Technical University of Crete
School of Electrical and Computer Engineering
Division of Intelligent Systems Laboratory

Microservice Placement Strategies in Kubernetes for Cost Optimization

---

Diploma Thesis

of

aznavouridis alkiviadis

Examination Committee : Professor Euripidis Petrakis (Supervisor)
Associate Professor Samoladas Vasileios
Associate Professor of Birkbeck, University of London,
Sotiriadis Stelios

Chania, February 2022

Technical University of Crete
School of Electrical and Computer Engineering
Division of Intelligent Systems Laboratory

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section.

(Signature)

Aznavouridis Alkiviadis
February 2022

Abstract

The current Thesis proposes efficient microservice placement strategies in Kubernetes for optimizing the total monetary costs for hosting an application running in a Cloud environment. The problem of service placement is formulated as an application's graph clustering one. The services form graphs with nodes representing services and edges representing communicating services. Both nodes and edges are labeled by resources consumed (i.e. mainly CPU and RAM resources) by the application's microservices and affinities between these microservices (i.e. messages load exchanged per second). The result of an algorithm is a set clusters comprising nodes (i.e. microservices) communicating heavily with each other. This in turn guides to placement of service clusters to nodes (or VMs) by using heuristic methods. For collecting the essential data and for monitoring an application, Service Mesh technologies, like Istio and its related services, are utilized to supervise the network communication and the resource usage of the microservices. These strategies are implemented and tested into two microservice-based applications, iXen and Google's OnlineBoutique eShop, into a homogeneous Cloud environment and particularly in the Google Cloud Platform. The experimental results reveal that the proposed microservice placement solutions reduce both the size of allocated resources and the network traffic of the host machines. This is achieved by reducing the number of the utilized machines and by increasing the internal communication volume of microservices in every host machine respectively. As a result, these solutions minimize the total monetary cost for the end-user.

Keywords

Kubernetes, Cost-optimization, Microservices, Heuristics methods, Clustering algorithms, Istio Service Mesh, Google Cloud Platform, iXen, OnlineBoutique eShop

Περίληψη

Η παρούσα διπλωματική εργασία προτείνει αποδοτικές στρατηγικές τοποθέτησης μικροϋπηρεσιών στο Kubernetes για την βελτιστοποίηση του συνολικού χρηματικού κόστους φιλοξενίας μιας εφαρμογής σε ένα περιβάλλον νέφους. Το πρόβλημα της τοποθέτησης μικροϋπηρεσιών διατυπώνεται ως πρόβλημα συσταδοποίησης του γράφου μιας εφαρμογής. Οι υπηρεσίες σχηματίζουν γράφους με τους κόμβους να αντιπροσωπεύουν μικροϋπηρεσίες και οι ακμές να αντιπροσωπεύουν επικοινωνούσες μικροϋπηρεσίες. Τόσο οι κόμβοι, όσο και οι ακμές, χαρακτηρίζονται από τους πόρους που καταναλώνουν (δηλαδή κυρίως πόρους CPU και RAM) από τις μικροϋπηρεσίες μιας εφαρμογής και από τις συσχετίσεις μεταξύ αυτών των μικροϋπηρεσιών (δηλαδή το φορτίο των μηνυμάτων που ανταλλάσσονται ανά δευτερόλεπτο). Το αποτέλεσμα ενός αλγορίθμου είναι ένα σύνολο συστάδων που περιλαμβάνει κόμβους (δηλαδή μικροϋπηρεσίες) που επικοινωνούν σε μεγάλο βαθμό μεταξύ τους. Αυτό με τη σειρά του οδηγεί στην τοποθέτηση συστάδων υπηρεσιών σε κόμβους (ή VMs) χρησιμοποιώντας ευρηματικές μεθόδους. Για την συλλογή των απαραίτητων δεδομένων και την επόπτευση της εφαρμογής χρησιμοποιούνται τεχνολογίες εξυπηρέτησης πλέγματος (Service Mesh), όπως είναι το Istio και οι επιμέρους υπηρεσίες του, για την επόπτευση της επικοινωνίας και της χρήσης πόρων των υπηρεσιών. Οι στρατηγικές αυτές εφαρμόζονται και ελέγχονται σε δύο εφαρμογές μικροϋπηρεσιών, το iXen και το Google's OnlineBoutique eShop, σε ένα ομοιογενές περιβάλλον νέφους και συγκεκριμένα στο Google Cloud Platform. Τα πειραματικά αποτελέσματα αποκαλύπτουν ότι οι προτεινόμενες λύσεις τοποθέτησης μικροϋπηρεσιών μειώνουν τόσο το μέγεθος των κατανεμημένων πόρων, όσο και την κίνηση του δικτύου των μηχανών φιλοξενίας. Αυτό επιτυγχάνεται μειώνοντας τον αριθμό των χρησιμοποιούμενων μηχανημάτων και αυξάνοντας τον όγκο της εσωτερικής επικοινωνίας των μικροϋπηρεσιών σε κάθε μηχάνημα φιλοξενίας αντίστοιχα. Ως αποτέλεσμα, αυτές οι λύσεις ελαχιστοποιούν το συνολικό χρηματικό κόστος για τον τελικό χρήστη.

Λέξεις Κλειδιά

Kubernetes, Βελτιστοποίηση Κόστους, Μικροϋπηρεσίες, Ευρηματικές μέθοδοι, Αλγόριθμοι ομαδοποίησης, Istio Service Mesh, Google Cloud Platform, iXen, OnlineBoutique eShop

to my family..

Acknowledgements

I would like to show my gratitude and appreciation to my supervisor professor Euripidis Petrakis for his continuous support and enlightenment in the conduction of this Diploma Thesis. His professional knowledge and progressive ideas have broaden my mind and expanded my opportunities in the upcoming working and educational challenges.

Next, i would like to mention how grateful i am to my colleague and assistant of my supervisor professor, in his MSc Diploma, Tsakos Konstantinos, for providing all the important information, assistance and knowledge on the utilized infrastructures and technologies. He was an essential support during the implementation of my Thesis and i wish him all the best in his new career.

Crucial factor on my successful undergraduate studies were the educational and mental support from my colleagues Maragkaki Maria, Katara Sotiria-Maria and Manara Christina. More than friends to me, i hope them to excel in their careers and may some day cooperate in future projects.

Finally, i would like to pay respects to my friends and family, and especially my mother Eleftheria, my father Pavlos and my brother Konstantinos, who helped me become the person i am today and provided me with all the important values to continuously chase my dreams and ambitions and achieve my goals. They have always persuaded me to set my standards and goals as high as possible so as to excel upon every aspect of my life.

Chania, February 2022

Aznavouridis Alkiviadis

Table of Contents

List of Figures

List of Images

List of Tables

Preface

This Thesis is developed for acquiring the Diploma from the Electronics and Computer Engineering Department of Technical University of Crete. It was originally suggested from the supervisor of the Thesis, Professor Euripidis Petrakis and his MSc student and laboratory stuff, Konstantinos Tsakos, who also implements microservice-based placement strategies in Kubernetes clusters and attempts to make a comparison between the proposed strategies and the ones he implements in his study for his MSc Thesis.

The contribution of both of them was crucial to my success in this stage of my studies and a broad range of possibilities appeared after the comprehension and experimentation with DevOps practises theoretically and practically. They led me to act and behave like an Engineer with Divide and Conquer skills for locating solutions in real life applications. The knowledge of these technologies and tools will support me for my future work and activities.

I would like to show my appreciation to my colleague, Evangelos Stamos, who implemented a LaTeX prototype for conducting various Diploma Thesis that can be utilized by students from different Technological Universities, as it was implemented in the current Thesis.

Hopefully, the DevOps practises and the Cloud Engineering will be my main area of interest in the upcoming years. With patience and continuous persistent i hope to broaden my skills and techniques on the Cloud infrastructures and extend my current work in future projects.

Chapter 1

Introduction

1.1 Problem Definition

Modern internet applications comprise of various innovative technologies, which can reduce the complexity of their creation, configuration, maintenance, deployment and observability. Containerization technologies, like Kubernetes, are common implementations in many developers' toolkits, enabling a more lightweight packing of services and facilitate the deployment of applications across different types of infrastructures and computing systems. These technologies are rapidly adapted by developers and organizations for increasing the scalability of their applications and reducing the run-time hosting and maintenance costs. Kubernetes clusters can efficiently host a wide range of different applications and secure the consistency of the run-time execution. These technologies are far from the monolithic approach and structure, making the troubleshooting process and fault isolation a simplified task for the developers. With the introduction of microservices in the modern era, applications comprise independent communicating services, each one executing a part of the application logic or even a simple task in the application. In this way, the complexity for constructing an application is reduced and developers are less about services updates or about extending the application with new services.

Furthermore, the continuous evolution of the internet applications and infrastructures, especially in the area of the Cloud environments, consists a vital factor upon deploying modern internet applications. Cloud computing provides an alternative solution to the monolithic on-premises data centers, where developers must configure the data center's hardware, network, virtualization and provide maintenance to successfully host an application. With Cloud computing, Cloud providers are responsible for all the above, providing a wide variety of software and platform as a service. Cloud services can reduce the cost of operations, complexity of the applications and increase the productivity of developers. They provide users and organizations with essential tools to monitor, optimize and extend each running application on their infrastructure.

All the above tools, improvements on applications and the Cloud infrastructures increase the overall performance and reduce the complexity of creating and managing the implemented applications. The utilization of Kubernetes can significantly reduce the run-time costs and the infrastructure management of running applications. However, configuring a Kubernetes cluster is not a simple task, if optimization of the network communication be-

tween the microservices and the execution costs are taken into consideration. The Service Placement (SP) problem [7] has raised concerns about the optimal placement solution of an application's services into a Cloud infrastructure's host machines. The solution to this problem is highly dependent on the factors that need to be optimized each time, either separately or simultaneously. Taking into consideration the monetary cost of a Cloud's infrastructure, existing strategies for service placement may not always lead to a cost optimized result, which is the desired result for this Thesis. Most Cloud vendors require extra payment for the traffic communication of the utilized host machines and their allocated computing resources and thus cost is considered a crucial factor upon building and deploying an online application, especially on Cloud infrastructures. The placement of these microservices in the available host machines can affect the cost and the network traffic of the cluster and therefore efficient strategies must be implemented to optimize these factors and further reduce the additional utilized resources and the overall Kubernetes cluster's costs. Additionally, the resource supply in CPU, RAM and storage, the availability and the geographic reach of the applications' host machines may be limited and further improvements of the service placement may be applied to address these issues and optimize the performance of each application.

1.2 Scope of Thesis

The scope of Thesis is to discover and implement microservice-based placement strategies to improve the cost of a Kubernetes cluster running on a Cloud infrastructure. We will attempt to solve the SP problem by taking into consideration the monetary cost factor and specifically to reduce the infrastructure's fee charges to the minimum. The infrastructure's total cost is related to the utilized computing resources and the network traffic between services placed in different host machines (normally network traffic between services placed in same node (or VM) comes for free). The goal is to provide placement solutions to fit every Kubernetes cluster running on every available infrastructure without requiring a prior knowledge of the application's microservices and their connections. The goal is to optimize all the factors connected to the cost function of each application's hosting, so as to provide an optimal solution to the SP problem (i.e. minimize the total monetary cost). In order to achieve this, we will apply graph clustering algorithms to locate an efficient service placement that can reduce the run-time costs of a Kubernetes cluster under different application workloads and performance measures. The proposed clusters will be placed effectively into the Cloud infrastructure by implementing heuristic methods, as a post-processing step.

1.3 Chapters Structure

This Thesis is organized in 5 main chapters, including the Introduction, and the Appendix chapter. In chapter 2, the related work on the SP problem and on graph partitioning algorithms will be presented. Furthermore, the theoretical background of Kubernetes, microservices, the various metric tools and agents and the stressing tools, that will be utilized for this Thesis, will be analyzed thoroughly. Chapter 3 includes the system design of the

implemented Kubernetes clusters and the benchmark applications that will be utilized for the performance testing of the placement strategies. Based on the graph partitioning algorithms presented on chapter 2, the modifications and optimizations on the presented algorithms will be explained to match to the utilized Cloud infrastructure. The performance measures will be properly calculated and will be collected from the metric tools and agents of the application to enable the estimation of each running Kubernetes cluster's cost. Chapter 4, the experimental phase of the Thesis, presents the service placement strategies and the Cloud infrastructure that will be utilized for the experimentation process. This chapter presents the actual calculations of the cluster's cost in the Cloud environment so as to execute the proposed strategies and produce the metrics data and the graph results according to each optimization factor. Finally, the evaluation of the results will be made to reveal the factors that can be optimized to reduce the infrastructure's cost according to each placement strategy, each benchmark application and each performance measure. In chapter 5, the Thesis implementation and experimentation process will be revised and the results will be commented on whether the Thesis goal is achieved. Moreover, improvements for future work and projects that can improve the proposed strategies and provide a more optimal solution to the SP problem will be presented. The last chapter, which is the Appendix, provides information about the methods for collecting the required data for executing the proposed placement strategies.

Chapter 2

Background and Related Work

In this chapter the theoretical background of the Thesis will be described and explained in detail. Related technologies are also presented and discussed. This includes algorithms, metric tools and agents that will be utilized and modified for the purposes of the implementing each applications and placement strategy.

2.1 Related Work on Service Placement

Service Placement (SP) problem has been discussed and analyzed rigorously in the last years in various publications and scientific magazines. In [7] and [8], authors present existing contributions to the SP problem for the Cloud and Fog environments respectively. In these papers the work on SP problem is categorized by the optimization strategy and the utilized infrastructure. The SP problem can be different for the various environments like the Cloud, Fog or Edge. For each environment a divergent approach should be used to face the problem.

In Cloud infrastructures, either on single Cloud [9], Multi-Cloud [10] or even Edge-Cloud infrastructures [11], the proposed strategies in each paper respectively managed to improve the overall performance (in terms of cost reduction) of the implemented infrastructure. In [9], service deployment strategies for minimizing the response time of a graph-based application are presented by transforming a graph-based model of the application into a minimum k-cut problem. Although a reduction in response times of services is achieved, only one VM is utilized for the conduction of the experiments and thus the strategies must be tested in a larger scale environment. In [10], authors present an efficient scheduling of microservices across different types of Cloud infrastructures. Their study and implementation of the proposed scheduling strategies reduced the communication traffic of microservices, decreased the turn-around time of requests (i.e. amount of time taken to fulfill a request) and led to higher satisfaction of user demands. They utilize weighted affinities with heuristic-based strategies for scheduling the microservices across the various Cloud environments. In [11], authors present a two-scale framework for joint service placement and request scheduling in Edge Clouds for data-intensive applications. The proposed framework improved the service placement performance and achieved a near-optimal placement of services. However, they applied only synthetic and trace-driven simulations for testing their strategy.

Service placement strategies have been also applied in Fog environments. In [12], an architecture of service placement strategy is presented in order to minimize the energy consumption in the fog computing paradigm by formulating a service placement plan to utilize resources efficiently. Their architecture is not implemented into an actual Fog environment and so their strategy is not tested. In [13], a decentralized microservices-based application placement policy is applied for heterogeneous and resource constrained Fog environments. The placement is achieved by assigning each microservice to the nearest data center in order to minimize latency and network usage. The proposed strategy improved the latency and reduced the delay in the network communication. However, none of these papers take into consideration the run-time costs of an application in the Fog environment and the network fees for the communication of application's services.

In this Thesis, we utilize Kubernetes to orchestrate the proposed applications and attempt to improve the performance of the SP problem in a Cloud Environment. We try to solve the SP as a problem of graph partitioning in a microservice-based application. We will focus in minimizing the number of partitions (equivalently the number of Kubernetes nodes) of an application and at the same time minimize the amount or traffic (i.e. message load exchanged per second) among these nodes. In fact, the methods attempts to maximize intra-node (i.e Ingress network) communications (for which a the end-use is not charge) and the same time minimize inter-node (i.e. Egress network communication for which the user is charged). Our goal is to locate service placement strategies that will achieve these goals. There have been several studies on the SP problem in Cloud infrastructures using the Kubernetes. In [14], two Kubernetes schedulers are implemented to improve the scheduling processes of services in a private Cloud infrastructure to serve the network requirements of a University Campus. The proposed schedulers take into consideration the historical data of requests and their priority queue on each service and apply an improved scheduling strategy (i.e. strategy for accelerating the students' requests) in comparison with the default Kubernetes Scheduler. Their strategy led to better resource utilization, decreased the scheduling time and increased the task throughput of processes. Finally, in [15] and in [16], service placement strategies are presented to increase the performance of a Kubernetes cluster and reduce the number of Nodes needed to host an application. In the former paper, an adaptation mechanism based on the service affinities is implemented in order to rearrange the services into the existing host machines initially placement from the Kubernetes Scheduler. In the latter study, two graph-partitioning algorithms, the Binary Partion and the K-Partition, and a post-processing placement algorithm, the Heuristic Packing, are applied on a graph-based application to improve the service placement of the services in the Cloud infrastructure. Both strategies present an increase on the successful placement ratio (i.e theoretical optimal solution compared to the produced solution) and a decrease on the number of VMs needed to host the proposed applications. However, both strategies are tested under mock (i.e not realistic) evaluations and experimental environments and they are not tested in real-time applications on a Cloud infrastructure. Our work is primarily based on the proposed algorithms in [15] and in [16], by implementing and combining the proposed algorithms into our realistic benchmark applications on the Google Cloud Platform (GCP). We want to test their behavior and their placement

performance under realistic application's and various synthetic workloads.

2.2 Related Work on Graph Partitioning

To address the SP problem, we will utilize graph partitioning algorithms, as mentioned in the previous section. In this section, the utilized partitioning algorithms and their theoretical background will be presented.

2.2.1 Theoretical Background of 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 [17][18]. Computationally, the problem is NP-hard, and the corresponding decision problem is NP-complete. Despite its worst-case hardness, optimal solutions to very large instances of the problem can be produced with sophisticated algorithms like first-fit algorithm which provides a fast but often non-optimal solution. A variant of bin packing that occurs in practice is when items can share space when packed into a bin. Specifically, a set of items could occupy less space when packed together than the sum of their individual sizes. This variant is known as Virtual Machine (VM) packing since when virtual machines are packed in a server, their total memory and CPU requirements could decrease due to pages shared by the VMs that need only be stored once. If items can share space in arbitrary ways, the bin packing problem is hard to even approximate. However, if the space sharing fits into a hierarchy, as is the case with memory sharing in virtual machines, the bin packing problem can be efficiently approximated.

First-Fit Algorithm

The First-Fit algorithm is one kind of the solution to the bin packing problem and it involves placing each item into the first bin in which it will fit [17]. The algorithm scans the items in any order and every item is attempted to be placed in the available bins sequentially [18]. If it does not fit into existing bins, then a new bin is created to place the item. It requires $\Theta(nlogn)$ time, where n is the number of items to be packed. The algorithm can be made much more effective by first sorting the list of items into decreasing order. However, this still does not guarantee an optimal solution, and for longer lists may increase the running time of the algorithm.

Minimum K-Cut

The minimum k-cut, is a combinatorial optimization problem that requires finding a set of edges whose removal would partition the graph to at least k connected components [19]. These edges are referred to as k-cut.

The goal is to find the minimum-weight k-cut. The problem assumes that given an undirected graph G = (V, E), where V is the Node set and E the edge set, with an

assignment of weights to the edges and an input k value of desired partitions the algorithm is to find a k-cut of minimum total weight of edges whose ends are in different components. The problem is NP-Complete and for fixed k it can be solved in polynomial time and specifically in $O(|V|^{k^2})$. Given a service-based application, we can represent it as a graph, where the Nodes represent services and the weights of edges represent the traffic rate. Specifically, the traffic rate from service s_i to service s_j and the rate from service s_j to service s_i are represented as two edges respectively in the graph. Hence, finding a minimum k-cut of the graph is equivalent to partitioning the application into k parts while keeping overall traffic between different parts to a minimum. However, for arbitrary k, the minimum k-cut problem is NP-hard [16].

Contraction Algorithm

In computer science and graph theory, the contraction algorithm, or as it is also known as Karger's algorithm, is a randomized algorithm to compute a minimum cut of a connected graph [20]. The basic idea of the Karger's algorithm is to randomly choose an edge from the graph with probability proportional to the weight of edge and merge the Nodes assigned to these edge into one Node (called edge contraction). In order to find a minimum cut, the algorithm iteratively contracts the edges which are randomly chosen until the required number of Nodes remain. The edges that remain are the output by the algorithm [16]. When the graph is represented using adjacency lists or an adjacency matrix, a single edge contraction operation can be implemented with a linear number of updates to the data structure, for a total running time of $O(|V|^2)$. Algorithm 2.1 presents the Contraction Algorithm for an undirected graph application and k = 2. The contraction algorithm will be used in conjunction with the partitioning algorithms, which will be presented in the next section, in order to locate the application's services partitions. For minimum k-cut, the contraction algorithm is basically the same, except that it terminates when k nodes remain and returns all the edges left in the graph G[16].

Algorithm 2.1: *Contraction Algorithm (k = 2) [16]*

Input: G = (V,E)
Output: a cut of G
while |V| ≥ k **do**
  Choose and edge $e_{u,v}$ with probability proportional to its weight
  G ← G - $e_{u,v}$ *//Contract edge* $e_{u,v}$
end while
Return the cut in G

2.2.2 Service Affinities

Service affinity defines the relationship between services, specifying that the instances of one service are started either on the same host as the other service (affinity) or on a host different from the one used by the other service (anti-affinity)[21]. Affinities and Anti-Affinities can be introduced and modified only by the cluster administrators of each

application. Moreover, administrators can configure a service group to define the affinity and anti-affinity conditions, which means that the service placement decision cannot be adapted to the dynamic workload variations of an application. In this Thesis, service affinities are used in Kubernetes to define the Pod and Node relationships of an application. There are four affinity conditions:

- **Hard affinity:** Every service instance must run on the same host as the list of service(s) or service group(s).
- **Hard anti-affinity:** Every service instance must run on a host different from the list of service(s) or service group(s).
- **Soft affinity:** Every service instance is preferred to run on the same host as the list of service(s) or service group(s).
- **Soft anti-affinity:** Every service instance is preferred to run on a host different from the list of service(s) or service group(s).

2.2.3 Related Algorithms

In this section the algorithms that will be used during the implementation phase will be briefly presented and analyzed. The theoretical background, the time complexity and the pseudocodes of each algorithm will be displayed.

Heuristic First-Fit

Sampaio et al. on [15] propose a heuristic approach to optimize service placement in the current infrastructure by moving services with higher affinity (i.e. higher traffic communication rates) on the same host machine. The proposed algorithm, a Heuristic First-Fit variant algorithm, computes a new placement for the application services by accessing the resource usage of the cluster Nodes and Pods and the available host machines of the cluster. In this modified version of First-Fit, the algorithm reorganizes the microservices in the available host machines of the cluster so that microservices with high affinity are co-located, while microservices' resource usage and availability of resources at the host are taken into account.

Algorithm 2.2 presents the Heuristic First Fit Variant algorithm, which iterates over the affinities which are sorted in descending order and attempts to co-locate the microservices with higher traffic communication rates at the same host machine. For each associated pair of microservices m_i, m_j linked by an affinity, the algorithm attempts to place m_j onto the host of m_i (H_i). If H_i does not have enough resources, the algorithm tries to put m_i onto the host of m_j, H_j. If both hosts do not have enough resources to co-locate m_i and m_j, these microservices remain at their original hosts. When a microservice is placed into a new host, it is marked as moved and cannot move anymore, even if it is connected with another service in the application with less affinity traffic rates. In the end, a list of movements is generated containing microservice identities and their new locations. This

algorithm does not guarantee that the list of moves computed is optimal for a cluster given a set of microservices.

Algorithm 2.2: *Heuristic First Fit Variant [15]*

Input: Hosts (H), microservices (m), resources (r)
Output: Placement Solution
moved ← [ ]
//Affinities are in decreasing order
for every pair of affinities **do**
  $m_i \in H_i$ // m_i located at host H_i
  $m_j \in H_j$ // m_j located at host H_j
  $m_j \neq m_i$, $H_j \neq H_i$
  hasMoved ← False
  if r(m_i) + r(m_j) ≤ r(H_i) ∧ $m_j \notin$ moved **then**
    $H_j \leftarrow H_j$ - m_j
    $H_i \leftarrow H_i \cup m_j$
    hasMoved ← True
  else if r(m_i) + r(m_j) ≤ r(H_j) ∧ $m_i \notin$ moved **then**
    $H_i \leftarrow H_i$ - m_i
    $H_j \leftarrow H_j \cup m_i$
    hasMoved ← True
  end if
  if hasMoved **then**
    moved ← moved ∪ [m_i, m_j]
  end if
end for

Binary Partition

Yang et al. in [16] present a strategy for optimizing service placement. Initially, the application services are partitioned into groups of services to be placed into the available infrastructure's VMs. To be able to compare the resource allocations and capacities of the Heterogeneous VMs (i.e. VMs with different resource allocation and OS) of the infrastructure, available, allocated and requested resources from services must be initially normalized according to the maximum available resources of each host machine.

Considering multi-resource demands of different services, threshold α is introduced to determine the size of allocated resources each partition can have so as to be successfully placed into the application's VMs. Threshold α denotes the upper bound of the resource demands of partitioned parts, which means that the partition algorithms are executed continuously until the total resource demands from each part do not exceed α or no part contains more than one service. The value of threshold α ranges between $[0, 1]$ after the normalization process of the resource values and each part of the application partitions must not exceed this threshold.

Taking the set of services of an application in the input, the Binary Partition (BP) method of Algorithm 2.3 attempts to create smaller groups of services so that the source demands of each group (partition) does not exceed threshold α or no part contains more

than one service. This is achieved by dividing the processed part each time into two sub-partitions. The initial partition is $P = S$ and the algorithm's requirements are examined. Foe each algorithm's step execution (i.e for each processing application's partition) a service graph $G = (V, E)$ is constructed from the current part and the contraction algorithm for $K = 2$ is applied in order for the part to be divided according to the minimum K-cut. The contraction algorithm is repeated $n = |V|$ times, where $|V|$ is the total number of Nodes, and for each iteration the produced graph is compared with the minimum graph that has been calculated in previous iterations (according to the total sum of service affinity rates). After this process, a two new sub-partitions (S_x, S_y) of the current part is produced according to the minimum graph that has been previously calculated and the two sub-parts are stored into the vector of the application's partitions P. The algorithm is executed repeatedly until all requirements are met.

The time complexity of the Binary Partition is O($n^2 mlog^2 n$), where n is the number of services and m the total edges of the application. The algorithm can produce at most n partitions and for each iteration of the algorithm the contraction algorithm is executed n times at most. Finally, the contraction algorithm is executed in O($mlog^2 n$).

Algorithm 2.3: *Binary Partition [16]*

Input: Service-Based Application (S), threshold α
Output: Partition $P = \{S_1, S_2, ..., S_N\}$
$P \leftarrow \{S\}$
while exists part S_i in P that total resource demands exceed α and part S_i contains more than one service **do**
    P ← P - S_i //Remove partition i from application's partitions P
    Construct graph G = (V, E) based on S_i service affinities
    n ← |V|
    G_{min} ← G
    t ← 0
    while t ≤ n **do**
        Perform Contraction Algorithm (k = 2) to get a cut G'
        $G_{min} \leftarrow \min(G_{min}, G')$
        t ← t + 1
    end while
    Create two sub-partitions {S_x, S_y} from part S_i according to G_{min}
    $P \leftarrow P \cup \{S_1, S_2, ..., S_k\}$
end while
Return P

K-Partition

Similar to BP algorithm, K-Partition (KP) algorithm [16], attempts to produce a partition of the available services of the application, with the same requirements and algorithmic logic as Binary Partition algorithm. The main difference between these algorithms is that Binary Partition divides the processed part into two sub-parts by applying the contraction algorithm for $K = 2$, while K-Partition increases the value of K upon each iteration of the

algorithm. This can lead to the creation of affinity hubs instead of affinity tuples between the microservices and can accelerate the partition process. The basic process of the algorithm remains the same as the Binary Partition. The contraction algorithm is executed also N times, but the total partitions is increased upon each iteration, with $K = 2$ to be the initial value. By increasing the number of partitions at each iteration, the time complexity of the algorithm increases exponentially. Each iteration of contraction algorithm produces a minimum graph (compared to the total sum of service affinities) and the best solution produces the desired k (the number of partitions created upon every iteration) sub-partitions, $\{S_1, S_2,...,S_k\}$. Similarly, this process would be repeatedly performed until the resource demands from each part do not exceed threshold α or no part contains more than one service. K-Partition can decrease the execution calls of the contraction algorithm upon each step compared with the Binary Partition, but there is always the possibility of over splitting a partition and thus produce a larger number of application's partitions. The pseudocode of the K-Partition is presented in Algorithm 2.4.

ALGORITHM 2.4: *K-Partition [16]*

Input: Service-Based Application (S), threshold α
Output: Partition $P = \{S_1, S_2, ..., S_N\}$
$P \leftarrow \{S\}$
k $\leftarrow$ 1
while exists part S_i in P that total resource demands exceed α and part S_i contains more than one service **do**
    P $\leftarrow$ P - S_i //Remove partition i from application's partitions P
    Construct graph G = (V,E) based on S_i service affinities
    n $\leftarrow$ |V|
    $G_{min} \leftarrow$ G
    k $\leftarrow$ k + 1
    t $\leftarrow$ 0
    while t $\leq$ n **do**
        Perform Contraction Algorithm until k Nodes remain to get a k-Cut G'
        $G_{min} \leftarrow \min(G_{min}, G')$
        t $\leftarrow$ t + 1
    end while
    Create k sub-partitions $\{S_1, S_2,...,S_k\}$ from part S_i according to G_{min}
    $P \leftarrow P \cup \{S_1, S_2, ..., S_k\}$
end while
Return P

Bisecting K-Means

Bisecting K-Means (BKM) is a divisive hierarchical clustering algorithm [22] and is based on the K-Means algorithm. Given a data set, all available data points are initially assigned to a single cluster and the algorithm utilizes the K-Means algorithm to select the two best fit sub-clusters to partition the data points. Upon each iteration, Sum of Squares Error (SSE) is calculated in order for the inter-cluster dissimilarity (i.e the distance from the selected points to the centroids) to be measured and so the next centroids can be se-

lected respectively. Then, the proposed sub-cluster is selected to be divided according to the SSE producing the two new sub-clusters. This process is repeated until the algorithm reaches the desired number of K clusters, which is an input given by the users. The algorithm generates binary clustering hierarchy, it is highly affected from the initial centroids selection and may not converge to global optima. Algorithm 2.5 shows the pseudocode of the Bisecting K-Means algorithm.

Algorithm 2.5: *Bisecting K-Means [22]*

Input: Cluster C, number k of desired clusters
Output: k Clusters of Application
i ← 1
while i < k **do**
  Select a parent cluster, C to split
  for fixed number of iterations **do**
    Use K-Means to split C into C_1 and C_2
    Calculate inter-cluster dissimilarity for C_1 and C_2
  end for
  Select the sub-clusters with highest inter-cluster dissimilarity
  i ← 1 + 1
end while

However, the selection of K is an important factor and can cause large deviation between the results and, eventually, produce sub-optimal results [23]. Furthermore, the selection of sub-clusters, which are produced randomly, can increase the deviation and increase the errors of clustering the data points. The selection of K must be selected properly in order for the intra-cluster similarity to be increased and the inter-cluster difference to be decreased.

Heuristic Packing

Heuristic Packing (HP), which is presented also in [16], is a placement algorithm attempting to pack each part of the given application partitions into the application's host machines. This algorithm is executed as a post-processing step of the graph partitioning algorithm (BP and KP) to reduce the size of the utilized VMs to host the application's partitions. Without considering the traffic rate, the problem can be formulated as a classical multi-dimensional bin packing problem, which is known to be NP-hard. When there is a large amount of services involved in the application, it is infeasible to find the optimal solution in polynomial time. In this algorithm, two greedy heuristics are introduced, the Traffic Awareness (tf) and Most-Loaded Situation (ml) heuristics, considering the time complexity and the packing quality factors. Given a set of services S and a set of host machines m, the former heuristic is the sum of the traffic rate between the services in part S_i and the services that have been determined to be packed into machine m_j before (services from another already processed partition). The latter is a scalar value of the load situation between the vector of resource demands from part S_i and the vector of available resources on machine m_j. Cases in which Traffic Awareness factors are equal, the

Most-Loaded heuristic prioritizes the machines in which services will be placed.

Algorithm 2.6, presents the Heuristic Packing algorithm. Inputs to the algorithm are the application's partitions produced by the partitioning algorithms $P = \{S_1, S_2, ..., S_N\}$ and the available resources on each machine $V = \{(V_1, V_2, ..., V_M\}$. The algorithm will produce a placement solution if each part can be placed into at least one machine. Initially, all parts are processed sequentially and for each part and available VM, that can host that specific part, Traffic Awareness and Most-Loaded heuristics are calculated according to the services of the specific part using the formulas below. HP algorithm is affected from the produced application's partitions and will not always produce an optimized placement solution (compared to the utilized VMs). If each partition can be efficiently hosted in at least one infrastructure's VM, then HP will always produce an optimal placement solution.

Traffic Awareness is calculated as follows:

$$tf \leftarrow \sum t_{uv} \tag{2.1}$$

Where:

- $tf \rightarrow$ the total traffic rate of all services
- $t_{uv} \rightarrow$ traffic rate between services u, v

and Most-Load Situation is calculated by the below equation:

$$ml \leftarrow \sum_{k=1}^{R} \frac{d_i'^k}{v_j^k} \tag{2.2}$$

Where:

- $ml \rightarrow$ the most loaded situation value
- $R \rightarrow$ the set of resource types (CPU, RAM, Storage etc)
- $d_i'^k \rightarrow$ amount of resource r_k that service s_i demands
- $v_j^k \rightarrow$ amount of resource r_k available on machine m_j

The higher ml rate, the more loaded the host machine is. The idea of this heuristic is to improve the resource efficiency by packing each part to the most loaded machine. As our main goal is to minimize the inter-machine traffic, the algorithm is designed to first prioritize the machines based on the factors of tf. If the factors of tf are the same, it then prioritizes the machines based on the factors of ml. Finally, if there is no VM to host that specific part of the application's partition, then the algorithm terminates and the specific partition can not be placed into the available host machines.

The time complexity for the Heuristic packing is O($nM+ n^2$). Upon every algorithmic execution all n parts in the application's partition are processed and for each part all the available Mmachines are examined for hosting that part. For each part that can be packed into an available host machine factor tf is calculated. The overall time complexity of calculating the factor tf in one execution of the algorithm is O(n^2).

ALGORITHM 2.6: *Heuristic Packing [16]*

Input: Partition P = (S_1, S_2...S_N) of application, vectors of available resources on each machine V = (V_1, V_2...V_M)
Output: a placement solution X
Calculate vectors of resource demands of each part a ($D_1^{'}$, $D_2^{'}$...$D_N^{'}$)
X ← $[x_{ij} = 0]$ for every part and host machine
for i ← 1; i ≤ $N^{'}$; i++ **do**
  tf ← 0, ml ← 0, y ← 0
  for j ← 1; j ≤ M; j++ **do**
    if part S_i can be packed into machine m_j **then**
      tf ← $\sum t_{uv}$
      ml ← $\sum_{k=1}^{R} \frac{d_i^{'k}}{v_j^k}$
      if $tf_j \geq$ tf **then**
        tf ← tf_j, ml ← ml_j, y ← j
      else if $tf_j ==$ tf and $ml_j >$ ml **then**
        tf ← tf_j, ml ← ml_j, y ← j
      end if
    end if
  end for
  if $y == 0$ **then**
    Return null
  else
    V_y ← V_y - $D_i^{'}$
    x_{iy} ← 1
  end if
end for
Return X

2.3 Infrastructure and Tools

In the section the infrastructure, in which applications will be hosted and orchestrated, and the various metric tools and agents, that will be utilized to collect the desired data, will be presented. The microservice architecture, in which are applications are based, and the Kubernetes orchestration platform will be explained. The idea of Service mesh will be introduced and analyzed through the Istio Service Mesh. Finally, the essential tools of Istio, which will be used to monitor the infrastructure and collect all the essential data, and the stressing tool for testing each application will be presented.

2.3.1 Microservices

Microservice architecture is an architectural style that structures an application as a collection of services that are highly maintainable and testable, loosely coupled and independently deployable. The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack [1]. As applications grow in size and become more complex, microservices architecture enables high availability and much easier expansion and scalability. De-

veloper teams can co-operate and apply changes to different parts of the application, isolate problematic services and application errors and fix the problems separately without altering other services in this process. When using microservices, software functionality can be isolated into multiple independent modules that are individually responsible for performing precisely defined, standalone tasks. These modules communicate with each other through simple, universally accessible Application Programming Interfaces (APIs). Figure 2.1 displays the microservices architecture and the way of communication from different end-users.

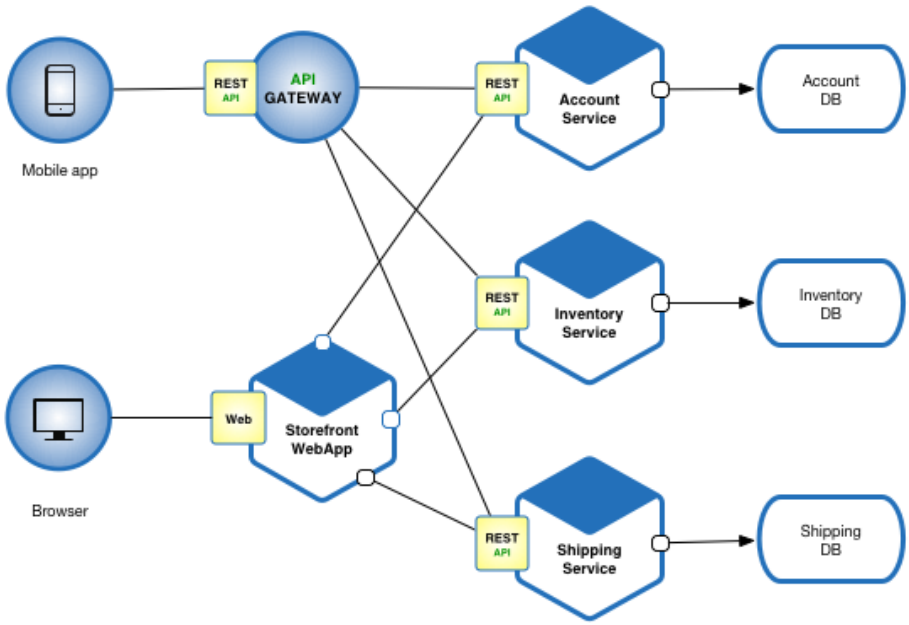

Figure 2.1. *Microservice Architecture [1]*

2.3.2 Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation [24].

The application of service containers is preferred among the other traditional options of physical servers or VMs running on a specific computer and Operating System (OS) due to the fact that containers have relaxed isolation properties to share the OS among the applications. They are considered lightweight and they are capable of allocating specific resources and disk space on their own. They are highly portable across Clouds and OS distributions. Containers achieve high efficiency on image creation and ease of use, continuous development, good observability on the application and metrics, they provide resource isolation, utilization and they decouple applications from the infrastructure.

Kubernetes provides all the essential tools and a framework to run distributed systems resiliently and handle the behavior and maintenance of containers. It handles the scaling of the application and containers, the fail over situations and provides efficient deployment patterns. It must be mentioned that Kubernetes does not limit the types of applications supported, nor does it deploy automatically source code or build the application. Fur-

thermore, it does not provide developers with dictating logging, monitoring or alerting solutions.

Kubernetes main features are:

- Service discovery and load balancing
- Storage orchestration
- Automated roll outs and rollbacks
- Automatic bin packing
- Self-healing
- Secret and configuration management

Related Components

By deploying Kubernetes into an application, a cluster is initialized. A Kubernetes cluster consists of a set of worker machines, called Nodes, that run containerized applications. The worker Nodes host the Pods that are the components of the application's workloads. The Control Plane manages the worker Nodes and the Pods in the cluster. In production environments, the Control Plane usually runs across multiple computers and a cluster usually runs multiple Nodes, providing fault-tolerance and high availability [2].

Control Plane consists of several components, the most important being the kube-apiserver, etcd and kube-scheduler. The API server is a component of the Kubernetes Control Plane that exposes the Kubernetes API to be accessed externally or internally. Etcd component is a consistent key-value storage used as Kubernetes backing repository for all cluster data. Kubernetes Scheduler is a component, which schedules newly created pods to nodes (VMs) and will be further analyzed in the next paragraphs.

Apart from the Control Plane, which is the brain of Kubernetes, there are some Node components that run on every Node, maintaining running Pods and providing information about the Kubernetes runtime environment, known as kubelet, kube-proxy and container runtime. Kubelet is an agent that runs on each Node in the cluster. It makes sure that containers are running in a Pod. Kube-proxy is a network proxy that runs on each Node in the cluster, implementing part of the Kubernetes orchestration logic and maintains the network policies of Nodes (i.e network traffic open ports, protocols of communication etc). Finally, container runtime is the software that is responsible for running containers. All these components, along with some extra Add-ons services are presented below in Figure 2.2, which displays the Kubernetes components.

Cluster Infrastructure

Implementation of Kubernetes refers to the initialization of a cluster into the desired infrastructure. The cluster consists of one or more Nodes with predefined or upon demand resource allocation of any type. Upon the initialization of the cluster, the requirements of

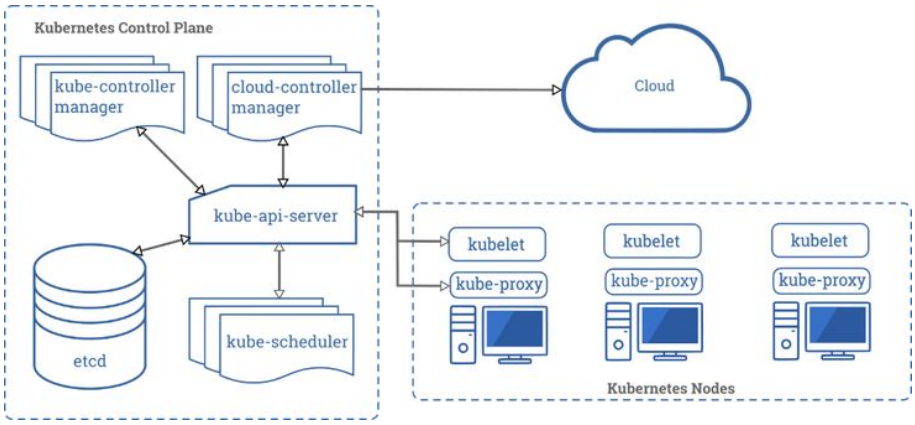

Figure 2.2. *Kubernetes Components [2]*

Nodes in CPU, RAM and Storage, as well as the total number of VMs (Nodes), should be configured properly according to each application's requirements.

Each application consists of Kubernetes manifests that are applied on an existing cluster and include all the essential components of an application to be deployed. These manifests are a set of YAML files, which are in JSON representation, contain key-value pairs describing the respective components of the application. These manifests describe the Deployment, the Services, the Stateful sets and Configurations, known as ConfigMaps, of each application. Deployments provide information about each application's Pod and their replicas. The container image and a label of each service must be also configured in order to for the Pod to be successfully initialized into the cluster. Service files enable the exposure of a specific Pod Deployment of the application into the network, whether it is on localhost or on a Cloud provider, and configure the network policies. Stateful sets are files, which preserve the identification of each created Pod that can not be modified after the rescheduling process of the Pod. It is mainly used for storing volumes and thus is utilized mainly for databases. A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can be connected with ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume. The manifests are applied to the Kubernetes cluster through the API server with a kubectl request command.

Deployment manifests can further specify the Pod requests in storage, RAM and CPU, the Pod Affinities and/or Anti-Affinities with other Pods of the application and many more attributes, which configure the Pod Deployments respectively. Specifically, for Pod Affinity/Anti-Affinity the label of the related Pod and the explicit Node must be described to enable these attributes. Except from the Pod Affinities, Node Affinities can be specified in Pod Deployments, so that Pods will or won't be deployed in specific Nodes of the cluster.

Furthermore, in Deployment and Service manifests, container port must be specified to enable the network communication among the services of the application. Additionally the network rules of the cluster should be configured accordingly to enable traffic communication between these ports. Service manifests specify the type of Service for each associated Pod, which can be configured mainly as Cluster IP, NodePort or LoadBalancer. Cluster IP refers to internal traffic from clients (i.e end-users) to internal IP addresses, which

are accessible only within the cluster's environment. NodePort enables the requests from clients between the cluster's Nodes. Each Pod Service defined as NodePort does not have a separate IP Address to communicate externally and uses the Node's External IP (allows communication from everywhere inside and outside the cluster). Finally, LoadBalancer enables the requests from clients through an Internal or an External IP of the network. Each Service defined as LoadBalancer is associated with a separate External IP from the Node IP. LoadBalancer handles efficiently the load balancing between the application's containers.

Scheduling Process

In Kubernetes, scheduling refers to assuring that Pods are matched to Nodes so that kubelet can run them. A scheduler observes for newly initialized Pods that have not assigned to any of the cluster's Nodes. For every Pod that is discovered, the scheduler becomes responsible for finding the most suitable Node for that Pod to run on [25]. Kubernetes provides the kube-scheduler, which is the default scheduler for Kubernetes Pods and runs as part of the Control Plane. It is designed to be easily extended, modified or customized according to the requirements of each application.

For every newly created Pod or other unscheduled Pods, kube-scheduler selects an optimal Node for them to run on. However, every container in Pods has different resource and scheduling requirements. Therefore, existing Nodes need to be examined according to these specific requirements to be able to host the new Pods. In a cluster, Nodes that meet the scheduling requirements for a Pod are called feasible Nodes. If none of the Nodes are suitable, the Pod remains unscheduled until the scheduler is able to place it.

The Kubernetes Scheduling process takes place into two cycles, the Scheduling and the Binding process of the Pod [3]. The former cycle, which is highly extendable and can be modified as required, attempts to find a feasible Node to host the desired Pod. This process is running serially and can handle only one Pod per scheduling cycle. The steps, also called plugins, of locating a feasible Node for the desired Pod are presented briefly below.

- **Sort:** Sorts the Pods that are going to be scheduled so that the serial scheduling process can initialize.
- **PreFilter:** Pre-process of Pod information. Examines specific requirements of cluster to deploy the Pod. Upon error the process is terminated.
- **Filter:** Exclude Nodes that cannot schedule the Pod according to configuration policies pre-defined on Scheduler. All policies must be fulfilled in order to deploy the Pod. Nodes may be evaluated concurrently.
- **PostFilter:** This plugin is called if there is no available Node to host the desired Pod. Examines the case whether some policies from the previous step are fulfilled and locates a feasible Node.

- **PreScore:** Implements pre-scoring tasks to generate the state of the desired Pod. Upon any failure the process is aborted.
- **Score:** Rank the feasible Nodes according to configuration file of Scheduler, which defines the weights of the priorities. Minimum and maximum score rates are examined for the feasible Nodes scoring.
- **Normalize Score:** Normalize the scoring values before the final ranking of the Nodes to optimize the ranking process. Upon any error the process is terminated.
- **Reserve:** Consists of two methods, the Reserve and Unreserve. The Reserve method is called to reserve adequate cache memory in order for the Pod to be deployed into a Node. The cache is reserved until the Reserve phase is completed. Upon a failure on the Reserve phase or a later phase, the Unreserve method is executed to free the reserved cache.
- **Permit:** Invoked at the end of the Scheduling cycle. It approves, denies or delays the scheduling of the desired Pod into the candidate Node. The wait process is occurred when a Pod is waiting for approval and includes a timeout time, in which the process will be terminated with failure.

The latter cycle, the Binding process, notifies the Kubernetes API server about the scheduling decision of the desired Pod into the specific Node. The Binding Cycle can not be extended further, but it can run concurrently for many Pods, which are selected for scheduling. The main steps of the Binding Cycle are presented briefly below.

- **WaitOnPermit:** A plugin executed when a wait signal is occurred and a wait process is called. It delays the binding process of the desired Pod until an approval or a denial signal occurs.
- **PreBind:** Pre-work for binding the Pod to the Node, like provisioning a network volume to be mount on the Node. Upon failure the binding process is terminated.
- **Bind:** Process to bind the Pod into the Node. It is executed after all the PreBind processes are completed.
- **PostBind:** Used to clean up associated resources for the binding cycle of the Pod.

The Scheduling Process is displayed in figure 2.3. Factors that need to be taken into account for scheduling decisions include individual and collective resource requirements, hardware, software and policy constraints, affinity and anti-affinity relationships on Pods and Nodes, data locality and so on. The filtering and scoring step are relied on the Scheduling Policies that have been prespecified on the Kubernetes Scheduler JSON configuration file. In this file, the policies of the predicates, for the filtering step, and the weights of the priorities, for the scoring step, are specified according to the demands of each Kubernetes Scheduler and by default predicates have predefined policies and priorities equal weights. For the scoring step, if more than one feasible Nodes gather the same amount of score, then kube-scheduler selects one of these Nodes randomly.

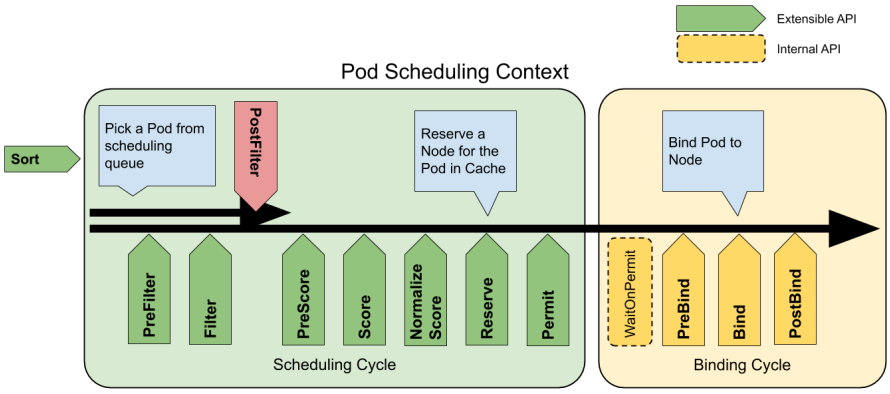

Figure 2.3. *Kubernetes Scheduling Process [3]*

In addition to this scheduling strategy, kube-scheduler can be further extended by specifying the Node and Pod Affinities or Anti-Affinities [26], which can be configured and overwritten in the YAML files by the end-users. With prior knowledge of each application's graph and communication edges between the application's services, Kubernetes Pods can be scheduled in specific Nodes and with associated Pods (Affinity) or vice versa (Anti-Affinity). This affinity can be either soft or hard depending on each application's requirements. With this strategy Kubernetes placement can be further improved and achieve better performance by co-locating services with higher affinity score into the same Node.

2.3.3 Service Mesh and Istio

A service mesh is a dedicated infrastructure layer that can be added to any application and it transparently allows adding capabilities like observability, traffic management and security without adding extra lines of code [4]. In a service mesh, requests are routed between microservices through proxies in their own infrastructure layer. Without a service mesh, each microservice needs to be coded with logic to govern service-to-service communication, which means developers are less focused on business goals [27]. It also means communication failures are harder to diagnose because the logic that governs inter-service communication is hidden within each service. A service mesh achieves better management of applications, especially in distributed environments, like Kubernetes-based systems, as they grow in size and complexity. The service mesh is usually implemented by providing a proxy instance, called a sidecar proxy, for each service instance. Sidecars Proxies handle inter-service communications, monitoring, and security-related concerns. This way, developers can handle only the development, the support, and the maintenance of the application's structure and code of the microservices. Sidecars Proxies monitor the application and each sidecar collects metrics for the resources of the application for benchmarking and testing purposes.

Istio is an open source service mesh that is configured and installed onto existing distributed applications to monitor them [4]. Istio exploits the benefits of Service Mesh and injects istio sidecar proxies into the application associated with each Pod in order to monitor the traffic, resources and add an extra layer of protection for users authentication. Istio is designed for extensibility and can handle a diverse range of deployment needs. Istio's

Control Plane, which will be presented next, runs on Kubernetes, enabling the connection of the deployed application to the mesh, extending the mesh to connect various clusters, or even connecting VMs or other endpoints running outside of Kubernetes. Figure 2.4 shows how the Istio service mesh works.

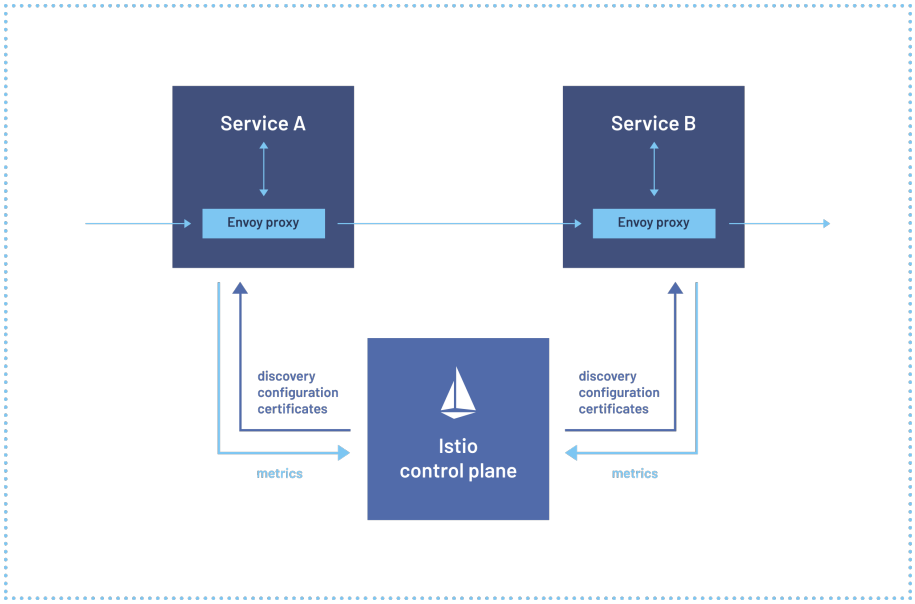

Figure 2.4. *Istio Architecture [4]*

Istio consists of two components, the Data Plane and the Control Plane [4]. The former is responsible for the communication between application's microservices and is achieved by injecting the Istio Envoy Proxies on each application's service running in VMs, enabling to identify the network traffic of the application, the protocols of communication and the type of data exchanged between the source and destination communicating services. The latter monitors the network traffic and dynamically programs the envoy proxies, according to the configuration policies specified by the developers, allowing or denying communication between specific services or specific types of requests. Furthermore, Control Plane secures service-to-service communication with Transport Layer Security (TLS) by specifying the policies of network external and internal inbound or outbound traffic in the respective configuration files.

Istio maintains an internal service registry containing the set of services and their corresponding service endpoints running on the service mesh [4]. This internal service stores all the discovery configuration certificates of the existing sidecar proxies. Istio utilizes the service registry to generate Envoy configuration and the Envoy proxies can then direct traffic to the relevant services. Most microservice-based applications have multiple instances of each service workload to handle service traffic, sometimes referred to as a load balancing pool. By default, the Envoy proxies distribute traffic across each service's load balancing pool using a round-robin model, where requests are sent to each pool member in turn, returning to the top of the pool once each service instance has received a request.

2.3.4 Metric Tools and Agents

In this section the various metric tools and agents, that will be utilized in the implementation phase, will be mentioned and described. These are the essential tools that will be installed into the Kubernetes clusters and cooperate with Istio Service Mesh to gather the desired Pod and Node metrics in order to implement the proposed service placement strategies.

Prometheus

Prometheus is an open-source system monitoring and alerting service, which collects and stores its metrics as time series data, which means that metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels [28]. It records real-time metrics in a time series database (allowing for high dimensionality) built using a HTTP pull model, with flexible queries and real-time alerting.

Prometheus has various components that work together to track and report on system health, behavior, and performance. The primary method of data collection is extracting metrics from instrumented applications and services, which expose metrics in a plain text format via HTTP endpoints. The Prometheus architecture facilitates the discovery of services and enables gathering data from these services. Prometheus stores the extracted data, which can be analyzed with the Prometheus Query Language (PromQL).

Prometheus extracts metrics from application's services by applying PromQL queries to that service. It stores all gathered samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts. Grafana or other API consumers can be used to visualize the collected data. It works well for recording any purely numeric time series. It fits both machine-centric monitoring as well as monitoring of highly dynamic service-oriented architectures. In a world of microservices, its support for multi-dimensional data collection and querying is a particular strength. It is designed for reliability and allows to quickly diagnose application problems. Each Prometheus server is standalone, not depending on network storage or other remote services.

Prometheus provides a functional query language called PromQL (Prometheus Query Language) that allows the user to select and aggregate time series data in real time. The result of an expression can either be shown as a graph, viewed as tabular data in Prometheus's expression browser, or consumed by external systems via the HTTP API of Prometheus. PromQL uses three data types: scalars, range vectors, and instant vectors. It also uses strings, but only as literals. It is also a nested functional language (NFLs), where data appears as nested expressions within larger expressions. The outermost, or overall, expression defines the final value, while nested expressions represent values for arguments and operands. PromQL enables the user to acquire the desirable metrics from the microservices or from the orchestration mechanism of each application by using the implemented functions and operations and specifying time series selectors.

In order to pull the desired metrics from the Cloud and the Kubernetes cluster of the application, it is vital to install some version of exporter. For this purpose, Prometheus Node Exporter is installed and injected into each VM of the cluster so it can pull Node data.

The Prometheus Node exporter is an exporter for physical and virtual machine metrics – hardware and kernel metrics that collects technical information from Linux Nodes. They must be configured to listen on a dedicated port (9100 by default). Because exporters are effectively single-purpose monitoring agents, to collect metrics from other services on the same host requires additional exporters with their own service management/supervision and dedicated network ports. The Node exporter enables measuring various VM resources such as memory, disk and CPU utilization.

Kiali

Kiali is a management console for an Istio-based service mesh. It provides dashboards, observability, and enables operating a service mesh with robust configuration and validation capabilities [5]. It shows the structure of the service mesh by inferring traffic topology and displays the health of the mesh. Kiali provides detailed metrics, powerful validation, Grafana access and strong integration for distributed tracing with Jaeger, which traces the requests from source to destination services. It visualizes the service mesh topology and provides visibility into features like request routing, circuit breakers, request rates, latency and more. Kiali offers insights about the mesh components at different levels, from abstract applications to services and workloads in detail.

The Kiali graph provides a powerful way to visualize the topology of the service mesh. It displays the services' communication and their respective traffic rates and latencies between them, which helps visually identify and troubleshoot problem areas and quickly pinpoint issues. Kiali provides graphs that show a high-level view of service interactions and a low level view of workloads or a logical view of applications. It identifies security issues, request and configuration errors and alerts users visually about the status of application and requests. Along with all these attributes, Kiali graph enables traffic animation for visualization of traffic communication between the microservices in HTTP and TCP protocols.

Kiali is a management console for Istio, and as such, Istio is a requirement. It provides and controls the Service Mesh. Kiali needs to retrieves Istio data and configurations, which are exposed from Prometheus and the Cluster API. Kiali collects Prometheus data for the resources and the Service Mesh of the application and uses the API of the container application platform (Cluster API) in order to collect and resolve Service Mesh configurations. Through the API developers can assemble all the information provided by Kiali service and the Kiali graph. Figure 2.5 demonstrates the Kiali architecture.

Grafana

Grafana is a multi-platform open source analytics and interactive visualization application. When connected to supported data sources, it can visualize the gathered data in charts and graphs. It provides users with tools to turn time-series database (TSDB) data into graphs and visualizations [29]. Grafana supports querying Prometheus and is an essential tool to understand the application metrics produced by Prometheus. Prometheus data are injected into Grafana by end users, who can create complex monitoring dash-

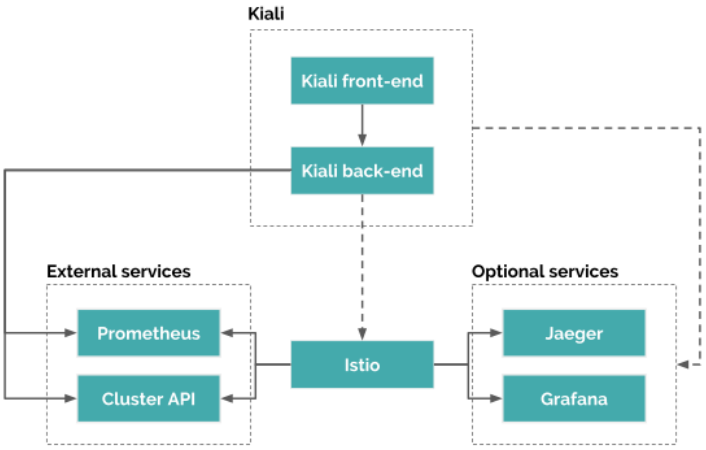

Figure 2.5. *Kiali Architecture [5]*

boards using interactive query builders or import a pre-build dashboard from the Grafana official website. Information about Node and Pod metrics, such as CPU and RAM usage or the availability in storage and resources, can be displayed visually in these dashboards and processed by the users accordingly.

2.3.5 Benchmark Stressing Tool

Apache JMeter

The Apache JMeter application is open source software, designed to load test functional behavior and measure performance [30]. It can be successfully used to test performance both on static and dynamic resources. It can simulate heavy load on a server, group of servers or network to test and analyze the overall performance under different load types and load distributions. It can easily load and test performance of application's microservices and their exchanged requests over HTTP protocol. Apache JMeter offers easy correlation through the ability to extract data to most suitable response formats, like HTML, JSON or XML, is a full multi-threading framework that allows concurrent and simultaneous sampling and it is highly extensible. Furthermore, it allows users to apply distributed requests of any type to an application to produce a more realistic stress testing.

Chapter 3

System Design and Benchmarks

In this chapter we will describe the implementation phase of our service placement strategy, by displaying the cluster's infrastructure and the various benchmarks that will be injected into the Kubernetes cluster. First and foremost, the cluster's architecture and the configuration of Istio Service Mesh in the cluster will be presented by analyzing its separate components and their associations in the cluster. Then, the performance metrics, which will be utilized to measure the traffic rates of the application's microservices, will be analyzed and their respective formulas will be displayed. As we attempt to solve the SP problem with graph partitioning algorithms, we will present the vital modifications on the benchmark algorithms to fit our cluster's and applications requirements and we will categorize them according to their purpose on the implemented strategies. The benchmark use-cases, in which we will implement the proposed service placement strategies, will be presented and, finally, the estimated cost of a Kubernetes cluster running on a Cloud infrastructure will be formulated to target the factors that impact the cluster's charges.

3.1 System Architecture

Each application running on Kubernetes must be initially configured properly to enable the communication of all its microservices. For our use-case applications, we describe each microsevice as a Deployment in the Kubernetes manifests and we utilize one Pod per each microservice (i.e one replica per each Pod) as we do not want to create a larger scale of each application. For each Deployment file, which is associated with a microservice's Pod, a respective Service file, linked to a specific port, must be configured to enable network communication with the other microservices and external services. For every Service described as NodePort or LoadBalancer type, the cluster's networks rules in the Cloud are configured to enable the network traffic from these ports. The type of Service is selected according to the requirements of each application. For our system, every LoadBalancing Service is converted into NodePort type in order to reduce the cluster's run-time costs, as we do not want to apply load balancing among the application's microservices. Horizontal and Vertical auto-scaling, which are extensions of Kubernetes for managing the Node and Pods size and resource allocation, are disabled, as we want to specify the initial number of Nodes that will be used to host each application and the respective resources for each Node.

Each Pod is associated with a respective Service component for enabling network communication. The implementation of the Istio Service Mesh into the application requires the configuration of Pods by injecting the Istio Envoy Proxies into every application's Pod. Upon every internal or external network communication of Pods, the traffic is re-directed through the Envoy proxies injected into the Pods. Envoys are responsible to receive any new request and forward it to the application's microservice through its respective Kubernetes Service component and vice versa. The Pod architecture, with the Istio Envoy injected in it, is presented in 3.1.

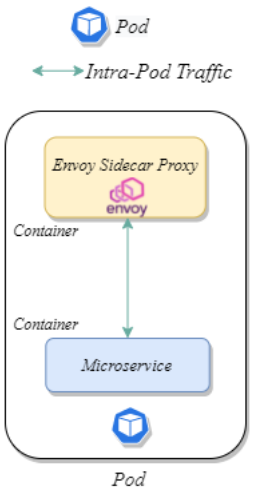

Figure 3.1. *Pod's Architecture*

Each application's Node can host a finite number of Pods, including some of the Istio Services and Kubernetes Components, depending on the available resources size of the Node and the requested resources size of the Pod. Istio is responsible for monitoring the application's network traffic and exchanged messages through the Istio's Data Plane. Every Pod's Envoy is communicating with all the other Pods inside the Node forming the Istio Mesh Traffic. Every Pod that is created into the application is injected with the Istio Envoy. Istio's Control Plane is responsible for monitoring the status of existing Envoys and for injecting Envoy Proxies to the newly-created Pods. The Envoy Proxy will send a configuration certificate to the Istio's Control Plane, so it can join the existing Mesh Traffic. Istio services are installed in the cluster as Pods inside the cluster's Nodes. Node Exporters are installed in every cluster's Node and they are responsible to monitor the Node's resources. Prometheus Server extracts these metrics by requesting the data from the Node Exporters. In this way, Grafana can request the data for every Node and Pod by applying the respective PromQL queries from the Prometheus Server and visualize them in a UI. In the same manner, Kiali collects these data from the Prometheus Server to create the application's graph. The Node's architecture and the communication among the Istio Services and the application's Pods is displayed in 3.2.

The installed Istio Services are randomly placed inside the cluster's Nodes according to the decision of the Kubernetes Scheduler. The Istio Services are not injected with Envoys Proxies, as they are not part of the application's Data Plane and extract only metrics

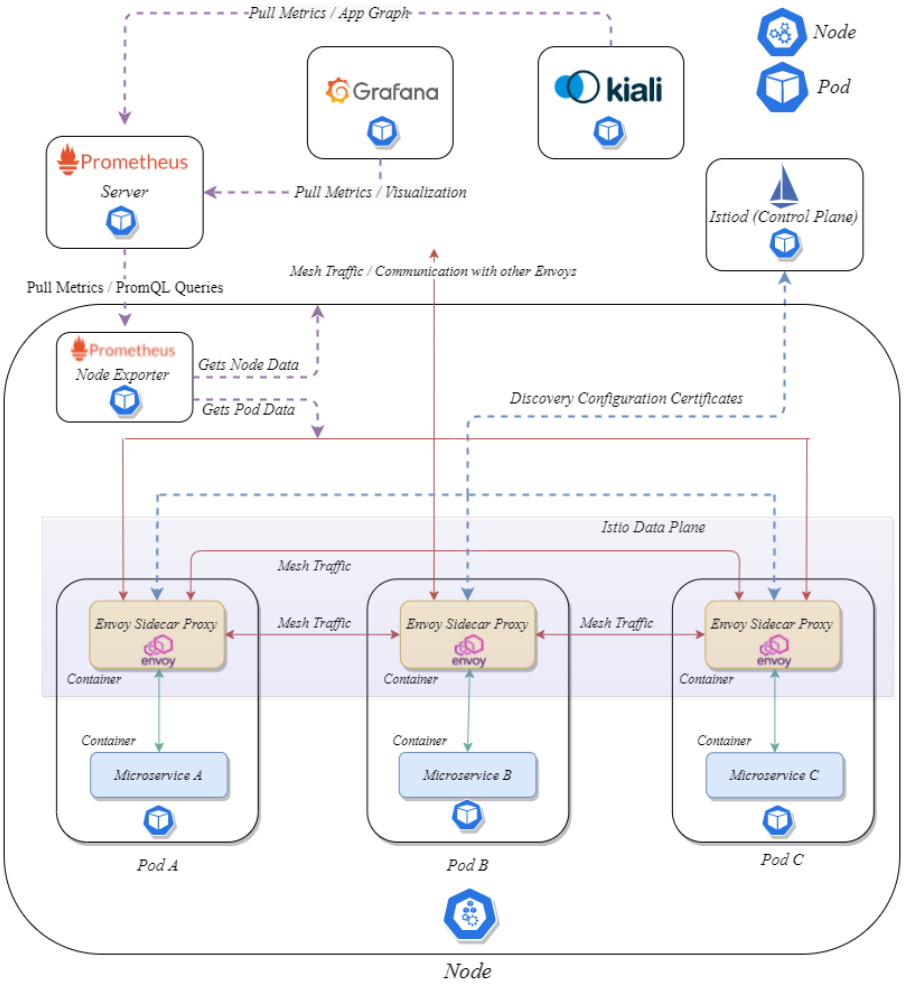

Figure 3.2. *Node's Architecture*

from the cluster's Pods and Nodes. Every Kubernetes cluster consists of a finite number of Nodes, according to each application's requirements. Every Node inside the cluster is communicating through the Istio's Data Plane via the Pods' Envoy Proxies. Kubernetes cluster is in control of creating and managing the Node and Pods. Any data extracted from external sources and services regarding the Nodes and Pods information is requested from the Kubernetes cluster. In 3.3, the cluster's architecture is presented in a Cloud infrastructure, like GCP. GCP provides users with two engines, the Kubernetes Engine and the Compute Engine, to manage the Kubernetes clusters and the cluster's Nodes (or VMs) respectively needed to host each application.

3.2 Microservices Performance Metrics

In this section, the methods for extracting the performance metrics of the microservices traffic rates will be presented and analyzed. These metrics will be collected from the respective metric tools and agents and will be utilized to implement the proposed service placement strategies. Two performance metrics are presented below, the Requests per Second (RPS) metric and the Weighted Bidirectional Affinity (WBA) metric. For the

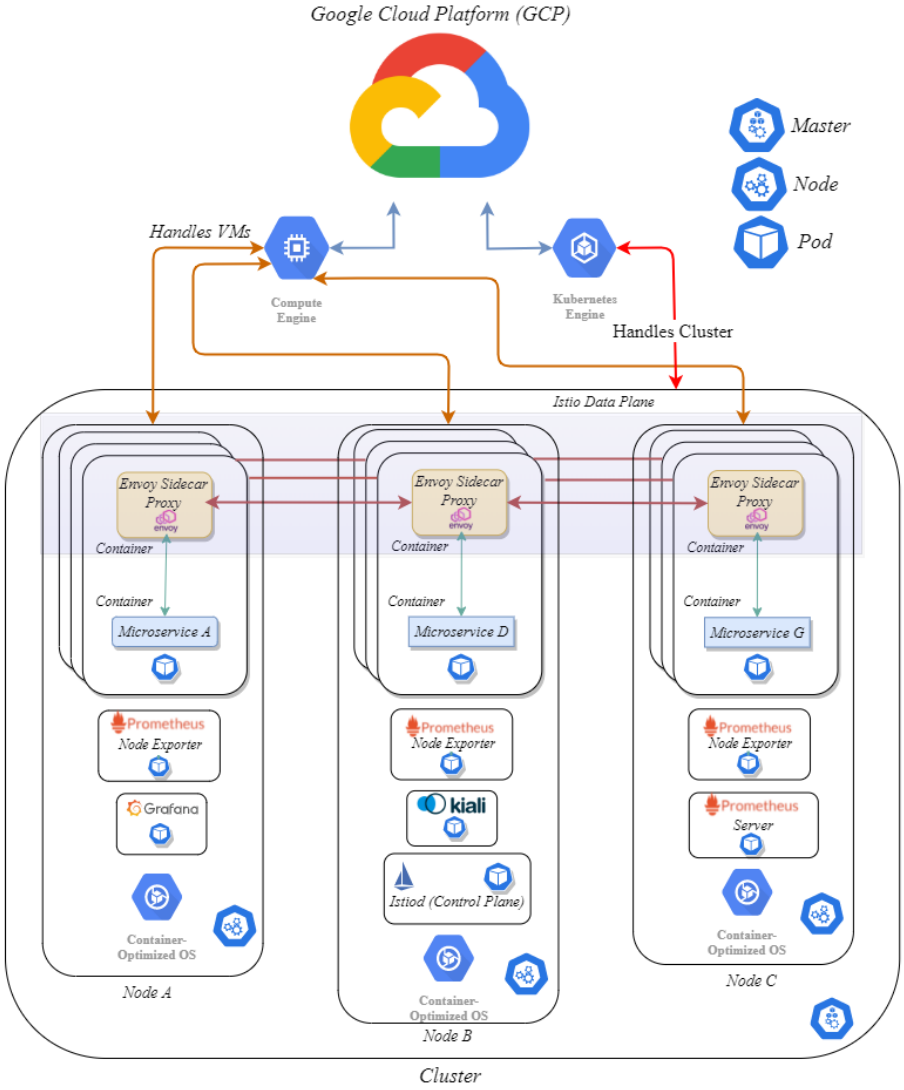

Figure 3.3. *Cluster's Architecture in GCP*

purpose of this Thesis, the performance metrics will also be referred as affinity metrics.

3.2.1 Requests per Second (RPS)

Requests per Second (RPS) is a performance measure of the amount of search traffic an information-retrieval system, such as a search engine or a database, receives in one second [31]. It's purpose is to calculate the traffic rates between two services and in our case we exploit this measure to define the communication relationships between the microservices of each implemented application. Additionally, it is a performance measure that can be easily acquired through the Kiali without any further modifications and calculations.

RPS performance metric is acquired from the Kiali Graph, which is collected through the Kiali API. The JSON file of the Kiali Graph contains information about the mean value of requests per second, which are applied on a specific time range of the application lifespan. It does not provide any further information about the size or count of messages, only the requested traffic rate from one service to another. Though it does not provide in-

formation about those additional metrics, it can provide useful information about the scale of the traffic rates and the microservices' affinities. The formula of RPS is presented in Figure 3.1. For every service i applying requests to another service j, RPS is calculated as the mean value of the total number of requests executed in one second for a specific time range.

$$RPS_{i \to j} = \frac{\sum_{t=1}^{T_{sec}} R_t(i \to j)}{T_{sec}} \tag{3.1}$$

Where,

- i is the source service
- j is the destination service
- t is the time (second) of measuring the requests
- T_{sec} is the total seconds of measurement
- R_t is the total requests applied in second t

In section 3.1, we mentioned that every Pod (or workload) is associated with a Service component to enable the network communication. Every other Pod, which applies requests to that specific workload, has to communicate initially with the Service of that specific Pod. The RPS rate of a workload (Pod) j is calculated as the sum of all the RPS metrics received at that Pod's service from every other service i in the application. The formula of the total RPS metric measured at a Pod from application's services that have an affinity edge is presented in Figure 3.2.

$$A_{j_{workload}} = \sum_{i} A(i \to j_{service}) \tag{3.2}$$

Where,

- $A_{j_{workload}}$ is the RPS metric measured on workload j
- $A(i \to j_{service})$ is the RPS metric from every workload i to the $j_{service}$

Further information about the collection of the RPS traffic rates from the Kiali graph will be presented in Appendix A.

3.2.2 Weighted Bidirectional Affinity (WBA)

Weighted Bidirectional Affinity is a microservice performance metric that is introduced in [15]. This metric exploits the size of the exchanged messages in bytes between the microservices and the total number of these messages to calculate the affinity metric between two microservices. The formula of the WBA is presented below.

$$A_{a,b} = w \cdot \frac{m_{a,b}}{m} + (1 - w) \cdot \frac{d_{a,b}}{d} \tag{3.3}$$

Where,

- $A_{a,b}$ 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 $\left\{w \in \mathbb{R} \mid 0 \leq w \leq 1\right\}$, used to define the significance of each affinity variable (size or count of messages)

The weight factor is selected according to the importance of the variables, which are calculated to get the total affinity metric between two microservices. For this Thesis, the importance for the variables of the size of messages and the count of messages will be equal, by assigning w with the value of 0.5. The reason behind this selection is because there is no strong preference between these two variables, nor is it assumed that giving different weights will produce a better solution for the service placement problem.

3.3 Benchmark Algorithms

In this section the modification and further analysis on the algorithms presented on the previous chapter will be applied. We categorize the algorithms into 2 major groups, the Clustering algorithms and the Adaptive Placement algorithms. It should be mentioned that these algorithms require prior knowledge of the cluster's data information and resource allocation. We utilize these algorithms and combine them to form the service placement strategies as will be implemented and will be described in the experiments chapter.

3.3.1 Clustering Algorithms

The Clustering algorithms do not require any prior knowledge about the initial service placement of the microservices in the Kubernetes cluster (e.g. one that is produced by another placement algorithm or by the default Kubernetes scheduler). These algorithms require only information about the service affinities, the service requests in CPU and RAM and the service list of each application.

The implemented algorithms are the Binary Partition, the K-Partition and the Bisecting K-Means. They require the construction of application's graph as a pre-processing step for each placement strategy. Graph G is constructed with the nodes representing the application's microservices and the edges representing the weights of the communication traffic rates (or the service affinity rates). Given an application's set of services and their affinities, graph G is constructed as presented in Algorithm 3.1

ALGORITHM 3.1: *Graph Construction [32]*

Input: Service List (S), Service affinities (A)
Output: Graph of application (G)
Initialize G = ()
for every source service u in A **do**
  for every destination service v in A(u) **do**
    if S contains (u,v) and G(u → v) does not exist **then**
      Create G(u → v)
    end if
  end for
end for
Return G

In the next paragraphs we will present the modified algorithms for the Contraction process of BP and KP algorithms and the modified Bisecting K-Means to match our directed graph-based applications. BP and KP algorithms will be utilized as presented in Algorithms 2.3 and 2.4.

Contraction Algorithm

Before analyzing the clustering algorithms, the vital modifications on the Karger's Contraction algorithm should be discussed, which is utilized by the Binary Partition and the K-Partition algorithms. The problem with contraction algorithm was that it is designed and presented in [16] so as to find a minimum k-Cut in an undirected graph. However, in our Microservice-based Applications the graph is a directed graph and thus the algorithm should be configured properly to meet these criteria.

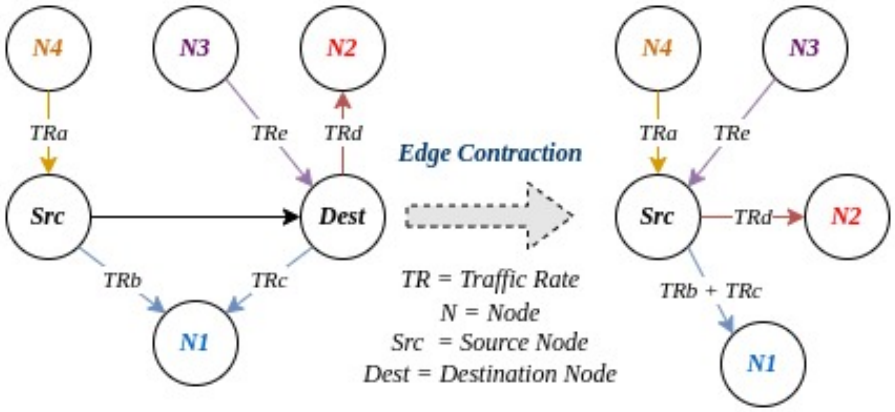

Figure 3.4. *Edge Contraction Process*

Algorithm 3.2 diplays the pseudocode for the modified contraction algorithm to find the minimum k-Cut in a DAG.

Given randomly a source service from G, modified contraction algorithm attempts to contract a destination node and rearrange the incoming and outgoing affinities from the destination service to the selected source service. The basic idea of the Algorithm 3.2 is to

ALGORITHM 3.2: *Contraction algorithm for Directed Graph*

Input: Application Graph (G) with Nodes (V) and Edges (E), Service affinities (A), k-Cut value (K)
Output: minimum k-Cut of the Graph

while V > K and E > K - 1 **do**
  Select random source Node (V_{src}) and random destination Node (V_{dest} from G
  if A(V_{dest}) is not empty **then**
    for every node Vi except (V_{src})) and (V_{dest}) with edge E_i in A(V_{dest}) **do**
      if A(V_{src}) contains E_i **then**
        $A(V_{src\to i}) = A(V_{src\to i}) + A(V_{dest\to i})$
      else
        $A(V_{src\to i}) = A(V_{dest\to i})$
      end if
    end for
    Remove V_{dest} from G
  end if
  for every node (V_i) in G except V_{dest} and V_{src} **do**
    if A(V_i) contains V_{dest} as destination node **then**
      if A(V_{src}) contains V_i as destination node **then**
        $A(V_{i\to src}) = A(V_{i\to src}) + A(V_{i\to dest})$
      else
        $A(V_{i\to src}) = A(V_{i\to dest})$
      end if
    end if
  end for
  Remove affinity A($V_{src\to dest}$) and edge E_i
end while
Return G

randomly choose an edge from the graph with probability proportional to the weight of the edge and merge the nodes assigned to this edge into one node (called edge contraction). In order to find a minimum cut, the algorithm iteratively contracts the edges which are randomly chosen until the required number of nodes remain. The algorithm iteratively selects a random edge in G and removes it until the graph contains only the desired K Nodes or K-1 edges. To remove an edge, we examine whether the selected destination node has affinity edges with other Nodes in G. If applicable, all affinities are redirected from the destination node to the selected source node, as it is displayed in Figure 3.4. The algorithm returns an updated graph G with only K nodes and $K-1$ edges. The time complexity of the algorithm is O($2N(N-K)$) or O(N^2) for a small K value compared to the size of N.

Bisecting K-Means

Bisecting K-Means (BKM) algorithm produces a finite group of services with high affinity traffic rates given a set of microservices. The specific number of K clusters or groups of services is selected by the end-user. As mentioned in the previous chapter, BKM relies on the K-Means algorithm and the Sum of Squared Errors (SSE), which is

calculated from every point in the cluster with the cluster's centroid point. These points are represented in Cartesian form and the calculation of SSE can be found from various formulas, according to the coordinates of these points. However, in a Cloud environment, microservices are just points which can't be represented in 2D form and thus the SSE can't be calculated accurately. To face this problem, we introduce a modified Bisecting K-Means algorithm, which is relied basically on the affinities edges of the microservices. Algorithm 3.3 shows the pseudocode which is used to implement the Bisecting K-Means for a Microservice Architecture.

ALGORITHM 3.3: *Bisecting K-Means for Microservices Architecture*

Input: Service-based application (S), Initial partition (P), Number K of desired clusters, Service affinities (A)
Output: Partition of Services in K Clusters
$P \leftarrow \{S\}$
while size$\{P\} < K$ **do**
  Select a cluster from P with the least sum of service affinities rates in total, C_i
  $P \leftarrow P - \{C_i\}$
  Pick and remove two microservices - centroids, ms_x and ms_y, from C_i with no or the least affinity rate between them
  $C_i \leftarrow C_i$ - $\{ms_x, ms_y\}$
  $C_x \leftarrow ms_x$
  $C_y \leftarrow ms_y$
  $P \leftarrow P \cup \{C_x, C_y\}$
  for every microservice (ms_i) in C_i **do**
    if A$(ms_i \rightarrow ms_x) >$ A$(ms_i \rightarrow ms_y)$ **then**
      $C_x \leftarrow C_x \cup \{ms_i\}$
    else if A$(ms_i \rightarrow ms_x) <$ A$(ms_i \rightarrow ms_y)$ **then**
      $C_y \leftarrow C_y \cup \{ms_i\}$
    else
      Select and place ms_i randomly among C_x and C_y
    end if
  end for
end while
Return P

The solution to match the Microservice Architecture restrictions for applying the BKM algorithm successfully is to estimate the error by the affinity metric of the microservices. Specifically, microservices with little or no affinity metric should be separated and produce two sub-clusters with higher affinity score. The non existence of a communication edge between these microservices is preferred, algorithmic logic will select them as two clusters and will not continue processing the next affinity edges. The remaining microservices, in the processed cluster, are assigned according to the affinity metric with the selected sub-clusters centroids or randomly if there is no affinity edge or strong preference between them. The fact that the centroid choice is made randomly among the microservices with the same affinity metric does not always lead to an optimal solution. Moreover, microservice assignment on the available cluster centroids is made only by the affinity score with the

centroid of the cluster and not with the other affinities pairs with the microservices inside the initial cluster. However, this version of BKM algorithm reduces the time complexity and produces most times a sub-optimal result, which can be further enhanced with the Heuristic Packing.

3.3.2 Adaptive Placement Algorithms

Clustering algorithms can effectively partition an application according to the microservice affinities traffic rates. However, it can not be guaranteed that these partitions can be successfully hosted into the application's Nodes (or VMs). In addition to the monetary cost optimization goal, we must ensure that these partitions would allocate the least amount of resources inside the cluster. For these reasons, We group the Heuristic Packing and the Heuristic First-Fit as post-processing algorithms, which require prior knowledge of the initial service placement or application's partitions and the Node resources to produce an optimized service placement result with heuristic methods. Heuristic Packing is implemented in conjunction with the Clustering algorithms, as it requires the partitioning result produced by them. However, Heuristic First-Fit is processed and executed independently from the other clustering algorithms in one step execution, without requiring the execution of Heuristic Packing. Finally, we will present a Top-Level algorithm to combine some of the clustering and the placement algorithms, called Placement Finding presented in [16].

Heuristic Packing

Heuristic Packing (HP) is used in conjunction with the Binary Partition and K-Partition and efficiently places each partition into the cluster nodes ensuring that the least amount of resources will be allocated. The algorithm requires prior knowledge of the Node resources and the Pod requested resources in order to locate a cost-optimized placement solution.

HP algorithm is mainly consistent with the Algorithm 2.6, presented in the previous chapter. The input to the algorithm is the application's partitions produced by BP, KP or BKM algorithms, the Node and Pod requested resources, the list of services and the service affinities list. For each part of the partition, the total CPU and RAM requests are calculated according to the services placed into that specific part and the algorithm examines and processes all the hosts that contain enough resources to host that specific part. For every Node that can efficiently host the processed part, the traffic rates between services in the processed partition and services that are already located into the specific Node are calculated. Prioritizing the traffic rates, then the CPU most-loaded resources and finally the RAM most-loaded resource requirements, we can determine the most suitable Node to host that specific part. The procedure is repeated for every application's partitions until all partitions are successfully hosted inside the cluster's nodes or there is at least one partition that can not be hosted into the available Nodes.

Prior to applying the HP algorithm after the clustering process, we calculate the available and allocated Node resources properly so as to locate a correct microservice placement solution. Upon collecting the Node data from the Metrics service and Agents, we take into consideration the initial service placement of each application. To effectively apply the HP

algorithm, we process the collected data so as to not include the requested data for each Pod. We must ensure that each Node contains only the metrics and Kubernetes services to find a placement solution.

Heuristic First-Fit

Heuristic First Fit (HFF) algorithm is a single-step processing algorithm, which is given as input the application's initial service placement and configures this placement by moving microservices with high affinity traffic rates in the same host machine. Additionally, the application's microservices affinities must be sorted in descending order. In this way microservices with higher affinity metric (or graph weight) are processed first and the produced service placement becomes as optimal as possible. Upon each iteration of the algorithm and for each processed affinity edge, we store the source and destination nodes, their initial hosting VMs and their resource requests in separate values to make easier the execution process of the algorithm. If two services belong at the same host machine, the algorithm stops the current iteration and services are marked as moved in order to remain at the same host machine until the termination of the algorithmic execution. On the other hand, if microservices belong at different host machines then the algorithm examines if the destination service can move to the source host or vice versa. Either way, available and allocated resources of the host machines are re-calculated and configured accordingly for the next iteration and microservice communication edge. The algorithmic logic remains the same as the original algorithm, presented in Algorithm 2.2.

Heuristic First-Fit will not attempt to perform some kind of clustering to the available services, instead it will apply a First-Fit variant packing, as mentioned previously, and will process and modify the current service placement produced by the default Kubernetes Scheduler. The algorithm is executed only once and the fact that microservices previously checked and marked as moved, and therefore can't be moved for the next iterations, produces a sub-optimal placement solution most of the executed times. Furthermore, it can't be guaranteed that a series of iterations of the algorithm for the current service placement will produce an optimal service placement. It can be concluded that the algorithm won't converge for a finite number of iterations and in some cases it can theoretically produce a non-optimal result with higher inter-machine traffic (Egress traffic) between the host machines of the cluster.

Placement Finding Process

Binary Partition and K-Partitions algorithms, in conjunction with the Heuristic Packing algorithm as a post-processing step, can not guarantee that a placement solution will be located upon each execution. In order to partition the application and attempt to place the produced partitions into the available host machines, threshold α is required for these placement strategies. However, giving an appropriate deterministic threshold α is difficult, as it cannot be guaranteed that the algorithm can find a placement solution through the randomized partition and the heuristic packing under a certain threshold α. Intuitively, the higher threshold α results in less application's partitions, which leads to less traffic

rate between different parts. Thus, Algorithm 3.4 presented in [16], the Placement Finding algorithm, is introduced in order to find the best choice of threshold α by enumerating from the larger value of α to smaller with a predetermined step Δ. For each value of threshold α, Binary Partition or K-Partition algorithms are executed in order to partition the application services. Then, the produced partition is processed by the Heuristic Packing algorithm for a placement solution to be found. If Heuristic Packing can not find an application placement solution, then threshold is decreased by Δ and the algorithm's steps are repeated until a placement solution is found.

ALGORITHM 3.4: *Placement Finding Process [16]*

Input: Service-based application S, vectors of available resources on each machine V = (V_1, V_2...V_M)
Output: a placement solution X
X $\leftarrow$ [$x_{ij} = 0$] for every part and host machine
$\alpha \leftarrow 1.0$
$\Delta \leftarrow 0.1$
while $\alpha \geq 0.0$ **do**
    P $\leftarrow$ Binary Partition(S,α) or K-Partition(S,α)
    $X^{'} \leftarrow$ Heuristic Packing(P, V)
    if $X^{'} \neq$ null **then**
        Calculate X according to $X^{'}$ and P
        Return X
    end if
    $\alpha \leftarrow \alpha$ - Δ
end while
Return null

3.4 Benchmark Applications

To implement and apply the proposed service placement strategies we utilize two benchmark applications, iXen and Google OnlineBoutique eShop. These applications are initialized into a separate Kubernetes clusters in a Homogeneous environment with the same resource capacities and infrastructure.

3.4.1 iXen

iXen [33] [34] is a prototype software architecture for an IoT scenario based on Service Oriented Architecture (SOA) principles. For the purposes of this Thesis, it was converted from a bare metal deployed SOA architecture to a microservice-based architecture. As a result, each microservice is independent from the others, becomes portable, has its own data storage, can be scaled independently and be orchestrated by container orchestrators such as Kubernetes. For each microservice we can monitor its traffic, define the minimum demands on resources and declare the infrastructure it will be deployed on.

iXen follows a 3-tier architecture model. At the first layer the Infrastructure Owners – System Administrators have the right to install and connect to physical devices, such

as sensors or actuators, of the same or different type that may be located in different geographic areas. At the second layer, application developers can create subscriptions to devices in order to exploit their data for building applications. At the third layer, end-users, called application customers, can create subscriptions to applications in order to access them. This 3-tier architecture help to extend the system to all 3 layers by adding more devices of different type and applications of different application fields. The iXen architecture is displayed on Figure 3.5. Each different color on edges depicts a separate type of request that can be applied in the application either internally or externally with a HTTP request.

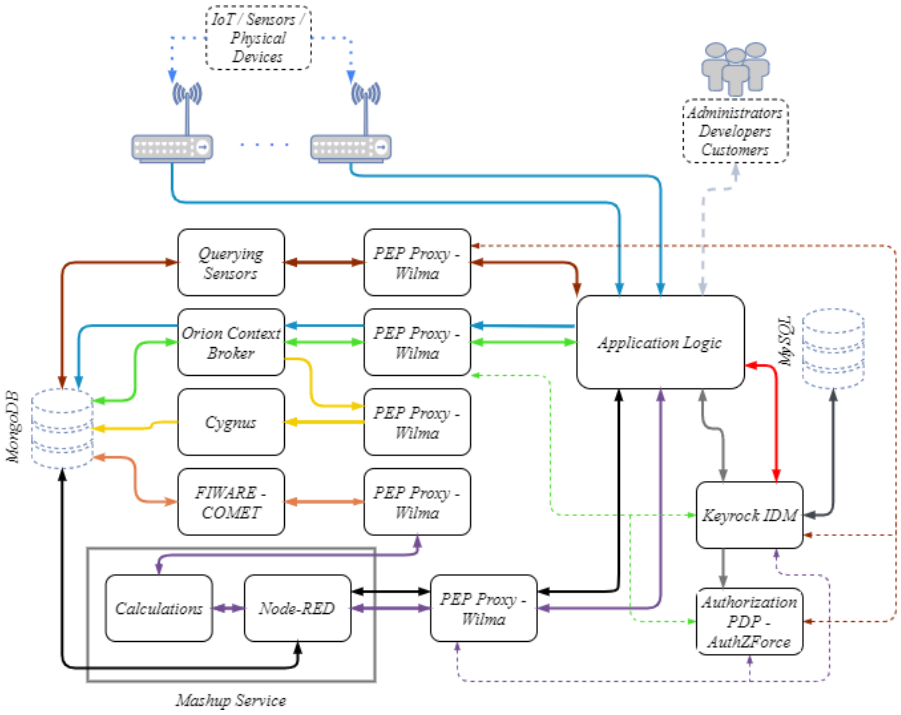

Figure 3.5. *iXen Architecture*

iXen consists of 15 different software units (containers) which are all orchestrated by one of them called application logic. All requests and responses are received from the application logic microservice coming from the Web Application or Devices and are dispatched suitably to the other Services depending on the request type a device or application operation triggers. Application Logic also exposes a Web UI for all User Operations and a Device Interface where all devices send their measures via HTTP protocol. Below the microservices of iXen are described briefly.

- **Keyrock Identity Management:** Service contains all users and their roles. It provides a REST API in order to register users , add policies about their rights to access resources on the application and authorize them via OAuth2.0 protocol. It is connected with a MySQL database in order to store the necessary information.
- **AuthzForce:** Provides a feature in order to apply more advances authorization policies to our applications via an OASIS standard in XACML format via an Attribute-Based Access Control (ABAC) framework.

- **Pep proxy:** Combined with Keyrock IDM and AuthzForce services in order to hide microservice APIs from unauthorized users and services according the policies that have been defined on Keyrock and AuthzForce. Every request is forwarded to the backend protected service by pep proxy, only if the client has the right to access it and conduct the relative operation.

- **Orion Context Broker:** A publish/subscribe mechanism which provides a REST API for storing and retrieving data from a Mongo Database. It stores data about devices with the NGSI format, e.g. as entity types with attributes like the name , the type and the current measure. Furthermore, it gives the ability to users and other services to subscribe on different device in order to be notified for new measure changes and other events referred to them.

- **Querying Sensors:** Service converts a custom query syntax to mongo queries on the Mongo Db where devices are stored as entities with attributes by the Context Brokers. It accepts query for device searching relatively their location, Model type, the type of measure they collect, the unit measure and their firm.

- **Cygnus:** Accepts data streams compliant formatted with the NGSI model and can store them on multiple types of Databases like MongoDB, mySQL, CKAN, DynamoDB. It can store data as RAW or aggregate them without being aware of the database which is used at the backend.

- **Comet:** Manages historic data as time series, which are produced by the attribute changes at the entities stored to the Orion Context Broker at a NGSI representation. Comet is used for data retrieval and exposes a set of different aggregated statistical information.

- **Mashup service:** Contains the Node Red service which creates multiple applications on the system combining data from the devices. There is a calculation service in the Mashup Service which contains all the algorithms for retrieving data from COMET service.

3.4.2 Google's OnlineBoutique eShop

Online Boutique is a cloud-native microservices demo application as described in the official GitHub page of the application [6] implemented by Google Cloud Platform (GCP) for benchmarking and demonstrating purposes of a microservice-based application in the Google Cloud. The application is a web-based e-commerce application, where users can browse items, add them to shopping cart and finally purchase them.

Online Boutique consists of 12 microservices communicating with Remote Procedure Calls (RPC) via gRPC and HTTP (HyperText Transfer Protocol) and users can access the frontend of the application via HTTP. Google exploits the application to demonstrate use of technologies like Kubernetes/GKE, Istio, Stackdriver, gRPC protocol and OpenCensus, but for the current thesis Kubernetes on GKE and Istio services will be applied to orches-

trate the microservices and gather the essential cluster metrics respectively to efficiently apply our proposed microservice placement strategies.

The gRPC protocol is applied instead of HTTP because it can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication [35]. It uses protocol buffers to transfer data among microservices, enables bi-directional streaming, can be implemented in a variety of programming languages and platforms and can be highly scalable as presented in the official website.

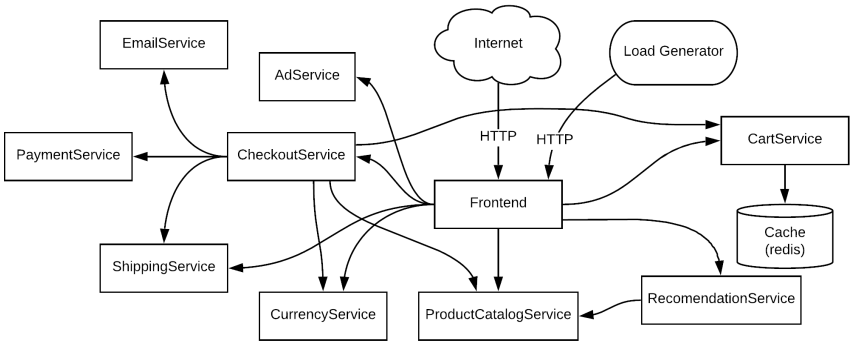

Figure 3.6. *Google Online Boutique Architecture [6]*

Figure 3.6 demonstrates the architecture of Online Boutique application and the communication edges between the microservices. All microservice logic is utilized to communicate via gRPC protocol and nearly all microservices are written in different programming languages for demonstrating purposes only. The application's various microservices are presented concisely below, along with the programming language of the implementation for each service and a short description about their usage in the current application.

It should be mentioned that this application is suitable for the implementation of the presented algorithms due to the fact that it is already tested, the main reason behind its creation was for benchmarking reasons and demonstrating purposes, contains no errors for initializing it into a Kubernetes cluster and the installation of this application into the Cloud infrastructure is described thoroughly in the documentation of the application on GitHub.

The application's microservices, as described in [6] are:

- **Frontend:** Written in Go. Exposes an HTTP server to serve the website. Does not require signup/login and generates session IDs for all users automatically.
- **Cart Service:** Written in C#. Stores the items in the user's shopping cart in Redis and retrieves it.
- **Product Catalog Service:** Written in Go. Provides the list of products from a JSON file and ability to search products and get individual products.
- **Currency Service:** Written in NodeJS. Converts one money amount to another currency. Uses real values fetched from the European Central Bank. It's the highest

QPS service.

- **Payment Service:** Written in NodeJS. Charges the given credit card info (mock) with the given amount and returns a transaction ID.
- **Shipping Service:** Written in Go. Gives shipping cost estimates based on the shopping cart. Ships items to the given address (mock).
- **Email Service:** Written in Python. Sends users an order confirmation email (mock).
- **Checkout Service:** Written in Go. Retrieves the user's cart, prepares orders and orchestrates the payment, shipping and the email notification.
- **Recommendation Service:** Written in Python. Recommends other products based on what is stored in the cart.
- **Advertisement Service:** Written in Java. Provides text ads based on given context words.
- **Load Generator:** Written in Python/Locust. Continuously sends requests imitating realistic user shopping flows to the frontend.

Figure 3.6 displays all the available application's microservices. However, there is an additional microservice that has direct communication with Cart Service over Transmission Communication Protocol (TCP) and stores the products, that the client has added to the purchase cart, remaining there until the order is submitted or deleted. This service, called Redis-Cart, is a storage microservice and it is installed in the existing application's Kubernetes cluster as a separate Pod, as it was implemented from the developers of the application.

Chapter 4

Experimental Results

In this chapter we will mainly present the experimental results upon executing the proposed service placement strategies in Kubernetes environment. The utilized Cloud infrastructure will be analyzed, along with the characteristics of the Kubernetes cluster. Next, the application stress testing modules, the type of requests and the distribution of these requests will be analyzed and presented for both applications, iXen and OnlineBoutique eShop. Before displaying the results for each placement strategy, application and affinity metric, we will specify the cost function calculated from the used Cloud provider cluster fees. Finally, the results of the implemented service placement algorithms will be presented in bar graphs for all the Clustering and Heuristic algorithms.

4.1 Service Placement Strategies

The benchmark algorithms presented in 3.3 categorize the proposed algorithms into two main categories, the partitioning of an application's graph into groups of services and the post-processing step for placing these partitions into the available host machines. As we attempt to minimize the partitions of the benchmark applications, minimize the traffic rates across different VMs, increase the intra-machine affinity and decrease the inter-machine affinities, we must combine some of these algorithms to produce the service placement strategies that will implemented in the cluster. The implemented service placement strategies are presented below.

- Heuristic First Fit (HFF)
- Binary Partition - Heuristic Packing (BP-HP)
- K-Partition - Heuristic Packing (KP-HP)
- Bisecting K-Means - Heuristic Packing (BKM-HP)

The partitioning algorithms presented in section 3.3.1 can effectively partition an application and create clusters with high intra-partition affinity. However, the produced parts must be placed in the Kubernetes cluster and thus a packing strategy is executed in conjunction with these algorithms. In section 3.3.2, a heuristic packing strategy was presented that can utilize the partitions produced by the clustering algorithms and place them into

the available host machines in order to reduce the size of VMs for hosting each application, increase the intra-machine affinity and reduce the inter-machine traffic.

The reason behind the utilization of HP algorithm as a post-processing step for our graph partitioning algorithms relies mainly on minimizing the monetary cost of the cluster. BP, KP and BKM algorithms can effectively partition an application, but they can not guarantee that each and every of these partitions can be successfully hosted in the cluster's nodes (or VMs). These algorithms are combined with HP to construct a two-step execution placement strategy to locate a cost efficient microservice placement strategy.

HFF placement strategy is based on the algorithm presented in 3.3.2 and does not require the HP algorithm to optimize the placement solution. The algorithm, as mentioned previously, does not attempt to partition the application, only to relocate the microservices for the purpose of reducing the intra-machine traffic and reserve as few hosts as possible for hosting each application. It is considered as a complete placement strategy that can be implemented into our Kubernetes cluster in a single execution step, optimize the service placement and reduce the cluster's cost.

4.2 Infrastructure

For the Kubernetes environment, we initialize a Kubernetes cluster into the Google Cloud Platform (GCP) and specifically in the Google Kubernetes Engine (GKE), which will host the desired cluster. GKE is responsible for monitoring and handling the Kubernetes clusters and provides a wide variety of tools to optimize their performance. Each Node or VM will be monitored and observed from the Compute Engine service of GCP, which communicates with GKE to locate the essential number of VMs with specific resources for efficiently hosting the Kubernetes cluster. In this section, we will present the cluster's structure and design to carry on the experiments.

Cluster Design

For each application, iXen and OnlineBoutique, we initialize a cluster in GKE with the same characteristics and variables. Each cluster is created in europe-west3-b Zonal location type. We disable horizontal and vertical autoscaling for the purposes of the Thesis, so that available VMs and their resource allocation remain the same for the implementation and experimentation of the algorithms, as mentioned in 3.1. We examine the performance of the placement strategies within an homogeneous environment with VMs allocating the same amount of resources.

Furthermore, GKE provides end-users with many optimizing tools for the initiated cluster, such as Load Balancing, Client certificates, Basic authentication and many more, however we are not going to exploit them for this Thesis. Last but not least, we enable Cloud logging and Cloud Monitoring at the existing clusters to access the cluster data and monitor its status and health. Table 4.1 displays the essential characteristics of the clusters for each application.

Table 4.1. *Cluster Characteristics*

Cluster Attributes	Option
Location Type	Zonal
Zone	europe-west3-b
Release Channel	Regular
Cluster Version Type	Stable
Cluster Version	1.20.9-gke.1001
Horizontal Autoscaling	Disabled
Vertical Autoscaling	Disabled
Cloud Logging	Enabled
Cloud Monitoring	Enabled

Each cluster contains a node pool to initiate Nodes (or VMs) for the existing Kubernetes clusters. The node pool creates an Instance Group, in which Virtual Machines are created upon the resize of Node Pool of the existing cluster to the desired number of VMs. This node pool is constructed upon the construction of the cluster and is responsible for communicating with the Compute Engine service of GCP to resize the number of cluster's Nodes. The node pools instantiate machines of e2-standard-2 type in europe-west3-b zone and images of Container-Optimized OS with Docker type. The autoscaling attribute is deactivated, so that we can manage the Node size that are created. The boot disk type is standard type with size of 100GB. The Nodes are not preemptible, which means that we allocate resources or reserve a finite number of VMs upon demand for initializing each application. Table 4.6 displays the main characteristics of the cluster's node pool for each application.

Table 4.2. *Node Pool Characteristics*

Node Pool Attributes	Option
Machine Type	e2-standard-2 (E2-standard)
vCPU	2
RAM	8GB
Zone	europe-west3-b
Image type	Container-Optimized OS with Docker
Autoscaling	Disabled
Preemptible Nodes	No
Boot Disk Type	Standard
Boot Disk Size	100GB

Kubernetes nodes are characterized by their type and the volume of the allocated resources, which is CPU and RAM. Both applications reserve resources, not only for Istio and the various Metric Tools and Agents, but also for the Kubernetes services, like kube-proxy, kube-dns, kubelet, metrics server etc. To implement the service placement strategies, we utilize 4 Node Machines with 2vCPU and 8GB RAM per each Node to host each appli-

cation adequately, according to the Pod and Node restrictions and for testing the cost optimization case, after each successful execution of the proposed strategies. Figure 4.1 displays the cluster infrastructure in GCP.

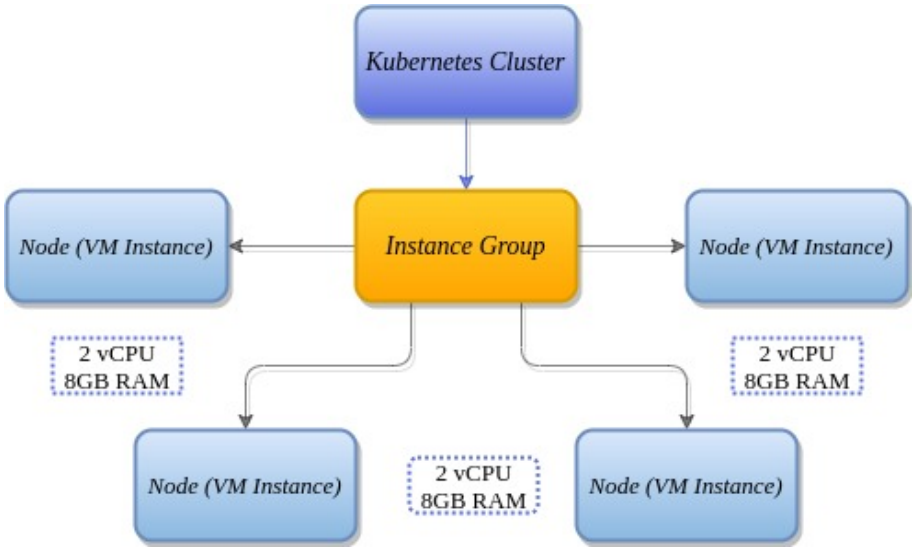

Figure 4.1. *Cluster infrastructure in GCP*

Upon consecutive initializations of the cluster, we have reached to the assumption that both applications require at least 2 VMs to host them efficiently, including Istio and Kubernetes services. The selection of the number of Nodes was made for presenting a realistic use-case with feasible requirements and restricted size of available VMs to test the cost optimization problem. This selection may applies in real-life challenges and projects, in which a DevOps team may have an upper bound to the available VMs for hosting an application with high restrictions on resource allocation. An initialized volume of Nodes greater than 4 for hosting these applications is considered to be a surplus and will only produce higher monetary cost-optimization rates as the size of Nodes grows, which will not provide additional information about the performance of the placement strategies.

4.3 Application Stress Testing

In this section we will present the Stress testing process, which is applied to create traffic flows to each application. This Stress testing can be achieved through various tools, but for this Thesis we will utilize Apache JMeter service [30]. Every Stressing module and technique that will be applied resorts to synthetic workloads and not realistic, as we do not apply a stressing technique according to historical data of requests for each application. Synthetic workloads enable to project various use-cases of each application according to the implemented type and size of requests without increasing the complexity of the stressing process. In the upcoming subsections we will thoroughly present the stressing methods, which are applied for both iXen and OnlineBoutique applications.

4.3.1 iXen Stressing

For iXen Stressing, we must initially create a test plan and configure the application's variables, which are the External IP (i.e endpoint of a Node or VM) of a cluster, the number of threads to be created and the service ports for accessing the application services from the Apache JMeter. For iXen Stressing we attempt to stress the application for a specific time period of 15 minutes. In total, 100 threads are created in the test plan and we apply distributed requests in the selected time range to the application's endpoints. These requests are selected among the available type of requests and some of them require additional information as input parameters. Below, on table 4.3, we present examples of all the types of requests, that will be applied in order to stress the application and create workflows among all microservices.

Table 4.3. *iXen Requests and Stressing test plan*

Request	Type	Requests Distribution
Login into the App	POST	12.5%
Access Device measurements	POST	12.5%
Access Device subscriptions	POST	12.5%
Deploy a new Mashup application	POST	12.5%
Search an existing application	GET	12.5%
Search for subscriptions	GET	12.5%
Make a new customer subscription	POST	12.5%
Access a Mashup application	GET	12.5%

4.3.2 OnlineBoutique Stressing

In OnlineBoutique application we utilize two synthetic workload modules for stressing the application. The first one is the stressing applied from LoadGenerator microservice, which is applied by default into the application. This microservice consists a part of the OnlineBoutique application and its role is to create network traffic into the application by applying random requests. The second stressing method is the additional stressing test plan implemented through Apache JMeter, as implemented in the iXen application. Both stressing methods are described and analyzed below. Both modules apply greater synthetic workloads in size and number of requests into the application than the iXen stressing.

Stressing with LoadGenerator Microservice

The former of these methods of stressing the application is the LoadGenerator microservice, which is initialized upon the launch of the application into the Kubernetes cluster. This microservice generates random HTTP requests towards the application's microservices. The initial Kiali graph (the graph is presented in Appendix - Figure A.2 is produced through the load stressing applied from the LoadGenerator microservice, so the application is constantly under stressing for nearly 3 requests per seconds. LoadGenerator microservice applies randomly generated requests from the available ones. Table 4.4 displays the

available application's requests and their request type.

Table 4.4. *Online Boutique Requests [6]*

Request	Type	Description
Index	GET	Return index page
Set Currency	POST	Change currency
Browse product	GET	Return random product
View Cart	GET	Access the cart page
Add to Cart	POST	Add random item into the cart
Checkout	POST	Buy the cart's products with customer information (Mock)

Stressing with Apache JMeter

The second stressing method is the stress testing applied through the Apache JMeter service. As mentioned previously on the iXen Stressing sub-section, we initially create a Test Plan for the stressing process of the application. Then, we define the desired user defined variables, which are the External IP (i.e Node or VM IP address) of one of the available Cluster Nodes to access the application via HTTP and the Node Port of the frontend service, from which we can send requests to all the application's microservices. Then we define four major Thread Groups to apply the desired requests. The first one, applies 450 requests by initializing the same number of Threads to fetch the index page of the application through the frontend service. The second one tests the successful checkout of the stored products into the cart. We initialize 150 Threads in total and we add two random products into the cart, after locating them in the products list, and finally submit the order with the current products of the cart. The third Thread Group, accesses the cart service for 350 Threads in total and finally the forth one attempts to change the payment currency randomly for 250 Threads. Only the Second Thread Group runs 5 type of requests for 150 threads each, so 750 requests are applied via this group. The difference between the iXen Stressing is that for this application we aim to submit a finite number of distributed requests, while on the iXen application we applied a finite and distributed number of requests for a specific time range of 15 minutes.

Totally, for OnlineBoutique application, 1800 normally distributed requests are applied within 1 minute time range via 1200 total threads. We repeat this Test Plan for 6 times in total and nearly 6 minutes in total time execution. We apply 10800 requests into the application, which can be calculated as nearly 30 requests per second for that specific time range. Table 4.5 makes the synopsis for all these thread groups by each type of request.

For the experiments part and only for the OnlineBoutique application, we will gather resources and produce a service placement solution for each of the proposed strategies, initially from the default stressing module, which is the LoadGenerator microservice and then we will apply additional stressing from the Apache JMeter service. In this way, we can test the application under different types of loads and monitor the behavior of the placement strategies to each stressing method.

Table 4.5. *Apache JMeter Test Plan for OnlineBoutique*

Thread Group	Threads	Total Requests	Requests Distribution
GET index	450	450	25.0%
POST products order	150	750	41.66%
GET cart	350	350	19.44%
POST change currency	250	250	13.88%
Total (One round)	**1200**	**1800**	16.66%
Application Stressing	**7200**	**10800**	100%

4.4 Cost Function

The final step prior to executing the service placement strategies and present the experimental results is to describe the cost function of our implemented cluster and locate the factors that can alter this function. In this section, we will present the cost function of our Kubernetes clusters and present the GCP pricing for the utilized resources.

The utilized system is running on a Cloud infrastructure orchestrated by Kubernetes. The cluster of each benchmark use-case consists of various Nodes and Pods with specific resource allocation and requirements. Moreover, Pods communicate either internally between Pods in the same Node (Ingress) or externally with Pods in a different Node (Egress) and store various data according to every application logic. All these factors are charged separately in a Cloud infrastructure and a cost function must be configured to estimate the cluster's fees. In table 4.6 we present the abbreviations that will need to define the cost function.

Table 4.6. *Cluster Abbreviations for Cost Function*

Description	Symbol
Cluster	C
Node	N
Number of Nodes	n
CPU allocation (Cores)	c
RAM allocation (GB)	r
Ingress Traffic (Bytes)	t_{in}
Egress Traffic (Bytes)	t_e
Storage (GB)	s
Machine Type	M
Time of usage (hours)	h
Region	R

There are many factors which can vary the cost of a Kubernetes cluster. Mainly, most Cloud Providers determine the cost according to the Network Traffic, especially between

VMs and even more between VMs locating in different zone areas (i.e Egress Traffic). Resource allocation is also a crucial factor upon an initialization of a Kubernetes cluster, which is mostly referred to CPU, RAM and Storage allocation. The different machine types or the image type of VMs can also affect the cost of the cluster. In the following equation, we will present an estimation of the cost function of a Kubernetes cluster according to the factors mentioned earlier and will be used next to formulate the actual cost of the utilized Kubernetes cluster.

$$TotalCost = Cost_{CPU} + Cost_{RAM} + Cost_{Traffic} + Cost_{Storage} \tag{4.1}$$

$$Cost_{CPU} = \sum_{i=1}^{N} cpu_{cost}(M_i) \cdot h_i \tag{4.2}$$

$$Cost_{RAM} = \sum_{i=1}^{N} ram_{cost}(M_i) \cdot h_i \tag{4.3}$$

$$Cost_{Traffic} = \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} [t_{in}(i \to j) \cdot cost_{ingress} + t_e(i \to j) \cdot cost_{egress}] \tag{4.4}$$

$$Cost_{Storage} = \sum_{i=1}^{N} s_i \cdot storage_{cost}(M_i) \tag{4.5}$$

It should be noted that these equations are described as a generalized function of the cluster's cost and there may be many factors to differentiate the total cost in a Cloud infrastructure. In some cases there are additional costs for the GPU usage according to the requirements, the optimizations tools (i.e load balancing, auto-scaling of Pods etc) of the cluster in the Cloud or even some extra network fees to enable external communication (i.e Egress Traffic or access outside the Cloud infrastructure).

Cloud Provider's Cost

In this Thesis, we utilize GCP as Cloud provider and therefore we will present the respective costs according to the GCP pricing. Although, the main equation of the total summary of costs (which presented previously) remains the same, the respective cost factors differentiate according to the costs given in the GCP documentation [36]. The following equations present the respective cost factors according to the GCP pricing.

$$Cost_{CPU} = \sum_{r=1}^{R} \sum_{i=1}^{N} cpu_{cost}\,[M_i(r)] \cdot h_i \tag{4.6}$$

$$Cost_{RAM} = \sum_{r=1}^{R} \sum_{i=1}^{N} ram_{cost}\,[M_i(r)] \cdot h_i \tag{4.7}$$

$$Cost_{Traffic} = \sum_{r=1}^{R} \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} [t_e(i \rightarrow j, r) \cdot cost_{egress}(r)] \tag{4.8}$$

$$Cost_{Storage} = \sum_{i=1}^{N} storage_{cost}(M_i) \cdot h_i \tag{4.9}$$

Cost of CPU and RAM relies only on the Machine types, the Region of the cluster and the run-time hours of the cluster's VMs. Storage, which is the disk size in our case, is connected with any new cluster initialization and varies according to disk type. Finally, Ingress Traffic is not charged in GCP Cloud, however Egress Traffic is charged according to the Zone and Region of each VM and the size of GB exchanged between the host machines.

Cluster Cost

Prior to presenting the actual cost equations for the experiments part, we must sum-up that we utilize a homogeneous environment in Kubernetes infrastructure, where every machine lies on the same Zone area and Region, with the same resource allocation attributes. In this way, the cost for initializing the requested number of VMs is the same and therefore the sum of each cost factor is replaced by the VM cost times the number of the VMs used. The size of each application, their services, Istio services and the data stored in the allocated disk of the cluster is relatively low in volume and thus the cost of storage is considered negligible. As we mentioned previously, the main equation of the summary of the costs for all the resource factors (Equation 4.1) remain the same, however the individual costs are modified according to the characteristics of the initialized clusters, as displayed below.

$$Cost_{CPU} = n \cdot 2vCPU \cdot cpu_{cost} \cdot h_i \tag{4.10}$$

$$Cost_{RAM} = n \cdot 8GB(RAM) \cdot ram_{cost} \cdot h_i \tag{4.11}$$

$$Cost_{Traffic} = cost_{egress} \cdot \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} t_e(i \rightarrow j) \tag{4.12}$$

$$Cost_{Storage} \simeq 0 \tag{4.13}$$

We can conclude that cost of CPU and RAM depends only on the hours of running each Node and the total size of Nodes in the existing cluster. Cost of storage is not charged additionally and the cost of Egress depends only on the requested data bytes from services that do not belong at the same Node (communication between the VMs). The responsed bytes of requests are considered as ingress traffic and there is no additional fees for them. As VMs belong in the same Zone and Region, the cost of Egress is fixed and respective to the total amount of GB requested from the Nodes in the cluster. In the following table 4.7 we present the actual GCP costs for the utilized cluster infrastructure and VMs.

Table 4.7. *GCP costs for e2-standard Machines*

Description	Cost (USD)
Predefined vCPU	$0.028103/vCPU/hour
Predefined RAM	$0.003766/GB/hour
VM-to-VM Egress traffic	$0.01/GB

As we initialize 4 VMs in our cluster and with prior knowledge of the hourly fees of the GCP, we can present the total monetary hourly cost function of our infrastructure in the following equation.

$$TotalCost = 0.345336 \cdot h_i + 0.01/GB_{h_i} \tag{4.14}$$

It must be referred that in the results section we display the cost results of the clusters per month of usage. The hourly costs presented in table 4.7 are modified accordingly to produce the value of the respective cost factors per month of usage.

4.5 Results

Making the summary of our implemented infrastructure, we initialize 4 Nodes into a Kubernetes cluster in GCP, with predefined resource allocation of 2vCPU and 8GB RAM per Node. The environment is homogeneous and all the Nodes reside at the same Region. Initially, we let the default Kubernetes Scheduler decide upon the initial service placement for each application into the existing Nodes without specifying the Pod and Node Affinities/Anti-Affinities. The scheduler distributes equally the services into the cluster Nodes according to the Pod and Node restrictions and requirements.

To implement the service placement strategies, we primarily access the Cluster API and the Metric tools to collect the important Node and Pod data (allocated space, resource requirements, available volume in resources etc) in order to produce the essential data structures for executing the proposed strategies. Then, we implement one by one the proposed placement strategies and produce the service placement for each use-case. Below, we display the results produced after applying the placement strategies. We divide this section into five major subsections, one for the selection of the K-Value of the BKM algorithm and the other four for each examination factor of the implemented placement strategies.

For OnlineBoutique and iXen applications, we utilize two performance measures as described in section 3.2, the RPS and the WBA performance measures. For both performance measures and for each microservice placement strategy we will apply the synthetic stressing workloads to create network communication among the applications' microservices. For the next subsections, we will present the produced graphs for each placement strategy for both applications, iXen and OnlineBoutique, both performance measures and specifically for OnlineBoutique application both stressing methods. For each produced graph, we will comment the results and justify their performance.

4.5.1 K-value Selection for BKM Algorithm

First and foremost, we need to justify the selection of the BKM K-value, which will be used to execute the BKM placement strategy and produce the graph results. Although we select the K-Value, mostly, to match the initialized number of VMs, we examine various cases of the K-value, which are the total number of hosts and the Egress traffic optimization upon executing the BKM placement strategy to the initial service placement. We examine only the case for the OnlineBoutique application and the WBA affinity metric. As both applications have almost the same Pod and Node resource requirements, we presume that the selection of the K-Value is independent on the scale of these applications. In Figures 4.2 and 4.3, we display the results for the produced host machines required to run the application for the various K-values.

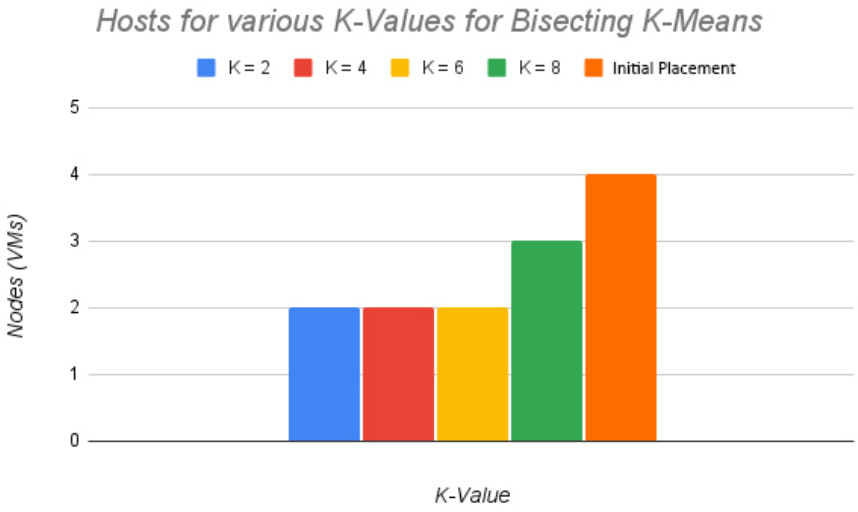

Figure 4.2. *Number of hosts for different K-Value of BKM algorithm*

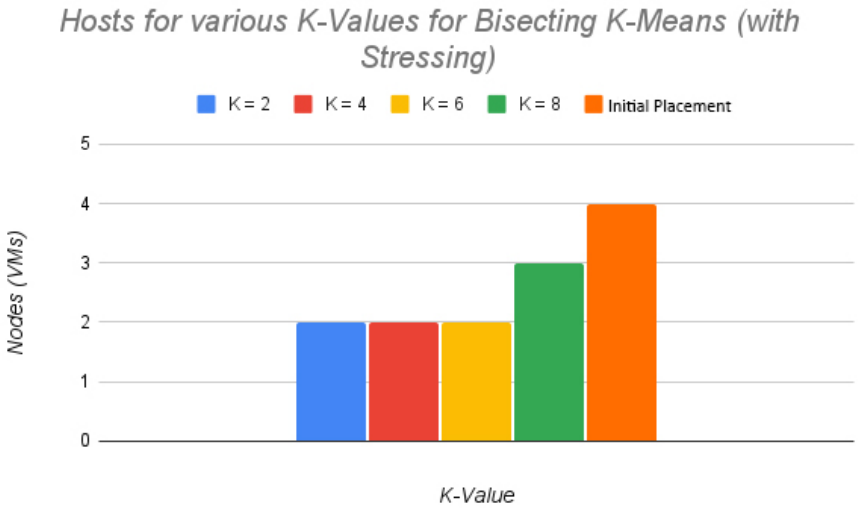

Figure 4.3. *Number of hosts for different K-Value of BKM algorithm with Stressing*

Below, in Figures 4.4 and 4.5, we display the results for the traffic optimization for the K-Values according to the initial service requested bytes among the Nodes.

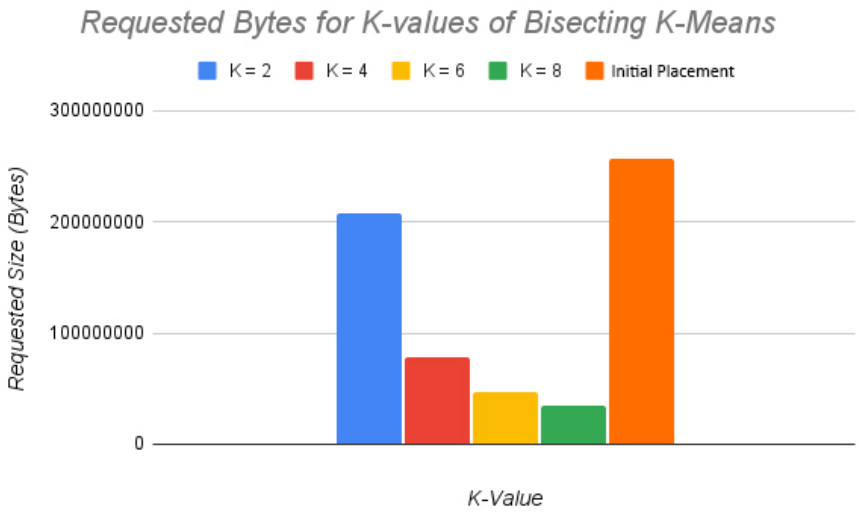

Figure 4.4. *Traffic Optimization for the various K-Values of BKM algorithm*

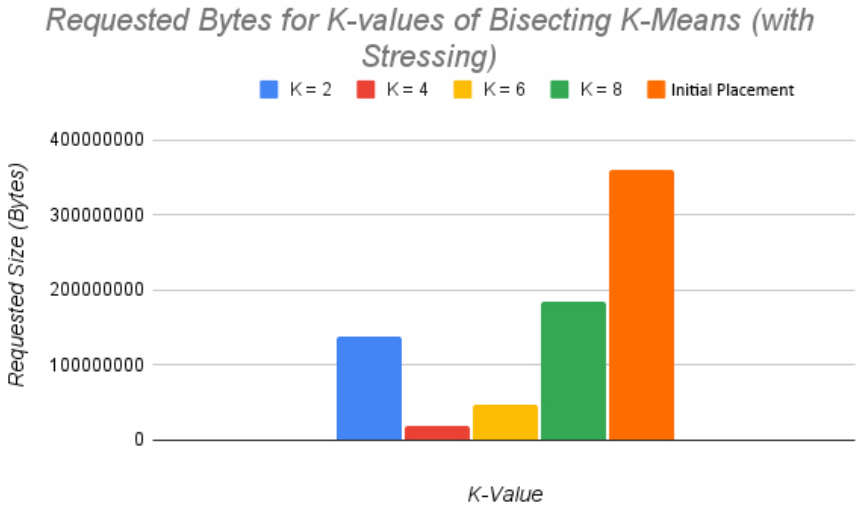

Figure 4.5. *Traffic Optimization for the various K-Values of BKM algorithm with Stressing*

We examined the case for 2, 4, 6 and 8 cluster centroids, which is the K-Value of the BKM algorithm, to partition each application into K clusters. As the cost of the cluster is relied basically on the Egress traffic and the allocated resources (or the number of nodes), we will select the K value according to the number that minimizes the total cost. From the produced graphs, we can conclude that number of hosts after each placement strategy remains almost the same for every case and increases slightly for the case of $K = 8$ but still is less than the initial placement, for both ways of Stressing.

The desired reduction in the service requested bytes in Egress traffic between the dif-

ferent VMs is achieved mainly for $K = 4$ and $K = 6$ clusters, with the former producing the best reduction for the case of the additional Stressing and the latter remaining almost the same for both ways of Stressing. However, BKM resides in random methods for the selection of the centroids, which may not always be optimal. Moreover, the scale of both applications is relatively small and we can not reach a safe conclusion for the behavior of the algorithm in a much more dense graph with many unconnected services. It can be concluded that for both $K = 4$ and $K = 6$ values the placement strategies would be optimal for these applications. We selected the $K = 4$ number to match the cluster's initial number of Nodes.

4.5.2 Execution Time of each Placement Strategy

An important factor of examining the placement strategies is the execution time of each implemented strategy and to crosscheck the theoretical time complexity of each algorithm with the measured execution time. We expect the time of executing the placement strategies, which utilize the Heuristic Packing and the Contraction algorithm, to be greater than the others. However, these execution times are expected to be relatively low as we use random methods to produce each application's partitions. In Figure 4.6, we display the results for the Online Boutique application for both performance measures (i.e RPS and WBA affinity metrics) and for all the proposed strategies.

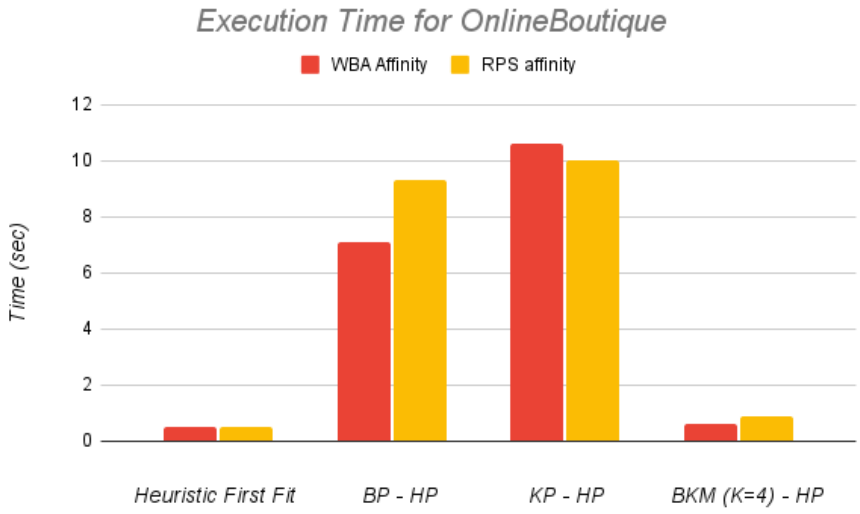

Figure 4.6. *Execution time of algorithms for OnlineBoutique*

Below, in Figure 4.7, we present the results for the Online Boutique application under the additional Stressing from the Apache JMeter service as described in the previous sections.

Finally, in Figure 4.8, we display the respective results for the iXen application for the utilized performance measures and placement strategies.

As for the execution time of each placement strategy, we can conclude that all the strategies are relatively fast and can produce a service placement within few seconds. All algorithms perform the same for both applications and performance measures, with only

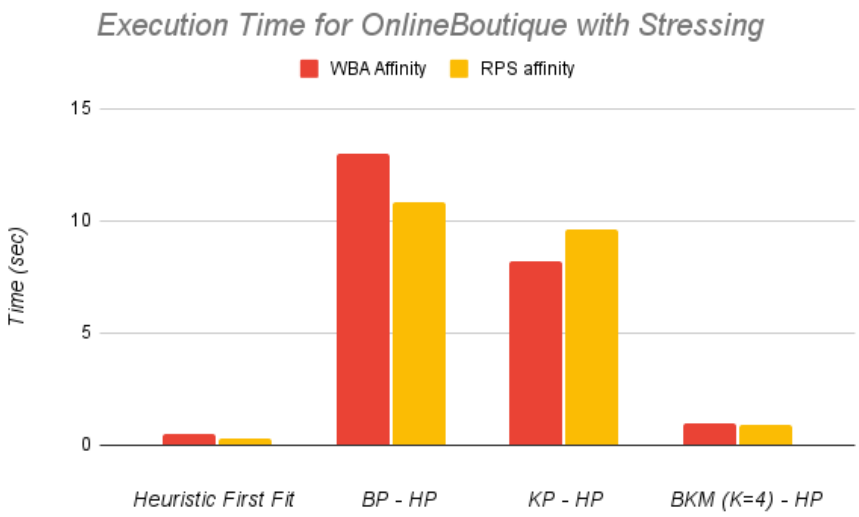

Figure 4.7. *Execution time of algorithms for OnlineBoutique with Stressing*

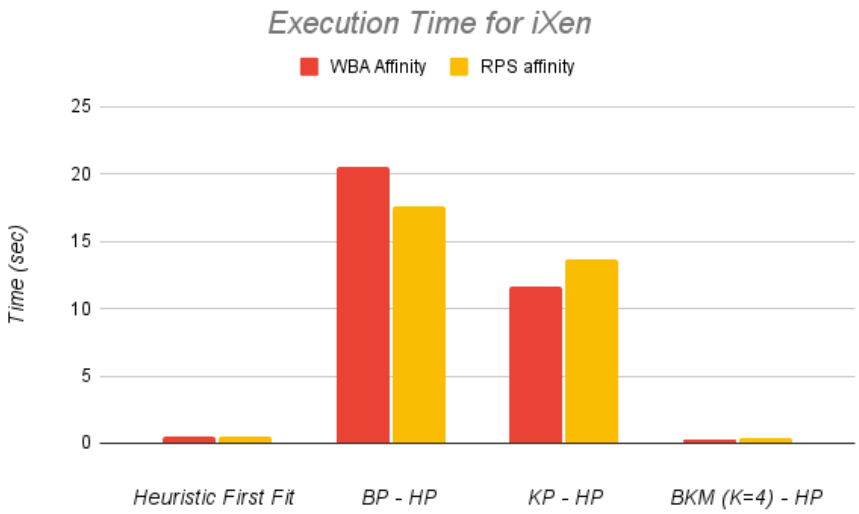

Figure 4.8. *Execution time of algorithms for iXen*

difference the execution time between BP and KP. We expect KP to have lower execution time than BP, which on the OnlineBoutique application for Stressing is accurate, but with the loadGenerator Stressing module we can observe that Binary Partition's execution time is lower. The reason behind the better performance in the time complexity of the KP is that, upon each iteration, K value increases and thus the complexity of the contraction algorithm is reduced. For iXen application, BP has higher execution time than the KP too, which is based on the better partitioning that is applied for small microservice-based applications, like iXen and OnlineBoutique. HFF and BKM strategies have the lower execution times and in most cases the former has lower execution time than the latter. This is because BKM algorithm utilizes the Heuristic Packing to place the partitions into the available hosts and thus its time complexity slightly increases. However, for the iXen application BKM has lower execution time and this can be due to the partitioning process

of the application's services.

It must be noted that the low rates of execution time is due to the fact that the applications are relatively small in number of microservices and communication edges between them. As the application services increase in size we expect an increase in the execution time of each placement strategy, mainly for the BP and KP algorithms. However, the fact that the partitioning algorithms rely on random methods to calculate each application's partitions will still have a better performance in time complexity than other algorithmic methods.

4.5.3 Number of Hosts

The first factor to determine the total cost of the cluster is the variation in the number of Hosts needed to run the application in the GCP cluster. The initial number of hosts is selected to 4 Nodes and in the following figures we present the results for each application and affinity method. In Figure 4.9, we display the results for the Online Boutique application for both performance measures and for all the proposed strategies.

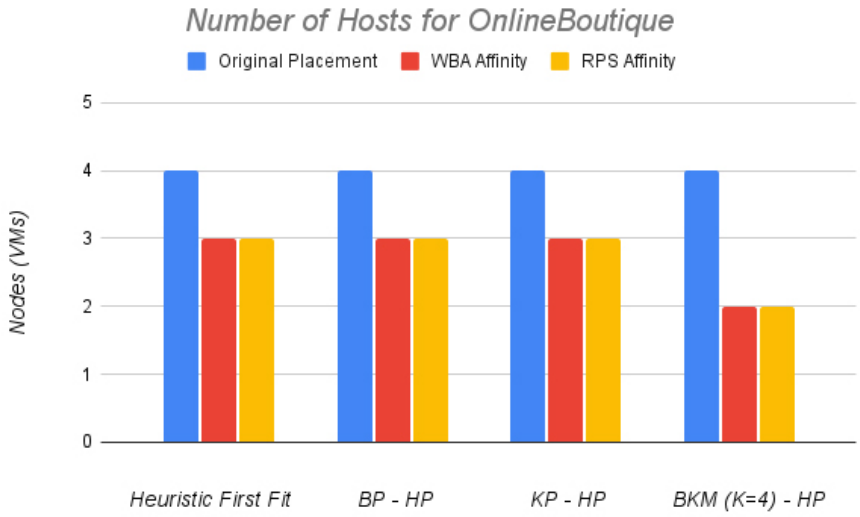

Figure 4.9. *Number of Hosts used for OnlineBoutique*

Below, in Figure 4.10, we present the results for the Online Boutique application under the additional Stressing from the Apache JMeter service as described in the previous sections.

Finally, in Figure 4.11, we display the respective results for the iXen application for the utilized performance measures and placement strategies.

As displayed above, for both applications and performance metrics and for all the service placement strategies, we can observe a significant reduction on the utilized VMs needed to host each application. For OnlineBoutique application, we can observe that almost all placement strategies reduce the number of utilized hosts by 25%, while BKM strategy reduces the number of hosts for both methods of Stressing by 50%. For iXen application, HFF resulted in utilizing 2 VMs, while most of the other service placement strategies resulted in 3 VMs.

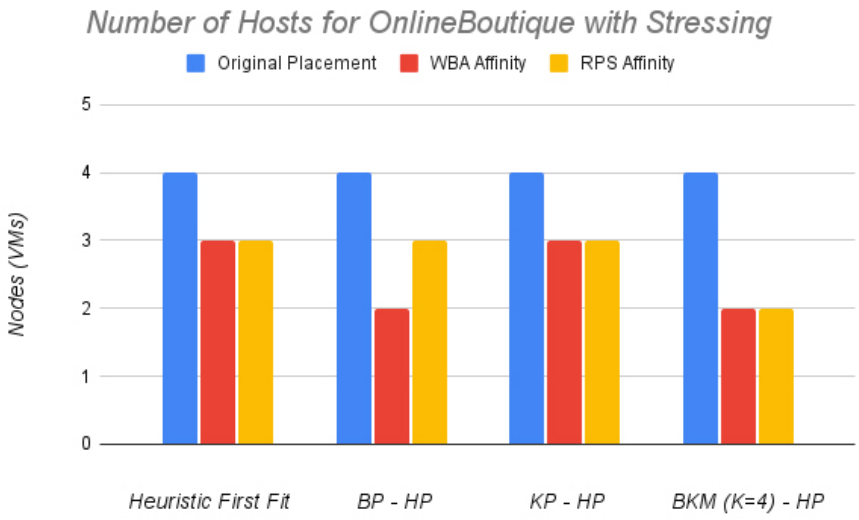

Figure 4.10. *Number of Hosts used for OnlineBoutique with Stressing*

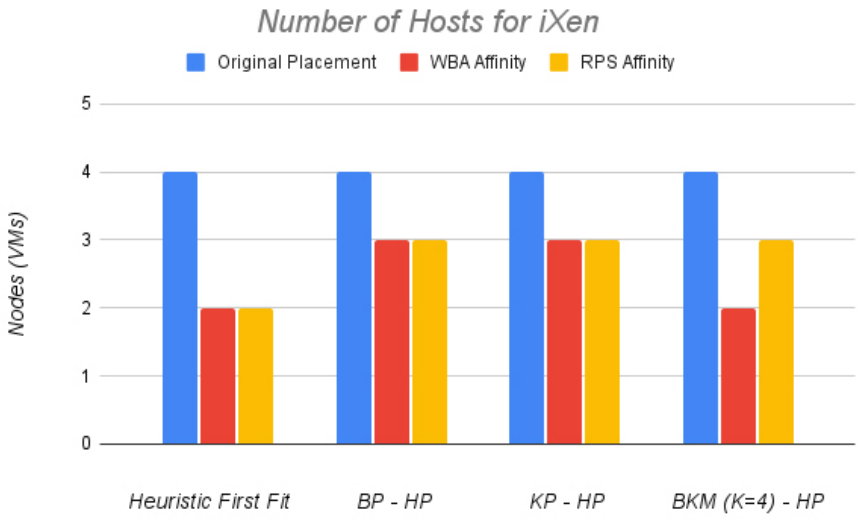

Figure 4.11. *Number of Hosts used for iXen*

We expected a reduction in the number of hosts, as have already mentioned, because each application requires only 2 VMs to operate successfully. The reason behind the utilization of 3 VMs instead of 2 for the most service placement strategies is different for the placement and the clustering algorithms. HFF has some restrictions upon the movement of microservices into other Nodes and therefore initial placement highly affects the produced results. For BP and KP strategies, algorithms produce partitions, which then are packed into the available VMs. However, the available VMs host also some Istio and Kubernetes Services and in conjunction with the random methods that are utilized from these algorithms result in a sub-optimal result most times. BKM algorithm resides also in random methods, however the initial placement and the affinity performance metric can vary the number of VMs, as shown in the iXen results.

4.5.4 Egress Traffic

Requested Bytes

The next case we examine is whether the traffic optimization is achieved by each placement strategy and specifically the reduction in the requested bytes from services belonging to different hosts. The goal is to reduce this amount from the initial's placement egress traffic rates. GCP charges only for the Egress traffic and therefore for the requests between services in different host machines. In Figure 4.12, we display the results for the Online Boutique application for both performance measures and for all the proposed strategies.

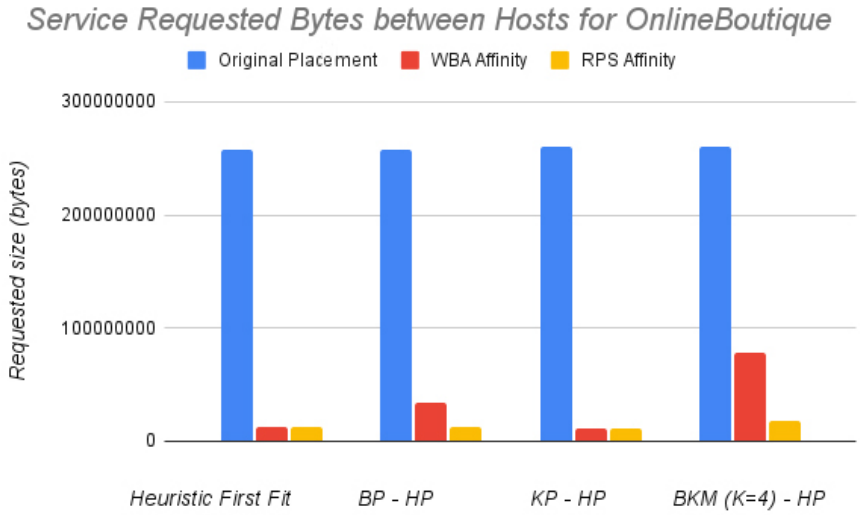

Figure 4.12. *Traffic Optimization by the Bytes of Requests for OnlineBoutique*

Below, in Figure 4.13, we present the results for the Online Boutique application under the additional Stressing from the Apache JMeter service as described in the previous sections.

Finally, in Figure 4.14, we display the respective results for the iXen application for the utilized performance measures and placement strategies.

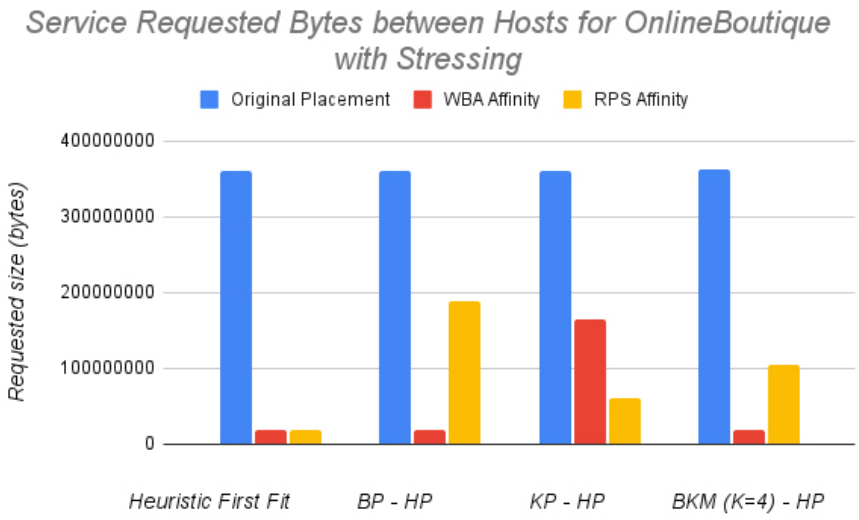

Figure 4.13. *Traffic Optimization by the Bytes of Requests for OnlineBoutique with Stressing*

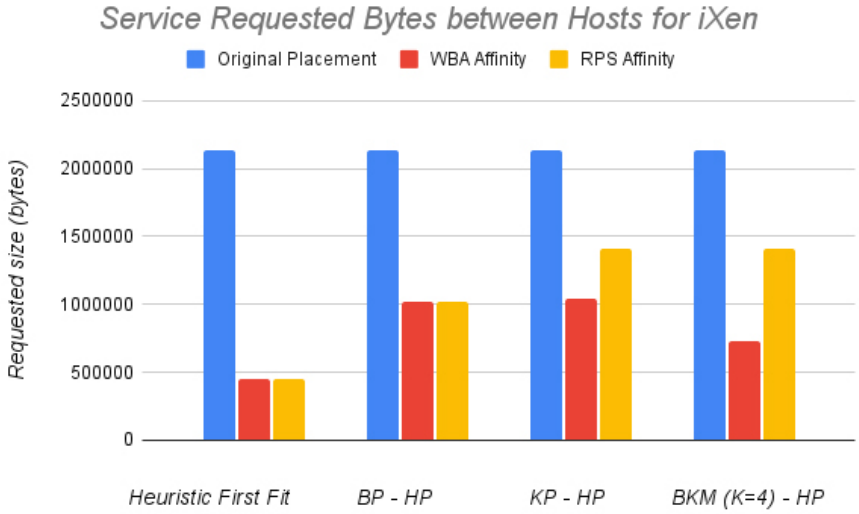

Figure 4.14. *Traffic Optimization by the Bytes of Requests for iXen*

Monthly Variation

Though traffic optimization shows significant reduction in the size of the Egress requested bytes between the cluster's Nodes, we present another case parameter, which is the monthly Egress Variation in GB for the proposed placement strategies. This measurement is used to further comprehend the reduction in the Egress Traffic and to calculate the monthly cost, which is displayed in the next section. In Figure 4.15, we display the results for the Online Boutique application for both performance measures and for all the proposed strategies.

Below, in Figure 4.16, we present the results for the Online Boutique application under the additional Stressing from the Apache JMeter service as described in the previous

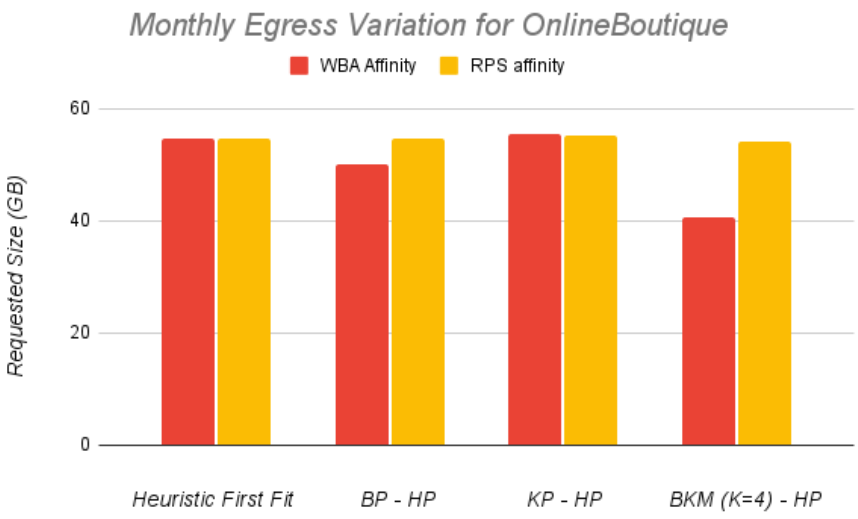

Figure 4.15. *Egress Variation per Month for OnlineBoutique*

sections.

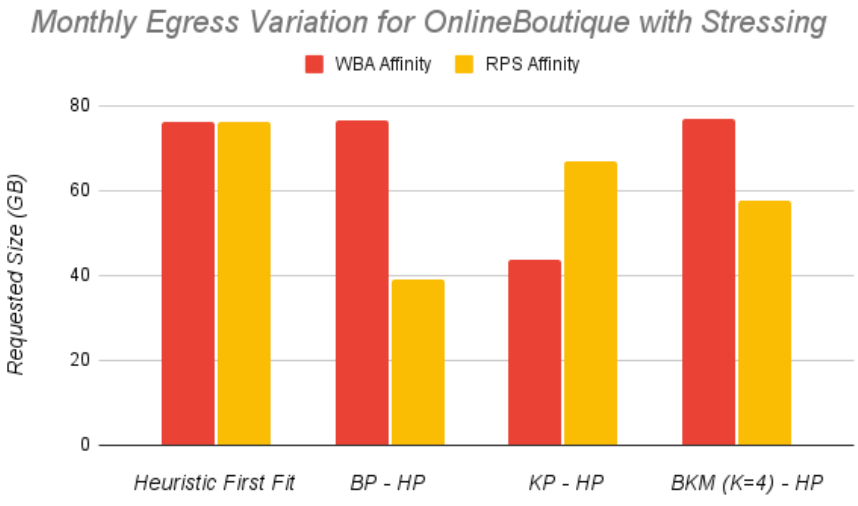

Figure 4.16. *Egress Variation per Month for OnlineBoutique with Stressing*

Finally, in Figure 4.17, we display the respective results for the iXen application for the utilized performance measures and placement strategies.

By examining the results on the Egress variation and the service requested bytes after each placement strategy, we can conclude that there is a major reduction in the traffic requested bytes and thus the monthly Egress variation after the implementation of the proposed service placement strategies. For both applications and performance measures, we can observe reduction greater than 50% in the egress traffic communication between the application's microservices.

For OnlineBoutique application, HFF is the only placement strategy with stable reduction rate in the traffic bytes. The other partitioning strategies perform almost equal

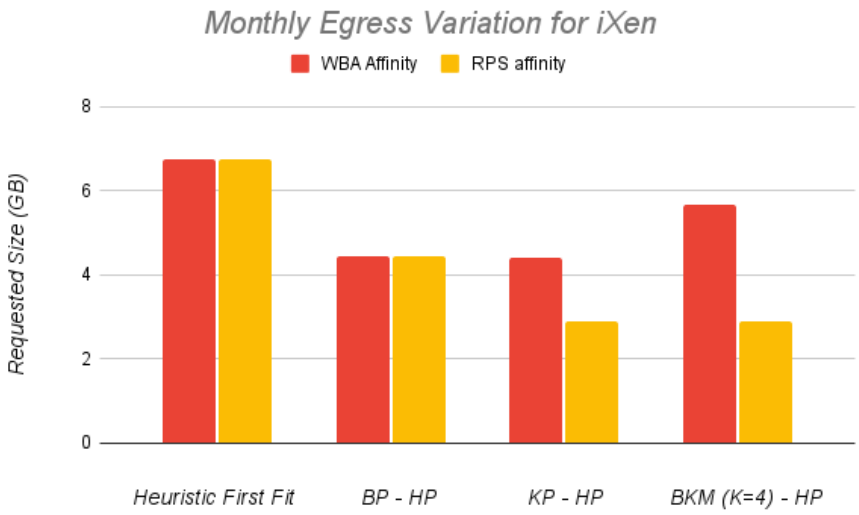

Figure 4.17. *Egress Variation per Month for iXen*

results and in every case they reduce the traffic bytes, although in some cases the rates of reduction varies. This is due to the fact that for every execution of each placement strategy, a new partition plan is proposed. This means that the final service placement may vary and thus the traffic requested bytes behave accordingly. All in all, the total traffic is reduced with the HFF producing the best results.

For iXen application, HFF and BP produce almost the same results for both performance measures, however we can observer a variation between the results of the KP and BKM placement strategy. In the case of the WBA affinity, the reduction in traffic requested bytes is greater than the reduction made with the RPS affinity. This is down to the performance measures and specifically the more reliable affinity metric, which is WBA affinity and can produce more accurate performance measures between each application's services. In this way the placement strategies can produce a more optimal placement strategy with the aid of the Heuristic Packing algorithm. Moreover, HFF has the higher reduction rate than the other strategies.

4.5.5 Total Monetary Cost of Cluster

Finally, we present the total cost of each cluster according to the parameters mentioned previously. Cost function resides mainly on the number of VMs and the Egress traffic bytes. In the previous results sections, we presented the reduction of these two parameters, so we expect the total cost to be also reduced. For better presentation of the results, we calculate the total cost of each cluster for a monthly period. Given the prices from the GCP documentation and the optimization on these parameters, we produce the below graphs for the total cost of each cluster. In Figure 4.18, we display the results for the Online Boutique application for both performance measures and for all the proposed strategies.

Below, in Figure 4.19, we present the results for the Online Boutique application under the additional Stressing from the Apache JMeter service as described in the previous sections.

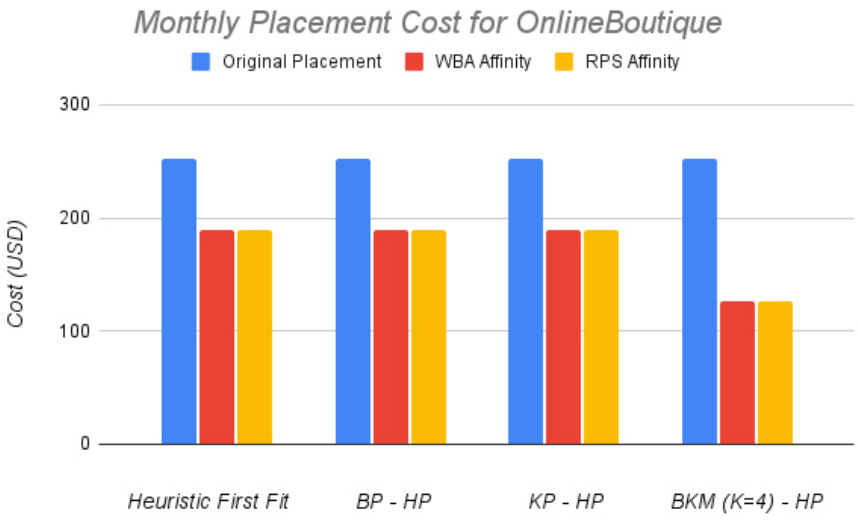

Figure 4.18. *Cluster cost per Month for OnlineBoutique*

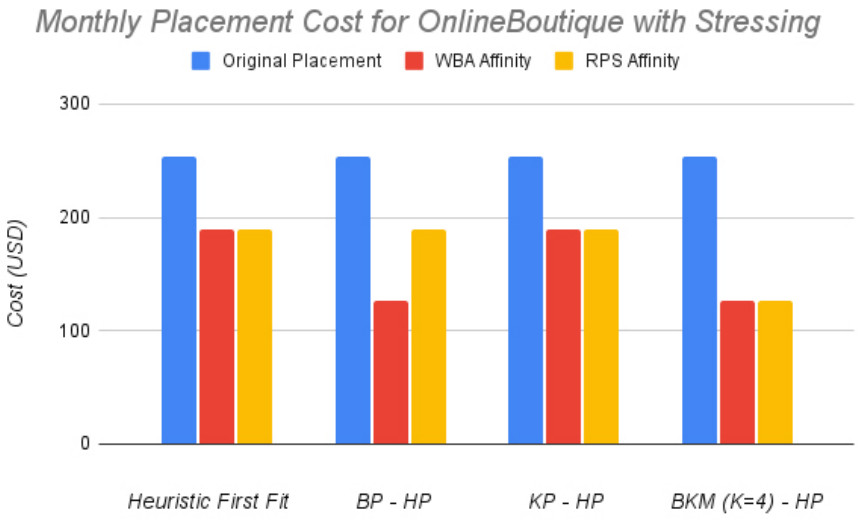

Figure 4.19. *Cluster cost per Month for OnlineBoutique with Stressing*

Finally, in Figure 4.20, we display the respective results for the iXen application for the utilized performance measures and placement strategies.

As cost optimization is the main goal of this Thesis, we can admit that we achieved a significant cost reduction for the Kubernetes cluster costs in comparison with the default Kubernetes Scheduler strategy.

The cost reduction in most cases is up to 25% and in some cases over 50%. This relies mostly on the reduction on the utilized Nodes needed to run the application, which means that less VMs, and therefore less CPU and RAM allocation, would result in a larger cost reduction. Traffic optimization can reduce the cost of the cluster too, however for these applications its significance is relatively low as the cost for the monthly Egress traffic is lower than 1 USD, when the total cost of the cluster rise up to 125 and 190 USD according to the number of hosts required for each cluster and placement strategy. For an application

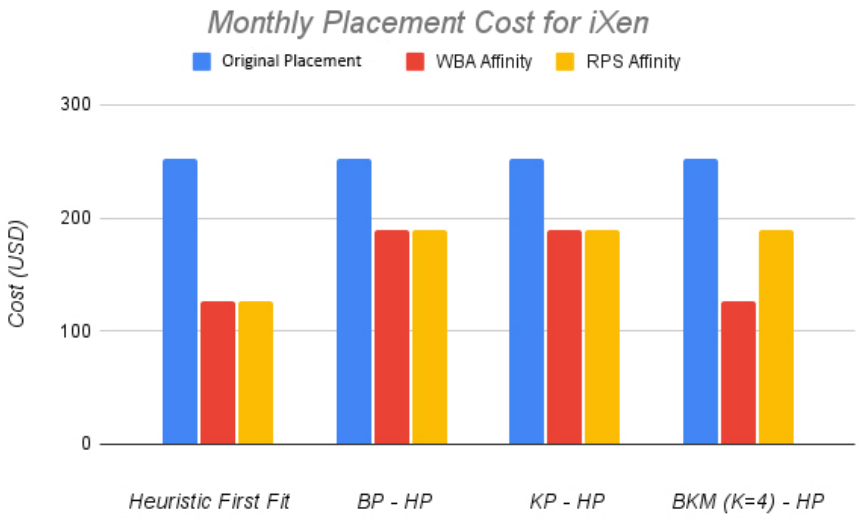

Figure 4.20. *Cluster cost per Month for iXen*

greater in scale with higher traffic communication rates in size of exchanged requests, we expect the Egress traffic to have greater impact on the cost function.

It must be referred that the time of the cluster's lifespan was different for conducting the experiments and thus some modifications were made to find the appropriate monthly cost. Specifically, OnlineBoutique application was running for about 3 hours after executing the proposed placement strategies. After the successful execution of each strategy, we applied the additional Stressing from the Apache JMeter service and repeated the same process. We multiplied, accordingly, the respective parameters to match the monthly time range and produced the monthly results, expecting the same behavior throughout the monthly time range. For the iXen application, the time of lifespan was about 15 minutes before executing the placement strategies and thus the monthly cost was modified accordingly.

4.6 Discussion

In this section we will evaluate the utilized applications, the performance metrics and the service placement strategies implemented in this Thesis and we will compare them between them. This evaluation will further support the justification of the produced results and the resulted placement solutions.

Comparison of the Performance Metrics

For the purposes of this Thesis we utilized two performance metrics between each application's microservices, the Requests per Second and the Weighted Bidirectional Affinity. The former can be easily collected from the Kiali graph, while the latter must be calculated according to the size and the number of the exchanged messages and thus it is considered as a more accurate metric. The important variations between the two performance measures can be observed when two services contain a high RPS affinity, but the size and number of messages is relatively low. This can lead to different application partitions and service

placement between the two performance measures, as the significance of each affinity varies from one affinity metric to another. For the most cases, we can conclude that WBA affinity has better overall performance over the RPS affinity, especially in traffic optimization. However, in some cases, RPS affinity might produce better results, but this can be due to various factors, as the initial service placement or the random methods selected for the clustering algorithms. We believe that for larger applications, WBA affinity will have even greater performance rates than RPS affinity.

Comparison of the Applications

Primarily, a comparison between the utilized applications must be made to better comprehend the produced results. Both applications are microservice-based applications running on a Kubernetes cluster in the same Region and Zone and with the same cluster resource allocation. In both applications we execute the placement strategies with the same parameters and restrictions. The number of services and affinity edges are relatively the same in size, so we can easily compare the placement strategies and come to a conclusion about their overall performance. Additionally, both applications can be hosted totally in 2 VMs, however we allocate 4 Nodes to benchmark the placement strategies.

However, iXen and OnlineBoutique have different Pod resource requirements, as iXen requires for every service nearly 2% for CPU and nearly 4% for RAM of the allocated space of each Node. On the other hand, Pod resource requests for the OnlineBoutique application is predefined and varies from 5%-15% for CPU and 1%-4% for RAM of the allocated space of one Node. Although the number of microservices is slightly lower for OnlineBoutique, their CPU requests are much higher than the CPU requests in iXen application. This difference can vary the service placement of each placement strategy and produce different results for each application.

Additionally, iXen application requires stressing to produce the application's graph and the performance measures, so we examine only the case for the distributed Stressing applied throughout the application's services. On the contrary, OnlineBoutique contains a microservice to produce traffic among the services, by applying random generated requests, and an additional stressing method with higher workload testing is applied through Apache JMeter, so we can observe the performance of the application under different types of loads. This can lead to better decision for the best service placement strategy.

Last but not least, these applications are different in the way they serve the application requests from user. In iXen application, we can access the various services directly to get the required information and data, while in OnlineBoutique every request is served through the frontend service. So, frontend service in OnlineBoutique has great impact to the service placement strategy and creates high performance measures with other services, while iXen has a more distributed system of communication among the microservices.

Comparison of the Placement Strategies

To conclude, considering all the factors and the produced results, we can admit that for small scale applications with small size of services and little affinity edges HFF is in

overall the better service placement strategy among the other options. BKM strategy has also a remarkable performance, as it decreases the number of hosts at the completely essential number, in most cases, and thus reduces the cost of the cluster, however it does not perform so well for the traffic optimization processes and might not perform the same for larger applications in scale. Both applications have significant low execution time and can produce sub-optimal results most times.

BP and KP reduce the total cost of the cluster and, in some cases, lead to traffic optimization, however their results are different from one execution to another and sometimes their service placement can be non-optimal. As the main goal of this Thesis is to reduce the overall cluster cost, we would recommend to exchange the random methods in these partitioning algorithms over the accuracy and stability that can be guaranteed from other clustering methods and algorithms.

Chapter 5

Conclusions and Future Work

In this last chapter we will briefly summarize the content and the results of this Thesis and propose future work and optimization strategies that can be applied to further improve the service placement problem and reduce the infrastructure's cost.

Our goal was to reduce the total running costs of a Kubernetes cluster in a Cloud environment by solving the SP problem. In order to reduce the overall cost, we had to reduce the number of VMs used to facilitate the implemented applications and the Egress Traffic between these VMs. To face the SP problem, we utilized graph-based partitioning algorithms and heuristic methods. We utilized two benchmark applications, the iXen and the Google's OnlineBoutique eShop to implement the proposed service placement strategies. We configured a Service Mesh, Istio, into these applications so as to monitor the Pods and Nodes resources and to calculate the performance measures needed to apply the placement strategies. By comparing the default placement strategy of Kubernetes Scheduler with the implemented strategies, we have reached to the assumption that run-time costs, in terms of money paid from users, can be reduced up to 25%-50%. The proposed strategies managed to reduce the size of the cluster and the Egress Traffic of VMs significantly by producing a placement solution within seconds for these small-scale applications.

Addressing the SP problem can be a hard task and the solution can vary according to the optimized application's parameters. For this Thesis, it is vital to trade-off the time needed to construct and apply a placement solution in order to locate an optimal placement solution and, eventually, reduce the cluster's run-time costs. By attempting to solve the SP problem from a different aspect, like reducing the response times between the applications' microservices, would require a divergent placement strategy from the proposed ones, like prioritizing the network latency between the communicating microservices. Reducing latency in microservices communication is based mainly on the infrastructure and is a crucial factor of running applications especially in Fog and Edge Cloud environments. Size of Kubernetes Nodes, their resource allocation and the Cloud environment can vary the placement solution of the SP problem. For our case, a Homogeneous Cloud environment, Node and Pod resources highly affected the outcome of the implemented placement strategies and their selection was made to fit the Thesis goal requirements.

An important comment must be made for the optimization strategies that are provided from the Kubernetes Scheduler. The Kubernetes Scheduler can locate a sub-optimal or

even an optimal solution by defining the Pod and Node Affinities and Anti-Affinities. However, defining the microservices affinities would require a prior knowledge of the application and their weights on the microservices communication edges of the application's graph. For our experimental process we did not contain the optimization methods of the Kubernetes Scheduler to test the efficiency of the placement strategies to locate an optimized placement solution in a completely unknown application or in an application, where we do not know the application's graph weights. This also applies in large scale applications, where the assigning process of Node and Pod Affinities and Anti-Affinities is considered a hard task.

Having achieved the monetary cost optimization goal in a Kubernetes cluster using microservice-based placement strategies, we propose further improvements that can be examined and applied in a Kubernetes cluster to test its performance on different environments and applications.

First and foremost, we suggest the examination of various clustering algorithms that are not relied on random methods to increase the efficacy of the cost optimization strategies. A comparison between the random partitioning algorithms and the clustering algorithms for graph based applications should be made to verify the significance of the time execution parameter of these random tactics.

Secondly, as we implement our cluster infrastructure in a Homogeneous environment, we utilize VMs with the same resource requirements in the desired size of CPU and RAM. However, in real-life applications, there might not always be the same volumes in VMs in size and in resource allocation. We propose an implementation of these strategies in Heterogeneous environment, so as to validate the consistency of the results in terms of cost reduction. In conjunction with an Heterogeneous environment, we propose the execution of these placement strategies in a Multi-Cloud environment, where Egress traffic is a crucial factor and can significantly modify the cost of the Kubernetes cluster according to the fees of each Cloud provider.

Taking into consideration the size of each application used in this Thesis, we suggest to test the performance also in more realistic benchmarks with large number of services or large number of replicas for each Pod. By increasing the scale of the application utilized in the Kubernetes cluster, we can better observe the behavior of the placement algorithms and their performance under heavy load in a much more dense application graph.

Furthermore, we suggest to utilize and process the affinity hubs that are created among the microservices, when a specific request is occurred. In this Thesis, we utilized affinity tuples between microservices with a communication edge and we did not take into consideration the complete traces of the requests. By implementing service placement strategies that use the affinity hubs, we strongly believe that the placement's performance will increase and the clustering algorithms will further produce more accurate partitions. Istio's tracing service, like Jaeger and Zipkin, can be configured into the Kubernetes cluster to monitor the tracing of a request among the application's microservices.

To achieve the cost optimization in an application on a Cloud infrastructure, we must execute the placement strategies and apply the produced placement result in the existing Kubernetes cluster. There is no automatic way to apply the changes in the service

placement and update the cluster status, so as to optimize the traffic and resource allocation. We propose the implementation of a dynamic Kubernetes placement strategy so as to update automatically the Kubernetes Nodes through the cluster YAML files and the Kubernetes API and re-arrange efficiently the Pods in the existing cluster.

Finally, during the experimentation part of this Thesis, we examined and attempted to optimize the latencies between services with a communication edge. We expected that a relocation of two services in the same host machine with high affinity metric rate will reduce the latency between these two services. However, in the real-time experiments, we discovered that not only the latency is not reduced, but in most cases it would increase significantly. A service placement strategy taking into consideration both factors of latency and service affinities or only only the latency factor should be implemented to test the performance of the proposed placement strategies, especially in Fog and Edge Cloud environments.

Appendices

A

Cluster Data Collection

In this chapter, we will present how the data are collected from the Metric tools and agents in order to be processed by the placement strategies. After the initialization of the Kubernetes cluster, the implementation of each application into the cluster and the configuration of Istio, the Node and Pod data must be collected in order to be processed and produce the placement solutions. Istio enables the collection of the vital data and the visualization of them so as to be better comprehended and utilized. The source code of application is implemented in PyCharm and Jupyter Notebook with Python programming language. All placement and clustering algorithms, as well as the module for collecting the required Node and Pod data, are implemented in Python Classes to achieve better organization of source code, high extensibility on class methods and enable the inheritance, which is an attribute of Object-Oriented Programming. Each of these Classes can be executed independently by every Python program given the required inputs to the suitable format and the required libraries. In the next sections, the way of collecting and visualizing the cluster data from the Istio services will be presented.

A.1 Prometheus Data

The main source of the data collection is from Prometheus service, which applies PromQL queries to collect the cluster data and communicates with other Istio services to provide them with these data. Prometheus provides essential information about the cluster status, the Node and Pod resource requirements and the collected data from the configured metric tools and agents, like the Node Exporters. Prometheus executes the PromQL queries and receives the respective results in JSON format through the Prometheus API, which can be also displayed visually in Prometheus UI in a table or in a graph representation. To assemble all the important information, verify the proper configuration of the cluster and better comprehend the collected data, we executed PromQL queries in the Prometheus UI, as depicted in Image A.1, which displays an example of an executed PromQL query about the requested CPU of the Kubernetes cluster's Nodes.

For accessing the Prometheus UI, we had to convert the Service type of Prometheus into NodePort enabling a TCP communication port in the range of 30000 and 32767 in the Cloud provider. To request data (GET method) from the Prometheus API, a valid url of

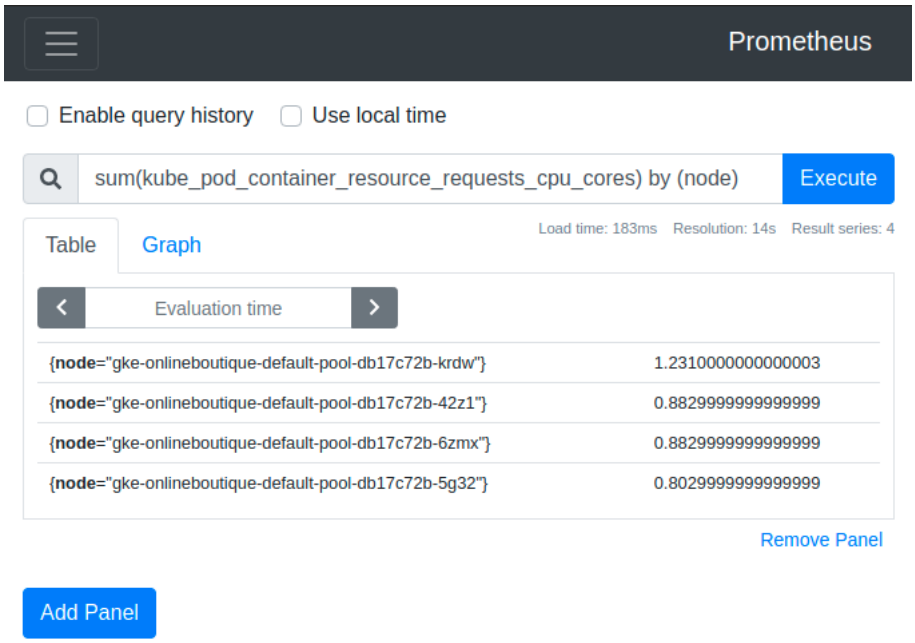

Image A.1: *PromQL query in Prometheus UI*

the API in the configured network port and a valid query must be given as input. In the next tables, we will present the queries executed to collect the Prometheus data. In Table A.1 and in Table A.2 the queries for collecting the Node and Pod resources respectively are displayed. Each executed request returns a JSON text file as a result, which is processed to store only the essential information about the Node and Pod resource, like the size of CPU and RAM allocation.

Table A.1. *Node PromQL Queries*

Description	Query
Node Requested CPU	sum(kube_pod_container_resource_requests_cpu_cores) by (node)
Node Requested RAM	sum(kube_pod_container_resource_requests_memory_bytes) by (node)
Node CPU Allocation	kube_node_status_allocatable{resource='cpu'}
Node RAM Allocation	kube_node_status_allocatable{resource='memory'}

After extracting the essential data from the PromQL queries, we store them in Python dictionaries and process them to calculate the performance measures and finally to be processed from each proposed placement strategy to produce a placement solution.

Table A.2. *Pod PromQL Queries*

Description	Query
Pod Requested CPU	sum(kube_pod_container_resource_requests_cpu_cores) by (pod)
Pod Requested RAM	sum(kube_pod_container_resource_requests_memory_bytes) by (pod)
Pod size of Requested Messages in Bytes	istio_request_bytes_sum{response_code = '200', connection_security_policy = 'mutual_tls', source_app != 'unknown', destination_app != 'unknown'}
Pod size of Responded Messages in Bytes	istio_response_bytes_sum{response_code = '200', connection_security_policy = 'mutual_tls', source_app != 'unknown', destination_app != 'unknown'}
Pod size of Requested Messages	istio_request_bytes_count{response_code = '200', connection_security_policy = 'mutual_tls', source_app != 'unknown', destination_app != 'unknown'}
Pod size of Responded Messages	istio_response_bytes_count{response_code = '200', connection_security_policy = 'mutual_tls', source_app != 'unknown', destination_app != 'unknown'}

A.2 Kiali Graph

Kiali service communicates internally with Prometheus to acquire all the collected data stored in it and produce the application graph for each application. The Kiali graph displays the application workloads and services, the protocol of communication, the traffic rates between the application's microservices and their affinities and finally the health status of each respective component. In Table A.3, we present the associated the symbology of the Kiali graph.

Table A.3. *Kiali Graph Symbology*

Symbol/Color	Explanation
Grey Rectangle	Kubernetes Workload (Pod) for a Microservice
Grey Triangle	Kubernetes Service for a Microservice
Green Edge	HTTP/gRPC Communication
Blue Edge	TCP Communication
Purple Arrow Symbol	Module applying HTTP requests

Upon each edge, the traffic rates of the communication between two services is depicted and it corresponds to the mean value of network communication for the selected time range of the graph. The traffic rates, which are displayed upon each edge, are used to calculate the RPS performance measure. Image A.2 displays the produced Kiali graph for the OnlineBoutique eShop application for a given time range. The produced image is associated with a unique timestamp of capturing the graph.

In Image A.3, the Kiali graph of the iXen application is displayed.

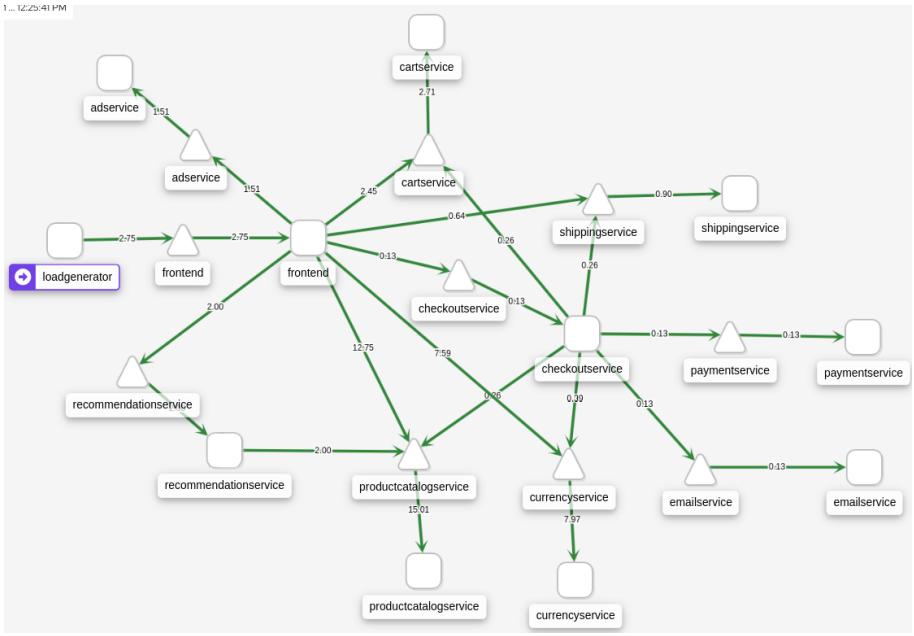

Image A.2: *Kiali Graph for OnlineBoutique eShop application*

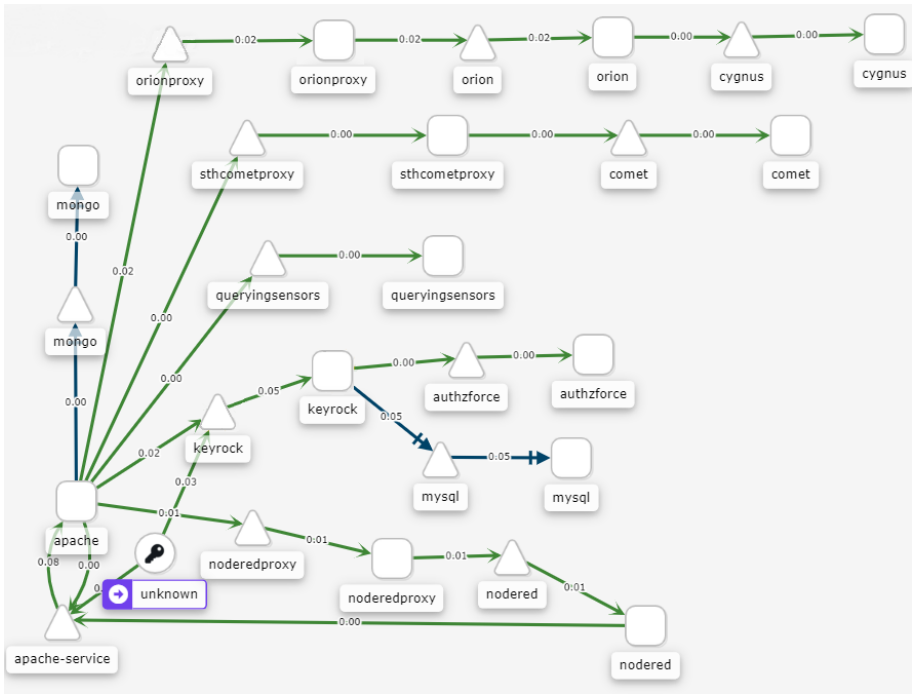

Image A.3: *Kiali Graph for iXen application*

We access the Kiali API and request each application's graph in a JSON file for a duration of 30 minutes. Kiali's Service is also converted to NodePort Service type and connected to an available port in GCP to access the API and collect the graphs. The API GET command requires the URL of the Kiali API, the headers, if exist, and the type of Data (duration, namespace, graphType) that we require to collect. We process the JSON file to collect the traffic rates from each communication edge and we store them in Python dictionaries so as to be utilized from the placement strategies.

A.3 Grafana Visualization

Another service that gathers data from Prometheus is Grafana. Grafana is a visualization service for PromQL queries and users can insert the desired PromQL queries and produce visually the application graphs about resource utilization for different timestamps. Users on Grafana can create their own visualization monitor system or import some predefined templates available on the Grafana website. In this Thesis, template with ID 11074 will be imported, which is a template to fetch the data from Node Exporters that are installed and initialized on each Node of the cluster. Image A.4 displays the acquired Node data of the Cluster for the last 15 minutes until the time of screen capture for all the availed Nodes in the Kubernetes cluster of the OnlineBoutique eShop application. Grafana's Service is also configured properly in GCP to access the Grafana UI from a specific network port.

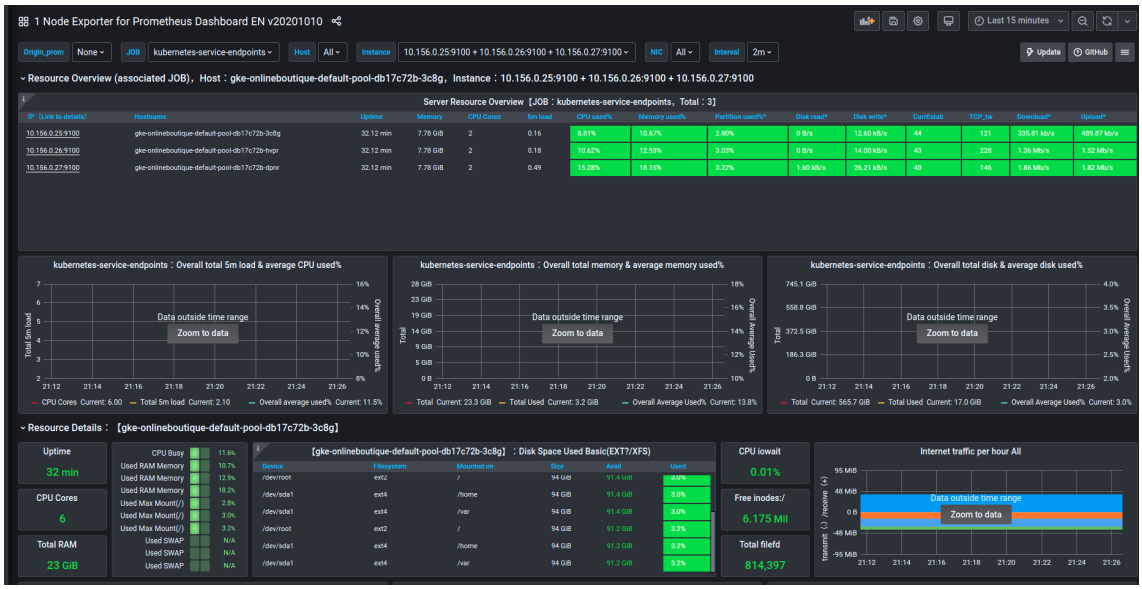

Image A.4: *Grafana Node Data*

Image A.5 displays the Node resources graphs about CPU, RAM, Storage and other cluster's I/O operations.

By importing this template on the Grafana UI, we can easily acquire the PromQL queries for each visualized graph depicted on the above images, store them and use these queries into the source code of the Thesis executable program to collect the Node data about CPU and RAM usage, allocated and available space, which will be processed by the placement strategies. Another benefit of the Nodes' visualization is that we can monitor the behavior and health of the application's Nodes under a Stress testing.

Image A.5: *Grafana Resource Graphs and I/O Operations*

Bibliography

[1] *Microservices*. https://microservices.io. Date inspected: 13-09-2021.

[2] *Kubernetes Components*. https://kubernetes.io/docs/concepts/overview/components/. Date inspected: 27-08-2021.

[3] *Kubernetes Scheduling Process*. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/. Date inspected: 16-11-2021.

[4] *Istio Service Mesh*. https://istio.io/latest/docs/. Date inspected: 12-08-2021.

[5] *Kiali*. https://kiali.io/documentation/latest/features/#_overview. Date inspected: 12-08-2021.

[6] *Google OnlineBoutique eShop Application*. https://github.com/GoogleCloudPlatform/microservices-demo. Date inspected: 10-08-2021.

[7] Ameni Hedhli, Haithem Mezni. *A Survey of Service Placement in Cloud Environments*. *Journal of Grid Computing*, 2021.

[8] Farah Ait Salaht, Frédéric Desprez, Adrien Lebre. *An overview of service placement problem in Fog and Edge Computing*. *ACM Computing Surveys, Association for Computing Machinery*, 2020.

[9] Kuo Chan Huang, Bo Jun Shen. *Service deployment strategies for efficient execution of composite SaaS applications on cloud platform*. *ELSEVIER, The Journal of Systems and Software*, 2015.

[10] Deval Bhamare, Mohammed Samaka, Aiman Erbad, Raj Jain, Lav Gupta, H. Anthony Chan. *Multi-objective scheduling of micro-services for optimal service function chains*. *ResearchGate*, 2017.

[11] Vajiheh Farhadi, Fidan Mehmeti, Ting He, Tom La Porta, Hana Khamfroush, Shiqiang Wang, Kevin S Chan, Konstantinos Poularakis. *Service Placement and Request Scheduling for Data-Intensive Applications in Edge Clouds*. *IEEE/ACM Transactions on Networking*, 2020.

[12] Hemant Kumar Apat, Bibhudatta Sahoo, Prasenjit Maiti. *Service Placement in Fog Computing Environmen*. *2018 International Conference on Information Technology (ICIT)*, 2018.

[13] Samodha Pallewatta, Vassilis Kostakos, Rajkumar Buyya. *Microservices-based IoT Application Placement within Heterogeneous and Resource Constrained Fog Computing Environments*. *IEEE/ACM 12th International Conference on Utility and Cloud Computing (UCC '19), December 2–5, 2019, Auckland, New Zealand. ACM, New York, NY, USA*, 2019.

[14] Wang Z., Liu H., Han L., Huang L., Wang K. *Research and Implementation of Scheduling Strategy in Kubernetes for Computer Science Laboratory in Universities*. *MDPI, Journal Information*, 2021.

[15] Adalberto R. Sampaio Jr., Julia Rubin, Ivan Beschastnikh, Nelson S. Rosa. *Improving microservice-based applications with runtime placement adaptation*. *Journal of Internet Services and Applications*, 2019.

[16] Yang Hu, Ceesde Laat, Zhiming Zhao. *Optimizing Service Placement for Microservice Architecture in Clouds*. *MDPI*, 2019.

[17] *Bin Packing Problem*. `https://en.wikipedia.org/wiki/Bin_packing_problem`. Date inspected: 28-08-2021.

[18] Kumaraswamy S., Mydhili K. Nair. *Bin packing algorithms for virtual machine placement in cloud computing: a review*. *ResearchGate*, 2019.

[19] *Minimum K-Cut*. `https://en.wikipedia.org/wiki/Minimum_k-cut`. Date inspected: 28-08-2021.

[20] *Karger's Algorithm*. `https://en.wikipedia.org/wiki/Karger%27s_algorithm`. Date inspected: 28-08-2021.

[21] *Service Affinity*. `https://www.ibm.com/docs/tr/pcfs/1.1?topic=services-service-affinity`. Date inspected: 13-09-2021.

[22] Shreya Banerjee, Ankit Choudhary, Somnath Pal. *Empirical Evaluation of K-Means, Bisecting K- Means, Fuzzy C-Means and Genetic K-Means Clustering Algorithms*. *ResearchGate*, 2015.

[23] Jian Di, Xinyue Gou. *Bisecting K-means Algorithm Based on K-valued Self-determining and Clustering Center Optimization*. *Journal of Computers*, 2017.

[24] *Kubernetes Introduction*. `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. Date inspected: 27-08-2021.

[25] *Kubernetes Scheduler*. `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/`. Date inspected: 17-08-2021.

[26] *Pod and Node Affinities/Anti-Affinities*. `https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-affinity-and-anti-affinity`. Date inspected: 16-11-2021.

[27] *Service Mesh explanation*. https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh. Date inspected: 08-11-2021.

[28] *Prometheus*. https://prometheus.io/docs/introduction/overview/. Date inspected: 13-08-2021.

[29] *Grafana*. https://grafana.com/grafana/. Date inspected: 13-08-2021.

[30] *Apache JMeter*. https://jmeter.apache.org. Date inspected: 23-09-2021.

[31] *Request per Second (RPS)*. https://en.wikipedia.org/wiki/Queries_per_second. Date inspected: 10-11-2021.

[32] *Graph Construction in Python*. https://www.geeksforgeeks.org/generate-graph-using-dictionary-python. Date inspected: 20-09-2021.

[33] Xenofon Koundourakis and Euripides G.M. Petrakis. *iXen: context-driven service oriented architecture for the internet of things in the cloud*. *Science Direct*, 2020.

[34] Koundourakis Xenofon. *Design and Implementation of service oriented architecture for deploying IoT applications in the cloud*. Διπλωματική εργασία, Electronics and Computer Engineering, Technical University of Crete, 2019.

[35] *gRPC Protocol*. https://grpc.io/docs/what-is-grpc/introduction/. Date inspected: 10-08-2021.

[36] *GCP Pricing*. https://cloud.google.com/compute/all-pricing. Date inspected: 01-10-2021.

List of Abbreviations

SP	Service Placement
VM	Virtual Machine
OS	Operating System
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
TLS	Transport Layer Security
TCP	Transmission Control Protocol
HTTP	HyperText Transfer Protocol
API	Application Programming Interface
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
UI	User Interface
CPU	Central Processing Unit
RAM	Random Access Memory
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
DAG	Directed Acyclic Graph
RPS	Requests per Second
WBA	Weighted Bidirectional Affinity
BP	Binary Partition
KP	K-Partition
BKM	Bisecting K-Means
HP	Heuristic Packing
SOA	Service Oriented Architecture