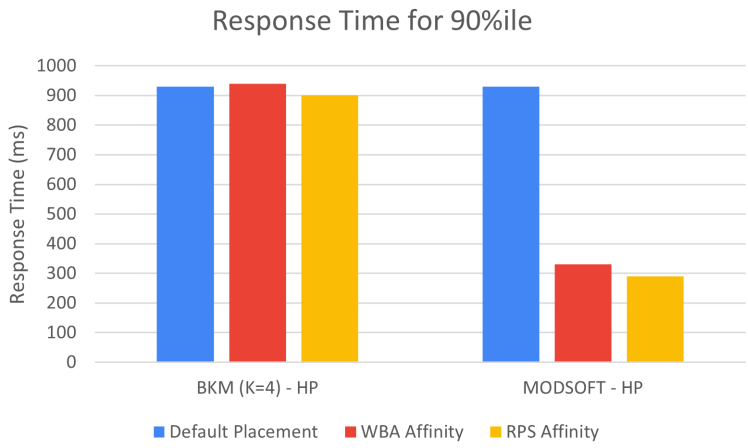


Figure 4.15. Response Time for the 90%ile of Requests for Online Boutique e-Shop

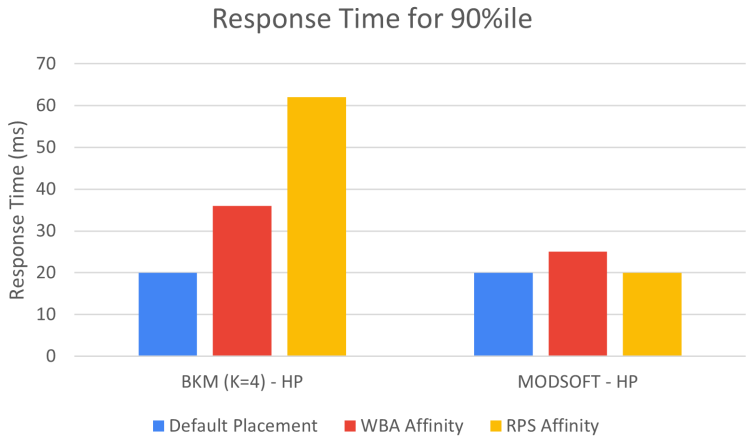


Figure 4.16. Response Time for the 90%ile of Requests for iXen

In Figure 4.16, we notice that the response times for the 90% faster responses are a lot different than the average response times for the iXen application. By examining a percentage of the faster responses, we leave a 10% of the "slower" responses out of our results. The remaining 10% of the iXen requests includes 2 requests that utilize the Node-red service and produce significantly slower response times, while the rest of the

requests have approximately the response times. This explains the difference between the average and the 90%ile response times, and by including the 5% of these "slower" requests in Figure 4.18, we come with a distribution of response times which is a lot closer to the average response time results.

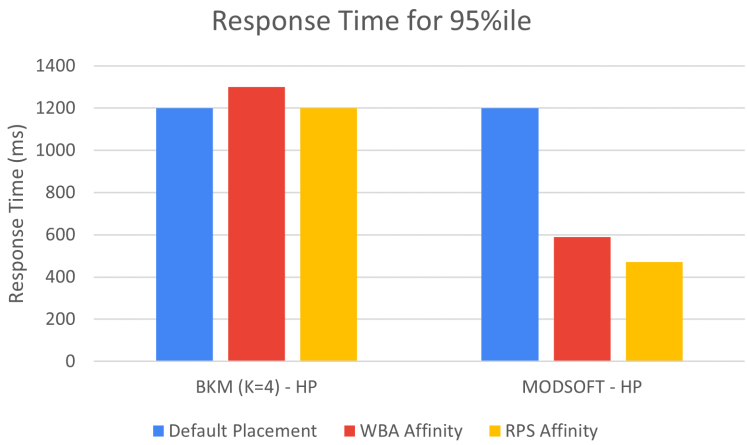


Figure 4.17. Response Time for the 95%ile of Requests for Online Boutique e-Shop

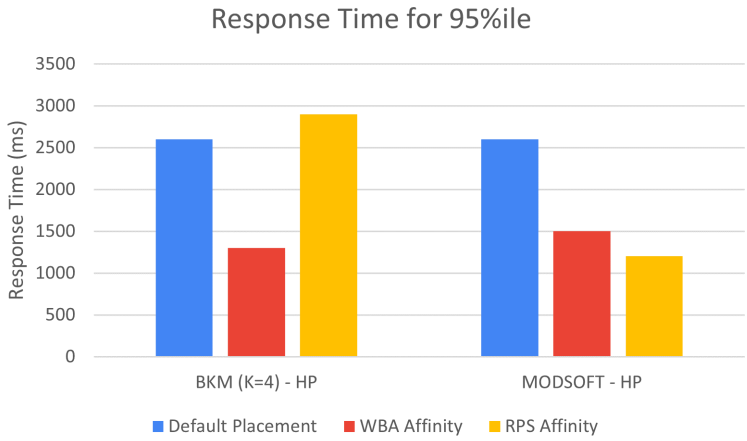


Figure 4.18. Response Time for the 95%ile of Requests for iXen

During our experiments, we noticed that for the e-Shop application, the Initial and the BKM-HP placement had many requests failing, mainly because the Frontend Pod was receiving a lot of requests per second. This happened because all the applied requests are being forwarded to their respective services, through the frontend service, resulting in a Pod crash. The requests that were performed until Kubernetes restarted the Pod, were failing and Figure 4.19 presents the percentage of these failed requests. We can see that the MODSOFT-HP placement is more stable and fail-proof due to the fact that it offers two instances of the Frontend Pod (based on the presented placement in Figure 2.2)

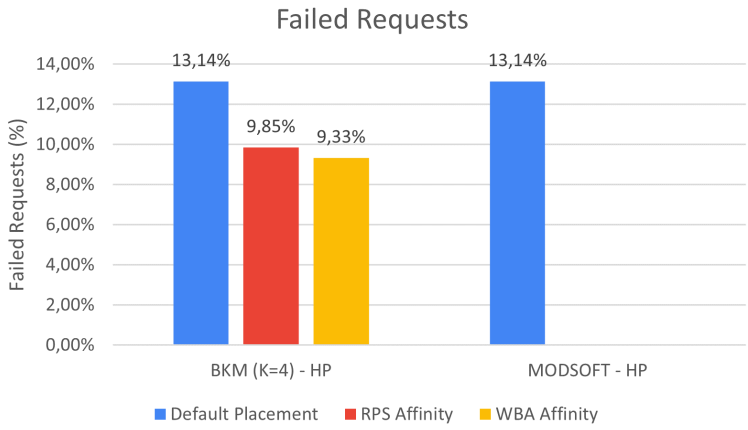


Figure 4.19. Percentage of Failed Requests for Online Boutique e-Shop

After examining the response time results, we can conclude that the affinity metrics, and the reduction of the egress traffic does not really affect the response time of the applications. The main factor that the response time relies upon, is the fuzziness of the placement, meaning how many of the highly-utilized services are running in multiple instances on the cluster.

General Conclusion and Future Work

This work aimed to reduce the response time of applications deployed in a Kubernetes cluster while optimizing the monetary cost of the infrastructure by solving the service placement problem using fuzzy techniques. Our fuzzy solution to the service placement allows services to run in multiple instances on the cluster, resulting in faster request processing by distributing the load between instances. We addressed the service placement problem using a graph-based fuzzy partitioning algorithm. Our placement strategy includes a heuristic method that attempts to place the partitions in the cluster's Nodes in a way that optimizes network traffic along with resource usage, and we manage to produce a placement that optimizes the cloud hosting monetary cost.

We utilized two benchmarking applications, Google's Online Boutique e-Shop and the iXen IoT application, to test the efficiency of our proposed solution to the service placement problem. Prometheus, Istio Service Mesh, and Kiali tools were configured in our GCP cluster to monitor traffic between services and the allocated Node resources. We extracted the traffic affinities by applying requests to the applications' endpoints and performing queries to the Kiali API.

Our experiments showed that fuzzy placement strategies could reduce the applications response times up to 50%-70%, compared to default Kubernetes placement and hard service placement strategies. The experiments also showed that our fuzzy placement strategy could reduce the cost of hosting the applications in the cloud by up to 25%. Furthermore, the fuzzy placement strategy resulted in significant egress traffic reduction compared to the initial Kubernetes placement. In some cases, we achieved more significant egress traffic reduction than the hard service placement strategy we tested. Even though egress traffic reduction doesn't affect the monetary cost in a homogeneous cloud environment, it will result in significant cost reduction in a Heterogeneous environment. Our work can also prove useful in Fog and Edge Cloud environments, where latency between services is significant. Applying our fuzzy service placement strategy will result in an increase in low-latency ingress traffic and a decrease in the high-latency egress traffic.

Having achieved the goals of our Thesis, including application response time reduction and optimization of the hosting infrastructure's monetary cost using fuzzy algorithms for microservice placement in a cloud environment, we propose additional work that can be implemented. First and foremost, we suggest an optimal placement solution that dynamically adapts to application workloads. Our placement strategy adapts to application workloads, but a complete solution where the strategy is re-examined every time the workload changes drastically can be implemented on top of our work, using our automated deployment solution for faster deployment. Examining this solution in a monthly period, with multiple simulated workloads may prove to be even more effective than our static solution since the initial Kubernetes placement does not adapt to workloads.

Furthermore, we suggest that the proposed fuzzy placement strategy is examined in a Heterogeneous Fog - Edge environment where the egress traffic reduction is expected to provide much more significant monetary cost reduction, while the fuzzy services will offer lower latency and better response times. Machines with more allocated resources could be deployed in the cloud layers of the architecture, and some services could be deployed but remain idle until the placement solution utilizes them.

Finally, we propose an examination of the service placement problem using latency between services as the affinity metric. Latency metrics will have more impact on VMs placed in different networks, and regions and because response times depend on microservices' latency, it might prove a more accurate affinity metric for response time optimization.

References

[1] 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.

[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] L.A. Zadeh. “Fuzzy sets”. In: *Information and Control* 8.3 (1965), pp. 338–353. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(65)90241-X. URL: https://www.sciencedirect.com/science/article/pii/S001999586590241X.

[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. URL: https://www.sciencedirect.com/science/article/pii/0020019094001480.

[7] 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.

[8] 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.

[9] *What are microservices?* https://microservices.io/. Accessed: 2022-06-13.

[10] *Use containers to Build, Share and Run your applications*. https://www.docker.com/resources/what-container/. Accessed: 2022-06-13.

[11] *Kubernetes Documentation*. https://kubernetes.io/docs/home/. Accessed: 2022-06-13.

[12] *Istio Documentation*. https://istio.io/latest/about/service-mesh/. Accessed: 2022-06-20.

[13] *Prometheus Documentation*. https://prometheus.io/docs/introduction/overview/. Accessed: 2022-06-20.

[14] *How to Setup Prometheus Node Exporter on Kubernetes*. https://devopscube.com/node-exporter-kubernetes/. Accessed: 2022-06-20.

[15] *Kiali Architecture*. https://kiali.io/docs/architecture/architecture/. Accessed: 2022-06-20.

[16] *Locust Documentation*. https://docs.locust.io/en/stable/what-is-locust.html. Accessed: 2022-06-20.

[17] Beschastnikh Ivan and Rosa Nelson S. “Improving microservice-based applications with runtime placement adaptation.” In: *Journal of Internet Services and Applications* 10.4 (2019).

[18] *MODSOFT GitHub Repository*. https://github.com/ahollocou/modsoft. Accessed: 2022-05-12.

[19] *Online Boutique*. https://github.com/GoogleCloudPlatform/microservices-demo. Accessed: 2022-05-12.

[20] 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.

[21] *GCP Pricing*. https://cloud.google.com/compute/all-pricing. Accessed: 2022-06-25.