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

Microservice Placement Strategies in Kubernetes for Cost Optimization

Diploma Thesis

Alkiviadis Aznavouridis

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

Chania, February 2022

Table of Contents

01

Introduction and Problem Definition

Introduction

- Modern applications utilize various **innovative technologies** (like Kubernetes)

- **Kubernetes** clusters **can efficiently host applications** and secure the consistency of their run-time execution

- Additionally, Cloud computing **provides an alternative for monolithic on-premises** data centers
  - Cloud providers are responsible for hardware, security, storage and network configuration

- **How to schedule application's services efficiently to reduce infrastructure's costs?**
  - **Service Placement (SP)** of services for increasing performance is a well-known problem!

Problem Definition

- Cloud providers **do not apply cost-optimization policies** in running applications
- Kubernetes **can improve run-time costs** by increasing the availability, however it **does not automatically apply cost-optimization strategies** for running a cluster
- **Excess supply of resource allocation** leads to higher costs for the end-user
- **Monetary cost is an important factor** for the end-users!

Goal of the Thesis

Solve the SP problem by **minimizing** the total monetary cost of a Kubernetes cluster

⇩

❖ Convert application into a graph G = (V, E)
  - V = application's microservices
  - E = communication edges (directed)

❖ Apply:
  - Graph-partitioning algorithms to create groups of microservices with high affinity traffic rate
  - Heuristic methods to efficiently place each partition to the infrastructure's VMs

❖ **Minimize** the volume of allocated resources (Number of VMs)

❖ **Maximize** intra-communication (Ingress) and **minimize** inter-communication (Egress) network traffic

02

Theoretical Background

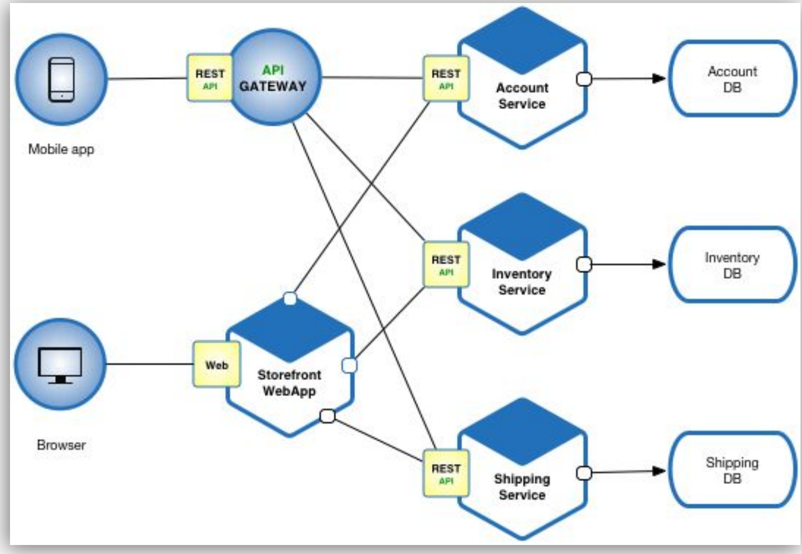
Related Work

- ❖ In Cloud environments:
  - Service deployment strategies in graph-based applications for reducing network latency
  - Scheduling of microservices in Multi-Cloud
  - Service placement and requests scheduling in Edge Clouds for data-intensive applications

- ❖ In Cloud environments with Kubernetes:
  - Scheduling processes of services to serve the network requirements of a University Campus
  - Adaptation mechanism for service placement based on the service affinities to rearrange the services into the existing cluster
  - Service placement using graph-partitioning algorithms and heuristic methods for packing

Microservices

- Architectural style which structures an application as a collection of services
- Microservices can be independently deployed, configured and expanded
- Easier isolation on problematic services and application errors
- Each microservice can be accessible from Application Programming Interfaces (APIs)



Kubernetes

- ❖ Platform for **managing containerized workloads and services**
- ❖ **Facilitates** both configuration and automation of services

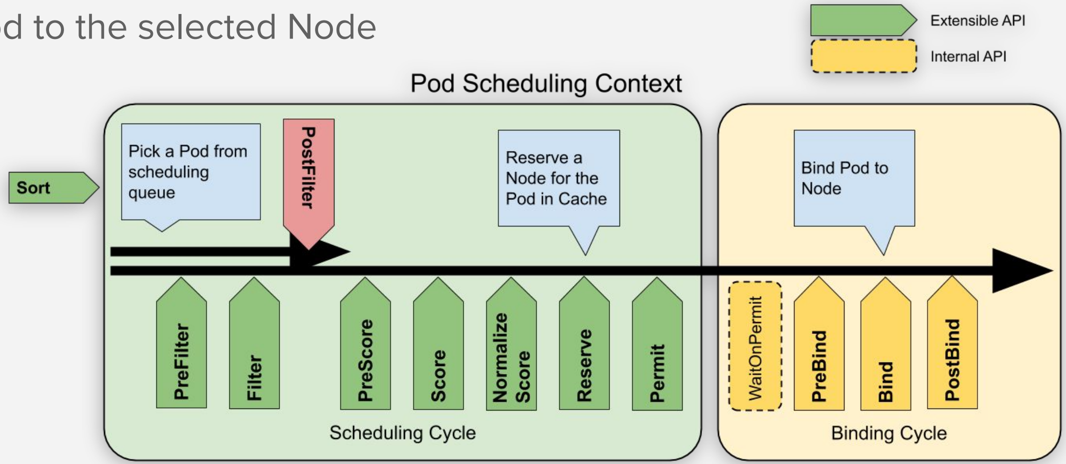
- ❖ **Provides:**
  - Efficient handling of application’s containers
  - Affinities/Anti-Affinities of services for deploying applications
- ❖ **Handles** the scaling of the application and containers and the fail over situations

- ❖ **Cluster** consists of set of worker machines, called **Nodes**, that run containerized application
- ❖ Nodes host **Pods**, which are the application’s workloads

- ❖ **Control Plane** of Kubernetes is responsible for managing the Nodes and Pods

Kubernetes Scheduling Process

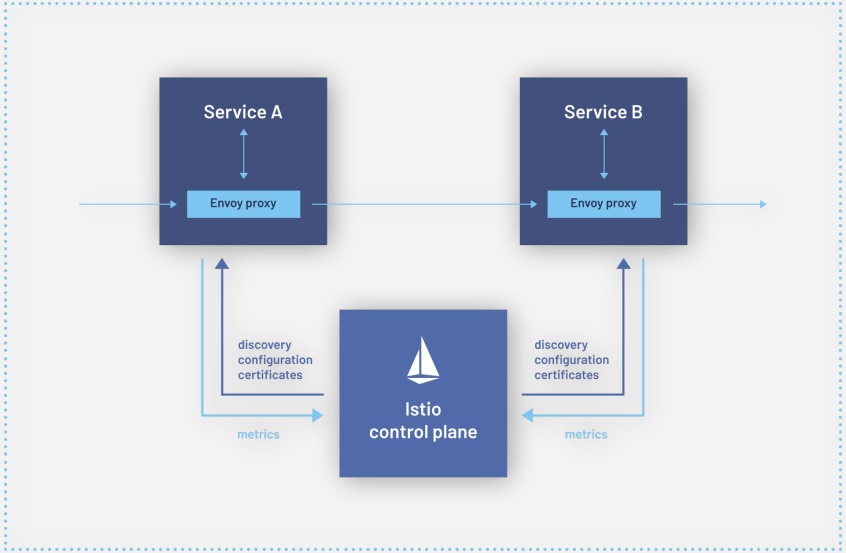
- A scheduler observes for newly initialized Pods that have not assigned to any of the cluster's Nodes
- **Scheduling** Cycle ⇒ Locate a “feasible” Node to host the Pod
  - **Filtering:** Find a set of Nodes “feasible” to host the Pod
  - **Scoring:** Ranks the Nodes from the filtering step
  - Can be extended for custom Scheduling policies
- **Binding** Cycle ⇒ Schedule Pod to the selected Node



Service Mesh and Istio

- **Service Mesh** is a dedicated infrastructure layer which allows adding capabilities into application like **observability, traffic management and security**
- Routing application's requests through **sidecar proxies**
- Easier troubleshooting process and monitoring

- **Istio** is an open-source Service Mesh
- Injects **Envoy sidecar proxies** into each service (Pod)
- Consists of:
  - **Data Plane:** Responsible for communication of application's microservices
  - **Control Plane:** Monitors network traffic and dynamically programs the Envoys



Metric Tools and Agents

- Services connected with Istio Service Mesh for monitoring an application and collecting data
- **Prometheus:**
  - **Monitoring system and alerting service** collecting and storing data as time series data
  - **Records real-time metrics** in a time series database
  - Extracts data by applying **PromQL queries** to an application
  - **Prometheus Node Exporters** can be installed into application’s Nodes and collect their data
- **Kiali:**
  - **Management console** for Istio Service Mesh
  - **Visualizes the application’s graph** by collecting data from Prometheus
  - Provides information about services, health status, traffic rates and protocols of communication
- **Grafana:**
  - Analytics and interactive visualization service
  - **Visualizes Prometheus Data** in graphs

Microservice Placement Strategies

- **Single-step execution** strategies:
  - Heuristic First Fit (HFF)

- **Two-step execution** strategies:
  - Binary Partition - Heuristic Packing (BP - HP)
  - K-Partition - Heuristic Packing (KP - HP)
  - Bisecting K-Means - Heuristic Packing (BKM - HP)

» Heuristic Packing is essential to:
  - **Verify** the successful placement of each partition
  - **Decrease** the volume of resources needed to host each application

Related Algorithms (1/4) - Heuristic First Fit (HFF)

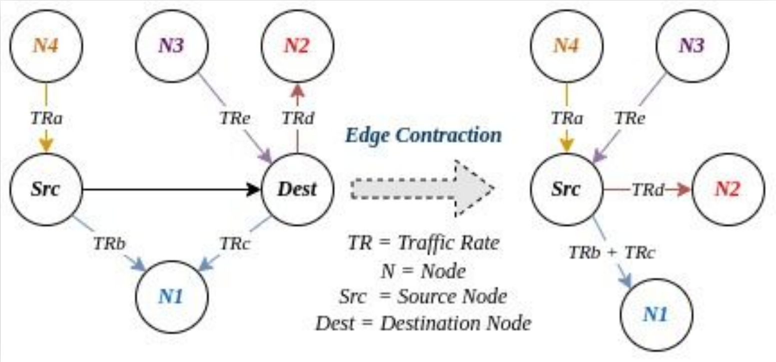
- **Heuristic approach to optimize** service placement in a current infrastructure
  - **Relocating microservices** with high affinity traffic rate into the same host
- **Input:** Initial microservice placement, Microservice affinities, Pod resource demands, Node available resources
- **Output:** Optimized microservice placement solution
- **Basic idea:**
  - **Sort** the microservice affinities **in descending order**
  - For each affinity examine whether **source node can relocate to destination's host node** or vice versa
  - If applicable or microservices are already at the same host **mark them as moved**
    - ➔ Marked microservices **can not be moved** for the next iterations
  - **Recalculate the Node available resources** according to the moved Pod resource demands

Related Algorithms (2/4) - Binary Partition (BP) and K-Partition (KP)

- **Input:** Application's graph G = (V, E), Resource demands of each microservice
- **Output:** Group of microservices (partitions) that can be placed into infrastructure's host machines
- **Process:**
  - ➔ Initial partition **P = {S}**, S = microservice-based application
  - ➔ **Repeat until:** Each partition contains **at least one** service and does not exceed **threshold α**
    - **Threshold α:** Upper bound of the partition's resource demands (in percentage)
  - ➔ At each **iteration:**
    - Create the partition's graph $G_{part} = (V_{part}, E_{part})$
    - Apply the **Contraction algorithm** in total $\mathbf{n = |V_{part}|}$ times
    - **Create K sub-partitions** according to the best result of Contraction algorithm and **insert** them into **P** (K=2 for BP)
    - For KP algorithm created sub-partitions are increased by 1 at each iteration (initial value = 2)

Related Algorithms (2/4) - Contraction Algorithm (Karger's Algorithm)

- **Randomized algorithm to compute the minimum K-cut** of a connected graph
- **K=2 for BP** and for **KP the value of K is respective to the iteration's produced sub-partitions**
- **Basic idea**:
  - Randomly choose an edge from the graph
  - Merge the Nodes connected to this edge (edge contraction)
  - Recalculate all the traffic rates connected to the selected Nodes according to the edge contraction



Related Algorithms (3/4) - Bisecting K-Means (BKM)

- **Graph-partitioning algorithm** based on a variant of K-Means algorithm
- Create **K clusters** (groups of microservices) with **high intra-affinity** and **low inter-affinity** traffic rates
- **Input:** Application’s graph G = (V, E), K value
- **Output:** K clusters (partitions) that can be placed into infrastructure’s host machines
- **Basic idea:**
  - **Iteratively split** a cluster into two sub-clusters until K clusters are created
  - **Initial Cluster:** Application’s graph G
  - At each iteration:
    - ➔ Select **a cluster** to be split according to **the minimum sum of traffic rates** among the cluster’s microservices
    - ➔ Select **two microservices** as centroids with **no communication edge or the with the lowest traffic rate** among the others
    - ➔ **Assign the rest microservices** between these two centroids according to their affinity

Related Algorithms (4/4) - Heuristic Packing (HP)

- ❖ Adaptive Placement and post-processing algorithm
- ❖ Attempts to pack of the given application's partitions into the infrastructure's host machines
- ❖ **Input:** Application's partitions, Node available and allocated resources, Resource demands of each Pod
- ❖ **Output:** Placement solution for the utilized infrastructure
- ❖ **Basic Idea:** Each partition **must be packed** in at least one Node

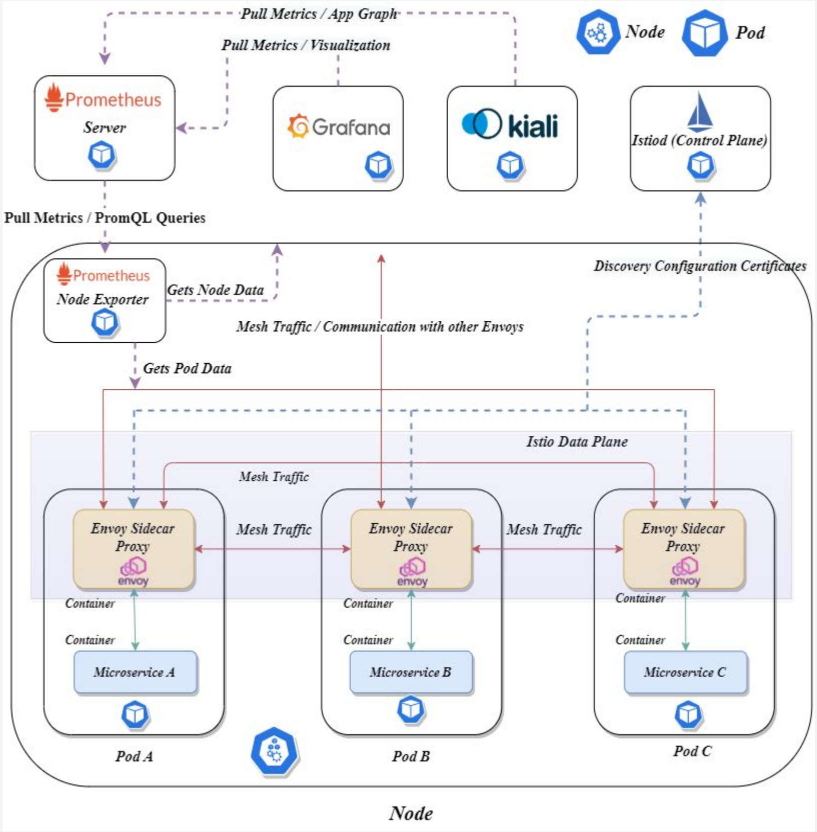
- ❖ Uses two greedy heuristic metrics to evaluate each partition:
  - **Traffic Awareness (tf):** Sum of traffic rates between partition's microservices and microservices already located in the processed Node
  - **Most-Loaded Situation (ml):** Scalar value of the load situation between the resource demands of partition's microservices and the available resource in the processed Node

03

System Design and Benchmarks

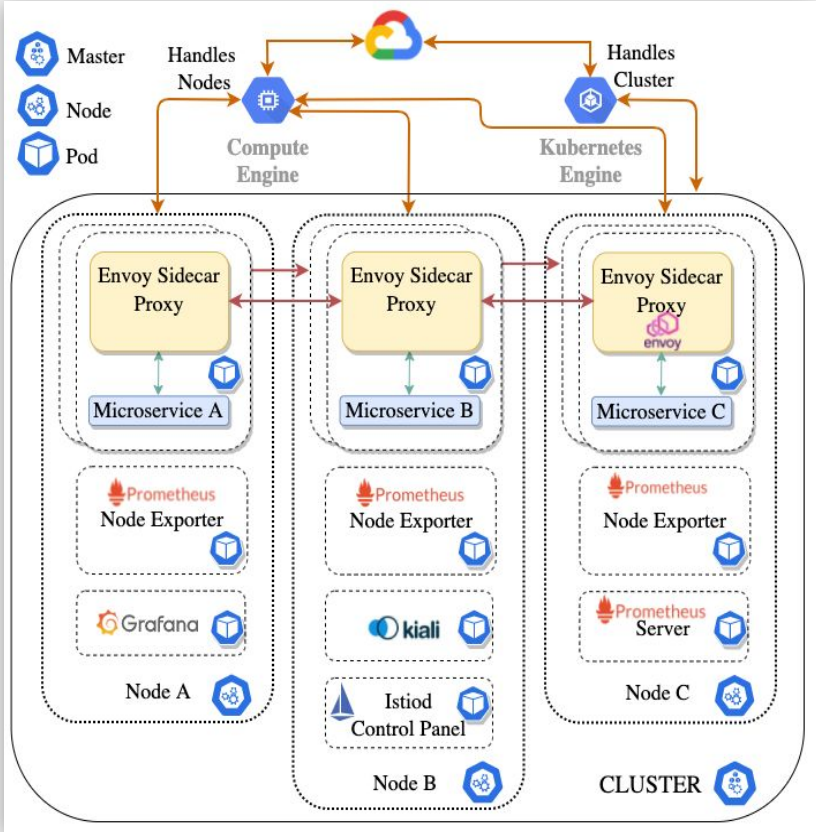
System Architecture (1/2)

- ❖ **Each Pod contains one Envoy sidecar proxy** handling inbound and outbound traffic
- ❖ Each newly created Pod (new Envoy proxy) **sends a discovery configuration certificate** to Istio's Control Plane
- ❖ Envoys communicate with all the cluster's Pods through Istio's Data Plane
- ❖ Prometheus Node exporter **collects Node and Pod data**
- ❖ Prometheus **pulls metrics** from Node exporters, **stores** them and **sends** them to Kiali and Grafana services



System Architecture (2/2)

- **Istio's Services are placed** into cluster's Nodes according to Kubernetes Scheduling decision
- Each Node **contains an instance** of a Prometheus Node Exporter and **is associated with a VM**
- **Cluster monitors and handles** all application's Nodes and Pods
- **Cloud providers** are responsible for managing the clusters and the utilized VMs



Performance Measures (1/2) - Requests per Second (RPS)

➢ Performance measures are utilized by the placement strategies **to calculate the microservices affinity traffic rates**

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

❖ **The amount of search traffic** a system receives in one second

❖ Calculated by the **application's graph,** which is collected by **Kiali service**

❖ **Mean value** is located for a specific time frame (i.e. specific timestamp)

Symbol	Description
i	Source service
j	Destination service
t	time (second) of the time frame
T_{sec}	total amount of time of the frame (seconds)
R_t	total number of requests per second

Performance Measures (2/2) - Weighted Bidirectional Affinity (WBA)

- Metric which **exploits** the size of exchanged messages in bytes and the total number of messages

- **A more accurate** performance measure than the RPS

- **Weight factor** is selected according to the importance of the function's variables

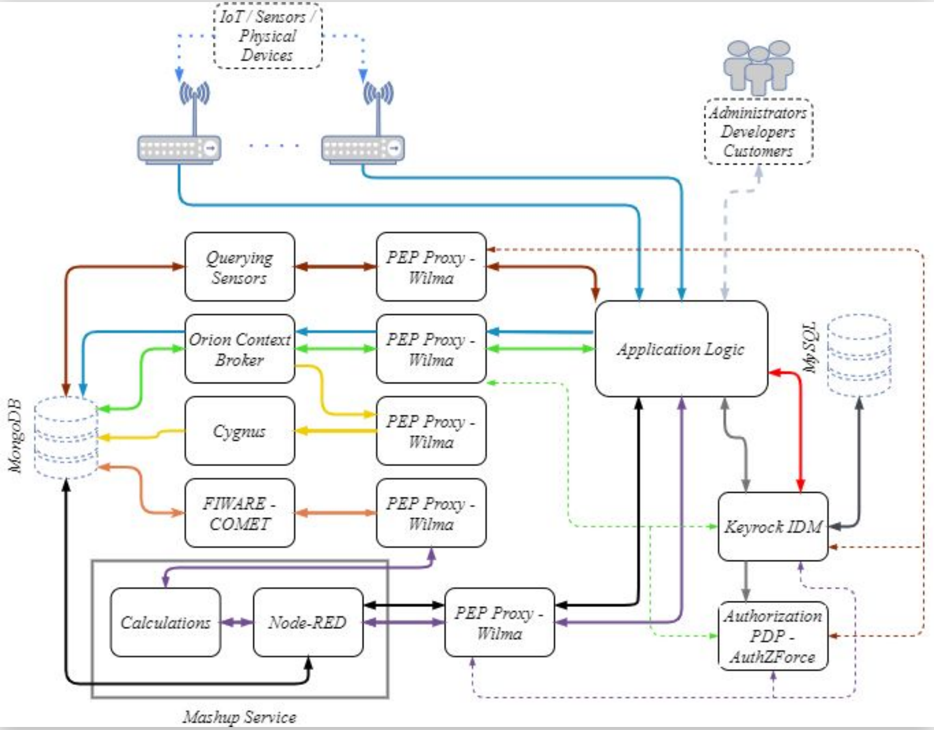
- There is no strong preference between the variables
  - **w = 0.5** for our implementation

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

Symbol	Description
$A_{a,b}$	affinity metric between edge connecting a and b
m	total number of messages exchanged
$m_{a,b}$	messages exchanged between a and b
d	total data exchanged in bytes
$d_{a,b}$	data exchanged in bytes between a and b
w	weight of significance of each affinity variable

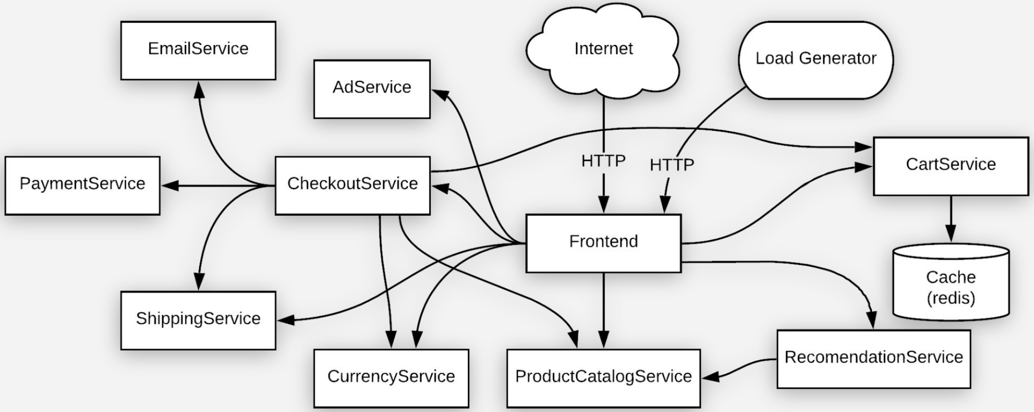
Benchmark Applications (1/2) - iXen

- Software architecture for an IoT scenario
- Based on Service Oriented Architecture (SOA) principles
- Converted from SOA architecture to microservice-based in Kubernetes
- Communication via TCP and HTTP protocol
- 15 Microservices - 30 communication edges



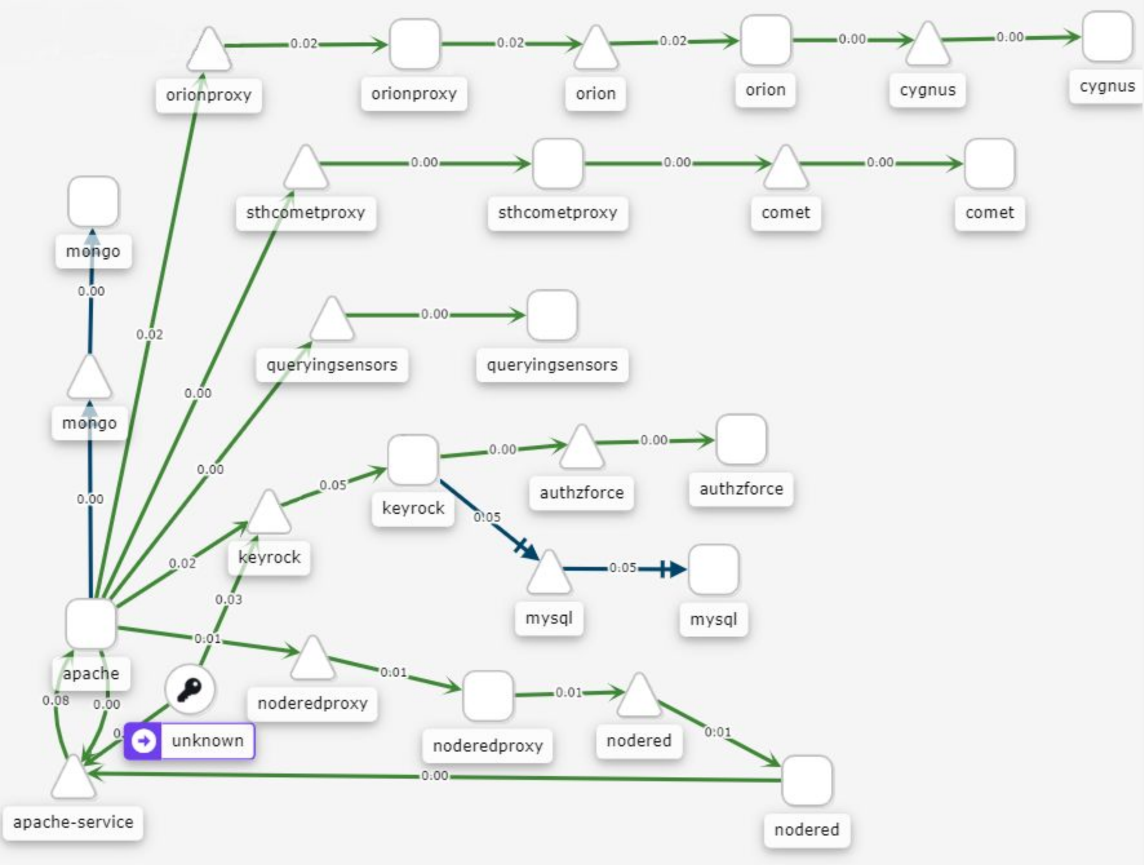
Benchmark Applications (2/2) - Google's OnlineBoutique eShop

- Google's Benchmark application - Mock eShop
- Communication via gRPC and HTTP protocol
- gRPC provides better support for load balancing, tracing and health monitoring
- Load Generator microservice applies stressing into the application with random generated requests
- 12 Microservices - 16 communication edges



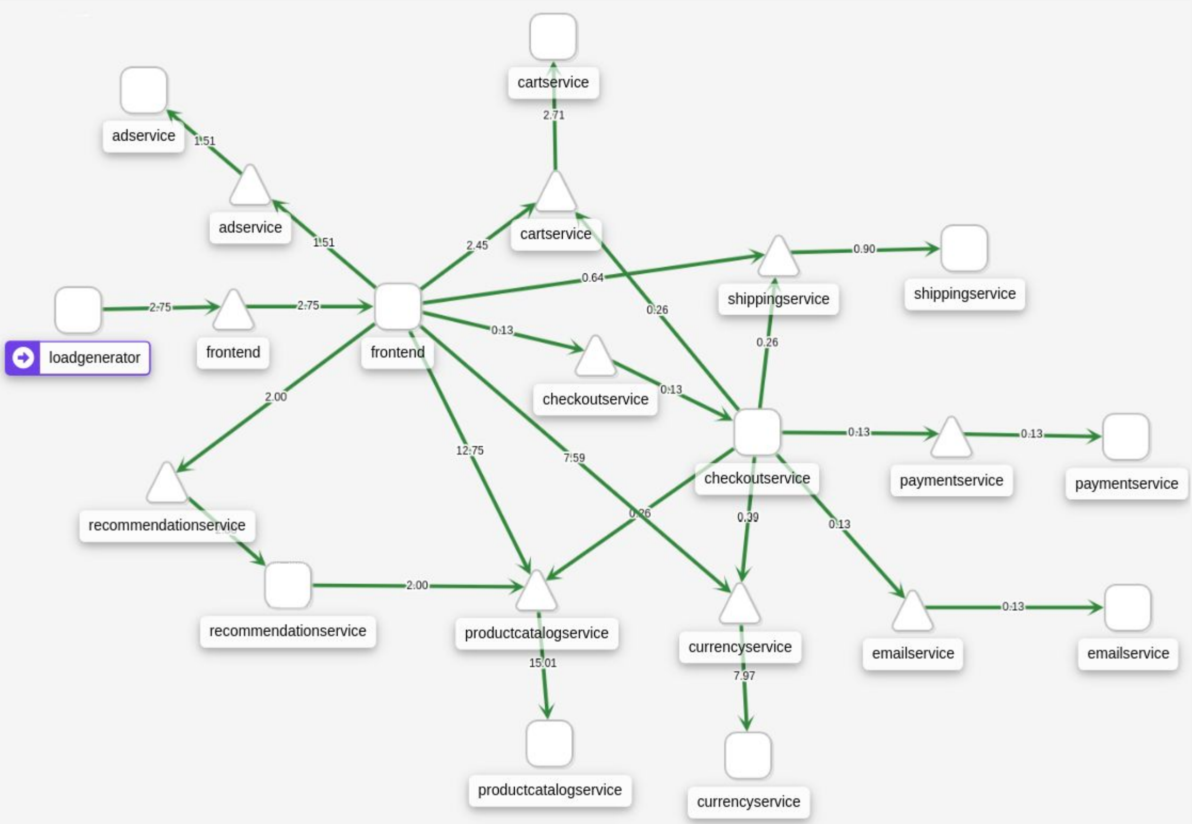
Kiali Graph (1/2) - iXen

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

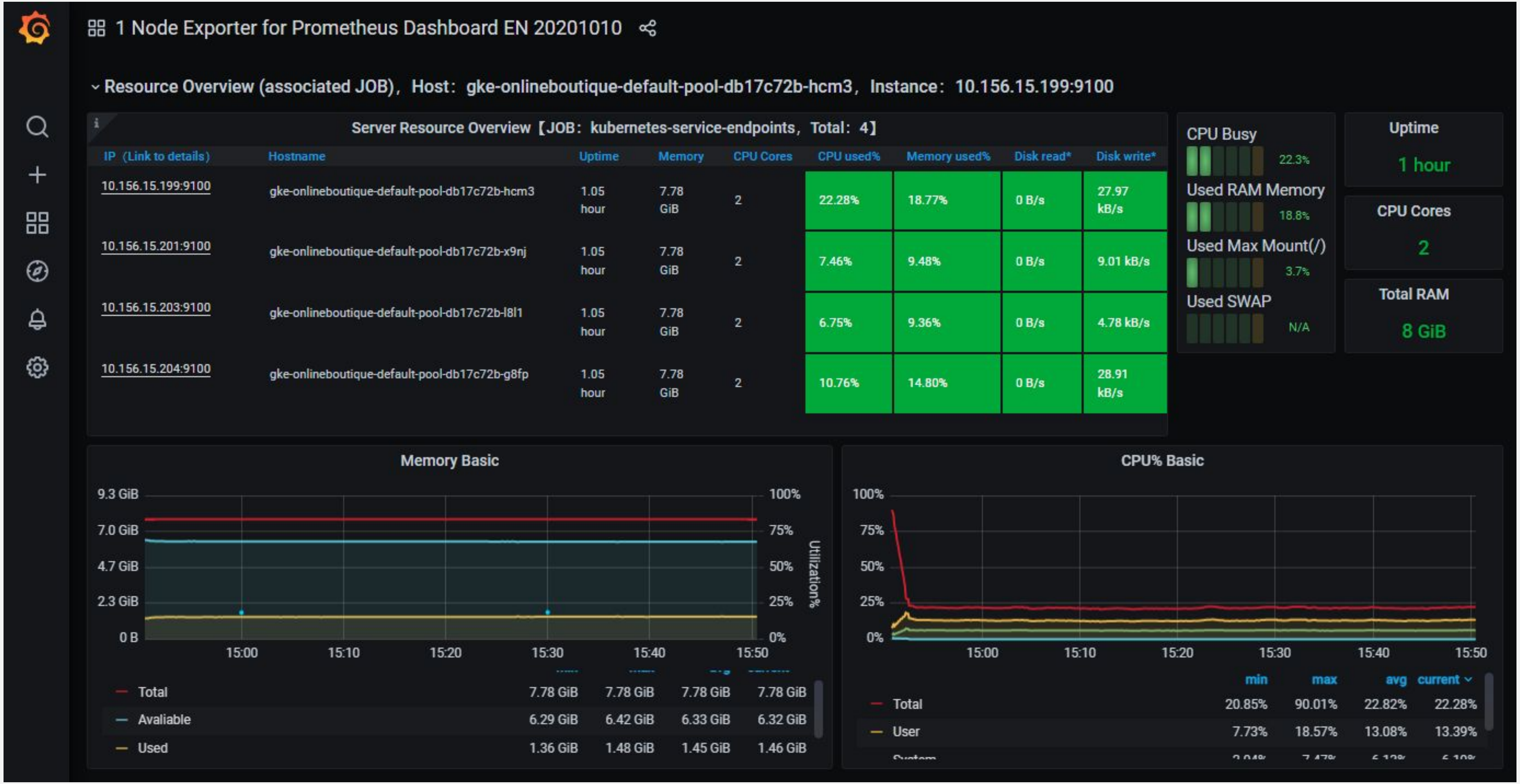


Kiali Graph (2/2) - OnlineBoutique

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



Grafana Visualization (OnlineBoutique)



04

Cost Function

Cost Function (1/2) - Cost Estimation

- Total monetary cost of a cluster is affected by **the volume of resource allocation** and the **network traffic**

- **Factors that can vary** this cost function:
  - Machine Types
  - Hours of operation
  - Storage volume
  - Respective cost of resources

- **Additional charges**:
  - GPU usage
  - Optimization tools for each infrastructure (i.e. load balancing of workloads and requests)

$$TotalSum = Cost_{CPU} + Cost_{RAM} + Cost_{Storage} + Cost_{Traffic}$$

$$Cost_{CPU} = \sum_{i=1}^{N} cpu_{cost}(M_i) \cdot h_i \qquad Cost_{RAM} = \sum_{i=1}^{N} ram_{cost}(M_i) \cdot h_i$$

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

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

Symbol	Description
N	Node
M	Machine type
h	Time of usage (hours)
t_{en}	Egress Traffic (bytes)
t_{in}	Ingress Traffic (bytes)
s	Storage size (GB)

Cost Function (2/3) - Kubernetes charges in GCP

- **Each machine type** is associated with a specific volume of CPU and RAM allocation
- Resources for each **VM are charged per hour of usage** (same amount of fee)
- **Ingress Traffic** is not charged
- **Egress Traffic** is charged according to the Region of each VM (total requests per GB)
    - **Responded messages** from different VMs are regarded as Ingress communication
- Cluster’s **storage is not charged**
    - There is **no additional volumes** attached

Cost Function (3/3) - Cluster cost in GCP

- Each Node (VM) allocates **2vCPU** and **8GB RAM**
- Implementation in **a homogeneous environment**
  - Same **volume of resources** per each utilized VM

$$Cost_{CPU} = n \cdot 2vCPU \cdot cpu_{monthly\ cost} \cdot h$$

$$Cost_{RAM} = n \cdot 8GB \cdot ram_{monthly\ cost} \cdot h$$

$$Cost_{Traffic} = egress_{monthly\ cost} \cdot \sum_{i=1}^{N} \sum_{j=1,\, j \neq i}^{N} t_e(i \rightarrow j)$$

$$Cost_{Storage} \simeq 0$$

Symbol	Description
n	Number of Nodes
h	Time of usage (hours)
t_e	Egress Traffic (bytes)

Description	Monthly Cost (USD)
Predefined vCPU	20.2342 /vCPU
Predefined RAM	2.711/GB
Egress Traffic	0.01/GB

05

Experimental Results

Infrastructure

- **Google Cloud Platform (GCP)** as Cloud provider
- Kubernetes cluster in Google Kubernetes Engine (GKE)
  - ⮡ GKE **manages** the cluster
- Each cluster creates a **node pool** responsible for handling the cluster’s Nodes (VMs)
  - ⮡ Compute Engine service **handles** VMs in GCP
- **4** Nodes with **2 vCPU** and **8GB RAM** each
  - The **node pool** is responsible for resizing the cluster Nodes
- Nodes in europe-west-3b Zone **(Homogeneous environment)**
- **Vertical and Horizontal auto scaling** and **load balancing** tools are disabled

Application Stressing (1/3) - Apache JMeter

- Designed to load test functional behavior and measure performance
- Can simulate heavy loads on systems to test and analyze the overall performance under different load types and distributions
- Applies stressing by applying requests via HTTP protocol
- Extracts data from responses in various types of response formats
- Full multi-threading framework that allows concurrent and simultaneous sampling
- Highly extensible

Application Stressing (2/3) - iXen Stressing

- **100 threads** applying randomly requests
- Various **application's endpoints** for applying requests
- **Stressing is required** to produce network traffic and in order for the application's graph to be created
- **Equally distributed** requests
- **15 minutes** of stressing
- **1-2 requests** per second
    - **Small volume** of stress testing

Requests Description	Type	Requests Distribution
Login into the App	POST	12.5%
Access device measures	POST	12.5%
Access device subscriptions	POST	12.5%
Deploy a new Mashup App	POST	12.5%
Search an App	GET	12.5%
Search for subscriptions	GET	12.5%
Make a new subscription	POST	12.5%
Access Mashup App	GET	12.5%

Application Stressing (3/3) - OnlineBoutique Stressing

- **Frontend microservice** is the single application endpoint
- **Two stressing techniques**:
  - Load Generator microservice
  - Apache JMeter stressing

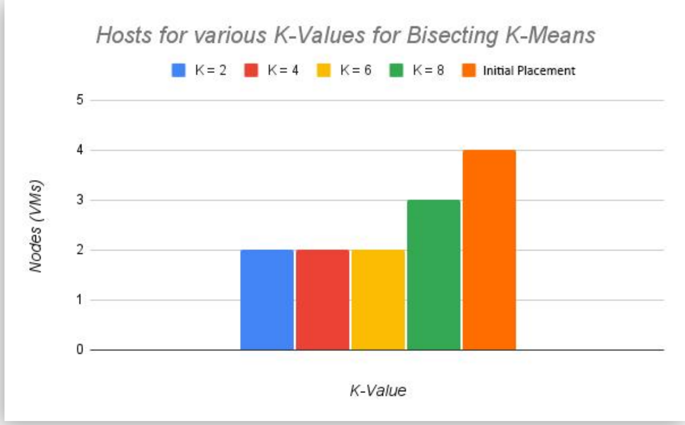
Requests Description	Type	Requests Distribution
Access Index Page	GET	25.0%
Submit an order	POST	41.66%
Show cart products	GET	19..44%
Change currency	POST	13.88%

- **Load Generator** microservice:
  - Applies **randomly generated requests** into the app
  - **2-3** requests per second

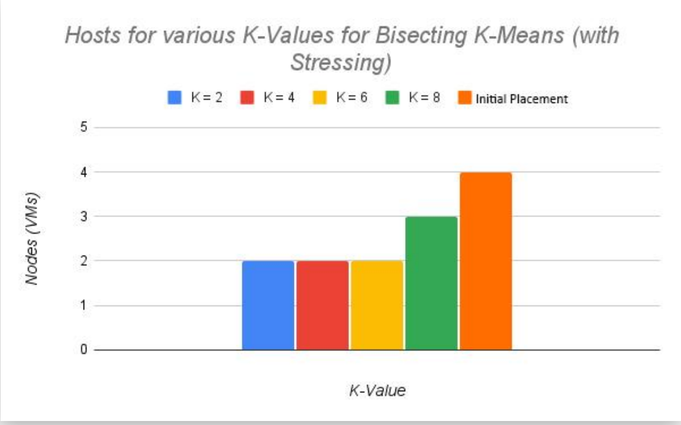
Note: Submit order request contains 3 types of requests for multi-products:
- Show a product (GET)
- Add product to cart (POST)
- Submit order (POST)

- **Apache JMeter** stressing:
  - **10 minutes** of stressing
  - **Not equally** distributed requests
  - Nearly **30 requests per second**

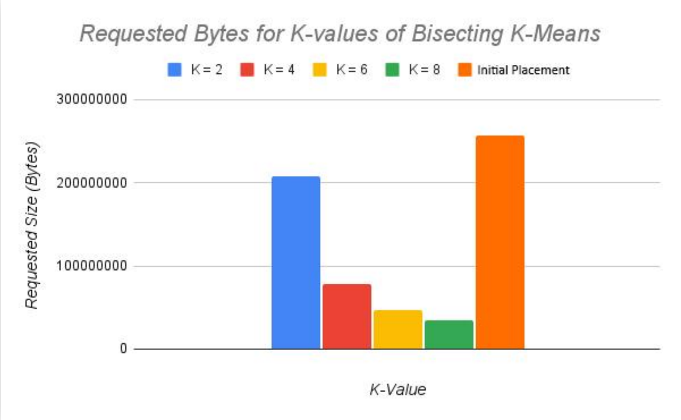
K-value selection for BKM algorithm (OnlineBoutique)



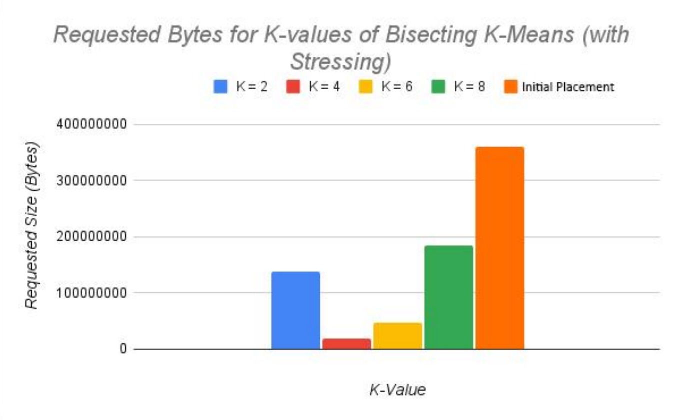
Number of Nodes



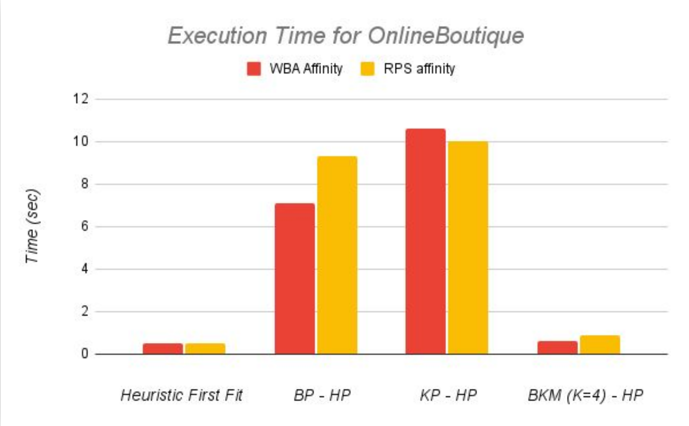
Initial Placement = Solution of Kubernetes Scheduler



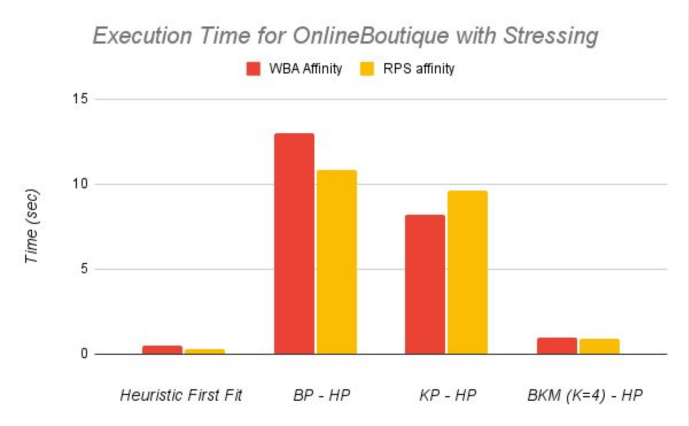
Egress Traffic

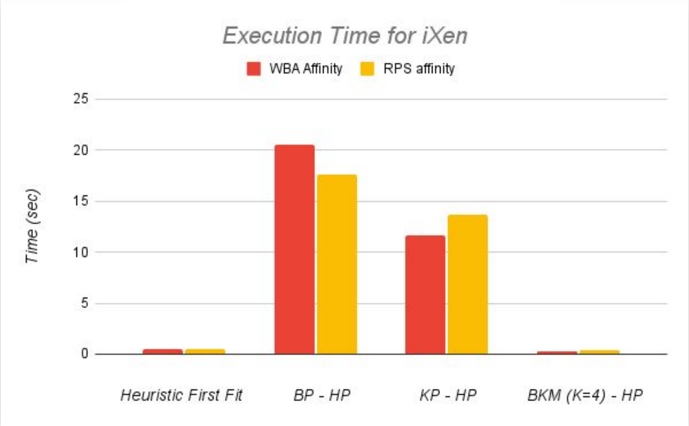


Execution time of each placement strategy

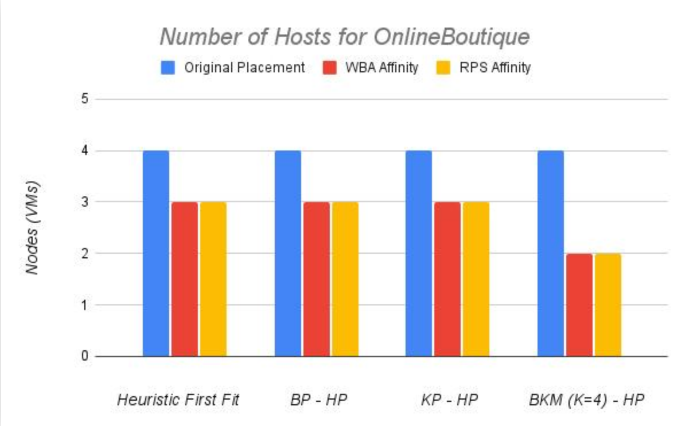


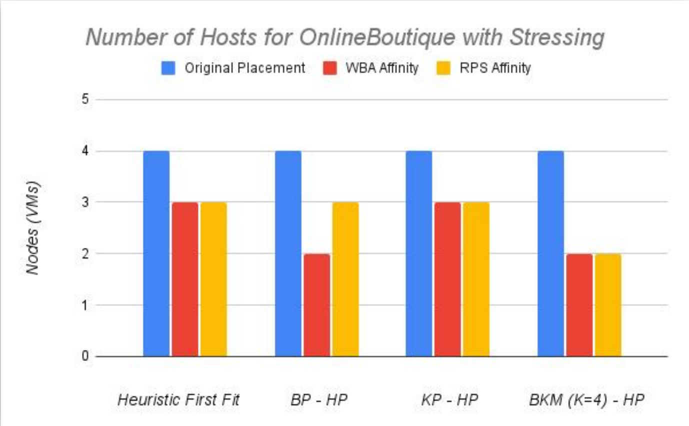
WBA / RPS = Performance Measures

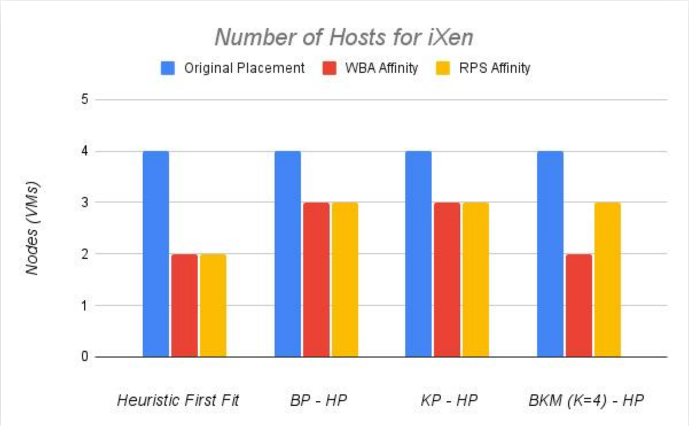




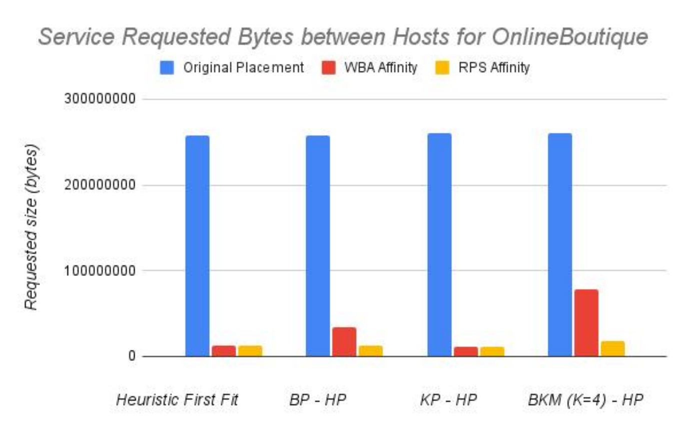
Number of utilized Nodes (VMs)

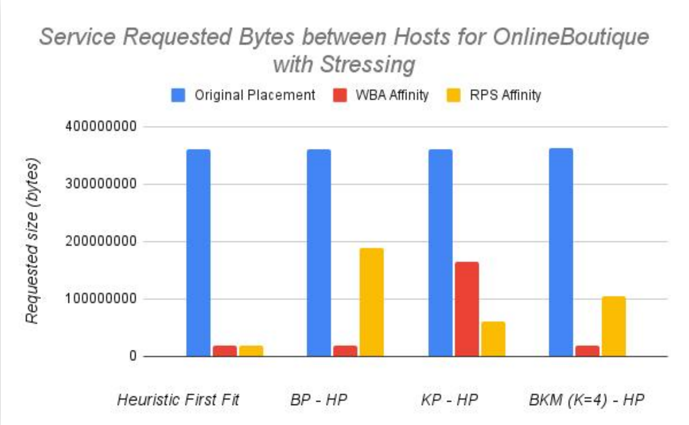


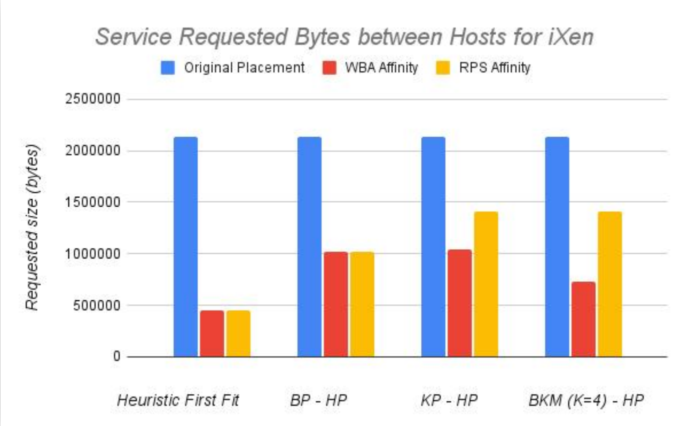




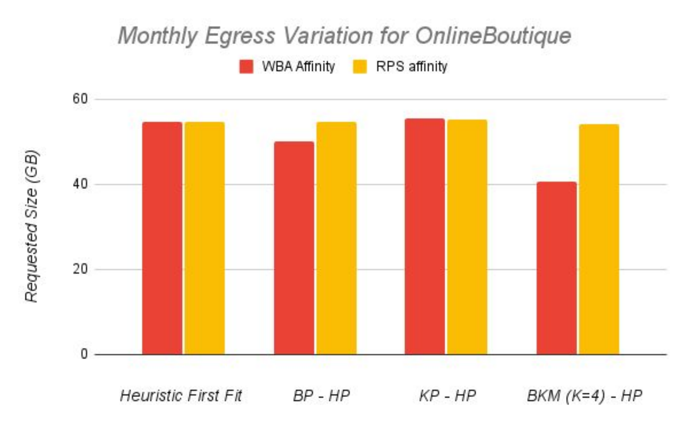
Egress Traffic (1/2) - Microservice requested size between Nodes in bytes

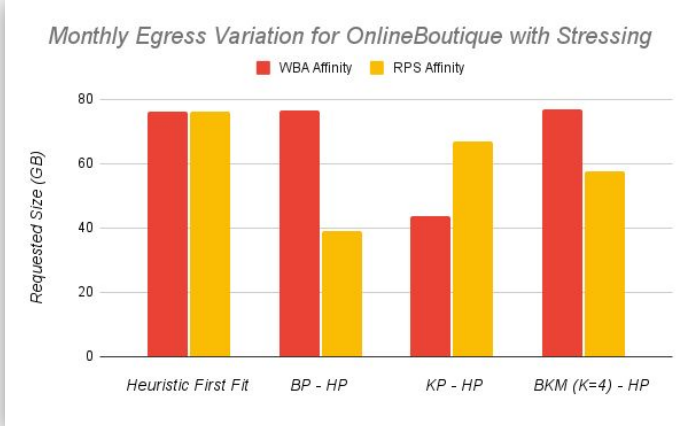


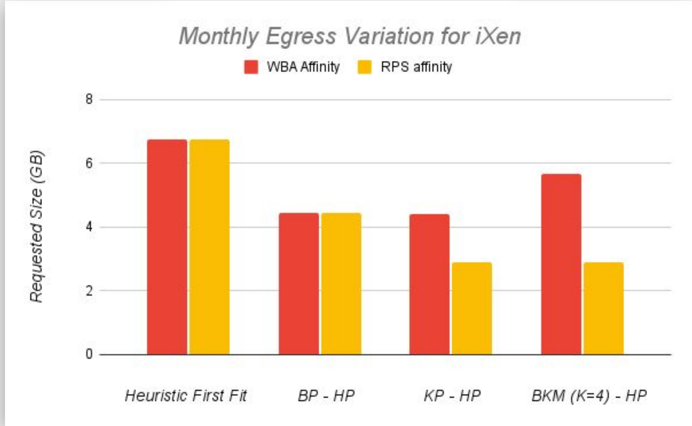




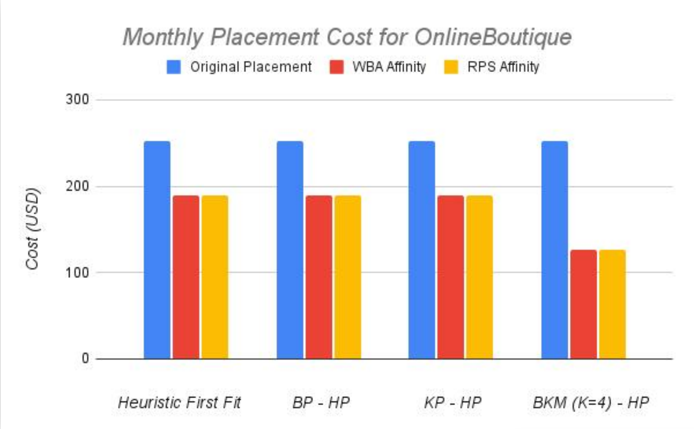
Egress Traffic (2/2) - Monthly Egress variation in GB

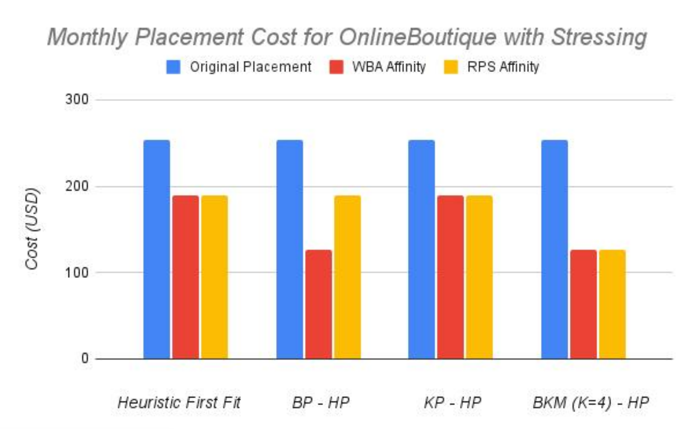


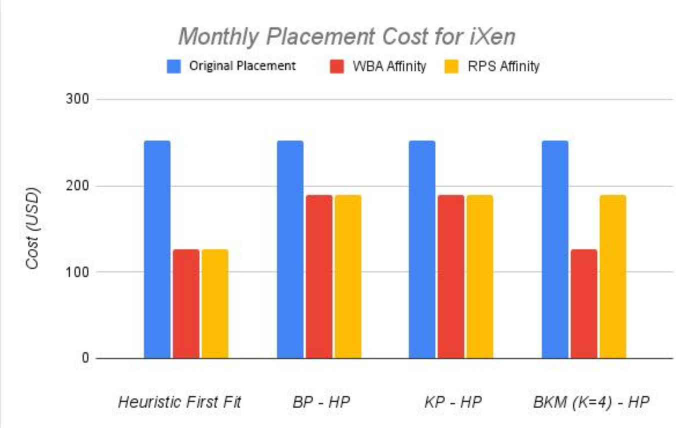




Total monetary monthly cost of cluster







Original Placement = Solution of Kubernetes Scheduler

WBA / RPS = Performance Measures

Discussion

- Comparison of **applications:**
  - **Small scaled** applications, easily comparable, can be hosted **efficiently in 2 VMs**
  - Different **Pod resource** requirements:
    - ➔ **iXen:** 2% of each Node for CPU and 4% for RAM
    - ➔ **OnlineBoutique:** 5-15% of each Node for CPU and 1-4% for RAM
  - **Load stressing and size of messages** in OnlineBoutique are greater than iXen
- Comparison of **performance measures:**
  - **RPS is collected** from the Kiali graph and **WBA is calculated** from the Prometheus metrics
  - **WBA is a more accurate** performance metric for measuring the affinity traffic rates
- Comparison of **microservice placement strategies:**
  - **BKM** is overall **the best** microservice placement strategy in terms of cost
  - **HFF reduces** Egress traffic to the minimum
  - **BP and KP** may produce a non-optimal application's partitioning result

06

Conclusion and Future work

Conclusion

- Microservice-based applications to graph representation
- Combination of graph-partitioning algorithms with heuristic methods
- Implementation of microservice placement strategies in Kubernetes on GCP
- Homogeneous environment with 4 Nodes (VMs) as initial number of Hosts

Reduction in the volume of utilized Resources ⇨ **25% - 50%**

Reduction in the Egress (between VMs) network traffic ⇨ **50% - 90%**

Minimization of the monthly monetary cost of the Kubernetes cluster ⇨ **25% - 50%**

Future work

❖ Trade-off random methods (**reducing** time complexity) with more accurate graph-partitioning algorithms (**increasing** partitioning efficiency)

❖ Apply microservice placement strategies in:

- **Large scaled applications**
- **Heterogeneous environments**
- **Multi-Cloud**

Greater impact of Egress traffic

❖ Implementation of:

- A placement strategy that **adapts to workload changes dynamically**
- Strategies in Kubernetes **reducing network latency**

Thank you!

Questions?