TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

INTELLIGENT SYSTEMS LABORATORY

MSc THESIS

Design and Evaluation of Microservice Placement Strategies in Cloud Infrastructures

Tsakos Konstantinos

COMMITTE:

Petrakis G.M. Euripides Professor, ECE, TUC (supervisor)

Deligiannakis Antonios Professor, ECE, TUC

Samoladas Vasileios Assoc. Professor, ECE, TUC

CHANIA 2025

ABSTRACT

The design and evaluation of microservice placement strategies in cloud infrastructures is crucial for optimizing resource utilization, reducing operational costs, and ensuring high performance. This thesis explores the Service Placement problem on cloud environments, with a focus on improving resource allocation and minimizing egress traffic. Service Placement is modeled as a graph clustering problem, and various clustering algorithms are investigated—specifically **Affinity Propagation**, **Maximum Standard Deviation Reduction (MSDR)**, and **Markov Clustering**—and are combined also with a placement strategy called **Heuristic Packing** to develop efficient service placement solutions.

The study is based on two benchmark microservice applications, **iXen** (IoT prototype) and **Online Boutique** (e-commerce platform), deployed on Kubernetes clusters in a Google Cloud environment. Through load stressing, criteria like the performance of different placement strategies in terms of node utilization, egress traffic reduction, execution time, and cost efficiency are evaluated. The results show that **Affinity Propagation with Heuristic Packing** and **Maximum Standard Deviation Reduction with Heuristic Packing** consistently outperform other strategies, offering low resource utilization, reduced egress traffic, and minimal costs in total.

The findings suggest that while **Affinity Propagation** provides fast execution, making it suitable for dynamic environments, **MSDR** offers superior long-term optimization at the expense of execution time. These strategies are recommended for applications with high inter-service communication and varying traffic loads. This work contributes to the field by providing insights into the application of clustering algorithms for microservice placement and suggests future directions for integrating machine learning and adaptive strategies to further optimize service deployment in cloud-based systems.

ΠΕΡΙΛΗΨΗ

Ο σχεδιασμός και η αξιολόγηση στρατηγικών τοποθέτησης μικρο-υπηρεσιών(microservices) σε cloud υποδομές είναι κρίσιμα για τη βελτιστοποίηση της χρήσης των πόρων, τη μείωση των λειτουργικών εξόδων και τη διασφάλιση υψηλής απόδοσης. Αυτή η διατριβή εξετάζει το πρόβλημα της τοποθέτησης Υπηρεσιών σε Cloud περιβάλλοντα, με στόχο τη βελτίωση της κατανομής των πόρων και την ελαχιστοποίηση της εξαγόμενης κίνησης (egress traffic). Η τοποθέτηση Υπηρεσιών μοντελοποιείται ως ένα πρόβλημα ομαδοποίησης κόμβων ενός γράφου, και διερευνόνται διάφοροι αλγόριθμοι ομαδοποίησης—συγκεκριμένα οι **Affinity Propagation**, **Maximum Standard Deviation Reduction (MSDR)** και **Markov Clustering**—τόσο μεμονωμένα όσο και σε συνδυασμό με μια στρατηγική τοποθέτησης ονόματι **Heuristic Packing** ένα είδος **Bin Packing αλγορίθμου**, για την ανάπτυξη αποδοτικών λύσεων τοποθέτησης υπηρεσιών.

Η μελέτη βασίζεται σε δύο εφαρμογές αναφοράς βασισμένων σε microservices, το **iXen** (πρωτότυπο IoT) και το **Online Boutique** (πλατφόρμα ηλεκτρονικού εμπορίου), οι οποίες έχουν αναπτυχθεί σε **Kubernetes clusters** στο **Google Cloud**. Μέσω φόρτισης του συστήματος (load stressing), αξιολογούνται κριτήρια όπως η απόδοση των διαφορετικών στρατηγικών τοποθέτησης ως προς τη χρήση των κόμβων, τη μείωση της εξαγόμενης κίνησης, το χρόνο εκτέλεσής τους στην λήψη απόφασης και η οικονομική αποδοτικότητα. Τα αποτελέσματά δείχνουν ότι ο **Affinity Propagation** αλγόριθμος σε συνδυασμό με **Heuristic Packing** και ο **Maximum Standard Deviation Reduction** σε συνδυασμό με **Heuristic Packing** ξεπερνούν σταθερά άλλες στρατηγικές, προσφέροντας χαμηλή χρήση πόρων, μειωμένη εξαγόμενη κίνηση και ελάχιστα κόστη συνολικά.

Τα ευρήματα υποδεικνύουν ότι, ενώ ο **Affinity Propagation** προσφέρει ταχεία εκτέλεση, καθιστώντας τον κατάλληλο για δυναμικά περιβάλλοντα, ο **MSDR** προσφέρει ανώτερη βελτιστοποίηση μακροπρόθεσμα εις βάρος του χρόνου εκτέλεσης. Αυτές οι στρατηγικές συνιστώνται για εφαρμογές με υψηλή επικοινωνία μεταξύ των υπηρεσιών και μεταβαλλόμενα φορτία κίνησης. Αυτή η εργασία συνεισφέρει στον τομέα προσφέροντας γνώση σχετικά με την εφαρμογή αλγορίθμων ομαδοποίησης για την τοποθέτηση microservices και προτείνει μελλοντικές κατευθύνσεις για την ενσωμάτωση τεχνητής νοημοσύνης και προσαρμοστικών στρατηγικών, με στόχο την περαιτέρω βελτιστοποίηση της τοποθέτησης υπηρεσιών σε cloud υποδομές.

ACKNOWLEDGMENTS

I would like to thank professor Euripides Petrakis for his valuable help and insightful comments. Also, I would like to thank the Graduated Student Alkis Aznavouridis for his contribution in this study. Finally, I would like also to thank the members of the laboratory for the excellent communication and collaboration.

1. Introduction

Service Placement problem in Cloud and Edge Computing environments has attracted the interest of many researchers, as a service placement strategy with an optimal solution is vital both for Cloud Users and Cloud Providers.

More and more enterprises are using **Cloud Computing** Technology in order to be more competitive increasing their speed to market, to avoid IT maintenance costs and to become scalable to customer needs. Cloud providers charge their customers by the amount of resources they allocate per a unit of time, such as number of CPUs per hour, Memory Usage per Hour and Storage in order to deploy their own services. In addition, there are also charges for Egress Traffic, i.e the traffic from a virtual machine to another, which is proportional to the size of data exchanged and to the number of the networking hops required to reach its destination. So, an optimal service placement strategy is crucial for minimizing users' expenses and for reducing applications' latency.

An optimal service placement solution is also important from the Cloud Provider's scope. They should effectively release and place their services in order to achieve high performance without infrastructure wastes. The solution takes into consideration challenges such as the geographical infrastructure heterogeneity and each service's defined specifications for placement.

Furthermore, the growing adoption of 5G networks has revolutionized how services are delivered, enabling ultra-low latency, high throughput, and massive device connectivity. These advancements are further augmented by technologies such as Network Function Virtualization (NFV) and Software-Defined Networking (SDN). NFV allows traditional hardware-based network functions to be virtualized and deployed flexibly on general-purpose hardware, while SDN provides programmable control over network operations, enabling dynamic and efficient traffic management. Cloud vendors are leveraging these technologies to reduce infrastructure expenses and enhance service distribution capabilities. For instance, NFV facilitates deploying Virtual Network Functions (VNFs) like firewalls, load balancers, and intrusion detection systems closer to end-users in edge environments, improving Quality of Service (QoS). Similarly, SDN supports dynamic path optimization, which is essential for latency-sensitive applications such as augmented reality (AR), autonomous vehicles, and remote surgery. However, achieving these benefits necessitates robust placement strategies for VNFs that consider factors like network latency, resource availability, and traffic patterns to ensure optimal service delivery to cloud users.

The **Service Placement Problem** has been studied from multiple and diverse perspectives, influenced by the complexity and heterogeneity of cloud environments. These environments include **Single Cloud**, **MultiCloud**, **Cross Cloud**, **Federated Cloud**, **Hybrid Cloud**, **Mobile Cloud**, and **Edge/Fog Cloud**, each presenting unique challenges and opportunities for service placement.

Service deployment modes can be classified as static or adaptive (also called dynamic). **Static placement** involves predefined service locations without runtime reconfiguration. It is suitable for environments where workloads and resource requirements are predictable, such as Single Cluster systems in one region with stable workloads. Static strategies are simpler and require minimal management overhead, but they lack flexibility to respond to changing conditions like load spikes or failures. **Adaptive or dynamic placement**, in contrast, involves real-time adjustments based on changing workloads, resource availability, or environmental factors. For instance, adaptive placement may relocate microservices to handle increased demand or recover from node failures. This mode is better suited for **Multi-Cluster** deployments across different regions or **Single Zone Multiple Clusters**, where dynamic conditions necessitate responsiveness to ensure performance and resource efficiency.

Placement strategies can also be categorized as reactive or predictive. **Reactive placement** responds to events or changes after they occur, such as reallocating services when a cluster becomes overloaded or a node fails. While simpler to implement, reactive strategies may introduce delays in achieving an optimal state, which can affect time-sensitive applications. **Predictive placement**, on the other hand, uses data analytics, machine learning, or historical workload patterns to forecast future demands and proactively place services. For example, predictive placement is ideal for scenarios involving **IoT services or BPaaS**, where demand spikes can be anticipated (e.g., during peak business hours). It is particularly valuable for **Multiple-Zone Single Clusters** or **Multi-Region Clusters**, as forecasting demand enables proactive resource allocation across zones or regions.

The suitability of each placement solution varies according to the deployment context. For single Clusters (one region), Static placement strategies can often suffice due to the limited geographical scope and relatively predictable workload patterns. Adaptive placement may still be applied if resource utilization optimization is critical or when workloads vary significantly. For multiple clusters (in same or different regions), adaptive or predictive placement becomes essential to manage latency, load balancing, and resource allocation across geographically distributed clusters. These solutions account for inter-cluster communication overhead and regional infrastructure differences. For single zone (single cluster) scenario, often benefits from static or reactive placement, as the geographical and resource scope is narrow. However, predictive placement may still add value for optimizing resource utilization over time. For multiple-zone single clusters, predictive or adaptive strategies are particularly advantageous, as zones within the same cluster may experience variable workloads or outages. Placement solutions need to minimize latency across zones while balancing the cost of inter-zone communication.

In addition to these considerations, placement strategies must evaluate criteria like security, latency, resource usage, cost, energy consumption, and application-specific requirements. For instance, services like **IoT applications** might prioritize low latency, while **BPaaS solutions** may focus on scalability and cost efficiency.

Microservices seem to be the most convenient architectural pattern for Service packaging and deployment. They are small and lightweight software units, easy and fast deployable on heterogeneous infrastructures. Software development becomes more effective, as Developer teams can divide the functional requirements of their application on each microservice and combine them later in order to succeed the overall functionality. Microservices provide such opportunities and they collaborate with each other to conduct each functionality via http communication exposing APIs via REST architectural pattern or gRPC protocol.

The continuous development of Microservice Based Software Architectures created some demands such as continuous monitoring and continuous managing and orchestration. Container Orchestrators (e.g Kubernetes) can deploy, scale, monitor and make microservices communicate each other via networking configuration

In this study, we apply different placement strategies on realistic Microservice Based Architectures deployed on Single Cloud via Kubernetes Orchestrator. Application services form graphs with nodes representing services and edges representing their affinities. Nodes and edges are labeled by the resources consumed (i.e. CPU and RAM) and by the affinities between services (i.e. number and size of requests within a time period), respectively. Graph clustering in the service application graph guides service placement. We chose promising clustering algorithms having already been applied successfully in different clustering problems combining them also with a bin packing solution to achieve both a great application performance through minimum latency during microservices communication and minimum cost through minimum resource allocation.

1.1 Problem Definition

Cloud providers charge customers for the computing resources they allocate and the outbound traffic propagated among Virtual Instances. Most of the times, DevOps Engineers analyze the Application's Operational Requirements and choose the appropriate amount of resources to be allocated. After the resource allocation, they use a Container Orchestrator to deploy and manage their apps. The placement scheduling by the orchestrator, does not contain any intelligence. For example, Kubernetes applies a Round Robin Logic deploying each microservice in the first node that it could fit related to its requested resources for Memory and CPU. An alternative second placement strategy that Kubernetes supports is the node selector and pod affinity and antiaffinity logic. These features give the opportunity to define for each pod in which Node must or should be deployed, or with which pod must or must not (should or should not) be deployed together. After choosing on which node a Pod will be deployed, Kubernetes Scheduler never replaces it. This happens only if a node fails and it seeks a node to replace its missing pods.

We are motivated by the idea for a more intelligent placement strategy which would place the microservices taking into consideration such as how often they talk to each other, how big are the messages they exchange and how to minimize the infrastructure

they allocate. All these choices will be made with an adaptive way based on the workload variations and the traffic distribution the application receives.

The main idea is to put microservices with high affinity in the same host machine if a suitable capacity is left. High affinity microservices form a cluster from each clustering algorithm scope. A post-processing step following graph clustering will attempt to further minimize the number of virtual machines (and therefore the amount of computational resources) by placing as many clusters as possible on each virtual machine. As a result, the process will also minimize the Egress traffic that occurs in communication between microservices on different virtual machines.

The present work compares the performance of placement by different clustering algorithms. For simplicity purposes, we constraint the problem in a Centric Cloud Environment, using Google Cloud Platform (GCP,) deploying the apps with Google Kubernetes Engine (GKE). When a GKE cluster is created, at least one default node pool is created by default. A node pool is a group of Compute Engine instances (VMs) that serve as the worker nodes in the Kubernetes cluster. Our solutions can be applied to clusters either having single or multiple node pools with different configurations for their worker machines (e.g. machine types, sizes). However, they are not applicable to clusters having autopilot mode enabled. In such a case, clusters can be used without directly managing node pools, but leaving GCP taking care of the infrastructure, including provisioning and scaling nodes automatically. Our solutions, have not been tested in clusters in different zones and regions. The deployment would introduce additional delays in the communication between the VMs that graph clustering cannot take into consideration.

To show proof of concept, we apply graph clustering on two Microservices based Architectures, one IoT prototype called iXen and a second one called OnlineBoutique E-Shop. We deploy each one on a different Kubernetes Cluster in a Google's Cloud Platform environment being orchestrated by Google Kubernetes Engine Service. After load stressing, we collect some meaningful metrics exploiting Istio. Based on the messages observed among microservices we construct a weighted graph with the weight values of the edges representing the affinity between each couple of connected microservices. We apply each clustering algorithm in each Architecture's representative Graph. At a second step, we apply a bin packing strategy to the clusters of the first step trying to further reduce the overall resource allocation.

Finally, we present meaningful results relatively with the final realistic monthly charges applied by the Cloud Provider based on the resource demands and the Egress traffic and also we observe the performance of each one strategy based on how fast they return a placement decision.

2. Related Work

As the bibliography is extended in many different scopes of the Service Placement Problem, in this study we are going to concentrate only to the Clustering Algorithms

having been applied on other domains trying to check if we can adapt them in our use case scenario, in order to extract dense clusters with microservices with very frequent communication.

In the first subsection we give an overview of recent studies related with service placement problem and how they approach it.

In the second subsection we describe clustering algorithms that have not yet been applied on Service Placement decisions.

In the third subsection, we describe some placement algorithms based on heuristic methods which have been applied as Service Placement strategies and which we could combine with the clustering algorithms to check if they can improve further the final service placement decisions as a second step.

Finally, in the fourth subsection, we present some useful technologies, architectural patterns and monitoring tools we will use to implement, apply and compare all different service placement strategies.

2.1 Recent Studies

Graph clustering belongs to the NP-Hard problems which means that it is as hard as the hardest problems of NP type, i.e whose solution can be verified as valid in polynomial time by a deterministic algorithm. NP-Hard problems are solved in polynomial time with heuristic methods [47], approximation algorithms [48], greedy algorithms [49], exact algorithms [50] or metaheuristics [51] without optimality guarantees. Lately, there are also studies based on Machine learning techniques and optimization-based frameworks, each aiming to balance computational efficiency with solution quality.

Placement techniques can be categorized in **static** and **dynamic**. In the static ones, the placement of services is determined initially and remains fixed during the operation. For the dynamic ones, the location of services is continuously adapted based on real-time data, such as fluctuating workloads, network traffic, user demands or resource availability.

Another categorization of recent studies comes according to the cluster's nature. Some of them use single clusters to apply their placement strategies while others support service placement in multiple clusters. For the single cluster studies, we can find strategies covering placement either in **common homogeneous** clusters or in **heterogeneous** clusters which are shaped by heterogeneous host units, i.e interconnected computers or nodes that have varying configurations, capabilities, and resources such as different CPU types, memory capacities, storage or even operating system.

Marchese et al. (2022) [33] propose a communication-aware scheduling approach for Kubernetes clusters that aims to reduce end-to-end latencies by scheduling microservices with high communication affinity on the same node. Using a custom scoring scheduler, they assign scores to nodes based on communication affinities and

network latencies, optimizing microservices placement to enhance response times for external users. TOSCA-based graph modeling is utilized to represent application topology, adjusting scoring parameters to prioritize either communication protocols or traffic history, depending on network demands. It is a single-cluster study optimizing response times by dynamically adjusting node scores according to communication affinities, despite an otherwise static framework.

Hu et al. (2018) [34] present ECSched, an efficient graph-based algorithm for container scheduling on single but heterogeneous clusters, which significantly outperforms existing schedulers in placement quality. The authors model deployment requirements as a minimum cost flow problem, achieving improved resource efficiency and speeding up container deployment. In their simulation-based experiments, ECSched demonstrated the capability to place 300 containers in just 3.4 seconds, highlighting its efficiency in large-scale environments. Future work will involve addressing container dependencies and adapting to resource dynamics, further enhancing the algorithm's applicability in heterogeneous environments.

A dynamic placement work in single clusters by Zhu et al. (2023) [35] explore traffic scheduling optimization for multi-replica containerized, stateless or stateful, microservices introducing a network-aware scheduling system called OptTraffic, which enhances high concurrency and fault tolerance. The authors implement a lightweight traffic estimation technique between container pairs, combining simple mathematical calculations with coarse-grained monitoring to optimize performance. The scheduling optimization problem is modeled as a Directed Acyclic Graph, allowing for a structured approach to traffic management and resource allocation in containerized environments. OptTraffic employs a local-first allocation scheme that minimizes cross-machine traffic while dynamically adjusting the deployment of containers to achieve load balancing and efficient resource utilization. In their evaluation, Zhu et al. (2023) demonstrate that their approach significantly reduces cross-machine traffic by 47%, leading to a 28-45% decrease in p99 latency for microservices without compromising resource usage balance. While this dynamic placement solution improves performance through reduced cross-machine traffic and latency, it does not focus on minimizing resource allocation, indicating a trade-off in this aspect.

Petrakis et al. (2023) [36] introduce the ModSoft-HP Scheduler, which employs fuzzy clustering for microservices placement in Kubernetes, achieving significant reductions in egress traffic and hosting costs. The authors report an impressive 90% reduction in egress traffic, a 20% decrease in response time, and 25% lower hosting costs when using ModSoft-HP compared to traditional scheduling algorithms. Unlike hard clustering algorithms that partition datasets, the fuzzy clustering approach utilized in ModSoft-HP calculates the probability of each element belonging to a cluster, enabling more flexible and efficient resource allocation. By leveraging fuzzy clustering combined with heuristic packing, ModSoft-HP effectively balances load and accelerates application response times, demonstrating superior performance compared to the default Kubernetes scheduler and other algorithms. The ModSoft algorithm operates in three stages: initialization of the membership matrix, calculation of the membership matrix, and

partitioning, ultimately producing fuzzy partitions for optimized microservices placement. Experimental evaluations conducted on realistic applications, including an IoT solution and an e-commerce platform within a real Kubernetes infrastructure, showcase the effectiveness of the proposed scheduler. The cost reduction achieved by ModSoft-HP is particularly beneficial for edge cloud environments, where it can further reduce latency and hosting expenses.

Senjab et al. (2023) [37] provide a comprehensive survey of Kubernetes scheduling algorithms, categorizing the literature into four sub-categories: generic scheduling, multi-objective optimization-based scheduling, AI-focused scheduling, and autoscaling. Through a thorough screening of 124 studies, the authors narrowed their focus to 47 key studies that specifically address Kubernetes scheduling algorithms, highlighting existing gaps in the research. The literature reviewed emphasizes the potential benefits of advanced scheduling algorithms in Kubernetes, including improved resource utilization and reduced latency, which are critical for optimizing cloud computing performance. Innovative scheduling algorithms such as Optimus, RUBAS, and Libra are proposed in the survey, showcasing their effectiveness in enhancing resource management and autoscaling capabilities within Kubernetes clusters. The authors note that methods like Dynamic Multi-level Autoscaling, RUBAS, and Libra demonstrate significant improvements in CPU and memory utilization, as well as reductions in runtime and response times, thereby enhancing overall system performance. They also highlight the advantages of integrating machine learning forecasting methods in Kubernetes scaling engines, which lead to better autoscaling decisions for cloud-based applications. The paper identifies key challenges in Kubernetes scheduling, such as finding optimal solutions and adapting to dynamic environments, while suggesting future research directions including advanced computation optimization and context-aware scheduling algorithms. The authors also discuss limitations and potential improvements within specific contexts like green computing, indicating a growing need for sustainable resource management practices in Kubernetes environments.

Edirisinghe et al. (2024) [38] introduce SpotKube, an open-source solution designed for cost optimization in microservices deployment on public clouds, specifically utilizing spot pricing options to enhance cost-effectiveness. The study highlights the development of an elastic cluster autoscaler powered by a genetic algorithm, which effectively balances cost and performance in real public cloud setups. Experimental results indicate that SpotKube can significantly reduce cloud costs while meeting user-defined service level objectives (SLOs) and ensuring optimal application performance. The system incorporates an Optimization Engine that predicts spot prices, optimizes node combinations, and calculates deployment costs, demonstrating a robust approach to resource allocation in Kubernetes clusters. Load testing with microservice applications including heavy traffic, database accesses, web access and compute-intensive tasks, coupled with monitoring tools like Prometheus, showed the effectiveness of SpotKube in managing network traffic and response times under various conditions. The research demonstrates a 25% cost saving with SpotKube compared to traditional Amazon EKS configurations using spot instances, highlighting its potential for cost-efficient microservices deployment. Looking ahead, the authors suggest enhancing SpotKube's

capabilities to support heterogeneous pod deployment, which could further improve resource utilization and cost savings in cloud environments. The study emphasizes the importance of application characterization and analytical modeling in determining the optimal number of pods based on SLOs and application behavior, which is crucial for efficient resource allocation.

Kaur et al. (2023) [39] explore the dynamic migration of microservices in 5G and 6G networks, highlighting the telecom industry's shift toward decomposing complex functions into containerized microservices to reduce service latency. The authors emphasize the importance of optimization criteria such as load balancing and latency control within distributed and virtualized architectures, which are critical for effective service function chains in telecom environments. Using Deep Reinforcement Learning techniques, the study addresses the dynamic placement of microservices, allowing for continual modifications to enhance performance metrics such as latency. The paper introduces a novel model for migrating microservices across heterogeneous cloud architectures, utilizing three heuristics that focus on optimizing placement, reducing runtime, and improving end-to-end latency during the migration process. They demonstrate a model structured as a three-layer tree, consisting of edge, regional, and centralized clouds, where latency is calculated based on geographic distribution and message exchange delays. The study compares different algorithms, such as the Greedy First Fit (GFF) and Greedy Best Fit (GBF) algorithms, for the placement of microservices in data centers, analyzing their effectiveness in minimizing latency and fragmentation. Simulation results demonstrate that the proposed migration strategy significantly outperforms static placement strategies, effectively reducing latency for microservices in dynamic network environments. The authors suggest that future work could incorporate deep learning approaches to further enhance microservices migration strategies, potentially leading to even greater latency reductions.

Attaoui et al. (2023) [40] provide a comprehensive review of recent advances in Virtual Network Function (VNF) and Cloud Native Function (CNF) placement within 5G networks, highlighting optimization techniques and the challenges associated with virtualizing network functions. The authors note that operators are increasingly transitioning to virtualized 5G core networks to achieve lower latencies and higher throughput, facilitated by technologies such as Software-Defined Networking (SDN) and Network Functions Virtualization (NFV). Network slicing is emphasized as a key feature of 5G deployment, enabling personalized connectivity services and allowing for dynamic network adjustments based on customer needs. The paper discusses the benefits of virtualization in 5G networks, including reduced latency, increased scalability, and improved energy efficiency, which are crucial for accommodating diverse service requirements. Research in the paper focuses on optimal VNF and CNF placement in various computing environments, such as cloud, edge, and fog computing, which play significant roles in reducing latency and improving service performance. The authors highlight the use of genetic algorithms and reinforcement learning as effective strategies for optimal VNF placement, addressing challenges such as energy consumption, cost reduction, and resource utilization. Looking ahead, the paper suggests that future research may focus on security-aware schedulers and the application of deep learning

algorithms to enhance scheduling policies for VNF and CNF placement in 5G networks. Attaoui et al. (2023) also emphasize the significance of multi-objective optimization approaches in solving complex placement problems, which are crucial for optimizing resource allocation while meeting diverse performance metrics.

Vasireddy et al. (2023) [41] provide a detailed review of Kubernetes load balancing solutions aimed at enhancing resource utilization and cluster efficiency. The authors categorize Kubernetes scheduling algorithms and critically evaluate the strengths and limitations of existing literature, highlighting areas for improvement in dynamic resource allocation. The paper discusses the integration of AI-driven strategies for scheduling in Kubernetes, which employ multi-criteria optimization to balance objectives and improve cluster performance. Vasireddy et al. (2023) highlight challenges in AI-driven scheduling, noting that a lack of real-world datasets limits the effectiveness of AI models, which must balance accuracy and computational complexity. The survey addresses dynamic scaling in Kubernetes, emphasizing how techniques like RUBAS and other container migration strategies significantly improve resource management and application performance. The authors suggest that future research should focus on developing advanced context-aware scheduling algorithms, as well as exploring Kubernetes' integration with serverless computing to further enhance resource utilization. Machine learning-based scheduling has gained prominence in Kubernetes, enabling adaptive resource balancing within clusters and showing promising results for future developments. The comprehensive survey by Vasireddy et al. (2023) offers a roadmap for future research in Kubernetes scheduling, emphasizing the need for new algorithms and enhancements to current approaches.

Dakić et al., (2024) [42] present a new platform integrating Kubernetes with High Performance Computing (HPC) environments, focusing on dynamic workload placement and performance analysis using machine learning scheduling. The authors highlight the unique challenges of containerizing HPC workloads, including hardware allocation limitations and the complex architecture of Kubernetes for latency-sensitive applications. The study develops a custom ML-based Kubernetes scheduler to improve workload placement and performance, tested on a multi-node setup with both CPU and GPU resources. Machine learning integration in workload scheduling is central to the study, as ML algorithms dynamically manage resource allocation to optimize both performance and energy efficiency. The custom ML-based scheduler achieved over 15% performance improvement compared to the default Kubernetes scheduler, emphasizing its utility for HPC applications. While the study presents effective ML scheduling, limitations include the focus on NVIDIA GPUs and the lack of demonstrated energy savings, making it less applicable to non-HPC applications. The platform includes a user-friendly interface with a drag-and-drop system, simplifying HPC workload deployment on Kubernetes by integrating Clickhouse for monitoring and performance tracking. Future research aims to extend the scheduler's capabilities to support FPGA and ASIC resources, along with further automation for performance optimization.

Datta et al., (2024) [43] present a novel microservice scheduling strategy designed to enhance cloud performance by improving fault tolerance, reducing network traffic, and

lowering latency. This study leverages particle swarm optimization (PSO) and the Round Robin (RR) algorithm to optimize microservice placement and resource management within cloud environments. The authors address the challenges of network traffic and latency in cloud-based microservices by proposing a replication strategy that places microservices near their dependencies to reduce communication costs. The proposed strategy was tested in a simulated environment using Alibaba and Google datasets, measuring network traffic, latency, and load balancing across physical machines. Their results show a 36% reduction in network traffic and an 84% reduction in latency in the first scenario, outperforming Alibaba's standard scheduling methods. The strategy assumes a cluster environment managed by Kubernetes, employing modified PSO for replica allocation and Round Robin scheduling for load balancing, effectively improving resource utilization and availability. With a combined PSO and RR approach, the strategy achieves optimal load balancing, fault tolerance, and latency reduction, marking significant improvements over baseline strategies. The authors suggest future research to refine PSO and RR convergence rates and explore machine learning models like Q-learning for more adaptive microservice scheduling.

Chen et al., (2024) [44] propose the multi-objective microservice allocation (MOMA) algorithm, which uses a genetic algorithm to optimize resource allocation within a container-based heterogeneous cloud environment. The MOMA framework addresses resource allocation challenges in microservices architecture, such as network transmission costs and maintaining reliability in heterogeneous cloud systems. The proposed framework integrates four main components—including a Kubernetes Management Unit for efficient workload deployment—to optimize microservice allocation and enhance cloud-native services. MOMA's optimization model focuses on maximizing resource utilization across multiple clusters and reducing network transmission overhead, particularly for edge computing scenarios. With a population size of 200 and a termination criterion set at 25,000 generations, MOMA uses specialized crossover and mutation operations to avoid local optima, adapting resource allocation to diverse workloads. Experiments were conducted on a simulated cloud environment with three cluster types, where tools like Prometheus and Grafana monitored resource utilization and Locust distributed the workload. MOMA demonstrates superior performance compared to Multiopt, Greedy, and Kubernetes' default algorithms, achieving higher resource utilization, reduced network overhead, and increased reliability. Chen et al., (2024) bi-objective model optimizes resource use and minimizes network overhead, with plans for future research to incorporate GPU management and expand the heterogeneous infrastructure.

Hossain et al. (2024) [45] present a three-layer Microservice Lightweight Execution (MLE) framework for decentralized microservice placement in multi-tier Fog networks, prioritizing high-priority Edge-required microservices. The MLE framework enables Master Fog devices to prioritize Edge microservices by sorting Fog devices based on resources, thereby exempting Citizen devices from resource management tasks and allowing focus on essential microservices. Through strategies for microservice placement, scaling, and request escalation, each Master Fog device efficiently manages microservice demands across Citizen Fog devices. In simulations using iFogSim, the MLE

framework reduces Cloud dependency by one-third and lowers average application execution time by 65-70%, surpassing state-of-the-art frameworks in resource utilization efficiency. Future research plans include testing the MLE framework with real-world IoT applications and studying the impact of IoT device mobility on microservice request execution.

Keshavarz Haddadha et al. (2024) [46] review ML applications in service placement, noting that the problem's NP-hard nature limits classical optimization methods, making ML-driven heuristic approaches essential. Service placement in emerging paradigms like Multi-Edge Cloud (MEC) and Fog Cloud (FC) faces challenges like workload variability and service interdependencies, with ML—particularly reinforcement learning—addressing these through predictive insights based on historical data. The review provides a taxonomy of ML methods in service placement, highlighting that 81% of studies use multi-objective approaches and reinforcement learning comprises 56% of recent research. ML methods such as K-means, Federated Learning, and neural networks are used in service placement to enhance QoS, delay management, and energy efficiency, particularly in Internet of Everything (IoE) environments. Analysis indicates 96% of applications adopt distributed architectures, with reinforcement learning and deep Reinforcement Learning (RL) constituting nearly 30% of service placement methods, while on-demand resource estimation is used in 51% of studies. The IoE paradigm broadens the scope of service placement, emphasizing node heterogeneity, task correlation, and mobility, particularly for applications in Next Generation Critical Communication (NGCC) like disaster management and e-health. Future trends in ML for service placement include federated and transfer learning for enhanced data handling, and advancements in reinforcement learning to balance exploration and exploitation effectively.

The following table provides an overview of all various service placement methods in Kubernetes and cloud environments which are described in this section, detailing their placement type (static or adaptive), the underlying model used (e.g. graph-based, heuristics or machine learning), and the platforms where they have been applied (e.g. simulators, Kubernetes clusters, or specific cloud environments). The last one is our current study which is called Service Clustering for Optimal Resource Efficiency in Kubernetes (SCORE).

Reference Number	Method Name	Placement Type	Model	Platform
[33]	Communication-aware scheduling (Marchese et al.)	Static	Graph (TOSCA-based graph modeling; scoring based on communication affinity)	Kubernetes (single cluster)
[34]	ECSched (Hu et al.)	Static	Graph (Minimum cost flow problem)	Simulation-based experiments

[35]	OptTraffic (Zhu et al.)	Adaptive	Graph (Directed Acyclic Graph; traffic scheduling and local-first allocation)	Kubernetes (single cluster)
[36]	ModSoft-HP Scheduler (Petrakis et al.)	Static	Clustering (Fuzzy clustering with heuristic packing)	Real Kubernetes infrastructure (IoT and e-commerce applications)
[37]	Survey of Kubernetes Scheduling Algorithms (Senjab et al.)	Static and Adaptive	Non-Graph (Multiple heuristic and AI-based methods, e.g., RUBAS, Libra)	Not specific; general review of methods for Kubernetes environments
[38]	SpotKube (Edirisinghe et al.)	Adaptive	Heuristic (Genetic algorithm for cost-performance optimization)	Public cloud environments (Amazon EKS)
[39]	Dynamic Migration for 5G/6G Networks (Kaur et al.)	Adaptive	Non-Graph (Deep Reinforcement Learning for migration optimization)	Simulation (5G/6G heterogeneous cloud architectures)
[40]	VNF and CNF Placement in 5G (Attaoui et al.)	Static and Adaptive	Heuristic (Genetic algorithms and reinforcement learning)	Simulated 5G networks (cloud, edge, and fog computing)
[41]	Kubernetes Load Balancing Survey (Vasireddy et al.)	Adaptive	Non-Graph (AI-driven multi-criteria optimization)	Not specific; general review of Kubernetes scheduling methods
[42]	ML-Based Kubernetes Scheduler for HPC (Dakić et al.)	Adaptive	Non-Graph (Machine learning scheduling)	Multi-node setup with CPUs and GPUs for High-Performance Computing (HPC)
[43]	PSO and RR-Based Scheduling	Adaptive	Heuristic (Particle Swarm Optimization +	Simulated environment (Alibaba and

	Strategy (Datta et al.)		Round Robin scheduling)	Google datasets)
[44]	MOMA Algorithm (Chen et al.)	Adaptive	Heuristic (Genetic algorithm for multi-objective optimization)	Simulated cloud environment (Prometheus, Grafana, Locust monitoring tools)
[45]	MLE Framework for Fog Networks (Hossain et al.)	Adaptive	Non-Graph (Fog-based prioritization and scaling)	Simulation (iFogSim)
[46]	ML-Driven Heuristic Approaches for Service Placement (Keshavarz Haddadha et al.)	Adaptive	Heuristic (Reinforcement learning, K-means clustering, Federated Learning)	Not specific; general review of ML-driven approaches in MEC and FC
	Service Clustering for Optimal Resource Efficiency in Kubernetes (SCORE)	Static	Graph clustering algorithms (MSDR,AP, Markov) combined with Heuristic Bin Packing	Google Cloud Platform (GCP)

2.2 Clustering Algorithms

In this chapter, different clustering algorithms are analyzed and we explain why they could or could not be suitable for being applied to shape clusters of microservices with high affinity.

Clustering is an unsupervised machine learning technique which is used in order to extract knowledge about similar data included on a dataset. Clustering searches for patterns among data without any label addition in contrast to what supervised machine learning techniques (e.g classification) do. Clustering methods have been widely adopted in various use cases such as market segmentation [6], social network analysis [7], search result grouping [8], medical imaging [9], image segmentation [10] and anomaly detection [11].

Clustering algorithms are divided in some categories according to the definition of similarity between data points [52]. First of all, **connectivity model** considers data points closer in space to be more similar than others being farther away. Such models either put all points in different clusters and then aggregate them or classify all data points as a unique cluster and split them as a predefined subjective distance is increased. They provide an easy way to interpret the clusters but they do not perform well on big datasets. On the other hand,

centroid model matches a data point with a cluster if the centroid of this cluster is closer from all other cluster centroids to that data point. This model requires the number of clusters to be predefined, so a previous knowledge about the dataset is vital. Algorithms of this model (e.g K-Means) run until finding the local optima. Another clustering model called **Distribution model** is based on the idea of how possible is all data points of a cluster belonging to the same distribution, like Gaussian or Normal. One drawback of this model is the overfitting meaning that although it can find patterns on the existing datasets, it might fail to find patterns to new upcoming data points. Finally, **Density Model** scans the data space to find areas with varied density of data points. It forms clusters based on the amount of density an area presents.

In this study we will concentrate to algorithms compatible with the first two pre-described models, i.e Connectivity and Centroid Model, such as Markov Clustering, Affinity Propagation, Maximum Spanning Tree, Bisecting K-Means, as they can be applied for graph clustering problems which is the problem category where the service placement problem belongs to. We decided to avoid distribution and density model algorithms because they assume properties (spatial density, predefined statistical distributions) that do not align well with the structure and dynamic nature of Kubernetes microservices graphs.

2.3 Placement Algorithms

In this subsection we describe algorithms that have been applied in service placement problem and constitute heuristic methods or part of them which we are going to combine with clustering algorithms to investigate if they could improve further the placement decisions for the cost optimization based on the computing resources for which the cloud users are charged by the cloud providers.

2.3.1 Bin-Packing Problem

The bin packing problem [22] is an optimization problem according which a concrete number of items must be placed in the fewest number of available bins considering their varying capacities. It seems to have common objective and can be applied for our service placement for cost optimization problem solution. The bin packing problem is NP-hard and its corresponding decision is computationally NP-complete. Despite the problem's worst-case hardness, there are also many variations that can be solved with sophisticated algorithms. 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) [23] 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.

2.3.2 First-Fit Algorithm

The first-fit algorithm [22] constitutes one approach for the bin packing problem solution. Its main idea is based on scanning all items one by one at any order and trying to place each one into the first available bin from the sequence of all bin that it can fit in [23]. In case that there is not any bin with available place for an item, a new one is created for the item to be

placed. First-fit solution requires Θ(nlogn) time, where n represents the number of items to be packed. The algorithm can be faster if items are firstly sorted in a decreasing order. However, these extra steps do not make the algorithm optimal and can drop much more its performance for big in size lists.

2.3.3 Minimum K-Cut Problem

A minimum cut [24] is a partition of the vertices of a graph that is minimal in some metric. The minimum k-cut [25] problem is a combinatorial optimization problem that requires finding a set of edges whose removal would partition a graph to at least k connected components with the minimum-weight k-cut.

The problem assumes that for an undirected weighted graph G= (V, E) where V is the set of its Nodes and E is the set of its edges, and for a given input k which represents the desired number of partitions, we can find a k-cut of minimum total weight of edges whose ends are in different components. It is NP-complete and for fixed k it can be executed in polynomial time $O(|V|^{k2})$.

From the problem definition, if we think the services of an application as vertices of a graph and their connections with the corresponding affinity values as weighted edges, calculating a minimum k-cut of the graph is equivalent to partitioning the application into k parts while keeping the ones with minimum affinities in different parts. However, for arbitrary k, the minimum k-cut problem is NP-hard [26].

2.3.4 Contraction Algorithm

The Contraction Algorithm, or also known as Karger's algorithm [27], is a randomized algorithm to compute a minimum cut of a connected graph. The basic idea of 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 this edge into one Node (called edge contraction). By iteratively contracting the edges, which are randomly chosen, till reaching the required number of Nodes remain, the algorithm can find a minimum cut. The algorithm's output is the set of edges that remain [28].

When a 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 $O(|V|^2)$., where V the total number of graph nodes. Algorithm 2.1 presents the Contraction Algorithm for an undirected graph application with a required number nodes in the output k=2. For minimum k-cut the contraction algorithm is basically the same, except that it terminates when k nodes remain and returns all the edges in the graph [28].

In the context of our study, the contraction algorithm is used in combination with the partitioning algorithms, which are analyzed in section 3.2.

Algorithm 2.1	***Contraction Algorithm (k=2)***
1.	**Input:** G =(V,E)
2.	**Output:** a cut of G
3.	**while** |V| ≥ k **do**
4.	Choose an edge $e_{u,v}$ with probability propotional to its weight

5.	G ←G - $e_{u,v}$ //Contract edge $e_{u,v}$
6.	**end while**
7.	Return the cut in G

2.4 Adopted Technologies

In this subsection we are going to present each adopted technology for developing, packaging, hosting, orchestrating, monitoring and stressing a microservices based architecture.

2.4.1 Docker

Docker is an open-source platform which uses OS-level virtualization to deliver software in packages called containers. Each container contains an isolated application along with all its dependencies, such as libraries, binaries and configuration files, that can run on any type of Operating System or Cloud environment (public or private).

Packaging an application in multiple containers offer a number of benefits. First of all, it makes our application **portable**, which means that it will perform the same way regardless the environment on which Docker is running.

Furthermore, containers have much smaller footprint than virtual machines, which is a characteristic which makes them very lightweight for fast creation and quicker deployment. This brings a higher **performance** in container-based applications.

In addition, Docker containers enable the continuous integration and delivery of code boosting the **agility** of software development.

Finally, docker containers are **isolated** which is a characteristic that offers fault tolerance (if one service is down the other continues performing well) and scalability (one service can be replicated individually if high workloads are observed).

2.4.2 Kubernetes

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

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 failure over situations and provides efficient deployment

patterns. It must be mentioned that Kubernetes does not limit the types of applications supported, nor it deploys automatically source code or build the application. Furthermore, 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 an application using Kubernetes platform, 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.

The Kubernetes Control Plane consists of several components, from which the most important are the kube API Server, 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 2.4.2-1, which displays the Kubernetes components.

Pods can contact each other via another Kubernetes resource called Service. ClusterIP Services are accessible only within the cluster's environment. NodePort Services enable the requests from clients outside 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, which allows communication from everywhere inside and outside the cluster. Finally, LoadBalancer Services enable 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.

Cluster Infrastructure

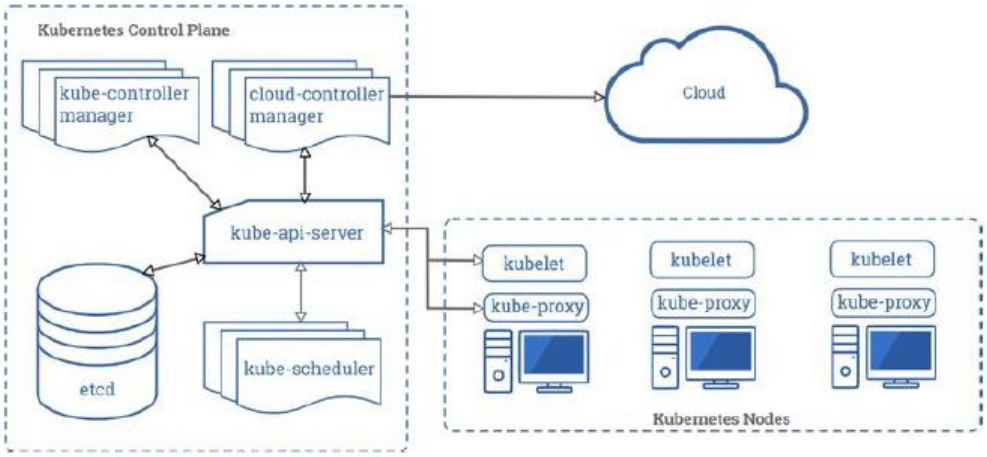


Figure 2.4.2-1: Kubernetes Components

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

Each application can be deployed in a Kubernetes cluster with Kubernetes API resources which are defined in yaml configuration files, called Kubernetes manifests. These manifests can describe resources like Deployments, Services, StatefulSets and Configuration data, known as ConfigMaps, for each application's Pod. Deployments provide information about each application's Pod and their replicas. Each service must have its container image and label configured to ensure the Pod is successfully initialized in 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, defining also the network policies. StatefulSets constitute resources, which preserve the identification of each created Pod that cannot be modified after the rescheduling process of the Pod. Statefulsets are mainly used for application units which must store a consistent state of data, thus they are utilized mainly for database services. A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can be connected with ConfigMaps to define environment variables in the executed source code, command-line arguments, or as configuration files in a volume. The manifests are applied to the Kubernetes cluster through the API server with a command line utility client called kubectl or via kube client libraries provided almost from all well-known programming languages.

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 the Pod's placement preferences feature. Besides Pod Affinities, also Node Affinities

or Antiaffinities can be specified in Pod Deployments configuration files, 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 among replica Pods of a specific Deployment.

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. 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 2.4.2-2. 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.

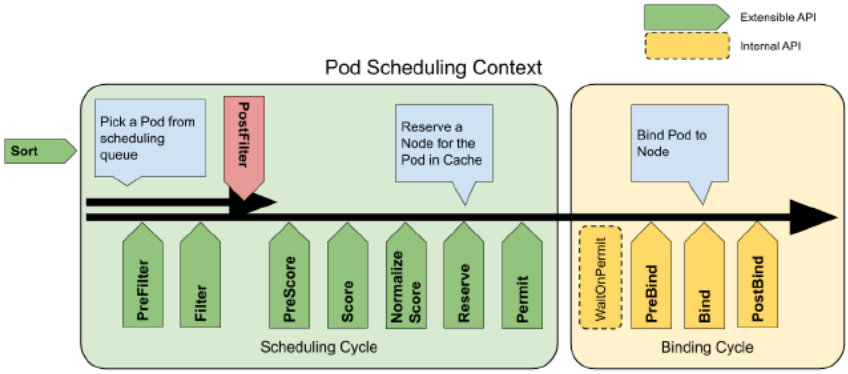


Figure 2.4.2-2: Kubernetes Scheduling Process

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.

2.4.3 Istio – A Service Mesh Implementation

Service mesh is an architectural pattern for managing communication between microservices in a microservices application.

When we move from monolith to microservice applications we introduce a couple of new challenges that we didn't have in monolith applications. For example, an online eshop application can consist of multiple microservices such as a webserver which gets the user interface requests, the payment microservice for the payment's logic and some more for functionalities such as shopping cart logic handling, product inventory, policies database and others.

This makes it clear that on a microservice based architecture each microservice contains each own business logic but for the overall functionality of the application, each service needs to talk to each other. So, it is vital to know how to **communicate** with each other, e.g. which is the endpoint to send traffic to a specific microservice and to be updated when a new microservice is being deployed. As a result, additional communication configuration is needed.

Furthermore, when deploying a microservices application in a Kubernetes Cluster we traditionally have firewall rules and a proxy in order to control the traffic crossing our application and have the suitable security around the cluster. However, such an approach doesn't provide security inside communication among microservices and every service inside the cluster can talk to any other service. From security perspective, if an intruder gets inside the cluster, he can do anything because there is no security layer inside cluster. Although this issue may not have any impact on a small application, in many scenarios such as online

banking or other applications with sensitive data, a higher level of security can be crucial. This is the reason why additional security configuration is needed before deploying each microservice.

In addition, a retry logic policy is demanding in order to make a microservices application more robust, in cases such as connection loss or a service's unavailability. Finally, metrics such as http errors, common receiving or sending requests by microservices, and how long does each request take in order a devops engineer to become aware for bottlenecks into an application are crucial for an application's monitoring procedure.

All these non-business logic operations we describe above, such as communication configuration, security and retry logic, metrics and tracing data must be applied to each microservice in order to handle all the above important challenges in a microservices based architecture. This means that developers of microservices are not working on the actual service logic, but are busy adding network logic for metrics, security, communication etc. for each microservice which also adds complexity to the services instead of keeping them simple and lightweight.

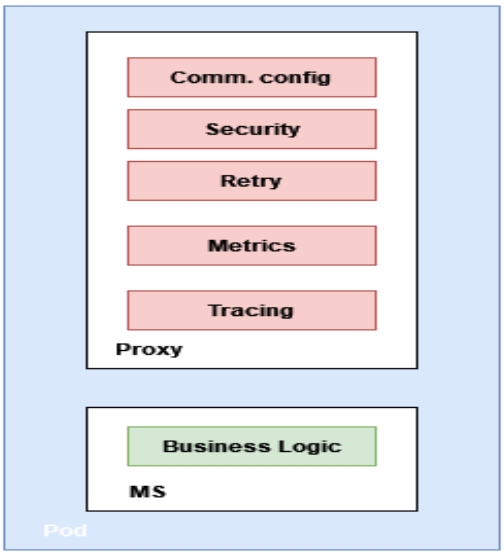


Figure 2.4.3-1: Sidecar Proxy outside of the Microservice's Business Logic

Service mesh solution, extracts all the non-business logic outside of the microservices adding a new sidecar application that handles all such functionality acting as a proxy [Figure 2.4.3-1]. This small application is a third-party application which cluster operators can easily configure without worry about the logic which is implemented. Furthermore, developers can eventually focus on the business logic of the application.

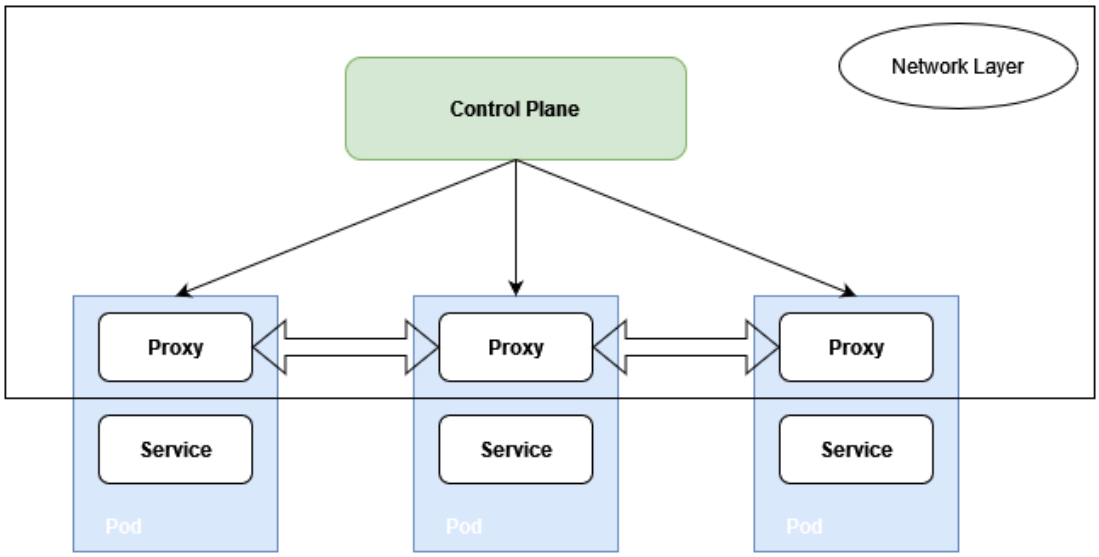


Figure 2.4.3-2: Network Layer Components in Service Mesh Paradigm

Service mesh has a Control Plane that it automatically injects this sidecar proxy in each Pod of an application. All microservices can talk to each other upon these proxies. All proxies along with the Control Plane form a network layer [Figure 2.4.3-2].

2.4.3.1 Istio[17]

Istio is one of the implementations of service mesh paradigm. Istio's architecture consists a data plane and a control plane. The data plane consists of multiple **envoy proxies[18]**, using them as sidecar proxies in the Pods. Envoy proxy is an open-source independent project which istio as well as other service mesh implementations use. On the other hand, Istio's Control Plane is called **Istiod** [Figure 2.4.3.1-1**]**.

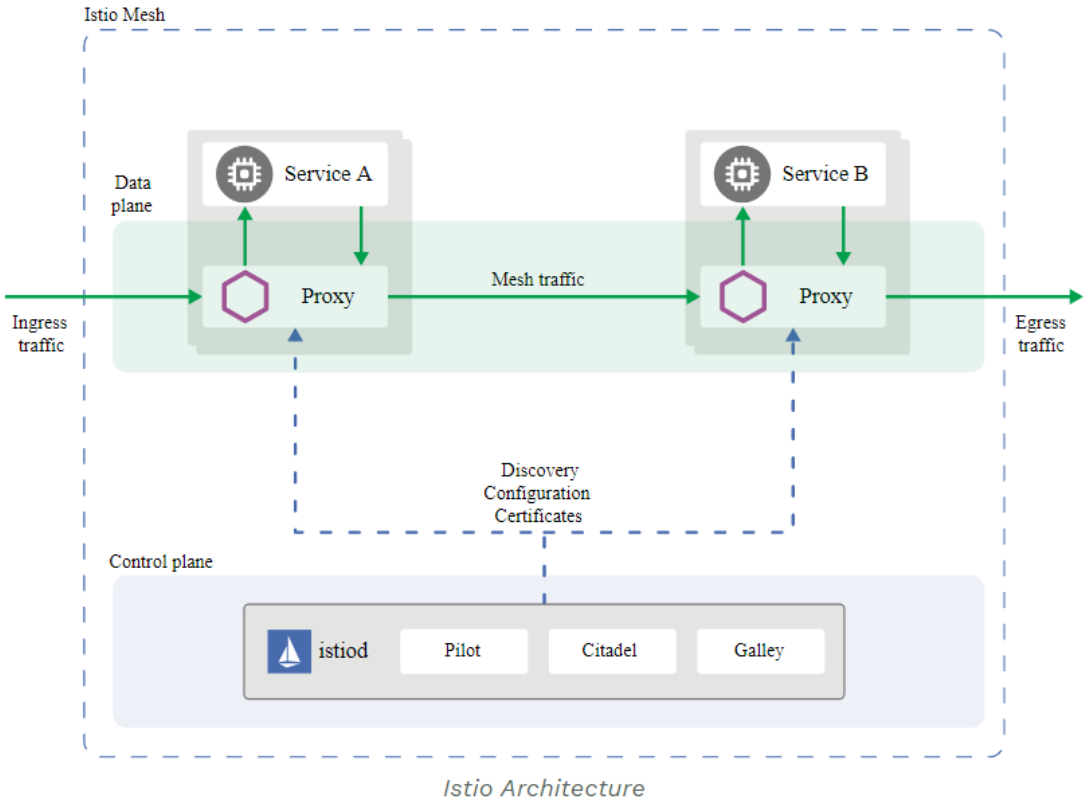


Figure 2.4.3.1-1: Istio's Architecture

Istiod injects the envoy proxies inside microservices pods. In earlier versions of Istio (until v1.5), Istiod was consisted by multiple components such as Pilot, Galley, Citadel and Mixer. After deploying Istio in a Kubernetes Cluster, multiple Pods were provisioned for the Istiod's different functions. However, all those components were merged to one single Istiod component in order to be easier for operators to configure and operate Istio.

Generally, Istio Architecture is comprised of the **Control Plane** which contains the Istiod component and manages the second part called Data Plane which is the group of all envoy proxies.

Configuring Istiod, it then converts high level routing rules and policies into Envoy-specific configurations and send it to each individual envoy proxy instance. Proxies can communicate without connecting to Istiod after receiving their configuration.

Istio is configured with Kubernetes YAML files. It uses Kubernetes CustomResourceDefinitions (CRDs) which are custom Kubernetes objects extending the Kubernetes API for configuring third party applications such as Istio, Prometheus, etc. As a result, without knowing any specific language users can configure Istio for traffic routing, traffic split, retry rules and many other network configurations.

The key building blocks of Istio's traffic routing functionality are the Virtual Services and the Destination Rules. With Virtual Services we can configure how to route traffic to a given destination and by destination rule components we can configure what happen to traffic for that destination, for example what type of load balancing to apply for the traffic reaching to

the specified destination service. Istiod converts these high-level routing rules into Envoy-specific configuration and propagates them into Proxy sidecars. Essentially, we don't configure Proxies, we configure Istiod component and proxies can communicate without connecting to Istiod, after receiving all the configuration they need.

Except of configuration features, Istiod has also an Internal Registry for Services and their endpoints. So instead of manually configuring the envoys, when a new microservice joins the cluster, it is subscribed to the registry automatically and no additional configuration is needed. As a result, each microservice knows each other's endpoint and can properly reach it if needed.

Furthermore, Istiod acts as a certificate authority (CA) and allows secure TLS communication between microservices inside a cluster.

Finally, Istiod gathers metrics and tracing data from envoy proxies which are later consumed by monitoring servers like Prometheus or Tracing servers like Jaeger and Zipkins, to have out of the box metrics and tracing data for the whole microservices application.

2.4.4 Kiali[19]

Kiali acts as an application for visualizing the traffic passing through microservices deployed along with Istio Service Mesh. The basic feature of Kiali needed in this work was the topology graph shaped and requested automatically by KIali's API.

2.4.5 Prometheus

Prometheus was created to monitor highly dynamic container environments like Kubernetes, Docker Swarm, Openshift etc. However, it can also be used in a traditional non container infrastructure monitoring applications deployed directly on bare metal servers.

Prometheus has become over the years the state-of-the-art tool for monitoring container & microservices infrastructure. As it is so hard for devops engineers to control the hundreds of simultaneous processes running by all microservices and be notified for unwanted issues such as response latencies, communication errors, resource shortages or server overloads and many more things that can go wrong in such complex infrastructures, it is so vital to be notified if an application is unavailable and for which reason, immediately after a failure without time waste for debugging and error fixing.

Describing the Prometheus architecture [Figure 2.4.5-1], we can distinguish **Prometheus Server** which does the actual monitoring work. It is consisted of **a Time Series database**, which stores and aggregates metrics data, such as CPU usage, number of exceptions, etc**, on disk or optionally in remote storage systems** , a **retrieval** worker which pulls metrics data from applications, services, servers and other target resources and sending them to the database, and third it has an **HTTP server** API which accepts queries for that stored data expressed with PromQL language and used to display them in a dashboard or a UI such as the Prometheus own Web UI or third party visualization tools like Grafana.

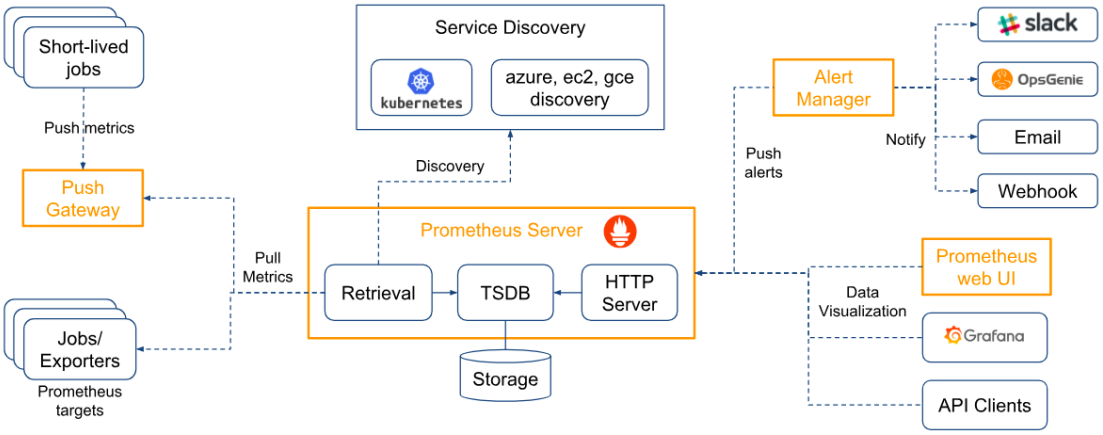


Figure 2.4.5-1: Prometheus Architecture

Generally**, Prometheus Server** monitors a particular entity which can be an entire Linux/Windows Server, an Apache Server, a single application or service like a database which is defined as a **target**. For a specific target there are several units which can be monitored such as CPU status, Memory/Disk Space Usage, Requests Count/Duration, Number of Exceptions etc. All these units are called metrics. Metrics are stored in Prometheus Server's database component and have a human-readable text-based format with TYPE and HELP attributes injected to increase their readability. HELP attribute expresses the description of what each metric represents and the TYPE can be a counter representing how many times something happened, a gauge which tells what is the current value of a metric now or a histogram which expresses how long or how big a size of a metric was over a time period.

All Prometheus configuration, about how to know what to scrape and when, is formatted in a YAML formatted file. Inside the configuration are defined different fields which express several policies such as how often Prometheus will scrape its targets, rules for aggregating metric values or creating alerts when a condition is met and what resources Prometheus monitors. Then it uses a service discovery mechanism to discover all targets.

In addition, Prometheus has another component called **Alert Manager** to trigger alert about a met condition. It is responsible for receiving alerts pushed to an HTTP Server and notify users with mechanisms like Email, Slack etc.

2.4.5.1 Prometheus Exporters

Although some servers exposing metrics to be pulled by Prometheus Server Retrieval component by default, some other need an extra component to expose metrics to Prometheus Server. Such components called **Exporters**.

Exporters fetch metrics from a targeted service and converts them to the correct format which is understandable by Prometheus. Then it exposes its own /metrics endpoint which can then be pulled by Prometheus Retrieval Component.

Prometheus provides a list of official third-party exporters for common used services like MySQL, ElasticSearch , Linux Server, build tools, Cloud Platforms and so on.

For example, in order to monitor a Linux Server, we can download and deploy a node exporter which then provides the suitable endpoint for Prometheus Server to collect metrics relatively with the available and used resources on it.

Exporters are also available as Docker Images, making easy to monitor several microservices only by deploying a sidecar container in a Kubernetes Pod along with the Service we would like to monitor. Furthermore, there are also libraries of several programming languages that Prometheus provides in order to implement our own exporters to custom services for exposing metrics endpoint to Prometheus for collecting data such as how many requests our service receives or sends, how many exceptions are thrown or how many server resources are used by the service.

Prometheus exporters is the main advantage that distinguishes Prometheus from other monitoring tools such as Amazon Cloud Watch, New Relic etc. In such tools Applications and Servers are responsible for pushing their metrics to a centralized collection platform. So, if we have a big number of microservices pushing their own data, then a high load of network traffic is created which makes our monitoring tool a bottleneck for the application's performance. Instead, Prometheus has a pull-based strategy for scraping metrics from all microservices which makes it possible for multiple Prometheus instances to pull metrics data and gives a better detection/insight if services are up and running.

However, Prometheus pull metrics strategy is not seemed suitable for short-lived jobs. Such jobs should push their metrics at their exit to a different Prometheus components called **Pushgateways**.

2.4.6 Grafana

Grafana is an open-source application for visualizing data. It allows us to build charts, graphs and dashboards of data we want to visualize. A simple case of Grafana Architecture is shown in Figure 2.4.6-1. It consists of a **Data Producer** which can be a Jenkins CI, Kubernetes Pods, a raspberry pi, a Virtual Machine in a Data Center or an IoT Sensor which produces the data and **a Data Source,** like Prometheus Server, an InfuxDB or a MySQL database, which connects with the Data Producer. Depending of the type of each Database, it might either pull data from the Producer or the latter is configured to push data into the Database. For example, as we described in 2.3.5 section, Prometheus reaches out the Data Producer (targets) and there is a dedicated endpoint for it to scrape the data from. Then it aggregates the data and provide them through an HTTP Server. Finally, a typical Grafana Architecture includes the **Grafana Server** which is actually the Front End which visualizes the data. In order to get the data Grafana queries the data source which then returns the requested data based on the query. These data are then displayed on a Grafana dashboard.

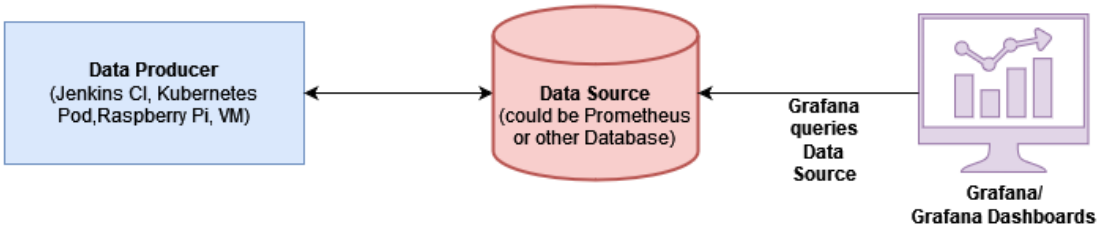


Figure 2.4.6-1: Grafana's Architecture

Basically, there are two types of data that are visualized in Grafana. The first category is **Metrics** and the second one is **Logs**. Metrics can be for example a CPU load, current Memory usage or a sensor's current Temperature value. In contrast, logs can be consisted of a timestamp with an error type and a small description about it. A typical Grafana Dashboard is shown in Figure 2.4.6-2.

Figure 2.4.6-2: Grafana's Dashboard Example

2.4.7 Apache J Meter

Software products demand excellent quality all the time. In order to ensure the suitable quality level, engineers involved in testing procedures adopting various testing approaches such as Functional UI, security and database performance. A well-known tool for such purposes is the Apache JMeter.

JMeter is a JAVA open-source software that is used as a load testing tool for analyzing and measuring the performance of a variety of services. Usually, we tend to follow the trends in testing and forget to pay importance to verifying whether a product meet expected or required performance. Nowadays, performance is an inevitable factor especially for web and mobile applications as the user strength is very huge for each one even if it is not to be expected all the time. In order to cope up with such situations of load we need a handy tool and that's exactly where JMeter comes into place.

Speaking about Performance Testing, it is defined as a type of software testing to ensure that software applications will perform well under their expected workload. It focuses on specific factors of a software program such as speed, scalability and stability. Performance

testing checks whether the application under tests, satisfies the required benchmarks on both load and stress. Load testing is testing to verify if a system or application under is able to handle the required number of concurrent user accesses on web server without any failure. The stress testing is testing to see how the web server copes up with high load and limited resources when it is under constrained conditions. It determines only the maximum load that the Web Server can handle. Generally, performance testing has much significance in real time particularly from a point of view of customer satisfaction and return on investment (ROI), which a performance measure used to evaluate the efficiency or profitability of an investment.

There are many advantages that make JMeter one of the most preferable testing tools. It is open-source written in JAVA, high extensible and platform independent. Also, it is very user friendly. It has a comprehensive GUI which helps to create a test plan and configure all its parameters. It also permits scripting but a user can run tests without knowing a bit of code. JMeter stores its test plans in XML format, which means that we could create a test plan using a common test editor. An additional advantage is the several types of testing which supports. Except of performance testing, for which was designed, it also supports non-functional testing such as stress stressing, distributed stressing, web service testing by creating test plans. In addition, it also provides support for protocols such as HTTP, JDBC, SOAP, JMS and FTP. Furthermore, it has a big community and a comprehensive documentation for satisfying support. Finally, it supports several built-in dashboards for test reporting.

3. Implemented Algorithms

3.1 Clustering Algorithms

In this section we describe the clustering algorithms we have adopted from the bibliography and the modifications we have made in order to apply them in our service placement use case. More specifically, we have implemented the maximum standard deviation reduction (MSDR) [2], Markov Clustering [4] and Affinity Propagation[3] algorithms. The basic reason for our choices was the unsupervised nature of the methods as they don't require the number of clusters as an input parameter.

Each algorithm takes the adjacency matrix of a graph as an input and tries to find clusters among nodes which are formed based on the weights of the edges which connect them. Each node represents a microservice of an application, called pod on kubernetes, and each edge represents the connection between two microservices. The weight of an edge is referred to the affinity between two nodes. The affinity between two microservices **a** and **b** is calculated by the following equation, adopted by [1]:

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

Where,

- m is the total of messages exchanged by all microservices,
- $m_{a,b}$ is the number of messages exchanged between microservices a and b,
- d is the total amount of data exchanged by all microservices,
- $d_{a,b}$ is the amount of data exchanged between a and b
- w is the weight, such that $\{w \in R \mid 0 \leq w \leq 1\}$, used to define the importance of each variable at the computation of the affinity.

We have adopted the above formula for affinity calculation, because two microservices could exchange a small number of messages but with a huge size. In such a scenario they should have a bigger probability to be placed on the same host machine because despite their small number of messages this communication will have a big amount of egress traffic in case that these microservices are placed on different hosts which is translated as higher charging by the cloud provider.

We apply the three prementioned algorithms in order to put all microservices into clusters taking only into consideration their affinities. This means that microservices which communicate often or exchange messages with a big size will be placed in the same host machine. This could lead our application to perform better, by reducing the network latency and the response time of each service.

At a second step, we apply a heuristic algorithm called bin packing, in order to reduce at the same time, the number of resources all these clusters will allocate. Essentially, the algorithm tries to place as many clusters of microservices that came up from the first step as possible to the same host machine.

Combining the two steps, i.e. a cluster algorithm and the bin packing solution, we would like to try a service placement solution according to which we will have the minimum amount of resource allocation and egress traffic propagation, i.e. the size of messages leaving from an instance to reach another. As a result, we could succeed the minimum charging for a microservice-based application in order to be deployed and orchestrated in a Kubernetes cluster.

3.1.1 Maximum Standard Deviation Reduction Algorithm (MSDR)[2]

The MSDR algorithm takes a set of points as input and constructs an **Euclidean Minimum Spanning Tree (EMST)** based on their pairwise distances. The algorithm begins by calculating the standard deviation of the edge weights in the tree. During each **round**, the algorithm evaluates whether removing a specific edge will result in a significant reduction in the overall standard deviation of the edge weights. At the end of each round, the edge that achieves the **maximum reduction in standard deviation** is removed, splitting the tree into two subtrees. This process is repeated iteratively until a termination criterion is met.

The final number of clusters corresponds to the number of subtrees at the point where the **standard deviation reduction function reaches a local minimum**, which is mathematically defined as the point where the **first derivative equals zero** and the **second derivative is positive**. At this stage, the algorithm achieves the **maximum standard deviation reduction**,

meaning that further cuts will no longer result in significant improvements. The algorithm has been successfully applied to image color clustering problems, where it automatically determined the optimal number of clusters without requiring users to fine-tune parameters.

The overall time complexity is influenced by the following factors:

- **O(E log V)** for MST construction,
- **O(E)** for calculating standard deviation in each iteration,
- **O(V)** for the number of iterations.

Thus, the total time complexity of the MSDR algorithm is **O(E log V + V * E)**, which simplifies to **O(V * E)** in the worst case. This is the overall complexity of the algorithm when considering both the MST construction and the iterative edge-removal process.

The main contributors to space complexity are:

- Adjacency matrix: O(N^2) (for storing the pairwise distances).
- MST structure: O(N) (for storing the tree).
- Subtree tracking (union-find or similar): O(N).

Thus, the overall space complexity is: **O(N^2)**

This is because the adjacency matrix dominates the space complexity, as it grows quadratically with the number of points.

In order to adapt the algorithm to our own problem, we chose to compute the Maximum Spanning Tree instead of the minimum one as the useful edges is those with the highest weight representing the microservices with the highest traffic which is the criterium selected for clustering our graph. Then, we follow the same procedure with MSDR until receiving a group of clusters containing all microservices of our architecture.

Algorithm: MSDR ()
Let S be the point set
Let T_0 be the EMST constructed from S
Let S_K be the set of disjoint subtrees of T_0
Let e be an edge in S_K
Let $\sigma(S_K)$ be the overall StdDev of all edges in S_K
Let $\sigma(T_j)$ be the StdDev of edges in subtree $T_j \in S_K$
Let $\Delta\sigma(S_K)[i] = 0$ be the maximum StdDev reduction
after the removal of an edge e at each iteration i
Let $\epsilon = 0.0001$

$S_K = \{T_0\}$
$\sigma(S_K) = \sigma(T_0)$
$i = 0$
Repeat
$i \leftarrow i + 1$
$temp = \sigma(S_K)$
/* Choose an edge that leads to max StdDev reduction
once it is removed from S_K */
For each $e \in S_K$
Assume e is removed from S_K *thus* $S_K = \cup_{j=1}^{i+1} T_j$
$\sigma(S_K) = \frac{\sum_{\forall T_j \in S_K} |T_j| \cdot \sigma(T_j)}{\sum_{\forall T_j \in S_K} |T_j|}$
/* Compute StdDev reduction */
If $\Delta\sigma(S_K)[i] < \sigma(S_K) - temp$
$\Delta\sigma(S_K)[i] = \sigma(S_K) - temp$
Remove e from S_K that corresponds to $\Delta\sigma(S_K)[i]$
$\sigma(S_K) = temp - \Delta\sigma(S_K)[i]$
until $|\Delta\sigma(S_K)[i] - \Delta\sigma(S_K)[i-1]| <$
$|\epsilon \cdot (\Delta\sigma(S_K)[i] + 1)|$

$f(j) = PolyRegression(\bigcup_{j=1}^{i} \Delta\sigma(S_K)[j])$
/* No. of clusters corresponds to the 1st local minimum */
$K = \min(j \in [1, i])$ that satisfies $f'(j) = 0$ & $f''(j) > 0$
Return $S_K = \{T_1, \ldots, T_K\}$

$$\sigma(Sk) = \frac{\sum_{\forall Tj \in Sk} |Tj| \times \sigma(Tj)}{\sum_{\forall Tj \in Sk} |Tj|}$$

3.1.2 Affinity Propagation[3]

Affinity Propagation (Brendan Frey, Delbert Dueck 2007) is an unsupervised machine learning technique. The algorithm does not require the number of clusters as an input. Each data point sends a message to all other data points informing them about how much **attractive** they seem for the sender. Then each data point replies to all senders with a message relatively with its **availability** to be associated with the sender which is derived by the attractiveness that is has received by all senders. The senders recalculate their attractiveness for each target and message to them back. The procedure is repeated until they reach a consensus. When a sender is associated with a target, that target becomes an exemplar for this sender. An **exemplar** is a data point that acts as the representative or "center" of a cluster. It is the data point that other points (senders) associate themselves with based on their mutual attractiveness and availability scores.

Points that share the same exemplar belong to the same cluster. Essentially, the exemplar is the most "attractive" or representative point for a group of associated data points.

The Affinity Propagation Clustering Algorithm calculates the similarity between two points I and k by the following equation:

$$s(i,k) = -\|x_i - x_k\|^2$$

This equation ensures that larger distances between two points correspond to smaller (more negative) similarity values, reflecting reduced similarity.

In order to adapt the algorithm for graph clustering, each node of the graph is treated as a data point, and the affinities between two nodes are used as the similarity values. However, affinities in a graph are typically positive values between 0 and 1, with higher values representing stronger relationships. To align these affinity values with the similarity metric used by Affinity Propagation (which operates on a negative scale), we subtract 1 from the affinity values. This transformation maps affinities to a similarity range of [−1,0]:

- Affinities close to 1 (strong relationships) are mapped to values near 0, reflecting high similarity.
- Affinities close to 0 (weak relationships) are mapped to values near -1, reflecting low similarity.

This mapping ensures that the algorithm interprets affinities in a way consistent with its original design. The similarity equation in this adapted context becomes:

$$s(i,k) = \text{Affinity}(i,k) - 1$$

As a result, our implementation of Affinity Propagation takes an adjacency matrix as input, where the values represent precomputed similarities between nodes. These similarities are derived from the affinities between nodes, but they are first normalized to a specific range and then adjusted to be negative [-1,0], as the algorithm operates on a negative similarity scale. This transformation ensures that the similarity values are consistent with the way the algorithm handles data points in an n-dimensional space, where larger distances correspond to more negative similarity values. Essentially, the similarity values in our graph-based adaptation are mapped to the negative range to match the expected input format of Affinity Propagation.

As we analyzed above, the higher the affinity value the bigger the probability of two nodes (microservices) to be members of the same cluster. As the affinity value equals the similarity value plus 1, it seems a suitable metric to replace the shortage of space dimensions our points present.

At the second step the algorithm calculates the responsibility matrix considering the similarity matrix as defined previously and the availability matrix initialized by zero values. In the Affinity Propagation algorithm, the **responsibility matrix** plays a critical role in determining how much a data point (node or microservice) should take on the role of a potential cluster representative, also called an **exemplar**. It is a key part of the iterative message-passing process that ultimately assigns points to clusters. The responsibility matrix, denoted as **r(i, k)**, represents the "responsibility" of point **i** to select point **k** as its exemplar. In simple terms, it indicates how much point **i** thinks point **k** is a good candidate to be its exemplar, taking into account the similarities between them and the availability of other points to serve as exemplars.

The following equation is used for the responsibility matrix calculation:

$$\mathrm{r(i,k)} \leftarrow \mathrm{s(i,k)} - \max_{\mathrm{k'} \neq \mathrm{k}} \{\mathrm{a(i,k')} + \mathrm{s(i,k')}\}$$

Then the availability matrix is updated by the following equations:

$$\alpha(\mathrm{i,k}) \leftarrow \min\left(0, r(k,k) + \sum_{i' \notin \{i,k\}} max\left(0, r(i',k)\right)\right), for\ \mathrm{i} \neq \mathrm{k}$$

and

$$\alpha(\mathrm{k,k}) \leftarrow \sum_{\mathrm{i'} \neq \mathrm{k}} \max(0, \mathrm{r(i',k)})$$

The above procedure is repeated until the clusters formed remain unchanged over a number of iterations (15 iterations we chose) or some predetermined number of iterations is reached (200 iterations we chose). The exemplars of the clusters are extracted from the final matrixes as those whose 'responsibility plus availability from themselves is positive' which means:

$$r(k,k) + a(k,k) > 0$$

This criterion ensures that exemplars are chosen as points that:

- Have high similarity to themselves (good candidates for cluster centers).
- Are widely agreed upon by other data points.

Each remaining data point ***i*** is assigned to the cluster of the closest exemplar ***k***, typically determined by:

$$k = \arg\max_{k'}\left(r(i,k') + a(i,k')\right)$$

Time Complexity: **O(T * N^2)**, where T is the number of iterations (up to 200), and N is the number of data points (nodes or microservices).

Space Complexity: **O(N^2)**.

3.1.3 Markov Cluster Algorithm(MCL)[4]

Markov is a clustering algorithm based on unsupervised learning. This means that it helps us find communities between associative data without requiring any input to form clusters.

The core idea of MCL is **the random walks** like the popular travelling salesman problem. Essentially, we have a weighted undirected graph consisted of a number of nodes, representing locations, and a number of edges, each one representing a connection between two locations. The weights of the edges represent the probability of the salesman being in a node to move from that node to a different one connected with it. Following a random path from location to location (node-to-node) forms a process called mathematically stochastic process or a random process. This is called a Random Walk.

MCL guesses that there is a system in which the next state is dependent upon the current state based on a probability or rule. This system called **a Markov chain**. A visit at a node is only dependent from the previous node visiting irrespective of the fact how one got there.

The algorithm takes as input the graph based on Markov chains. A Markov chain is a mathematical system that undergoes transitions from one state (or node) to another. The key feature of a Markov chain is that the future state depends only on the present state, not on the sequence of events that preceded it (this is known as the *memoryless property*). In this context, each node in the graph represents a state, and the edges between them represent possible transitions. A graph based on Markov chains is essentially a network where each edge has an associated probability that represents the likelihood of moving from one node (state) to another. This probability is determined by a transition matrix, which contains the probability values for moving from any node to any other node in the system. The transition matrix is a square matrix where each element represents the probability of transitioning from one node (microservice, in your case) to another. The value of an element at position (i, j) in this matrix indicates the probability of moving from node **i** to node **j**. For the MCL algorithm, this matrix is crucial because it represents the likelihood that one microservice will send traffic to another, given their connectivity (affinity). When MCL uses the graph based on Markov chains, it treats the nodes as locations and the edges as probabilistic transitions between those locations. The random walks (the process of moving from one node to another) in this graph help identify tightly connected communities (clusters of nodes), where the random walk is more likely to remain within a cluster rather than jump across clusters.

As our graph extracted from Kiali can be transformed to an adjacency matrix, describing the affinities between connected microservices, we have to convert this matrix to a transition matrix needed by MCL algorithm, in order to express the probability a microservice to send traffic to another one having non-zero affinity value.

Guess A is an NxN adjacency matrix, we calculate the transition matrix T by normalizing A with the following equation:

$$\mathrm{T[i,j]} = \frac{\mathrm{A[i,j]}}{\sum_{k=0}^{N} \mathrm{A[k,j]}}$$

After calculating the transition matrix, we apply the MCL algorithm based on two important operations **Inflation** and **Expansion**.

Inflation raises each element of a column to non-negative power and then re-normalizing. As a result, the strong neighbor values are strengthened and less popular neighbors (with low affinity) are demoted. The following equation express the Inflation calculations:

$$\mathrm{T'[i,j]} = \frac{T[i,j]^I}{\sum_{k=0}^{N} T[k,j]^I},$$

*where **T'** the matrix after inflation operation and **I** the inflation value*

On the other hand, the **expansion** operation makes distant nodes more accessible by raising the transition matrix to a higher power. This process increases the probability of reaching nodes that are farther apart by incorporating intermediate connections. Essentially, expansion helps to "connect" different regions of the graph by allowing the random walk to flow across clusters, even if they are not directly connected. The following equation describes the matrix **T''** after applying the expansion operation to **T'** with the exponent **e**.

$$T''[i,j] = T'[i,j]^e$$

The two operations, i.e inflation and expansion, are repeated until there is a convergence, i.e the values of the matrix are not changed any more.

The time complexity of the MCL algorithm is **O(K * N^3)**, where K is the number of iterations required for convergence and N is the number of nodes in the graph. In practice, K is typically small, but it depends on the graph's structure and the chosen inflation/expansion parameters.

The overall space complexity of the MCL algorithm is dominated by the matrices involved, thus it is **O(N^2).**

In the implementation of the MCL algorithm, the values for **inflation (I)** and **expansion (e)** are selected based on their effect on the clustering results. For our specific case, we chose e=2 and performed a search for the optimal inflation value (I) within the range of 1.2 to 2, using a step of 0.1. The value of I that gave the best modularity was selected. **Modularity** is a measure used to evaluate the quality of the clustering by comparing the density of edges within clusters to the expected density if the edges were distributed randomly. The modularity Q is given by the following equation:

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

Where:

- m is the total number of edges in the graph,
- Aij is the adjacency matrix element representing the weight of the edge between nodes i and j
- ki and kj are the degrees of nodes i and j
- ci and cj are the communities (or clusters) to which nodes i and j belong
- δ(ci,cj) is 1 if nodes i and j belong to the same community, and 0 otherwise.

By testing different inflation values in the given range, we found the combination of e=2 and the best I to yield the highest modularity, resulting in more distinct and well-formed clusters.

3.2 Placement Algorithms

In this chapter, we analyze useful preexisted heuristic methods relative with the Service Placement Strategies that we implemented and compared in this study.

3.2.1 Heuristic First-Fit

Sampaio et al. on [1] propose a heuristic approach to optimize service placement in an existed infrastructure by moving services with higher affinity (i.e. higher traffic communication rates) on the same host machine. The proposed algorithm, a variant of Heuristic First-Fit, computes a new placement for the application services by accessing the resource usage of the cluster Nodes and Pods and the available host machine 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 those with high affinity are co-located, while microservices' resource usage and availability of resources at the host are taken into account.

Algorithm 3.2.1 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 that m_j requests to be placed on it, then the algorithm attempts to place m_i onto the host of m_j, i.e. onto the host 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 be moved anymore, even if it is connected with another service in the application with more affinity traffic rates. At the end, a list of movements is generated containing microservices 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 3.2.1	***Heuristic First-Fit Variant [1]***
1.	**Input:** Hosts (H) , microservices (m), resources (r)
2.	**Output:** Placement solution
3.	*moved* ← []
	//Affinities are in decreasing order
4.	**for** every pair of affinities **do**

5.	$m_i \in H_i$ // m_i located at host H_i
6.	$m_j \in H_j$ // m_j located at host H_j
7.	$m_j \neq m_i$, $H_j \neq H_i$
8.	*hasMoved* ← False
9.	**if** $r(m_i) + r(m_j) \leq r(H_i)$ ^ $m_j \notin$ *moved* **then**
10.	$H_j \leftarrow H_j - m_j$
11.	$H_i \leftarrow H_i \cup m_j$
12.	*hasMoved* ← True
13.	**else if** $r(m_i) + r(m_j) \leq r(H_j)$ ^ $m_i \notin$ *moved* **then**
14.	$H_i \leftarrow H_i - m_i$
15.	$H_j \leftarrow H_j \cup m_i$
16.	*hasMoved* ← True
17.	**end if**
18.	**if** *hasMoved* ***then***
19.	*moved* ← *moved* ∪ $[m_i, m_j]$
20.	**end if**
21.	**end for**

3.2.2 Binary Partition

Yang et al. in [28] 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 form 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 3.2.2 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. For 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 $O(n^2 m \log^2 n)$, 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 $O(m \log^2 n)$.

3.2.2 Algorithm	***Binary Partition[28]***
1.	**Input:** Service-Based Application (S), threshold α
2.	**Output:** Partition P = {S_1,S_2,…,S_N}
3.	P ← {S}
4.	**while** exists part S_i in P that total resource demands exceed α and part S_i contains more than one service **do**
5.	P ← P - S_i //Remove partition i from application's partition P
6.	Construct graph G= (V,E) based on S_i service affinities
7.	n ← |V|
8.	G_{min} ← G
9.	t ← 0
10.	**while** t ≤ n **do**
11.	Perform Contraction Algorithm (k=2) to get a cut G'
12.	G_{min} ← min(G_{min},G')
13.	t ← t+1
14.	**end while**
15.	Create two sub-partitions {S_x,S_y} from part S_i according to G_{min}
16.	P ← P ∪ {S_1,S_2,…,S_k}
17.	**end while**
18.	Return P

3.2.3 K-Partition

K-Partition (KP) algorithm [28], 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 contraction algorithm for K=2, while K-Partion increases the value of K upon each iteration of the algorithm. This leads to the creation of affinity hubs instead of affinity tuples between microservices and can accelerate the partition process. The basic logic of the algorithms remains the same as the Binary Partition. The contraction algorithm is executed also N times, but the total number of 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 [art 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 3.2.3.

3.2.3 Algorithm	***K-Partition[28]***
1.	**Input:** Service-Based Application (S), threshold α
2.	**Output:** Partition P = $\{S_1,S_2,\ldots,S_N\}$
3.	P ← {S}
4.	k ← 1
5.	**while** exists part S_i in P that total resource demands exceed α and part S_i contains more than one service **do**
6.	P ← P - S_i //Remove partition i from application's partition P
7.	Construct graph G= (V,E) based on S_i service affinities
8.	n ← |V|
9.	G_{min} ← G
10.	k ← k+1
11.	t ← 0
12.	**while** t ≤ n **do**
13.	Perform Contraction Algorithm until k Nodes remain to get a k-cut G'
14.	G_{min} ← min(G_{min},G')
15.	t ← t+1
16.	**end while**
17.	Create k sub-partitions $\{S_1,S_2,\ldots,S_k\}$ from part S_i according to G_{min}
18.	P ← P ∪ $\{S_1,S_2,\ldots,S_k\}$
19.	**end while**
20.	Return P

3.2.4 Bisecting K-Means

Bisecting K-Means (BKM) is a divisive hierarchical clustering algorithm [29] 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, the 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 selected 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 3.2.4 shows the pseudocode of the Bisecting K-means algorithm.

3.2.3 Algorithm	***Bisecting K-Means[29]***
1.	**Input:** Cluster C, number k of desired clusters
2.	**Output:** k clusters of Application
3.	i ← 1

4.	**while** i < k **do**
5.	Select a parent cluster, C to split
6.	**for** fixed number of iterations **do**
7.	Use K-Means to split C into C_1 and C_2
8.	Calculate inter-cluster dissimilarity for C_1 and C_2
9.	**end for**
10.	Select the sub-clusters with highest inter-cluster dissimilarity
11.	i ← 1 + 1
12.	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 [30]. Furthermore, the selection of sub-clusters, which are produced randomly, can increase the deviation and the errors of the data points clustering. 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.

3.2.5 Heuristic Packing

Heuristic Packing (HP), which is presented also in [28], is a placement algorithm attempting to pack each given application partition into the application's host machines. This algorithm is executed as a post-processing step of 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 number of services involved in the application, it is feasible 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 machine in which services will be placed.

Algorithm 3.2.5 presents the Heuristic Packing algorithm. Inputs to the algorithm are the application's partitions produced by the partitioning algorithm $P=\{S_1,S_2,\ldots,S_N\}$ and the available resources on each machine $V=\{V_1,V_2,\ldots,V_N\}$. The algorithms will produce a placement 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:

$$tf = \sum t_{uv}$$

Where:

- *tf* → the total traffic rate of all services
- t_{uv} → traffic rate between services *u,v*

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

$$ml = \sum_{k=1}^{R} \frac{d'^{k}_{i}}{v^{k}_{j}}$$

Where:

- *ml* → the most loaded situation value
- *R* → the set of resources types (CPU,RAM,Storage etc.)
- d'^{k}_{i} → amount of resources r_k that service s_i demands
- v^{k}_{j} → 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 design 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 cannot be placed into the available host machines.

The time complexity for the Heuristic packing is $(nM + n^2)$. Upon every algorithmic execution all n parts in the application's partition are processed and for each part all the available *M* machines 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 (n^2) .

3.2.5 Algorithm	***Heuristic Packing [28]***
1.	**Input:** Partition *P={S₁,S₂,…,S_N}* of application, vectors of available resources on each machine *V={V₁,V₂,…,V_M}*
2.	**Output:** a placement solution X
3.	Calculate vectors of resource demands of each part *{D'₁,D'₂,…,D'_N}*
4.	X ← [X_{ij} = 0] for every part and host machine
5.	**for** *i* ← 1 ; *i* ≤ *N'* ; *i++* **do**
6.	*tf* ← 0, *ml* ← 0, *y* ← 0
7.	**for** j ← 1; *j* ≤ *M; j++* **do**
8.	**if** part S_i can be packed into machine m_j **then**
9.	*tf* ← $\sum t_{uv}$

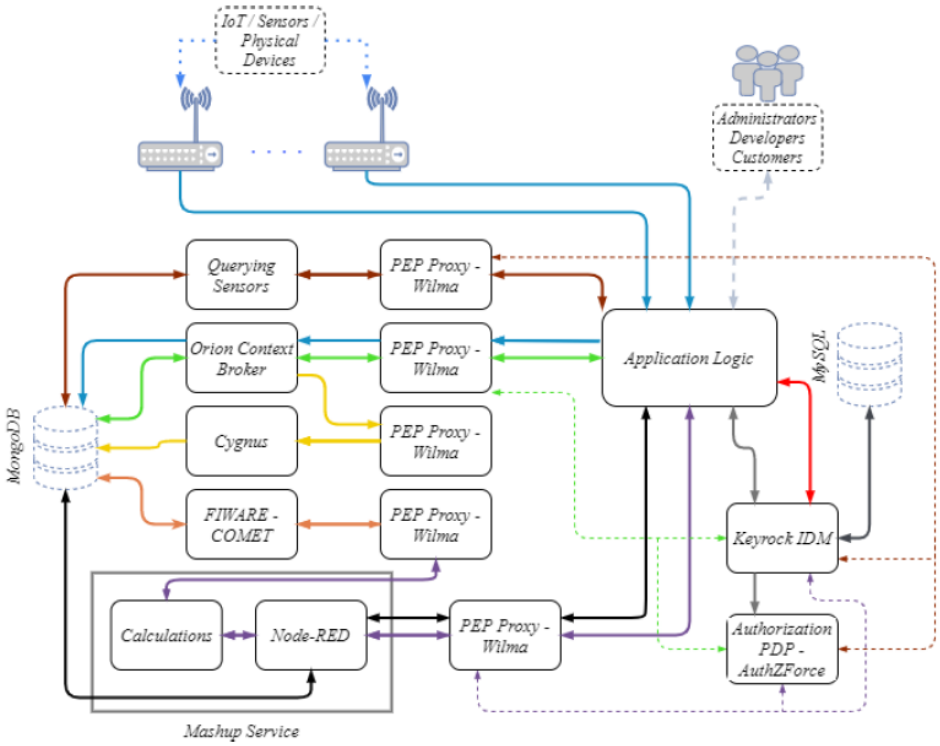
10.	$ml \leftarrow \sum_{k=1}^{R} \frac{d'^{k}_{i}}{v^{k}_{j}}$
11.	**if** $tf_j \geq tf$ **then**
12.	$tf \leftarrow tf_j$, $ml \leftarrow ml_j$, $y \leftarrow j$
13.	**else if** $tf_j == tf$ **and** $ml_j > ml$ **then**
14.	$tf \leftarrow tf_j$, $ml \leftarrow ml_j$, $y \leftarrow j$
15.	**end if**
16.	**end if**
17.	**end for**
18.	**if** $y == 0$ **then**
19.	Return null
20.	**Else**
21.	$V_y \leftarrow V_y - D'_i$
22.	$x_{iy} \leftarrow 1$
23.	**end if**
24.	**end for**
25.	Return X

4. Implementation

In this chapter we describe the two benchmark architectures we use for the placement strategies evaluation. In section 4.1 we demonstrate a microservice-based architecture for IoT use cases and in section 4.2 we present a google's prototype microservice architecture which consists of several services for an online e-shop. Finally, in section 4.3 we analyze the cloud infrastructure and all the components for our system architecture which are deployed on it for each app's functionality and their microservices orchestration, traffic management, traffic monitoring and metrics collection.

4.1 iXen Architecture[12]

iXen was a prototype which constituted a software architecture for IoT scenarios based on SOA principles. For the purposes of this thesis, it was converted from a bare metal deployed SOA architecture to a microservices-based architecture packaging each of its services as Docker Images. As a result, each microservice is independent from the other, becomes portable, has its own data storage, can be scaled independently and can be orchestrated by container orchestrators such as Kubernetes. For each microservice we can monitor its traffic, define the minimum demands on resource and declare the infrastructure it will be deployed on.



iXen follows a 3-tier architecture model. At the first layer the **Infrastructure Owners – System Administratos** 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.

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.

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 is 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 is 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 that an app created by Node Red can provide to its users.

In order to deploy the above predescribed microservices on a K8s cluster on GKE we applied the databases as Statefulsets with their own Persistent Volumes, Persistent Volume Claims and secret resources. For the stateless apps we define a deployment for each one and for each microservice needing configuration variables to be declared as sensitive data we define their own secrets. For microservices with their own source code such as application logic and querying sensors services we also mount their pods with a created NFS volume with an external NFS server hosting their codebases.

4.1.1 iXen Workloads for Stressing

In the following table we present examples of all the requests are sent to the application logic in order to stress the application and create workflows among all microservices. We present for each request the HTTP VERB, the endpoint, the body data and a short description of the function they trigger in the application:

Operation	HTTP VERB	ENDPOINT	BODY
Login into the Application via	POST	/index.php	USERNAME=ktsakos@isc.tuc.gr&PASSWORD=1234

Keyrock			

A device sends measures	POST	/packetR.php	{
			"identifier":
			"D1C19631103AD2AA9C780934FD62CE0A",
			"ambientLightInLux":"3584.3999997",
			"temperatureInCelsiusDegrees": 14,
			"pressure": "1.677721599609375E7"
			}

Searching for sensors measuring temperature or humidity	POST	/mongoParserAjax.php	(attrs.temperature!= undefined || attrs.humidity!= undefined)
A developer makes a subscription to a device giving its unique id	POST	/subcreate.php	["D1C19631103AD2AA9C780934FD62CE0A"]
Deploying a Mashup App which presents the highest temperature per day captured by 3 devices, save app infos to Context Broker	POST POST	/deploy.php /AppToCB.php	[{"appname":"abcdefgabc","info":[{"attribute":"temperature","operation":"HMAX","ids":["D1C19631103AD2AA9C780934FD62CE0A","D1C19631103AD2AA9C780934FD62CE0B"]}]}] {"appname :"abcdefgabc ","appdescription":"e.g: This app is showing cool stuff ","appid":"${appid-returned from the request after deploying the app}","App_scope":"assisted_living"}
Search apps of a specific developer and scope	POST	/Apps_to_retrieve.php	{"Username":"ktsakos@isc.tuc.gr","service_path":"assisted_living"}
Search Subscriptions to Sensors	GET	/MySensors.php	
Customer subscribes to specific apps	POST	/CustomerSubCB.php	
Access a Mashup Application	GET	/redirect.php/myappA	

4.2 Google's OnlineBoutique E-shop Benchmark Architecture[20]

Online Boutique is a cloud-native microservices demo application as described in the official GitHub page of the application [20] 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 orchestrate 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 [21]. 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.

The following figure 4.2-1 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.

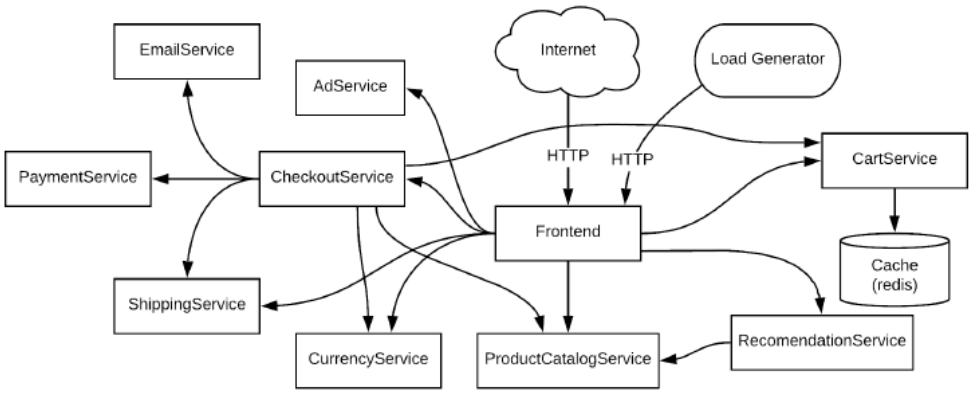


Figure 4.2-1: Online Boutique's Architecture

It should be mentioned that this application is suitable for applying the pre-described service placement 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 [20] 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.

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.

4.3 System Architecture

For each application's deployment, the networking configuration of each microservice and for all monitoring tools setup we use Kubernetes orchestrator. Each stateless application unit is deployed as a Deployment resource in the Kubernetes cluster with 1 replica (i.e. 1 Pod per Deployment). Also, each stateful microservice that stores application's data (e.g each database) is deployed in the Kubernetes cluster as Statefulset with 1 replica. For each deployment and statefulset we configure a corresponding Service linked to a specific communication port, in order to enable the network communication either internally among all microservices or externally for specific useful endpoints that need to be accessed for metrics collection or application stressing. All microservices are accessible internally with a ClusterIP service and those which are exposed outside of the cluster are accessible via NodePort Services. Finally, Horizontal and Vertical pod auto-scaling and cluster autoscaling which are Kubernetes mechanisms for managing the pods and Nodes size and resource allocation, are disabled, as we want to specify the initial number and size of Nodes that will be used to host each application and the respective requested resources for each Pod minimum requirements.

In order to enable application traffic management and monitoring for constructing the application graph and calculate the service affinities we exploit the Service Mesh architectural pattern advantages. More specifically, we deploy in our Kubernetes cluster Istio Service Mesh implementation with all its corresponding components for its Control Plane operations. Also, we enable the automatic injection of an Envoy Proxy sidecar container into each application's pre-existed pod. Upon every internal or external network communication of the Pods, the traffic is re-directed through the Envoy proxies to the backend microservice and vice versa. The Pod architecture, with the Istio Envoy injected in it and the backend service it serves, are presented in Figure 4.3-1.

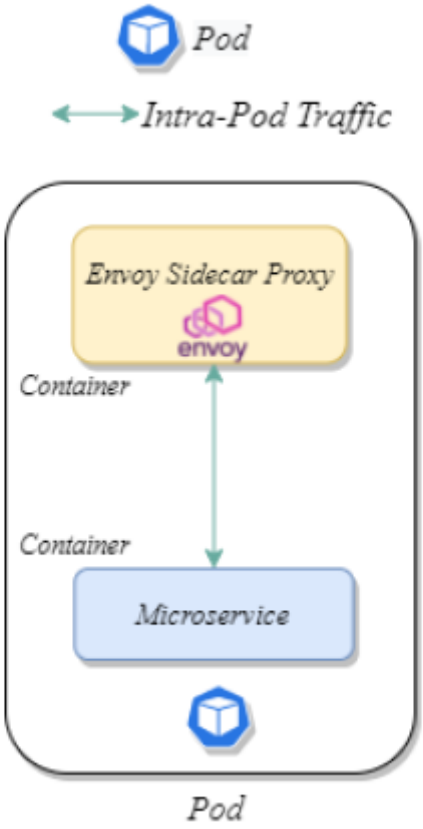


Figure 4.3-1: Pod Architecture with Injected Envoy Proxy by Istio

Furthermore, our kubernetes cluster consists of multiple nodes (e.g. virtual instances) which can host a finite number of Pods, including some of the Istio Services and Kubernetes components, depending on the available resources of the Node and the requested resources of each Pod. Istio monitors the application's network traffic and the exchanged messages through the Istio's Data Plane (the Pods with the injected Envoy Proxy's containers). Every Pod's Envoy communicates with all the other Pods inside the Node forming the Istio Mesh Traffic. The way that Istio configures each Pod to host an Envoy's sidecar container occurs by sending a configuration certificate to the Istio's Control Plane, so it can join the existing Mesh Traffic. Istio services are also deployed in the cluster's node as Pods. In addition, Node Exporters are installed in every cluster's node as Replicasets to monitor the available, the allocated and used node resources. These resources metrics are then available through Prometheus and via Kiali tools. Essentially, Prometheus Service scraps these metrics by requesting the data form the Node Exporters every specified period of time. Those metrics, can then be retrieved and consumed via the Prometheus UI with promql queries and visualized in Grafana. More specifically, we can have a brief look for each Node's and Pod's requested, used and available resources. In the same manner, Kiali collects these data from Prometheus Server to create the application's graph. The Node's architecture and the communication among the Istio Services and application's Pods are displayed in Figure 4.3-2.

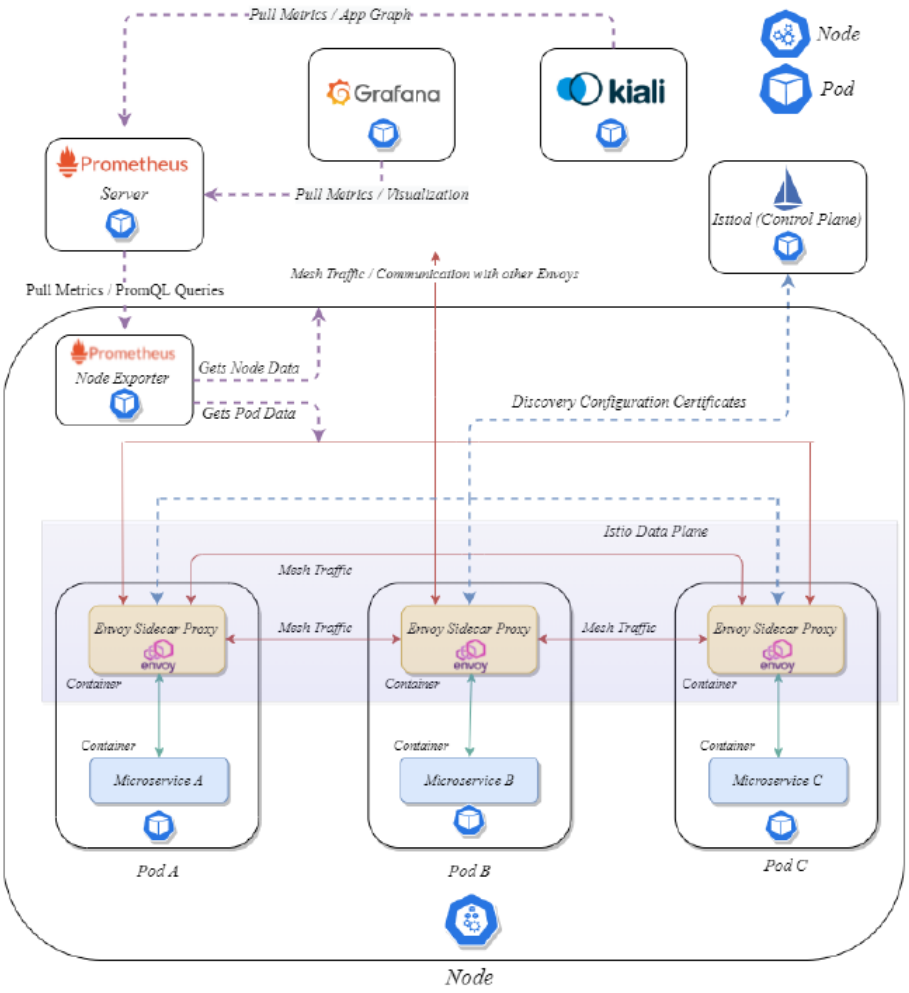


Figure 4.3-2: Application Pods placed in a Node with injected Envoy Proxies by Istio

The deployed Istio Services are randomly places inside the cluster's Nodes according to the decision of the Kubernetes Scheduler. The Istio Services are not injected with Envoy Proxies, they are deployed independently from each app's Pods in a different Kubernetes namespace, as they are not considered as part of the application's Data Plane and are used only to extract useful metrics from the cluster's Pods and Nodes for our placement strategies application and results evaluation.

Our Kubernetes cluster consists of a predefined finite number of Nodes, according to each application's requirements. The operator is responsible to estimate application's resource requirements and choose the appropriate number of nodes and their corresponding sizes in CPU, memory and storage (Instance types). Every Node communicates with each other inside the cluster through the Istio's data plane via the Pod's injected envoy Proxies. Kubernetes cluster has the control of creating and managing the Nodes and Pods. Any data extracted from external sources and services regarding the Nodes and Pods Information is requested from the Kubernetes cluster. In Figure 4.3-3, the whole cluster's architecture is presented in a Cloud Infrastructure, like Google Cloud Platform (GCP), which we use for this study's experiments. GCP provides users with two engines, the Kubernetes Engine and the Compute Engine. The first one provides Kubernetes as a Service to the user and the other

one provides pools of different types of VM Instances which are then managed and used by Kubernetes clusters to host each application's component (Pods).

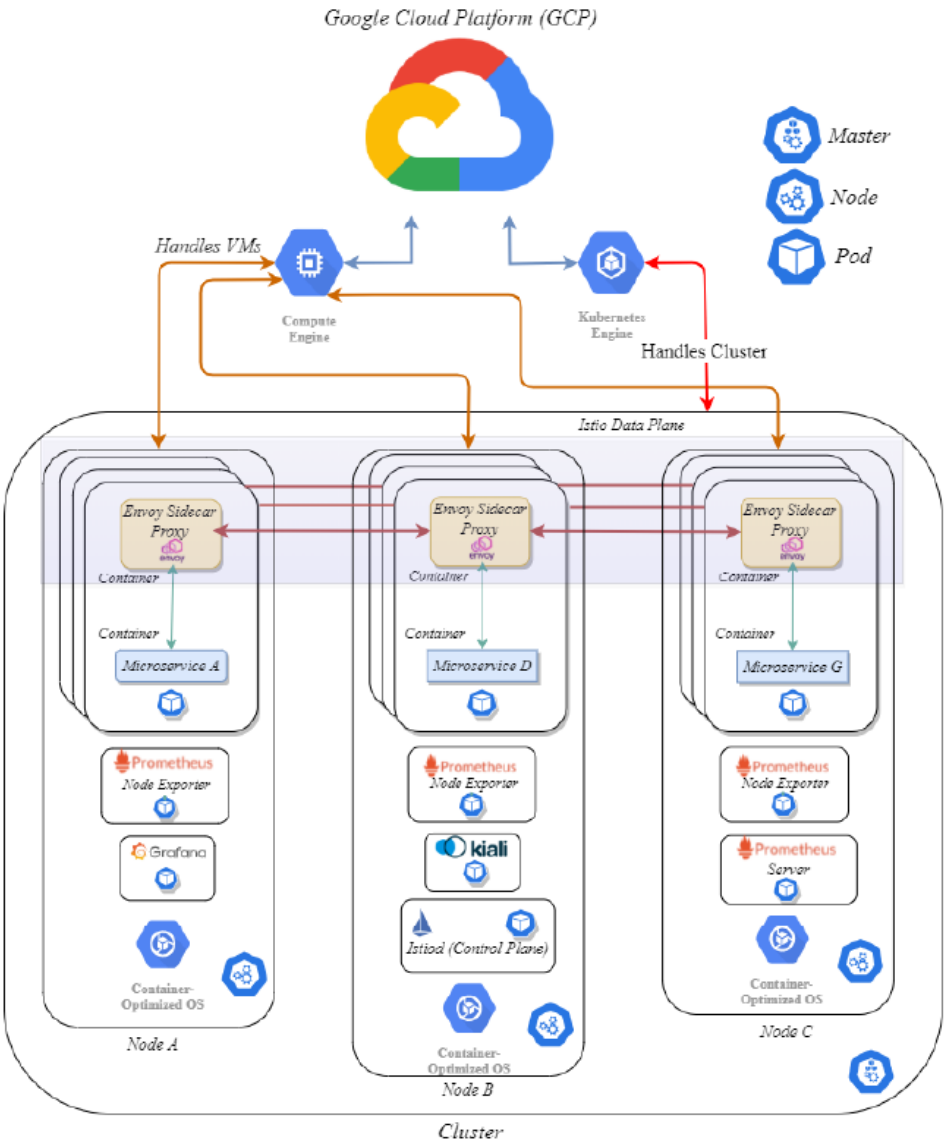


Figure 4.3-3: Kubernetes Cluster's Architecture with deployed Istio and Application's Pods in Google Cloud

5. Experimental Results

In this chapter we present the experimental results after applying some of the proposed service placement strategies in a Kubernetes environment. First of all, we discuss the algorithms or combination of algorithms which we apply. Then, we analyze the utilized Cloud Infrastructure for hosting all the tools and the benchmark applications used, along with the characteristics of the Kubernetes cluster we setup. Furthermore, we present each application's workloads sent during stress testing, and also the type of requests and the distribution of each kind of them for both applications, i.e iXen and OnlineBoutique eShop. Then we specify the generic cost function used by Cloud Providers to charge customers for

cluster resources allocation and finally, we present the results of the implemented and applied placement strategies in bar graphs related with the final cost after applying each placement method , the egress traffic reduction they achieved and the number of host machines needed eventually to host each app's microservices in comparison with the initial placement cost, egress traffic and number of host machines recorded while using the default Kubernetes approach.

5.1 Service Placement Strategies

In chapter 2 the algorithms were categorized in two main categories, the clustering algorithms which try to find clusters among microservices of an application based on their affinity's calculations only, and the placement algorithms which are based on partitioning algorithms combined with bin packing heuristic methods taking also into consideration the resource requirements of each microservice and the available resources of the nodes of the infrastructure.

In a first step we apply clustering algorithms which provide optimal results based on affinity calculations, which help reduce egress traffic effectively. In a second step, we aim to further reduce the cost of app hosting in a Kubernetes Cluster. This is done by packing the clusters into host machines using the best method from the placement algorithms category. The heuristic packing method (HP) is the most suitable approach for this step. HP can be applied to existing partitions of microservices and provides an optimal solution in polynomial time. This is possible as long as each cluster of microservices derived after placement decision can be placed on at least one host machine, considering the available resources.

Essentially, in the first step we try to achieve the highest intra-machine affinity and the lowest inter-machine communication for the maximum app performance by avoiding as much as possible network latency. In the second post-processing step we try to place those clusters by allocating as less as possible resources (host machines) for further cost reduction. The implemented and applied algorithms for the purposes of this study are the followings:

- Markov Clustering (MCL)
- Affinity Propagation (AP)
- Maximum Standard Deviation Reduction (MSDR)
- Markov Clustering + Heuristic Packing (HP)
- Affinity Propagation + Heuristic Packing (HP)
- Maximum Standard Deviation Reduction + Heuristic Packing (HP)

More specifically, in a first round of experiments we will compare the results of the three clustering algorithms (Markov Clustering, Affinity propagation and Maximum Standard Deviation Reduction) without taking into consideration the available and requested resources. In a second step we will combine these algorithms results with the heuristic packing algorithm to check if the results will be further improved by achieving further cost reduction and higher application's performance.

5.2 Infrastructure

In order to conduct the experiments of this study, we use the Google's Cloud Platform (GCP) infrastructure and its Kubernetes Engine (GKE) service, which will host and orchestrate each benchmark microservice based application and all needed tools for its monitoring and metric collection.

Cluster Design

For each application, iXen and OnlineBoutique, initial placement, stressing, monitoring and placement improvement purposes, we initialize a Kubernetes cluster in GKE with common characteristics. Each cluster is created in Europe-west3-b Zonal location type. We disable horizontal and vertical autoscaling for the purposes of this Thesis, so that available VMs and their resources allocation remain unchanged while running and stressing the apps and be lately chosen only from our own placement strategies/algorithms execution decisions. Essentially, we examine the performance of each placement solution within a Public Cloud Vendor's environment with VMs (kubernetes nodes) allocating the same amount of resources.

In addition, 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 purposes for simplicity and to highlight each strategy's achievement by itself. 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 5.2-1 displays the exact characteristics of the Kubernetes clusters for each benchmark application.

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 Pod Autoscaling	Disabled
Vertica Podl Autoscaling	Disabled
Cluster Autoscaling	Disabled
Cloud Logging	Enabled
Cloud Monitoring	Enabled

Table 5.2-1: Characteristics of Kubernetes Clusters

Each GKE cluster requires at least one node pool, which is responsible for provisioning virtual machines (VMs) that act as nodes in the cluster. When creating a node pool, an instance group is automatically generated, and virtual machines are created based on the desired number of nodes specified during the setup. The node pool is created along with the cluster and communicates with the GCP Compute Engine service to manage the scaling of the cluster by resizing the number of nodes as needed. More specifically, our node pool instantiates instances 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 with our own placement strategies decisions only. The boot disk type is standard type with size of 100GB. The Nodes are not pre-emptible, which means

that we allocate resources or reserve a finite number of VMs upon demand for initializing each application. Table 5.2-2 displays the main characteristics of the cluster's node pool for each application.

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

Table 5.2-2: Characteristics of Kubernetes Cluster Node Pool

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 other 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 application adequately, according to the Pod and Node restrictions and for testing the cost optimization case, after each successful execution of the proposed strategies. Figure 5.2-1 displays the cluster infrastructure on GCP.

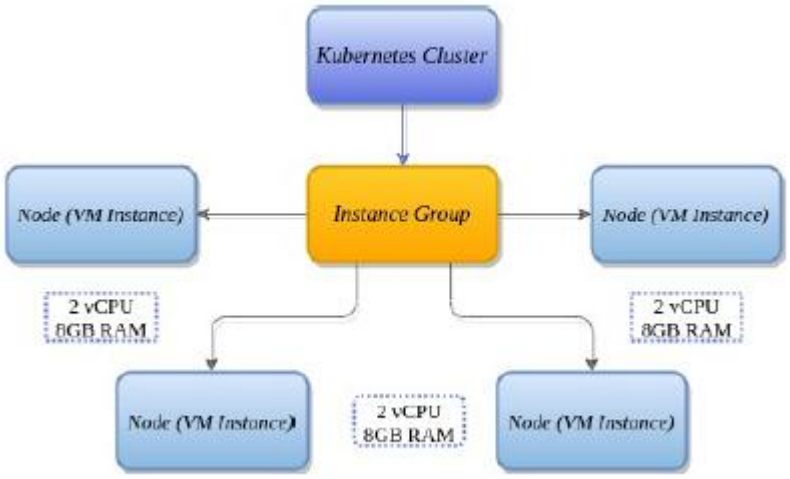


Figure 5.2-1: Kubernetes Cluster Infrastructure on GCP

Upon consecutive initializations of the cluster to deploy both applications, we have reached to the conclusion that both apps require at least 2 VMs to host them efficiently with the predescribed node pool characteristics. The selection of 4 Nodes for the initial placement 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 might be applied in real-life challenges and projects, in which DevOps teams 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 our benchmark application 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 safe information about the performance of our placement strategies in comparison with the initial Kubernetes placement decision of its own scheduler.

5.3 Application Stress Testing

In this section is presented the Stress testing process, which is applied to create traffic flows towards each application. Stress testing can be achieved through various tools, but for the experiments of this Thesis we utilize Apache JMeter[31].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 complexity of the stressing process. In the upcoming subsections we will thoroughly present the stressing methods, which are applied for both iXen and OnlineBoutique E-shop applications.

5.3.1 iXen Stressing

For IXen stressing, we initially create a test plan and configure the application's variables, such as External IP (i.e. endpoint of a Node or VM which is exposed to make the app accessible to the outside world) of the cluster, the number of threads to be created to simulate different users accessing the app and the service ports for accessing the application services from the Apache JMeter. 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 specified time period 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 in their body or their headers. The application's graph is presented as it is visualized through Kiali's User interface in Figure 5.3.1-1 while stressing the application. Grey rectangles represent the kubernetes pods and the yellow ones those ones with some internal error (e.g. any security issue or anything about readiness of the pod). Grey Triangles represent the kubernetes services and the yellow ones those who may have any issues regarding their healthy status. Finally, the green edges represent the HTTP communication and the blue ones the TCP communication. The purple arrow symbol is the module applying HTTP requests.

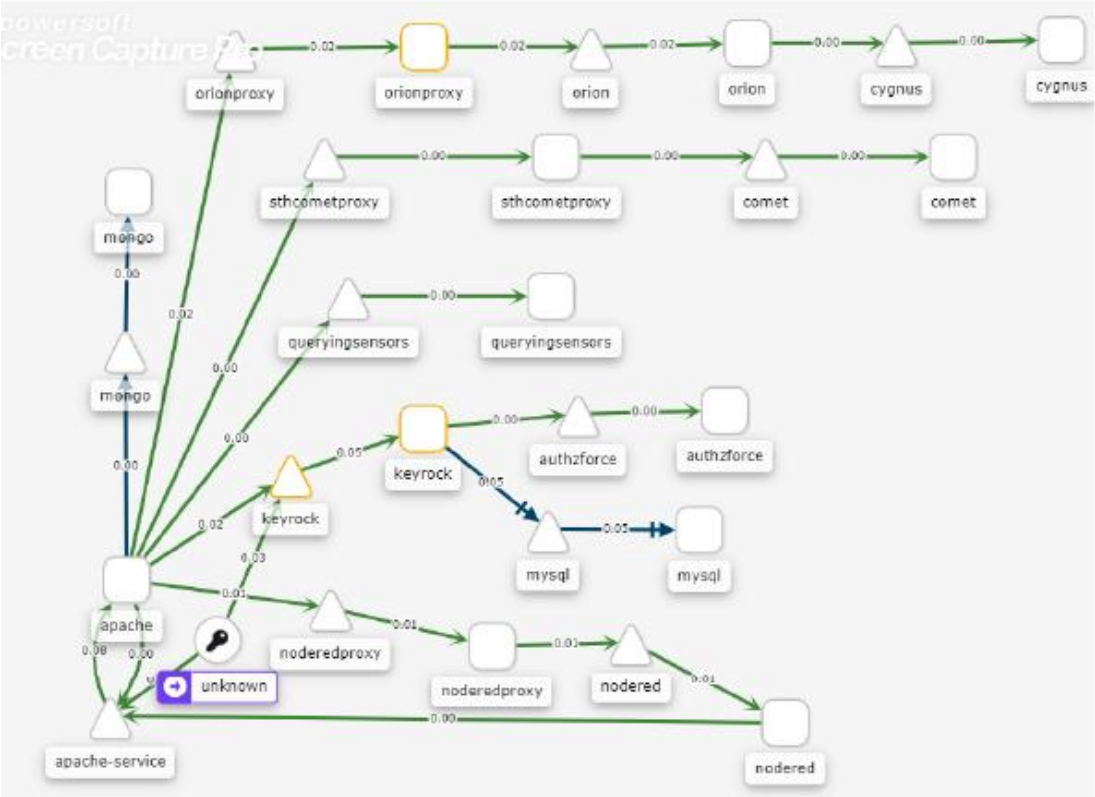


Figure 5.3.1-1: iXen Application's Graph visualized in Kiali GUI

Below, in Table 5.3.1-1, we present all the types of requests we make towards iXen in order to stress it and create all kinds of acceptable workloads among all its microservices serving its operations. In section 4.1 we presented all kinds of iXen requests with all headers and body format needed in detail.

Request	Type	Request Distribution
Login the App	POST	12,5%
Access Device measurements	POST	12,5%
Access Device Subscriptions	POST	12,5%
Deploy a new Mashup app	POST	12,5%
Search an existing app	GET	12,5%
Search for subscriptions	GET	12,5%
Make a new customer subscription	POST	12,5%
Access a Mashup app	GET	12,5%

Table 5.3.1-1: Types of Requests sent to stress iXen app

5.3.2 OnlineBoutique Stressing

In OnlineBoutique application we utilize two synthetics workload sources for stressing this application. The first one is the stressing applied from LoadGenerator, i.e a microservice constitutes a part of this app's architecture which creates network traffic towards application's endpoints by sending random kinds of requests. The second stressing source is the Apache JMeter tool, which is used also for iXen app stressing. Both stressing methods are described and analyzed below. Both stressing sources apply heavier workloads in size and number of requests into the application than the previous iXen stressing.

Stressing with LoadGenerator Microservice

The first method that we have available to stress the OnlineBoutique application is the LoadGenerator microservice, which is initialized upon the launch of the application in the Kubernetes cluster. This microservice generates random HTTP requests towards the application's endpoints. The initial application's graph is presented as it is visualized through Kiali's User interface in Figure 5.3.2-1. Grey Triangles represents the Kubernetes services and the grey rectangles the Kubernetes pods for each microservice. The green edges represent the HTTP communication between microservices and the weight shows the traffic rate (i.e. requests per second).

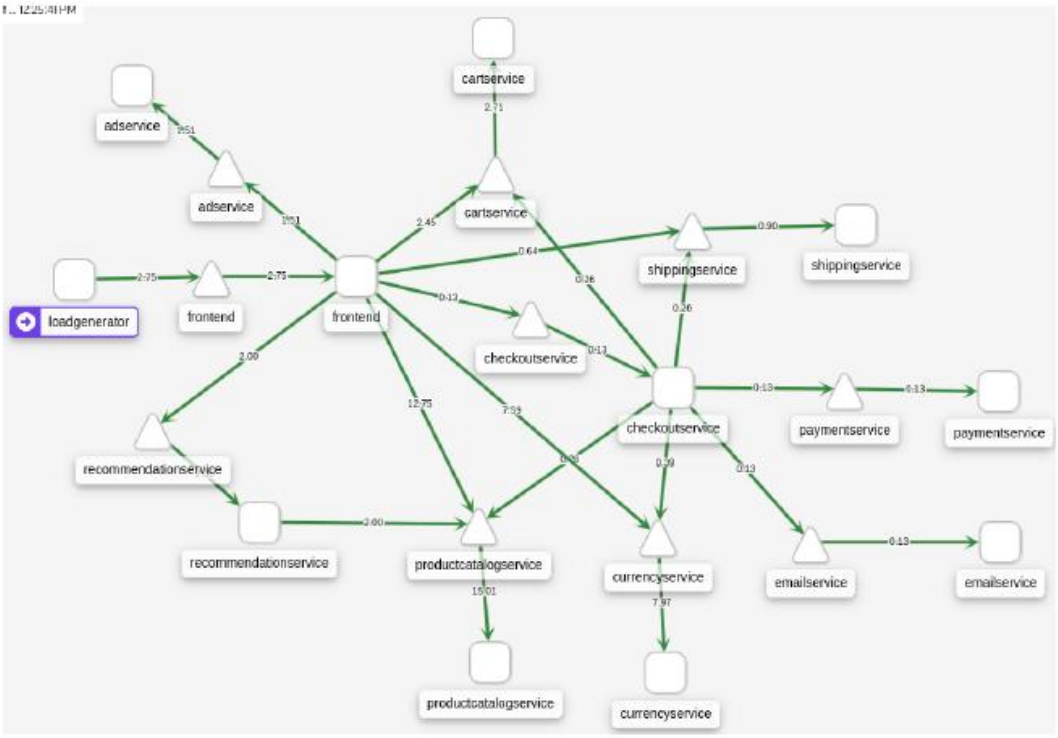


Figure 5.3.2-1: Online Boutique Application's Graph visualized in Kiali GUI

More specifically, the graph is produced during the application's stressing with LoadGenerator and it stresses the app with nearly 3 requests per seconds of random type. Table 5.3.2-1 displays the acceptable application's request types.

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)

Table 5.3.2-1: Types of Requests accepted by OnlineBoutique App

Stressing with Apache JMeter

The second method for stressing OnlineBoutique application is using the Apache JMeter tool as previously we analyzed for iXen stressing. Similarly, we initially create a Test Plan for the stressing procedure and then, we define the desired user defined variables which are the External IP (i.e. Node or VM IP address on which the app is exposed outside of the cluster) and the corresponding Node Port of the frontend service, from which we can send requests to all application's microservices. Then we define 4 major Thread Groups to apply the desired requests each one in different distributions. 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 fourth one attempts to change the payment currency randomly for 250 Threads. Only the Second Thread group sends 5 types of requests for 150 Threads each, so 750 requests are applied via this group. The difference between iXen Stressing and OnlineBoutique is that for the first one its stressing method sends all request types according to a uniform distribution, ensuring a consistent frequency across all request types. In contrast, the second method applied for OnlineBoutique introduces variability by dispatching each request type independently, with distinct and varying distributions that result in a randomized load profile for each request type.

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 send 10800 requests into the application, which can be calculated as nearly 30 requests per second for that specific time period. Table 5.3.2-2 makes the synopsis for all these thread groups by each type of requests.

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%

Table 5.3.2-2: Types of requests sent to stress Online Boutique App

For the experiments part and only for OnlineBoutique application, we will gather metrics and produce a placement result by executing each of the proposed strategies, initially from the default stressing module, i.e. the LoadGenerator microservice and then we will apply additional stressing from the Apache Jmeter service and re-execute the algorithms to compare the results. This approach enables an evaluation of the placement strategies by assessing whether their performance differs in response to variations in traffic type and volume.

5.4 Cost Function

Before presenting the experimental results after applying all service placement strategies we need to define the criteria upon which we can evaluate each strategy and compare it with each other. In this section, we set a cost function of our Kubernetes clusters presenting the GCP parameters for pricing the utilized and allocated resources.

Each benchmark application is deployed on Google Cloud Platform (GCP) Infrastructure and is orchestrated by Kubernetes Engine given as a Service by the provider. Kubernetes is responsible for managing and orchestrating a set of Nodes (Instances of GCP) and Pods deployed on them with specific resource requirements (memory, CPU, etc.). Moreover, Pods communicate either internally with other Pods (hosted on the same Node), or externally with Pods in a different Node. There are also manageable Kubernetes resources called persistent volumes which also demand disk storage, storing data to be mounted in the pods file system. All these requirements are charged separately in a Cloud Infrastructure and a cost function should be defined to estimate the cluster's fees. In Table 5.4-1, we present the symbols needed to set the 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*

Table 5.4- 1: Parameters describing Cost Function

Most of the Cloud Providers determine the billing factors for the users based on the Network Traffic passed between different instances (VMs) and even more between different instances which are located on different zone areas (Geographically). Also, the resource allocation is also considered as a crucial factor for charging which is mostly based on CPU, RAM and Disk Storage allocation. Cloud Providers define different machine types, known also as flavors, with specific pre-allocated resources and exact cost charging per time unit. Based on all these factors we define below a cost function which is going to be used to estimate the cost of a Kubernetes cluster to host and manage each benchmark architecture during serving all kinds of requests towards them.

$$TotalCost = Cost_{CPU} + Cost_{RAM} + Cost_{Traffic} + Cost_{Storage} \quad (5.1)$$

$$Cost_{CPU} = \sum_{i=1}^{N} cpu_{cost}(M_i) \times h_i \quad (5.2)$$

$$Cost_{RAM} = \sum_{i=1}^{N} ram_{cost}(M_i) \times h_i \quad (5.3)$$

$$Cost_{Traffic} = \sum_{i=1}^{N} \sum_{j=1, j\neq i}^{N} [t_{in}(i \rightarrow j) \times cost_{ingress} + t_e(i \rightarrow j) \times cost_{egress}] \quad (5.4)$$

$$Cost_{Storage} = \sum_{i=1}^{N} s_i \times storage_{cost}(M_i) \times hi \quad (5.5)$$

All the above equations are described as a unified way for Cloud Providers to estimate the cost charging of their customers. However, it is possible to face also additional charging criteria during cloud usage such as costs for GPU usage according to different application's requirements or fees for optimization tools usage that are provided such as load balancers, Pod auto-scaling mechanisms and external communication enabling.

Cloud Provider's Cost on GCP

Based on the Google Cloud Platform's documentation [34] the above equations with the cost charging factors differentiate as follows:

$$Cost_{CPU} = \sum_{i=1,}^{N} cpu_{cost}(M_{i,R}) \times h_i \quad (5.6)$$

$$Cost_{RAM} = \sum_{i=1,}^{N} ram_{cost}(M_{i,R}) \times h_i \quad (5.7)$$

$$Cost_{Traffic} = \sum_{i=1}^{N} \sum_{j=1, j\neq i}^{N} [t_e(i \rightarrow j, R) \times cost_{egress}(R)] \quad (5.8)$$

$$Cost_{Storage} = \sum_{i=1}^{N} s_i \times storage_{cost}(M_i) \times h_i \quad (5.9)$$

Cost of CPU and RAM depends only on the Machine type, 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, instead the Egress Traffic is charged according to the Zone and Region of each VM and the size of GB exchanged between the host machine (e.g. how much data one VM sends and how far is the destination VM from its own region and zone).

Actual Cluster Cost

Before presenting the cost calculations for the experimental part of our study, it is important to note that we operate within a homogeneous environment in Kubernetes. All virtual machines (VMs) are of the same type, located in the same zone and region, with identical resource allocations. As a result, the cost for initializing the requested number of VMs remains consistent, and the sum of the individual cost components can be simplified to the VM cost multiplied by the number of VMs used.

The storage requirements for the benchmark applications and monitoring tools are considered negligible, as the data volume is minimal and consistent across all service placement strategies. Therefore, the cost of storage is excluded from the cost calculations.

In terms of the cloud provider's cost function, specifically for Google Cloud Platform (GCP), the cost factors depend on the VM type, region, and runtime hours. CPU and RAM costs are determined by the machine type, while storage costs vary according to the disk type and cluster initialization. Egress traffic is charged based on the data size and the distance between VMs in different zones or regions.

Given the uniformity in VM type, zone, and resource allocation, the cost for CPU, RAM, and egress traffic can be calculated with the following equations:

$Cost_{CPU} = n \times 2vCPU \times cpu_{cost} \times h_i$ (5.10)

$Cost_{RAM} = n \times 8GB(RAM) \times ram_{cost} \times h_i$ (5.11)

$Cost_{Traffic} = cost_{egress} \times \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} t_e(i \rightarrow j)$ (5.12)

$Cost_{Storage} \cong 0$ (5.13)

In general, the cost of CPU and RAM depends only on the hours of the operation of each Node and the total size of Nodes consist the existing cluster. The cost of storage is too low to be considered in our evaluations. As VMs belong in the same Zone and Region, the cost of Egress is fixed and proportional to the total amount of GB leaving a Node to reach another one in the cluster. In the following Table 5.4-2 we present the actual GCP costs for the utilized cluster infrastructure and VMs.

Description	Cost (USD)
Predefined vCPU	$0,0285/vCPU/hour
Predefined RAM	$0,003819/GB/hour
VM-to-VM Egress traffic	$0,01/GB

Table 5.4-2: GCP's Prices for Kubernetes Cluster utilized resources

As we initialize 4 VMs on each cluster to deploy each benchmark application (heuristic chosen by the default scheduler till no pods pended to be deployed based on its requirements) and with prior knowledge of the hourly fees of the GCP, we can calculate the total monetary hourly cost function of our infrastructure for the initial placement with the following equation

$TotalCost = 0.350208 \times h_i + \frac{0.01}{GB_{h_i}} \times t_e$ (5.14)

, where $0.350208 \times hi = 4 \times (0{,}0285\$ \times 2vCPU \times hi) + 4 \times (0{,}003819\$ \times 8GB \times hi)$

It must be clarified that in the results section 5.5 we calculate the cost results of the clusters for 1 month period of usage. The hourly costs presented in Table 5.4-2 are modified accordingly to produce the value of the respective cost factors per month of usage for a greater estimation of each service placement strategy's expenses.

5.5 Results

After applying all placement strategies reported in section 5.1 for each of the two applications, we present in this section the number of host machines utilized, the egress traffic observed within 1 hour, the execution time for each algorithm to provide the output of the placement strategy decision and the total estimated monthly operational costs based on proportional egress traffic amount within 1 month period and the overall resource allocation.

All these metrics are presented in graph bars, providing a clear, comparative view of the infrastructure needs, egress traffic variations and cost efficiency of each placement strategy after being applied to each of the two benchmark applications.

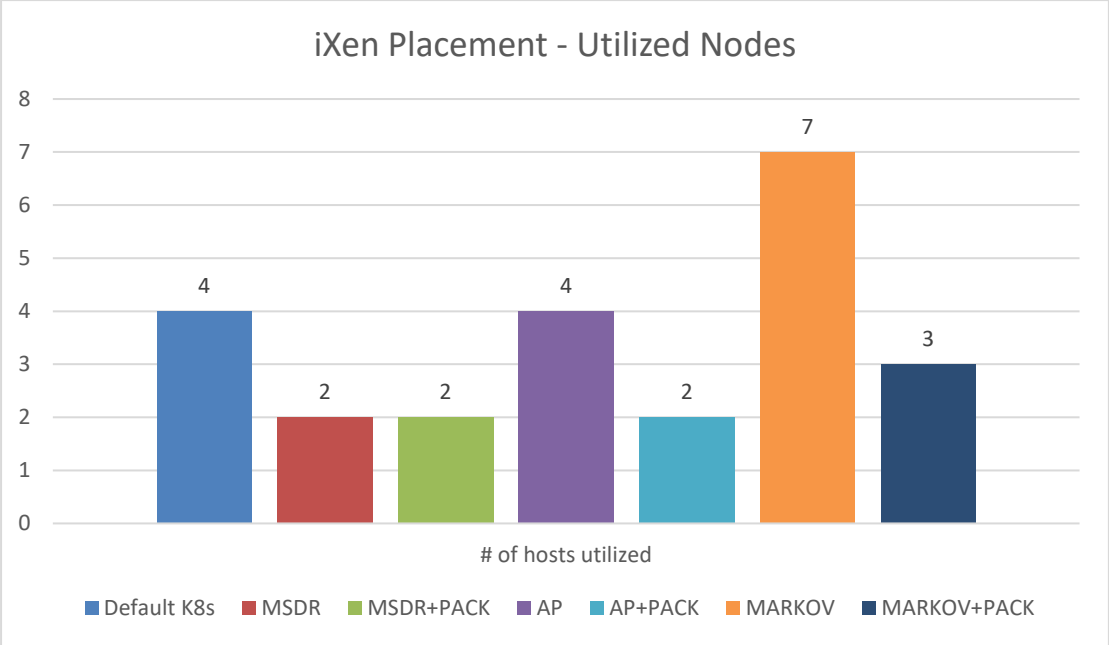
5.5.1 iXen Results

In this section we demonstrate and discuss the results after applying all placement algorithms and stressing iXen application.

From resource usage perspective, i.e the number of utilized nodes needed to host all pods of iXen application, the Maximum Standard Deviation Reduction (MSDR) Clustering (with and without Heuristic Packing) and Affinity Propagation (AP) with Heristic Packing (PACK) used the least nodes, specifically 2, optimizing node utilization well. This reduction in nodes translates to cost savings.

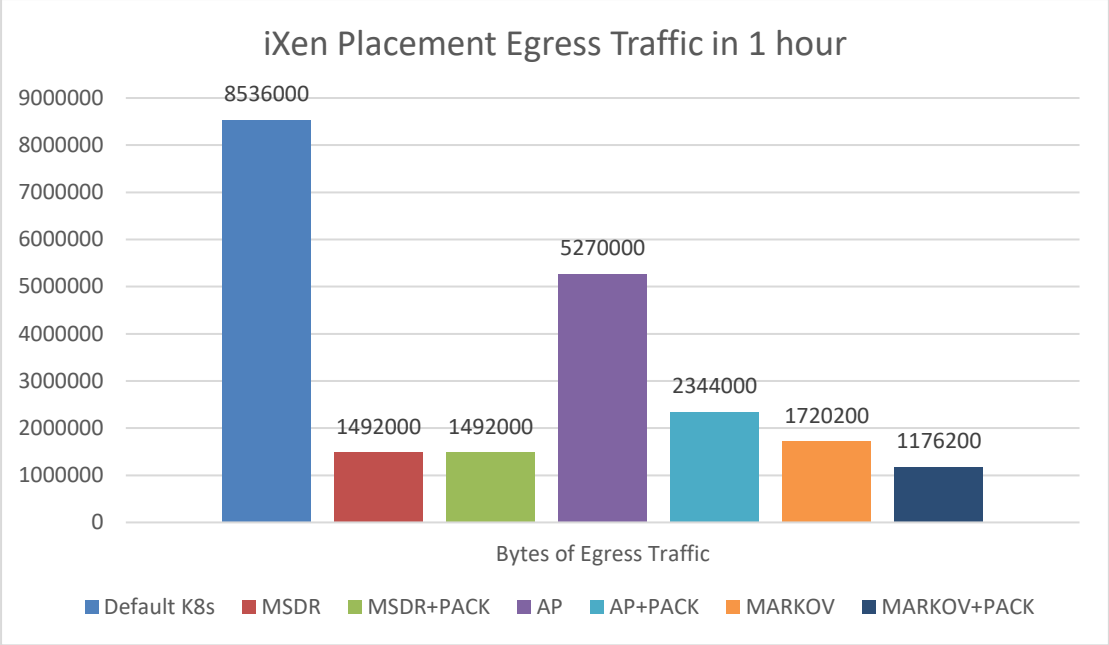
Markov clustering combined with Heuristic Packing also minimized node usage to 3, although standalone Markov Clustering was the most node intensive choosing 7.

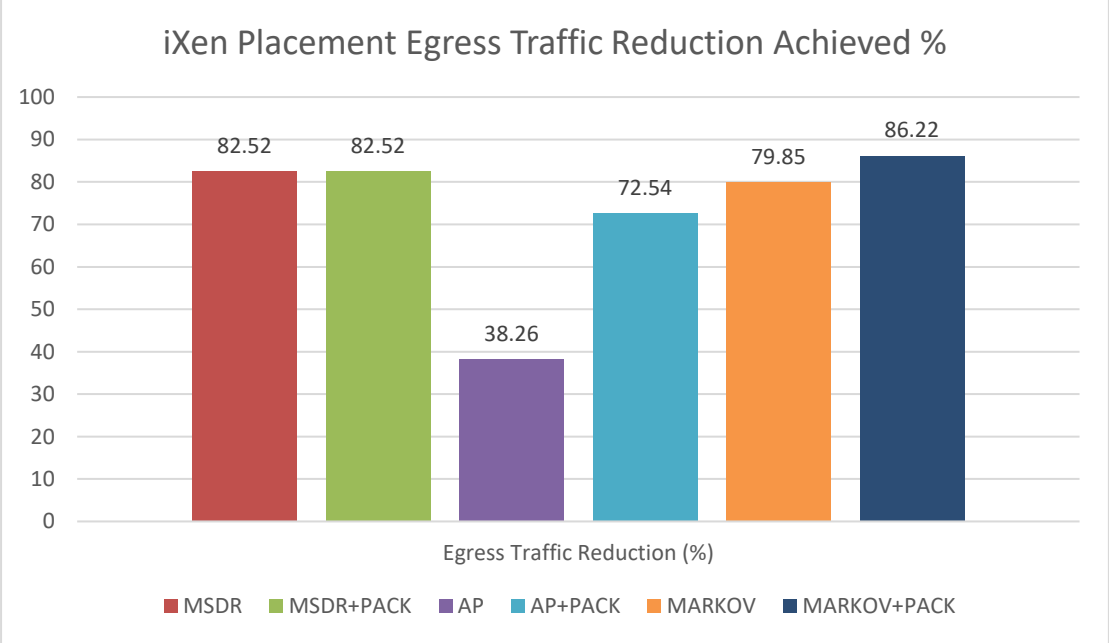
All metrics related with utilized Nodes chosen by each strategy for iXen placement are presented in the graph bars in the diagram below:



In the next two diagrams, we can see respectively the Egress Traffic in bytes observed within 1 hour during stress testing after each iXen's placement and the total percentages of Egress Traffic Reduction achieved by each strategy comparing it with the default k8s one which was applied initially.

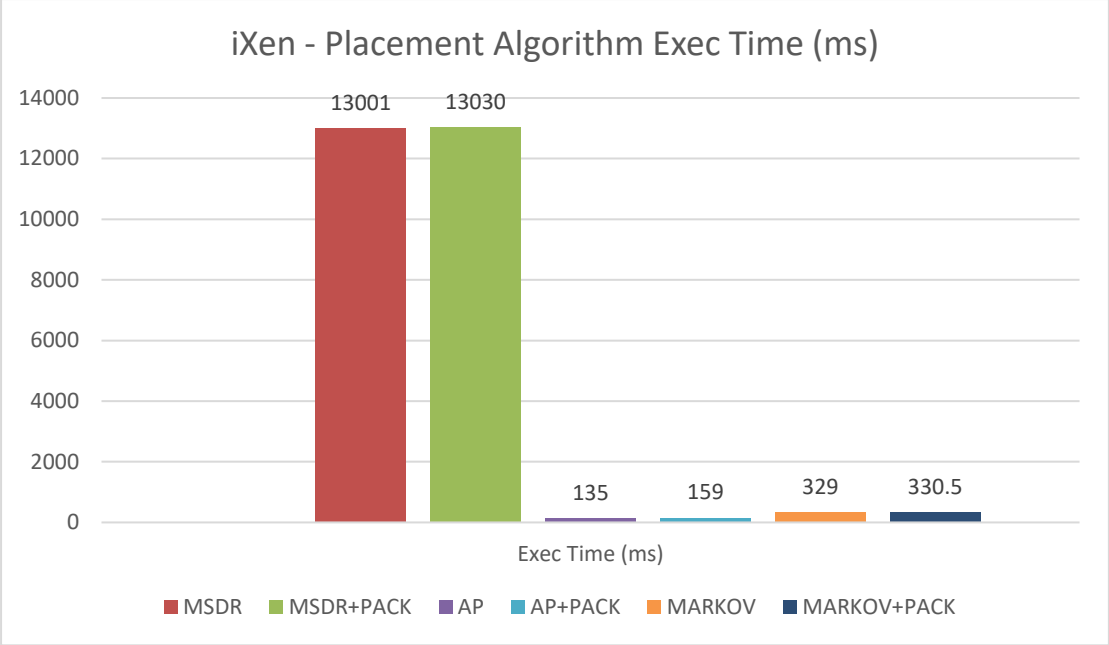
Markov Clustering combined with Heuristic Packing achieved the highest egress traffic reduction (86.22 %), followed closely by Maximum Standard Deviation Reduction Clustering (82,52%). These are strong reductions, indicating that these algorithms optimize inter-node communication effectively which has an impact especially on the application's performance and on cost savings for hosting and operating it.



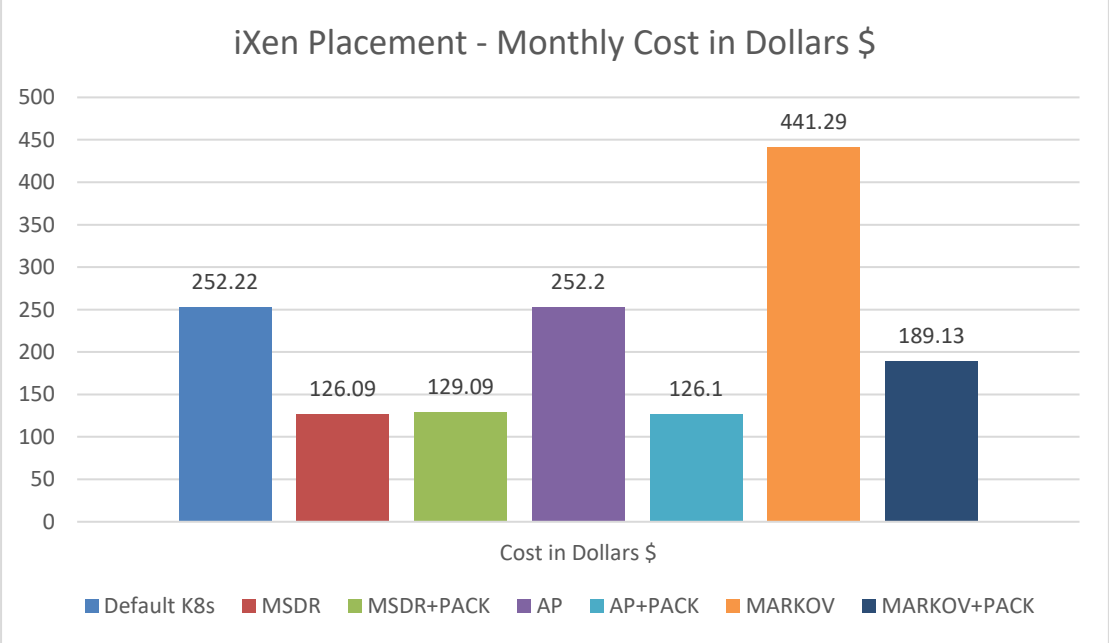


Speaking about each placement strategy's execution time, i.e how much time taken till they reach a final decision and give an output about each component's placement, we can observe in the following diagram that Affinity propagation (with and without Heuristic Packing) achieved the shortest execution times, both under 160 ms, which could be beneficial in environments requiring rapid adaptation.

Maximum Standard Deviation Reduction Clustering had the longest execution times (~13 seconds), indicating a heavier computational load. This may affect scalability, especially in dynamic environments with frequent redeployments.



From monthly cost perspective in US Dollars, we can see the calculations for each placement strategy in the diagram below:



Affinity Propagation combined with Heuristic Packing and Maximum Standard Deviation Reduction Clustering offered the lowest costs (~$126), making them the most cost-effective strategies. Markov Clustering alone, incurred the highest cost ($441), due to its higher node usage (7), while its combination with heuristic Packing reduced costs to $189 by bringing node usage down to 3 and achieving the highest egress traffic reduction (~86%).

These initial insights suggest that some strategies (like MSDR Clustering and Affinity Propagation with Heuristic Packing) balance cost efficiency, egress traffic reduction and number of utilized nodes optimization well, though they vary significantly in execution times.

5.5.2 Online Boutique Results

In this section we demonstrate and discuss the results after applying all placement algorithms and stressing OnlineBoutique application.

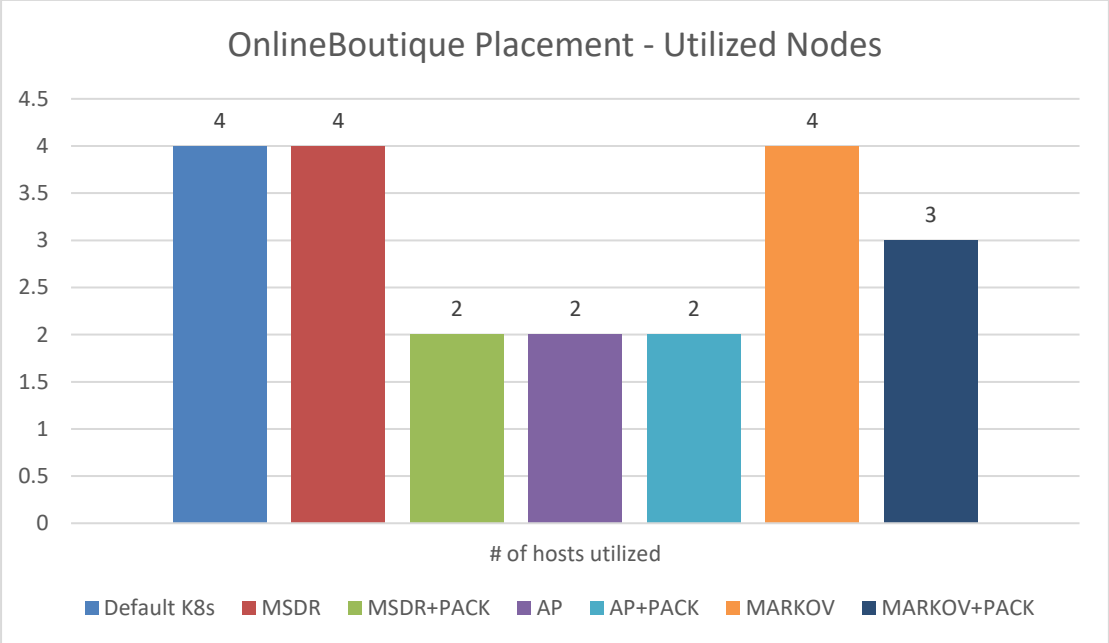
In section 5.5.2.1 are presented the results after stressing only with Load Generator component and in section 5.5.2.2 we can find the results after stressing OnlineBoutique application with both load generator and JMeter tools.

5.5.2.1 Stressing only with Load Generator

In this section we demonstrate and discuss the results after applying all placement algorithms and stressing OnlineBoutique application with only Load Generator.

Regarding utilized nodes presented in the diagram below, Affinity Propagation (with and without Heuristic Packing) and Maximum Standard Deviation Reduction Clustering with Heuristic Packing required the fewest nodes (2). Reducing node usage to this level can provide significant cost savings, as reflected in the monthly costs.

Markov Clustering with Heuristic Packing reduced nodes to 3 from initial 4, showing moderate improvement. The other methods, including the default k8s placement, did not achieve a node usage below 4.

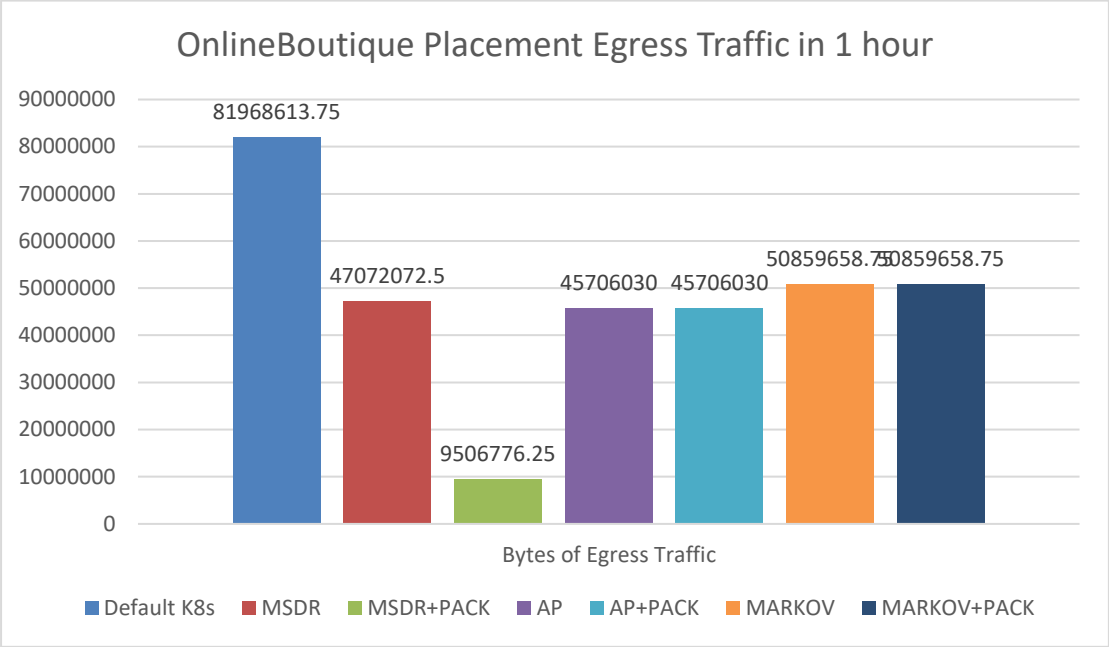


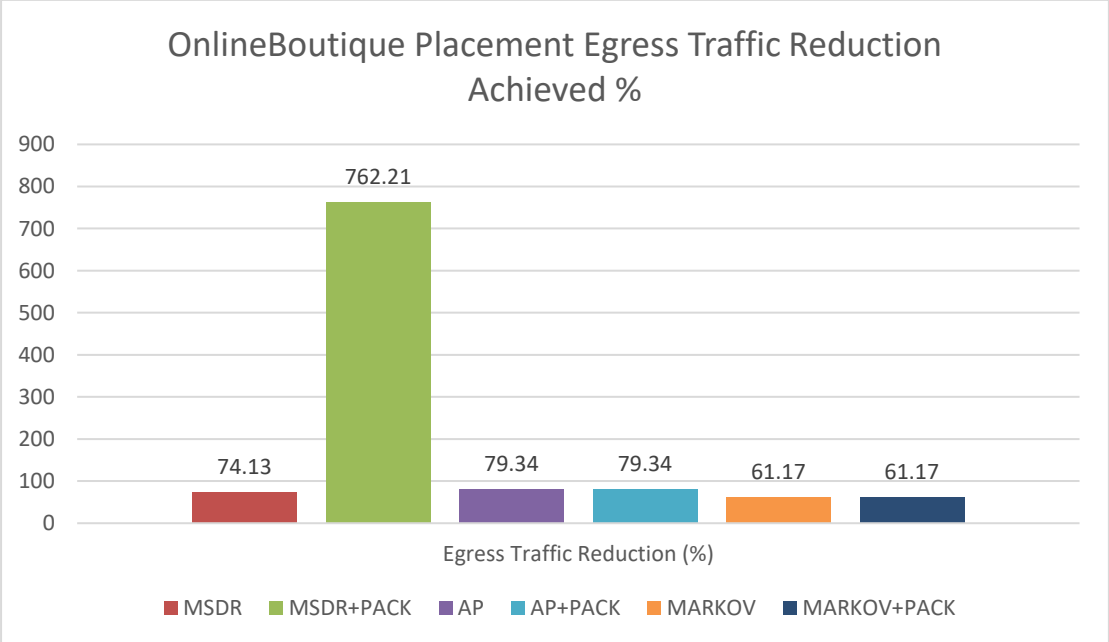
In the next two diagrams, we can see respectively the Egress Traffic in bytes observed within 1 hour during stress testing after each OnlineBoutique's placement and the total percentages of Egress Traffic Reduction achieved by each strategy comparing it with the default k8s one which was applied initially.

Maximum Standard Deviation Reduction Clustering combined with Heuristic Packing showed the most substantial reduction in egress traffic (~762%), a promising result indicating high efficiency in minimizing cross-node traffic.

Both Affinity Propagation approaches achieved a solid 79.34% reduction in egress traffic, second only to Maximum Standard Deviation Reduction with Heuristic Packing. These reductions show that these algorithms may be effective clustering approaches for applications with high communication demands.

Markov Clustering, in both configurations, had relatively modest reductions (61.17%), suggesting that it may be less effective in reducing inter-node traffic for this workload.



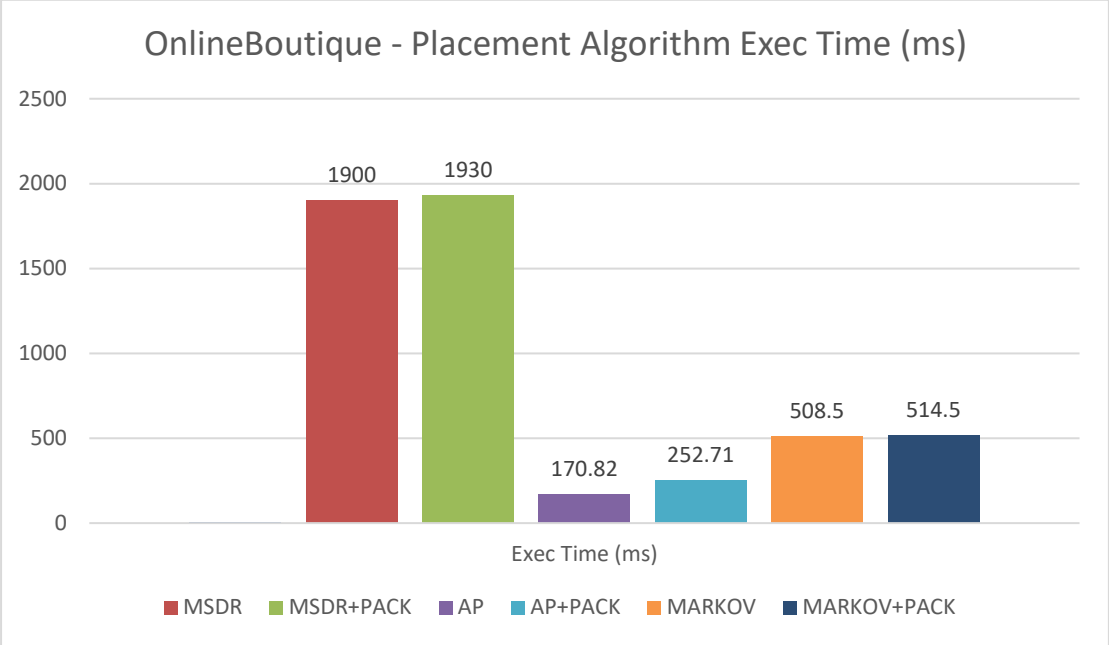


Discussing each placement approach's execution time, Affinity Propagation once again demonstrated the fastest execution times (between 170-253 ms), making it the most

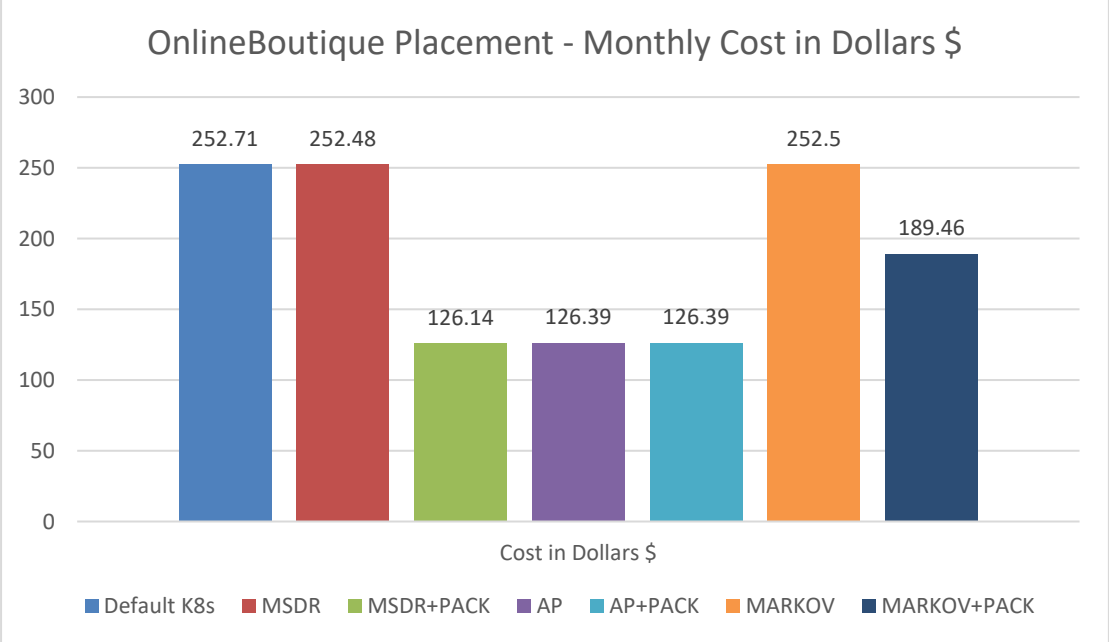
responsive strategy under this stress scenario. This could be valuable in dynamic environments where quick decision-making is crucial.

Maximum Standard Deviation Reduction Clustering took between 1.9 and 1.93 seconds, still manageable but much longer compared to Affinity Propagation.

Markov Clustering required around 500 ms, placing it in the middle of the pack.



From monthly cost perspective in US Dollars, we can see the calculations for each placement strategy in the diagram below:



Affinity Propagation and Maximum Standard Deviation Reduction with Heuristic Packing both achieved the lowest costs (around $126 USD), demonstrating that these algorithms balance node usage reduction and cost efficiency well.

Markov Clustering with Heuristic Packing reduced costs compared to the default and standalone Markov Clustering but remained higher than the aforementioned methods at ~$189 USD.

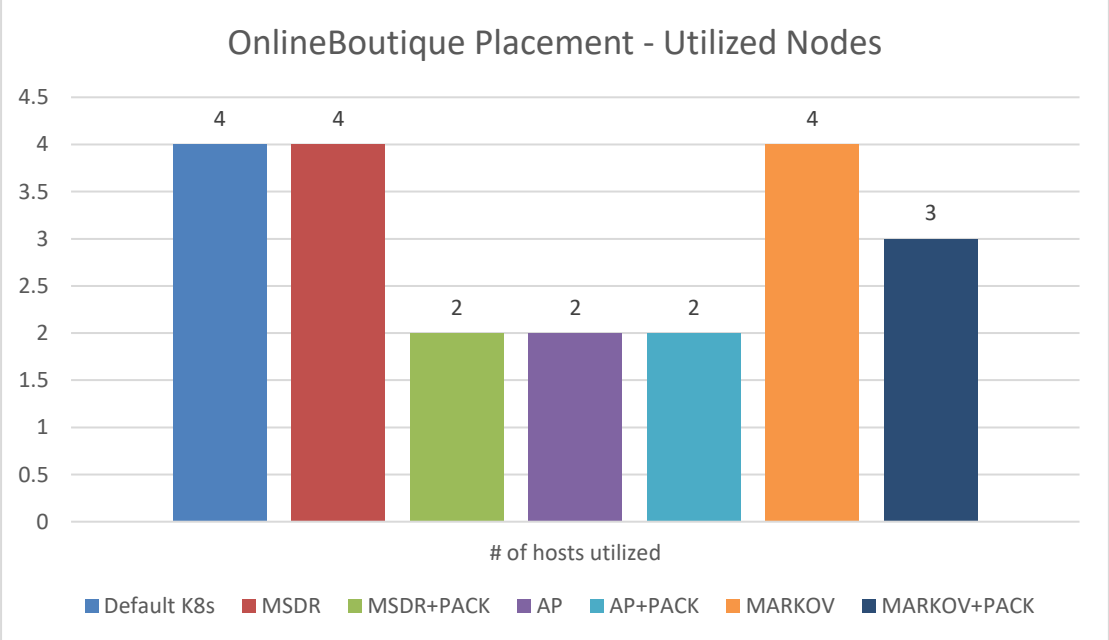
These insights indicate that Affinity Propagation (both with and without Heuristic Packing) and Maximum Standard Deviation Reduction Clustering with Heuristic Packing are leading strategies for the OnlineBoutique app under Load Generator stress, balancing egress traffic reduction, lower node usage and cost efficiency.

5.5.2.2 Stressing with both Load Generator & JMeter

In this section we demonstrate and discuss the results after applying all placement algorithms and stressing OnlineBoutique application with both Load Generator and JMeter tools. By comparing the outcomes from both load generation setups, we aimed to determine the sensitivity of the system's behavior to variations in load distribution.

From the presented node utilization in the diagram below, Affinity Propagation (with and without Heuristic Packing) and Maximum Standard Deviation Reduction Clustering with Heuristic Packing consistently minimized node usage to 2 across both stress scenarios, making them the most efficient in terms of node allocation.

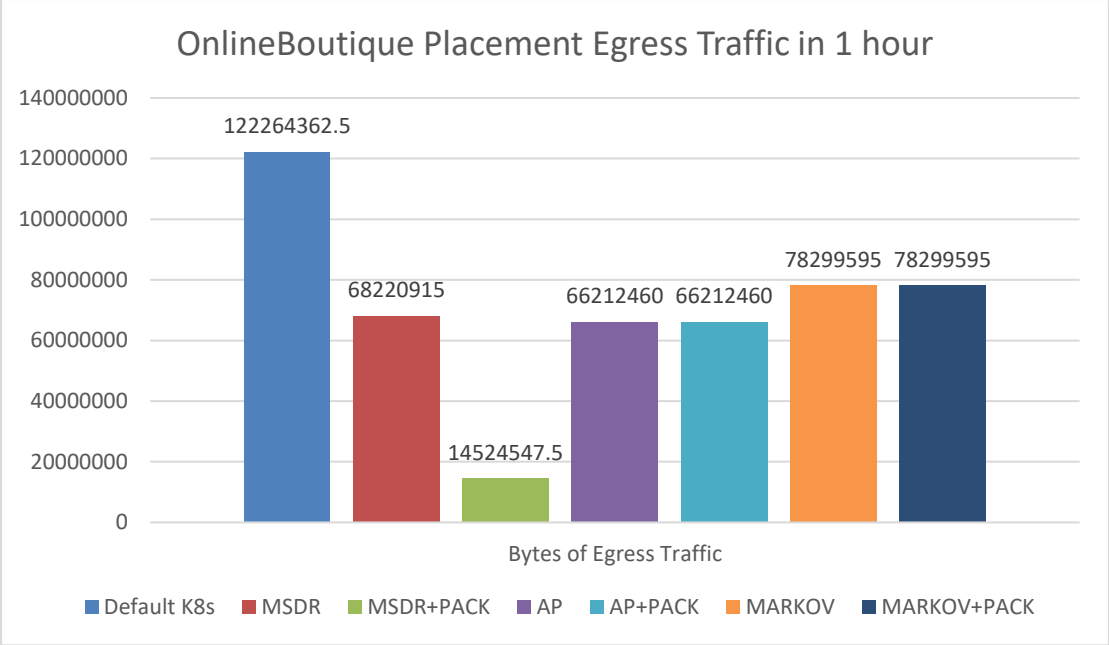
Markov Clustering with Heuristic Packing reduced nodes to 3, though standalone Markov Clustering and Default Kubernetes placement remained at 4 nodes in both tests.

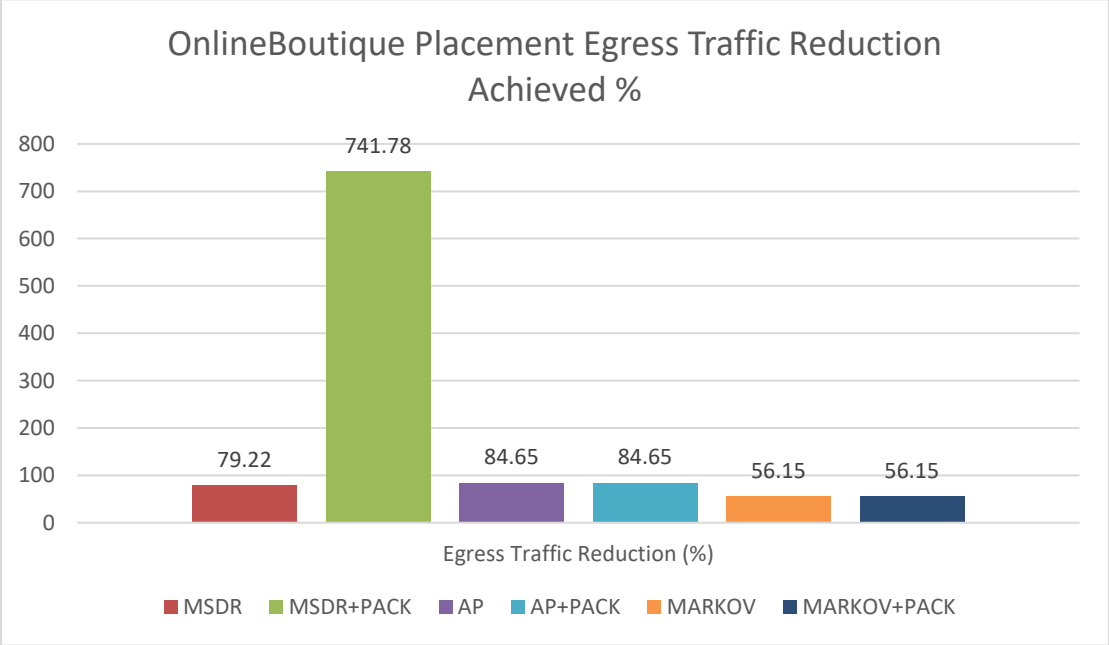


From Egress Traffic reduction scope shown in the two diagrams below, **Maximum Standard Deviation Reduction Clustering with Heuristic Packing** achieved again the greatest reduction (~742%).

Also, both Affinity Propagation methods provided a solid reduction again (~85%).

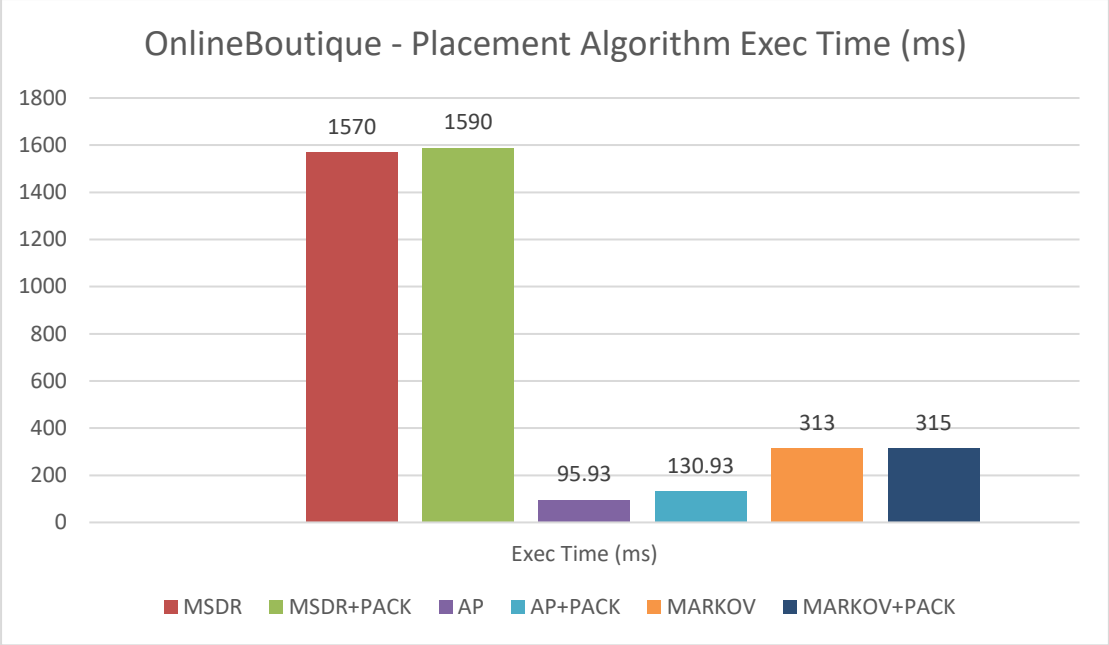
Markov clustering methods showed a more modest reduction (~56%).





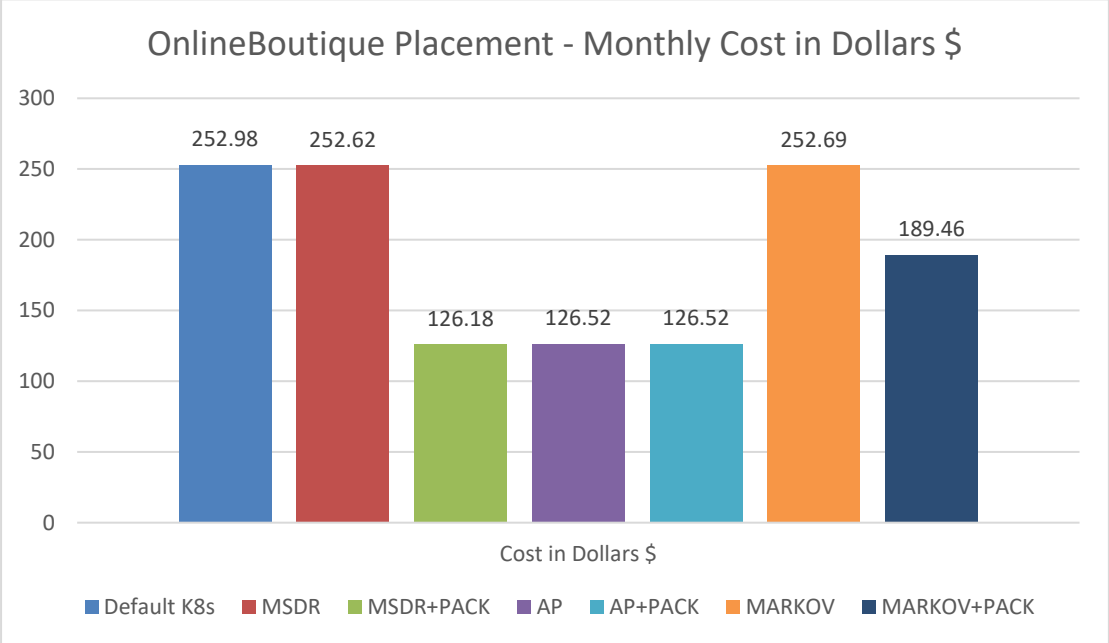
According to the execution time of each strategy shown below, Affinity Propagation (both with and without Heuristic Packing) executed the fastest, with times as low as 95.93 ms under the combined Load Generator and JMeter stress, indicating superior speed and responsiveness.

Maximum Standard Deviation Reduction Clustering consistently took longer (1.5 seconds), and **Markov Clustering** was again in the mid-range (~300ms).



Regarding the monthly costs, Affinity Propagation (with and without Heuristic Packing) and Maximum Standard Deviation Reduction Clustering with Heuristic Packing were the most cost-effective, keeping monthly costs around $126 in both stress scenarios.

Markov Clustering with Heuristic Packing reduced costs to $189 from the standalone Markov Clustering cost of $252-253, but it remained higher than the other optimized methods.



From sections 5.5.2.1 and 5.5.5.2 we can come to the conclusion that changing the load distribution by combining two stressing methods did not change the outcomes of the placement strategies comparison and did not have any impact in any of the criteria.

5.6 Discussion

Comparing the results after stressing and placing both application with all placement strategies chosen for this study, we can come to the following conclusions.

From Node Utilization perspective, both applications achieved reduced node utilization with Affinity Propagation combined with Heuristic Packing and Maximum Standard Deviation Reduction with Heuristic Packing. The further reduction of nodes utilization with Heuristic Packing application is expected because the clustering algorithms by themselves do not consider the node resources availability but only communication affinities among microservices which cannot lead to an optimal solution from resource savings scope.

Commenting the Egress Traffic Reduction results, OnlineBoutique produced higher egress traffic values under both stress scenarios than iXen, likely due to its workload profile and communication patterns. The placement algorithms had a slightly greater impact on egress traffic reduction in OnlineBoutique than in iXen, suggesting these methods may be more effective for applications with higher communication demands.

Regarding the execution time, across both applications, Affinity Propagation methods were significantly faster than Maximum Standard Deviation Reduction Clustering, with Markov Clustering consistently falling between these two. This speed difference could make Affinity Propagation preferable for real-time or frequently changing workloads.

Finally, considering the cost efficiency, in both applications, the same three methods—Affinity Propagation with Heuristic Packing, Maximum Standard Deviation Reduction with Heuristic Packing, and Affinity Propagation alone—achieved the lowest monthly costs.

6. Conclusion and Future Work

Based on the results of this study, we can conclude that optimizing service placement in microservices-based applications can significantly reduce infrastructure costs and improve overall performance. The findings suggest that clustering algorithms, such as **Affinity Propagation** and **Maximum Standard Deviation Reduction**, when combined with **Heuristic Packing**, offer the best balance between minimizing resource usage and reducing inter-service communication overhead. These strategies can be particularly beneficial for applications that experience high traffic between services, such as IoT applications (iXen) and e-commerce platforms (Online Boutique).

While **Affinity Propagation** demonstrated the fastest execution times, making it suitable for dynamic environments with fluctuating workloads, **Maximum Standard Deviation Reduction** achieved better long-term resource optimization, albeit at the cost of slower execution. Therefore, the choice of placement strategy should consider the application's specific requirements in terms of responsiveness versus long-term cost savings.

Future work could investigate how these clustering and placement strategies scale as the number of microservices and the complexity of workloads increase. This would help assess the strategies' performance in larger, more complex applications or multi-cloud environments.

An interesting direction would be also to explore hybrid strategies that combine the strengths of fast execution algorithms (e.g., Affinity Propagation) with the resource

optimization of slower, more computationally expensive strategies (e.g., Maximum Standard Deviation Reduction). This could result in more adaptive and balanced solutions.

Another potential area for future research is the integration of machine learning techniques to predict optimal placement strategies based on historical workload data. Machine learning models could be trained to forecast system load patterns and dynamically adjust placement strategies in real-time.

Future work could explore how these algorithms might be adapted to optimize for energy consumption, which is becoming increasingly important in cloud and edge computing environments.

Expanding the service placement problem to multi-cloud and edge environments could introduce new challenges and opportunities. Investigating how these strategies perform in distributed, heterogeneous environments (with more complex network topologies and diverse infrastructure) would provide valuable insights into the viability of these methods in next-generation cloud architectures.

Finally, ambient mesh exploitation, a new data plane mode that Istio provides without sidecars, could leverage the full potential of service meshes to dynamically adjust service placements based on real-time traffic patterns and network performance, presenting a promising area for future research. This could enable even more granular control over service interactions, traffic routing, and resource allocation, enhancing both performance and scalability in large, distributed cloud environments.

REFERENCES

1) Sampaio Junior, Adalberto & Rubin, Julia & Beschastnikh, Ivan & Rosa, Nelson. (2019). Improving microservice-based applications with runtime placement adaptation. Journal of Internet Services and Applications. 10. 10.1186/s13174-019-0104-0.
2) Grygorash, Oleksandr & Zhou, Yan & Jorgensen, Zach. (2006). Minimum Spanning Tree Based Clustering Algorithms. Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI. 73-81. 10.1109/ICTAI.2006.83.
3) Frey, Brendan & Dueck, Delbert. (2007). Clustering by Passing Messages Between Data Points. Science (New York, N.Y.). 315. 972-6. 10.1126/science.1136800.
4) Stijn van Dongen, *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, May 2000.
5) Hedhli, Ameni & Mezni, Haithem. (2021). A Survey of Service Placement in Cloud Environments. Journal of Grid Computing. 19. 10.1007/s10723-021-09565-z.
6) Bose, Avishek & Munir, Arslan & Shabani, Neda. (2017). A Comparative Quantitative Analysis of Contemporary Big Data Clustering Algorithms for Market Segmentation in Hospitality Industry.

7) Ouyang, Guang & Dey, Dipak & Zhang, Panpan. (2017). Clique-Based Method for Social Network Clustering. Journal of Classification. 37. 10.1007/s00357-019-9310-5.
8) Bansal, Diksha & Grover, Rahul & Saha, Sriparna. (2021). A Multi-view Multiobjective Partitioning Technique for Search Results Clustering. 758-763. 10.1109/SMC52423.2021.9658944.
9) Li, Jian & Pan, Haiwei & Zhang, Minghui & Han, Qilong & Feng, Xiaoning. (2012). Graph-based medical image clustering. 153-158.
10) Li, Zhimei & Zhang, Wanzheng & Liu, Zhiyong. (2020). An Improved Fast Fuzzy Clustering Image Segmentation Algorithm. 403-406. 10.1109/ICVRIS51417.2020.00101.
11) Shi, Peng & Zhao, Zhen & Zhong, Huaqiang & Shen, Hangyu & Ding, Lianhong. (2020). An improved agglomerative hierarchical clustering anomaly detection method for scientific data. Concurrency and Computation: Practice and Experience. 33. 10.1002/cpe.6077.
12) Petrakis, Euripides & Koundourakis, Xenofon. (2021). iXen: Secure Service Oriented Architecture and Context Information Management in the Cloud. Journal of Ubiquitous Systems & Pervasive Networks. 14. 01-10. 10.5383/JUSPN.14.02.001.
13) https://www.microfocus.com/documentation/enterprise-developer/ed40pu5/ETS-help/GUID-F5BDACC7-6F0E-4EBB-9F62-E0046D8CCF1B.html
14) https://en.wikipedia.org/wiki/Docker_(software)
15) https://www.docker.com/
16) https://kubernetes.io/
17) https://istio.io/
18) https://www.envoyproxy.io/
19) https://kiali.io/
20) https://github.com/GoogleCloudPlatform/microservices-demo
21) https://grpc.io/docs/what-is-grpc/introduction/
22) https://en.wikipedia.org/wiki/Bin_packing_problem
23) https://www.researchgate.net/publication/332546043_Bin_packing_algorithms_for_virtual_machine_placement_in_cloud_computing_A_review
24) https://en.wikipedia.org/wiki/Minimum_cut
25) https://en.wikipedia.org/wiki/Minimum_k-cut
26) https://www.researchgate.net/publication/336973943_Optimizing_Service_Placement_for_Microservice_Architecture_in_Clouds
27) https://en.wikipedia.org/wiki/Karger%27s_algorithm
28) Hu, Yang & Laat, Cees & Zhao, Zhiming. (2019). Optimizing Service Placement for Microservice Architecture in Clouds. Applied Sciences. 9. 4663. 10.3390/app9214663.
29) Banerjee, Shreya & Choudhary, Ankit & Pal, Somnath. (2015). Empirical Evaluation of K-Means, Bisecting K- Means, Fuzzy C-Means and Genetic K-Means Clustering Algorithms. 10.1109/WIECON-ECE.2015.7443889.

30) Di, Jian & Gou, Xinyue. (2018). Bisecting K-means Algorithm Based on K-valued Selfdetermining and Clustering Center Optimization. Journal of Computers. 588-595. 10.17706/jcp.13.6.588-595.
31) https://jmeter.apache.org/
32) https://cloud.google.com/compute/all-pricing *(Date Inspected: 1-10-2021)*
33) Marchese, Angelo & Tomarchio, Orazio. (2022). Communication Aware Scheduling of Microservices-based Applications on Kubernetes Clusters. 190-198. 10.5220/0011049300003200.
34) Hu, Yang & Zhou, Huan & Laat, Cees & Zhao, Zhiming. (2018). ECSched: Efficient Container Scheduling on Heterogeneous Clusters: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27 - 31, 2018, Proceedings. 10.1007/978-3-319-96983-1_26.
35) Zhu, Xianzhi & Li, Yongkun & Yao, Lulu & Qi, Zhihao & Xu, Yinlong & Wang, Pengcheng & Wang, Weiguang & Zhu, Xia. (2023). On Optimizing Traffic Scheduling for Multi-replica Containerized Microservices. 358-368. 10.1145/3605573.3605646.
36) Petrakis, Euripides & Skevakis, Vasileios & Eliades, Panayiotis & Aznavouridis, Alkiviadis & Tsakos, Konstantinos. (2023). ModSoft-HP: Fuzzy Microservices Placement in Kubernetes. Electronics. 13. 65. 10.3390/electronics13010065
37) Senjab, Khaldoun & Abbas, Sohail & Ahmed, Naveed & Khan, Atta ur Rehman. (2023). A survey of Kubernetes scheduling algorithms. Journal of Cloud Computing. 12. 10.1186/s13677-023-00471-1.
38) Edirisinghe, Dasith & Rajapakse, Kavinda & Abeysinghe, Pasindu & Rathnayake, Sunimal. (2024). Cost-Optimal Microservices Deployment with Cluster Autoscaling and Spot Pricing. 10.48550/arXiv.2405.12311.
39) Kaur, Kiranpreet & Guillemin, Fabrice & Sailhan, Francoise. (2023). Dynamic Migration of Microservices for End-to-End Latency Control in 5G/6G Networks. Journal of Network and Systems Management. 31. 10.1007/s10922-023-09773-w.
40) Attaoui, Wissal & Sabir, Essaid & Elbiaze, Halima & Guizani, Mohsen. (2023). VNF and CNF Placement in 5G: Recent Advances and Future Trends. IEEE Transactions on Network and Service Management. PP. 1-1. 10.1109/TNSM.2023.3264005.
41) Vasireddy, Indrani & Kandi, Prathima & Gandu, SreeRamya. (2023). Efficient Resource Utilization in Kubernetes: A Review of Load Balancing Solutions. International Journal of Innovative Research in Engineering and Management. 10. 44-48. 10.55524/ijirem.2023.10.6.6.
42) Dakić, Vedran & Kovač, Mario & Slovinac, Jurica. (2024). Evolving High-Performance Computing Data Centers with Kubernetes, Performance Analysis, and Dynamic Workload Placement Based on Machine Learning Scheduling. Electronics. 13. 2651. 10.3390/electronics13132651.
43) Datta, Amitava & Hassan, Ghulam Mubashar & Alelyani, Abdullah. (2024). Optimizing Cloud Performance: A Microservice Scheduling Strategy for

Enhanced Fault-Tolerance, Reduced Network Traffic, and Lower Latency. IEEE Access. 10.1109/ACCESS.2024.3373316.

44) Chen, Qi-Hong & Wen, Chih-Yu. (2024). Optimal Resource Allocation Using Genetic Algorithm in Container-Based Heterogeneous Cloud. IEEE Access. PP. 1-1. 10.1109/ACCESS.2024.3351944.
45) Hossain, Md Razon & Whaiduzzaman, Md & Barros, Alistair & Fidge, Colin. (2024). Dynamic microservice placement in multi-tier Fog networks. Internet of Things. 26. 101224. 10.1016/j.iot.2024.101224.
46) Keshavarz Haddadha, Parviz & Rezvani, Mohammad & Mollamotalebi, Mahdi & Shankar, Achyut. (2024). Machine learning methods for service placement: a systematic review. Artificial Intelligence Review. 57. 10.1007/s10462-023-10684-0.
47) García-Díaz, Jesús & Cornejo-Acosta, J. Alejandro. (2024). A Greedy Heuristic for Graph Burning.
48) Eppstein, David & Har-Peled, Sariel & Sidiropoulos, Anastasios. (2015). Approximate Greedy Clustering and Distance Selection for Graph Metrics.
49) Gudapati, Naga Venkata Chaitanya & Malaguti, Enrico & Monaci, Michele. (2021). In search of dense subgraphs: How good is greedy peeling?. Networks. 77. 10.1002/net.22034.
50) Benati, Stefano & Ponce, Diego & Puerto, Justo & Rodríguez-Chía, Antonio. (2021). A Branch-and-price procedure for clustering data that are graph connected. 10.48550/arXiv.2104.05454.
51) Miasnikof, Pierre & Bagherbeik, Mohammad & Sheikholeslami, A.. (2023). Graph clustering with Boltzmann machines. Discrete Applied Mathematics. Volume 343. 208-223. 10.1016/j.dam.2023.10.012.
52) https://en.wikipedia.org/wiki/Cluster_analysis#Algorithms