School of Electrical and Computer Engineering
Intelligent Systems Laboratory

Minimum cost microservice placement strategies in Kubernetes environment for hybrid Cloud-Fog infrastructures

---

Diploma Thesis of
Zacharopoulos Apostolos

Committee:
Prof. (Supervisor) Euripides G.M. Petrakis
Asc. Prof. Vasilios Samoladas
Prof. Michail G. Lagoudakis

Abstract

In this thesis, we explore innovative techniques for minimizing cost in microservice placement within Cloud-Fog environments, aiming to enhance application response latency and optimize end-user experience in distributed computing contexts. The advent of the Internet of Things (IoT) has led to an expansion of internet-connected devices, generating vast data volumes. While cloud computing remains a dominant deployment method, Fog computing-leveraging computational nodes proximal to the user-offers significant benefits in handling this data deluge.

Our study focuses on a static deployment approach within a specifically structured topology consisting of five nodes, each hosting a Kubernetes cluster environment. This setup includes one Cloud node, two Fog nodes, and two Edge nodes, with a descending resource availability gradient as we move away from the Cloud. We analyze two microservice-based applications, conceptualized as Directed Acyclic Graphs (DAGs), to demonstrate our approach.

A central contribution of this thesis is the development and presentation of a heuristic algorithm for microservice placement. This algorithm, thoughtfully designed for effectiveness within our multi-layered infrastructure, leverages metrics garnered from comprehensive application benchmarking. These metrics are crucial in forming affinities among microservices, serving as the primary factor for the algorithm in determining the deployment priority of each microservice. By applying this algorithm, our objective is to establish a framework that not only optimizes resource allocation but also ensures reduced response latencies, significantly enhancing the overall user experience in Cloud-Fog computing environments.

Περίληψη

Σε αυτή την εργασία, διερευνούμε καινοτόμες τεχνικές για την ελαχιστοποίηση του κόστους τοποθέτησης μικροϋπηρεσιών σε περιβάλλοντα Cloud-Fog, με στόχο τη βελτίωση του χρόνου απόκρισης εφαρμογών και τη βελτιστοποίηση της εμπειρίας του τελικού χρήστη σε κατανεμημένα περιβάλλοντα υπολογιστών. Η έλευση του Διαδικτύου των Πραγμάτων (IoT) οδήγησε σε επέκταση των συσκευών που είναι συνδεδεμένες στο Διαδίκτυο, δημιουργώντας τεράστιους όγκους δεδομένων. Παρότι το υπολογιστικό σύννεφο παραμένει η κυρίαρχη μέθοδος ανάπτυξης και φιλοξενίας εφαρμογών στο διαδίκτυο, ο συνδυασμός υπολογιστικού Σύννεφου - Ομίχλης - αξιοποιώντας υπολογιστικούς κόμβους κοντά στον χρήστη - προσφέρει σημαντικά οφέλη στον χειρισμό αυτού του κατακλυσμού δεδομένων.

Η μελέτη αυτή επικεντρώνεται σε στατικές τοποθετήσεις εντός μιας ειδικά δομημένης τοπολογίας που περιλαμβάνει πέντε κόμβους, ο καθένας από τους οποίους φιλοξενεί ένα περιβάλλον Kubernetes. Αυτή η τοπολογία περιλαμβάνει έναν κόμβο Cloud, δύο κόμβους ομίχλης και δύο κόμβους Edge, με φθίνουσα διαβάθμιση διαθεσιμότητας πόρων καθώς απομακρυνόμαστε από το Cloud. Αναλύουμε δύο εφαρμογές, που βασίζονται σε μικροϋπηρεσίες, ως Κατευθυνόμενα Άκυκλους Γράφους (DAGs), για να αναδείξουμε την προσέγγισή μας.

Κύρια συμβολή της παρούσας διπλωματικής εργασίας είναι η ανάπτυξη και παρουσίαση ενός ευρετικού αλγορίθμου για την τοποθέτηση μικροϋπηρεσιών. Αυτός ο αλγόριθμος είναι σχεδιασμένος για τοποθετησεις μικροϋπηρεσιών εντός της πολυεπίπεδης υποδομής μας, χρησιμοποιώντας μετρήσεις που προέρχονται έπειτα από εκτενές benchmarking των εφαρμογών και δημιουργώντας τις συγγένειες μεταξύ των μικροϋπηρεσιών απο τις μετρήσεις αυτές. Αυτές οι μετρήσεις είναι ζωτικής σημασίας για το σχηματισμό συγγένειας μεταξύ των μικροϋπηρεσιών, χρησιμεύοντας ως πρωταρχικός παράγοντας για τον αλγόριθμο στον καθορισμό της προτεραιότητας τοποθέτησης κάθε μικρουπηρεσίας. Εφαρμόζοντας αυτόν τον αλγόριθμο, στοχεύουμε να δημιουργήσουμε ένα πλαίσιο που βελτιστοποιεί την κατανομή πόρων, αλλά και εξασφαλίζει χαμηλότερες καθυστερήσεις απόκρισης, βελτιώνοντας έτσι σημαντικά τη συνολική εμπειρία χρήστη σε περιβάλλοντα υπολογιστικού Σύννεφου - Ομίχλης.

Acknowledgments

I would like to express my sincere thanks to my professor, Euripides G.M. Petrakis, for his guidance and support on this Diploma Thesis. His expertise has been invaluable.
A special thank you to my family for their endless love and patience, and to my friends for their support and good cheer.

1. Introduction

1.1 Problem definition

The concept of Cloud-Fog computing tackles the crucial challenge of improving response times and minimizing latency in distributed applications, a significant concern in today's advanced computing landscape. This paradigm positions Cloud computing as a centralized, remote data processing center, complemented by Fog computing, which extends processing power closer to data sources at the network's edge. The primary goal of Cloud-Fog computing is to forge a distributed infrastructure adept at smartly assigning application components, thereby orchestrating computational tasks between the Cloud and Fog layers to significantly reduce latency for end-users.

Incorporating microservice placement within this Cloud-Fog computing framework adds a layer of complexity. Microservices, which are small, independent units within an application, must be strategically placed within this distributed infrastructure. The challenge lies in determining the optimal location for each microservice — whether in the centralized cloud for robust processing power or in the proximity-based fog nodes for quicker data access and reduced latency. This placement decision is pivotal to harnessing the full potential of Cloud-Fog computing, particularly in scenarios where resource constraints at the fog layer and the varying computational demands of microservices come into play.

This approach is crucial not only for enhancing data processing efficiency but also for ensuring optimal resource utilization across the distributed network. Such strategic placement of microservices within the Cloud-Fog continuum is a critical area of research and development, especially for applications that demand real-time responsiveness, including the Internet of Things (IoT), augmented reality, autonomous vehicles, and other cutting-edge technologies.

1.2 Scope of Thesis

This thesis focuses on implementing and evaluating microservice-based application placement methods on a distributed infrastructure using Kubernetes, within the Cloud-Fog computing paradigm. The primary objective is to assess the performance and Quality-of-Service(QoS) benefit when deployed in a distributed environment.

The research involves benchmarking these applications and collecting critical metrics such as CPU and RAM consumption, communication wise metrics between microservices, and individual microservice processing latency. This study aims to model these microservice applications as Directed Acyclic Graphs (DAGs) and develop a heuristic placement algorithm to strategically allocate microservices within the distributed infrastructure in a static manner. The algorithm will consider factors such as minimizing end-user latency while taking into account the resource constraints on computational nodes. By achieving effective placement of microservices closer to end-users while optimizing resource utilization, this research contributes to the advancement of Cloud-Fog computing and its potential to enhance the performance of real-time, latency-sensitive applications.

2. Background and related work

2.1. Cloud and Fog Computing Evolution

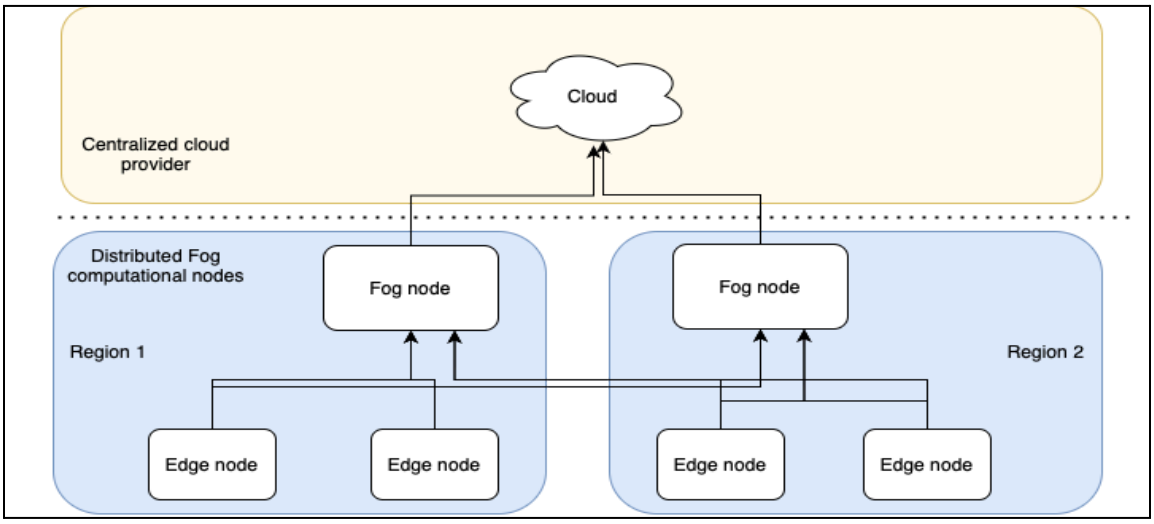
Cloud computing emerged as a revolutionary concept, promising centralized data processing, scalability, and cost-efficiency. Based primarily on virtualization technology, data centers worldwide can provide computational resources to multiple users, adhering to one's needs. Thus, it removes the complexity barrier of owning and managing a computer infrastructure, as well as solving the scalability problem when resources do not suffice.

Over time, as the demand for real-time data processing and low-latency applications surged, with the estimated number of devices connected to the global network being approximately 50 million[1], fog computing evolved as an extension of cloud computing, moving computational capabilities closer to the network edge. This shift was driven by the need to reduce latency and enhance the performance of applications like IoT, autonomous vehicles, and augmented reality, where responsiveness is critical.

The distinction between Cloud and Fog layers in distributed infrastructures lies in their proximity to data sources across the network. Cloud computing refers to a centralized remote data center, offering extensive and scalable computational power, and storage.
On the other hand, Fog layer computing is the latter's extension to the network edge, where devices such as IoT sensors and small servers are the data source. While Cloud computing can handle any heavyweight task and data storage demand, the Fog layer's priority is to serve

low-latency applications with real-time processing close to the data source, in the new realm of data-driven applications.

Together, cloud and fog computing have redefined how we utilize computing resources, providing a dynamic infrastructure that adapts to the requirements of modern, data-intensive applications.



[Cloud-Fog architecture]

2.2. Related work

Recent years have witnessed a surge in research within the Cloud and Fog computing domain, particularly concerning the efficient utilization of distributed infrastructures with a focus on application placement methodologies. Extensive research has been conducted to discover the most optimal strategies for deploying microservices across diverse computational nodes, with the main goal of enhancing end-user efficiency. Despite the emergence of numerous algorithms, it is crucial to acknowledge that the distributed nature of Fog computing comes with inherent complexity.

In their research documented in reference [2], the authors introduce an algorithm with a primary focus on Quality-of-Service (QoS) within the context of Cloud-Fog infrastructure for Internet of Things (IoT) applications. They demonstrate their approach using an example involving a fire alarm application, which emphasizes the critical significance of network latency between a set of such sensors and an application hosted in the Cloud.
The study initiates by establishing a broad set of variables and gathering relevant data that contains key decision criteria for the algorithm. These criteria contain values such as network latency and bandwidth characteristics of the infrastructure's network links, the interconnections among computational nodes, as well as their respective hardware and

software capabilities. Additionally, they consider similar to the latter, the application's requirements to meet the prerequisites necessary to define an acceptable deployment policy.

They introduce a three-part algorithm, each with its distinct function. These algorithms are tasked with the critical job of assessing the suitability of individual application components for various computational nodes. They evaluate factors such as hardware and software compatibility, and network capabilities, gathering a collection of deployments that meet the eligibility criteria. Subsequently, the remaining two algorithms step in to validate and refine these eligible deployments.
Within the scope of their research, they have developed a practical prototype application known as FogTorch[5]. This software provides recommendations for strategically placing different application components within distributed infrastructures. However, it does not work under a real use case scenario but rather an artificial one.

In [3], the authors propose a heuristic service placement approach for IoT applications on distributed and resource-constrained Fog nodes. The research focuses on IoT applications, known for their bandwidth-intensive nature and sensitivity to latency, with the added flexibility of deploying and scaling individual components on demand. In this scenario, the Fog paradigm comes to help with offloading the workloads across the network and avoiding congestion.
The experiment topology consists of nodes placed in hierarchical order, where nodes closer to the cloud have more resources.
In the proposed algorithm, they introduced load-balancing and service discovery capabilities in the nodes, granting each node a role in the decision-making process during microservices placement. With the constraint that the client module of the application, representing the user interface component, should always be placed at the lowest level, they designed an algorithm that utilizes the ability to scale the microservices among nodes at the same level, achieving a noticeable reduction in latency and bandwidth usage.
While the clustering capability at the Fog level is an interesting addition with added value, the experiment results remain under the simulation environment.

In reference [4], the authors present an infrastructure topology designed to accommodate specific modules facilitating dynamic service placement within a distributed infrastructure. Their research is based on a real-world scenario, specifically centered around a Smart City application developed to gather data and invent sustainable solutions within the city of Lisbon, Portugal.
Adhering to the continuously changing nature of the distributed Fog nodes, they propose an architectural framework comprising key components responsible for network data collection, microservice placement decision-making, and dynamic service allocation across diverse computational nodes. The core components are the Information Collection module, responsible for recurrently capturing network usage data across the topology and microservices, the Service Orchestrator, which leverages the information gathered to formulate

placement decisions, and the Service Repository, serving as a repository for storing deployment configurations required for all services.

The main objective is to achieve optimal network latency between end-users or sensors and the corresponding services they interact with. To reach this goal, the authors have implemented a dynamic allocation algorithm, empirically tested under two distinct scenarios. The first scenario involves random initial service placement. In contrast, the second scenario integrates predictive network analysis, seeking to anticipate an efficient initial placement strategy using available information before any service deployment.

2.3. Infrastructure and Tools

2.3.1. Microservices and Containerization

Virtualization technology has been a foundational concept in the field of IT infrastructure. It allows for the creation of virtual instances, such as virtual machines, that can emulate computer systems with their operating systems [6]. These virtual machines provide a realistic and isolated execution environment, allowing for efficient resource utilization while maintaining security and isolation. However, as cloud computing and the demand for more flexible and scalable infrastructures emerged, a lightweight virtualization technology known as containerization gained popularity.

Containers emerged as a response to the need for more efficient resource utilization and faster deployment of applications in distributed infrastructures. Containers provide abstraction at the application layer, allowing for the packaging of an application along with all its dependencies, libraries, settings, and system tools into a single image. These container images can be constructed from filesystem layers, making them lightweight and taking up considerably less space than virtual machines. The flexibility and efficiency of containers have made them highly suitable for cloud-native technologies and microservices architectures. Moreover, containers offer benefits in terms of ease of deployment, testing, and composition for developers.

One of the key advantages of containers is their efficient resource utilization and rapid deployment capabilities. Unlike traditional virtual machines, containers share the host system's kernel and do not require a separate guest operating system for each instance. As a result, they consume fewer resources and can be started and stopped quickly, making them ideal for dynamic and distributed infrastructures.
Additionally, containers provide a consistent and reproducible runtime environment for applications, ensuring that they can run consistently across different computing environments.

This allows developers to package their applications with all the necessary dependencies and configurations, eliminating compatibility issues and making it easier to share and deploy applications across different systems and platforms. Furthermore, containers offer isolation and security by ensuring that each application runs in its isolated environment. This isolation prevents interference between applications and enhances the overall security of the system.

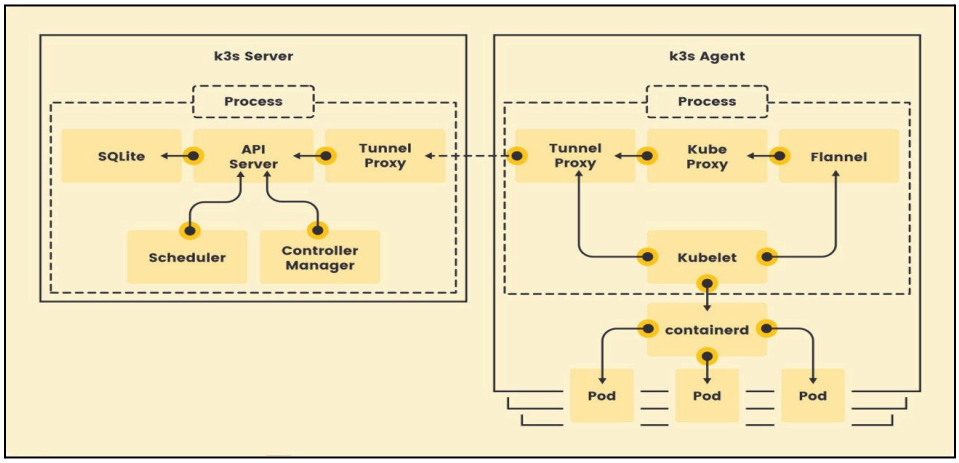
Docker[7] played a significant role in the establishment of containers as the industry standard. Launched in 2013, Docker provided a user-friendly platform for creating, managing, and deploying containers. Its ease of use and portability made containers accessible to a broader audience and contributed to their widespread adoption.
The combination of Docker's user-friendly interface, portability, and robust ecosystem played a pivotal role in establishing containers as the industry standard, revolutionizing the way applications are built, deployed, and managed in the digital age.

2.3.2. Container orchestration - Kubernetes and k3s

In recent years, the adoption of containerization technology has skyrocketed. Containers, which package software and its dependencies into a standardized unit, offer numerous benefits such as portability, scalability, and reproducibility. However, as the use of containers became more widespread, managing and orchestrating them at scale became a challenge. This led to the emergence of container orchestration, a practice that enables the efficient management of containers across various environments. Container orchestration is necessary because as the number of containers and containerized applications grew, so did the complexity of managing them.
Container orchestration platforms, such as Kubernetes[8], have quickly become the de facto standard for managing containerized applications at scale. Kubernetes, originally developed by Google, has gained widespread adoption due to its robust features for automating deployment, scaling, and management of containerized applications. One of the key advantages of Kubernetes is its ability to provide a unified platform for deploying and managing applications across diverse infrastructures, whether on-premises, in the cloud, or in hybrid environments. Its declarative configuration and self-healing capabilities make it a powerful tool for ensuring the availability and reliability of containerized workloads.
As the adoption of containerization continues to grow, Kubernetes is expected to play a pivotal role in shaping the future of modern infrastructure and application deployment practices. However, while Kubernetes offers numerous benefits, it can also be complex and challenging to learn and implement for many developers. This is where K3s [9] emerges as a pivotal solution in this context..

K3s is a lightweight Kubernetes distribution that addresses the complexity and resource requirements of standard Kubernetes deployments. It simplifies the deployment and management of Kubernetes by reducing its resource requirements, making it more accessible to developers with limited resources or expertise in Kubernetes. K3s achieves this by eliminating unnecessary components and slimming down the installation process, resulting in

a more lightweight and easily deployable Kubernetes solution. This makes it ideal for edge computing and resource-constrained environments where efficiency and simplicity are essential.



[k3s architecture- https://k3s.io]

2.3.3. Service mesh

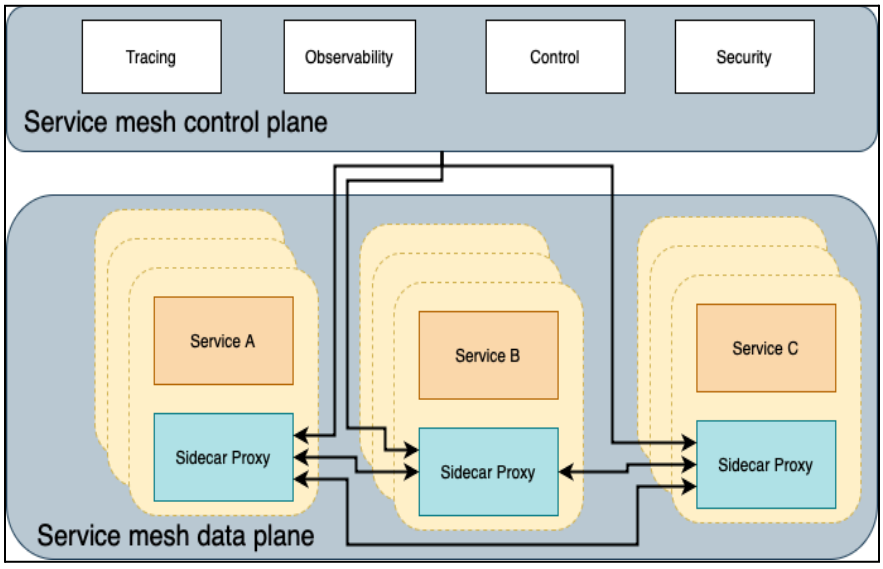
Service mesh technology is an essential component for handling communication between microservices within a Kubernetes environment. In addition to facilitating microservice communication through features like service discovery, load balancing, and security policies, it also offers capabilities such as traffic management, observability, and fault tolerance.

One of the key components of service mesh technology is the sidecar proxy, which is deployed alongside each microservice. This proxy intercepts and manages all inbound and outbound traffic, enabling communication between services to be more reliable and secure.
In addition to this, service mesh technology also facilitates observability by providing metrics, logs, and traces of communication between microservices. This visibility into the communication patterns helps in better understanding the performance and behavior of the entire application.
By integrating service mesh technology in Kubernetes environments, one can achieve better control, security, and observability of their microservices architecture, leading to improved reliability and performance of their applications.

2.3.3.1. LinkerD

Among the service mesh tools available, Linkerd [10] is a well-regarded choice for Kubernetes environments due to its lightweight and efficient solution. It seamlessly integrates with Kubernetes deployments and offers features such as automatic mTLS encryption, request-level telemetry, and dynamic service discovery. This simplifies the implementation of communication between services while providing insights into microservices' performance and behavior. Additionally, Linkerd's transparent proxy architecture reduces the operational

overhead associated with service mesh deployment, making it appealing when aiming to improve the resilience and observability of their Kubernetes-based applications.



[Service mesh architecture schema]

This architecture is divided into two distinct planes: the data plane and the control plane. The data plane is responsible for handling the actual transport of service-to-service communication within the mesh. It typically consists of a network of lightweight proxies, deployed alongside application code, which intercept and route requests, thus ensuring reliable data transfer, load balancing, and encryption. On the other hand, the control plane functions as the administrative and configuration core of the service mesh. It is responsible for managing and configuring the proxies in the data plane, enforcing policies, aggregating and reporting telemetry data, and providing the necessary tools for service discovery, security, and observability.
Some of the key operational points of LinkerD are:

- Load Balancing
- Traffic Management
- Reliability and Fault Tolerance
- Security
- Observability

This segregation into two planes allows for a high degree of control and flexibility, enabling developers and operators to manage inter-service communications more effectively, without embedding this logic into the microservices themselves, thus maintaining the agility and scalability essential in modern cloud-native environments.

2.3.4. Locust

Benchmarking tools play a crucial role in the development and optimization of cloud-native applications. These tools enable developers to gauge the performance, scalability, and reliability of their applications in a cloud environment. By simulating real-world scenarios and workloads, benchmarking tools provide valuable insights into the behavior of an application under varying conditions.

One of the key benefits of benchmarking tools is their ability to identify potential bottlenecks and performance issues within an application. This allows developers to make informed decisions about resource allocation, architectural improvements, and optimizations to enhance the overall performance of their cloud-native applications.

Moreover, benchmarking tools help in setting performance baselines and comparing different versions of an application or different cloud providers. This enables developers to make data-driven decisions when choosing the optimal configuration or infrastructure for their applications.

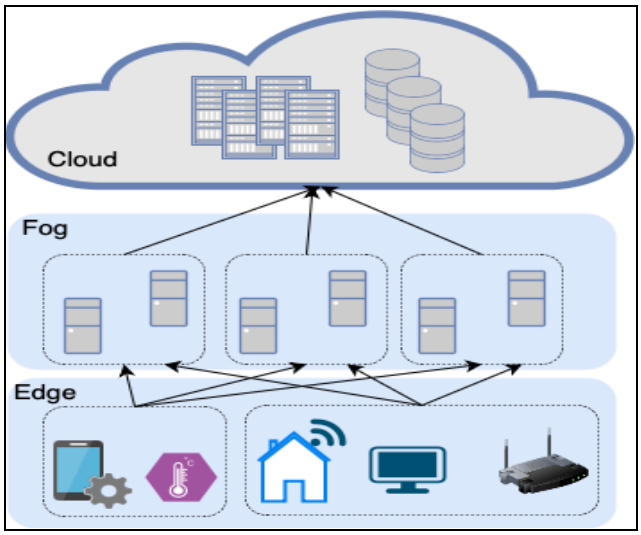
For this particular study, we used Locust [11] for all of the tests presented later on.
Locust is an open-source, Python-based load-testing tool famous for its scalability and flexibility in simulating user behavior for web applications and services. Unlike traditional testing tools, Locust allows testers and developers to script user behaviors and test scenarios directly in Python, offering a great level of customization and complexity in testing.
This feature makes it especially suitable for complex applications that require detailed testing strategies.

Locust is capable of simulating millions of simultaneous users, making it a powerful tool for understanding how much load an application can handle. It also supports distributed testing across multiple machines, ensuring robust performance analysis. With its web-based interface, users can easily initiate tests, monitor real-time user simulations, and analyze results through intuitive charts and statistical data.

3. Distributed topology architecture

3.1. Overview

In this section, we outline the architecture and design of the topology developed for hosting the experiments of this study. Our goal is to closely replicate an actual distributed topology, with its inherent complexity and unique characteristics, to mirror a real-world scenario. This involves strategically deploying an application across our artificially made geographically distributed computing nodes, aiming to serve users efficiently and effectively.



[Cloud-Fog-Edge topology]

To accomplish this objective, we utilized services provided by Google Cloud to establish the nodes of our topology. Specifically, we chose Compute Engine, a virtual machine (VM) service, as the foundation for hosting the necessary tools. On top of these VMs, we constructed an interconnected multi-cluster environment using Kubernetes, employing K3S for its lightweight architecture. Furthermore, we integrated LinkerD as a plugin to implement a service mesh across all microservices. The following section will provide a detailed exposition of this implementation.

3.2. Cloud-Fog infrastructure

3.2.1. K3S cluster overview

In our infrastructure, each virtual machine comprises a single-node K3S cluster tasked with the management and orchestration of all containerized deployments. The resource allocation for these virtual machines is predetermined at the time of provisioning and remains fixed throughout their operational lifespan. The efficiency of K3S, attributed to its ability to run components in a singular process, facilitates minimal resource utilization within our virtual machines. The server component of K3S operates several essential elements, including the core control-plane components of Kubernetes (API, controller, and scheduler), SQLite, and a reverse tunnel proxy. Conversely, the K3S agent component is responsible for managing the kubelet, kube-proxy, container runtime, load-balancers, and other ancillary components essential for pod management.

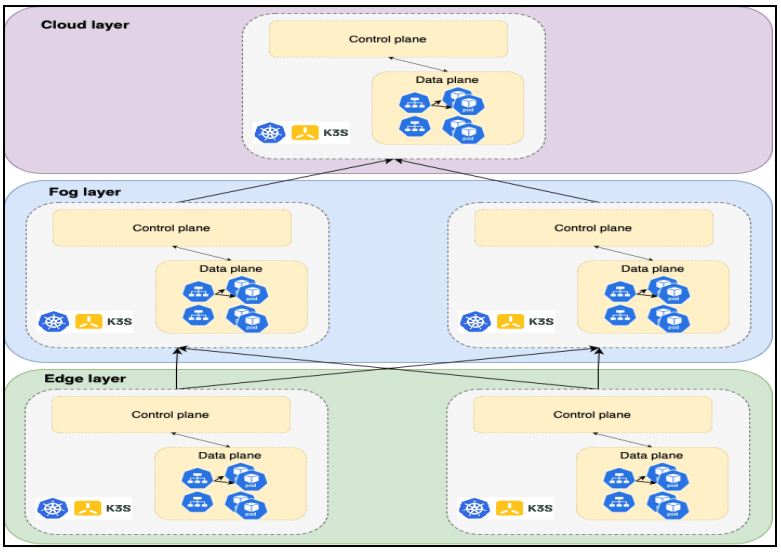
On each host, the Deployment service enables the running of multiple pods, akin to standard Kubernetes usage, where each pod can encapsulate multiple containers. Additionally, we employ the Service API [13], by creating the necessary services that provide a mechanism to

expose a network endpoint with static details (such as IP and port) for one or more designated pods. This feature ensures streamlined communication among the containers.

3.2.2. Multi-cluster architecture with service mesh

To imitate a distributed infrastructure, we need to employ multiple VMs, each of them hosting a K3S cluster. The architecture consists of three layers depicting the Cloud-Fog-Edge hierarchy.

In this context, 'Cloud' denotes a centralized data center, theoretically offering unlimited resources for an application's use without constraints. Following this in the hierarchical structure is the 'Fog' layer, encompassing computing nodes situated in various geographical locations. These nodes are positioned nearer to the network's edge and are characterized by their constrained resources, in contrast to the expansive capabilities of the Cloud. Concluding the topology is the 'Edge' layer, situated in closest proximity to the data origin, where users or sensors generate network traffic for applications.



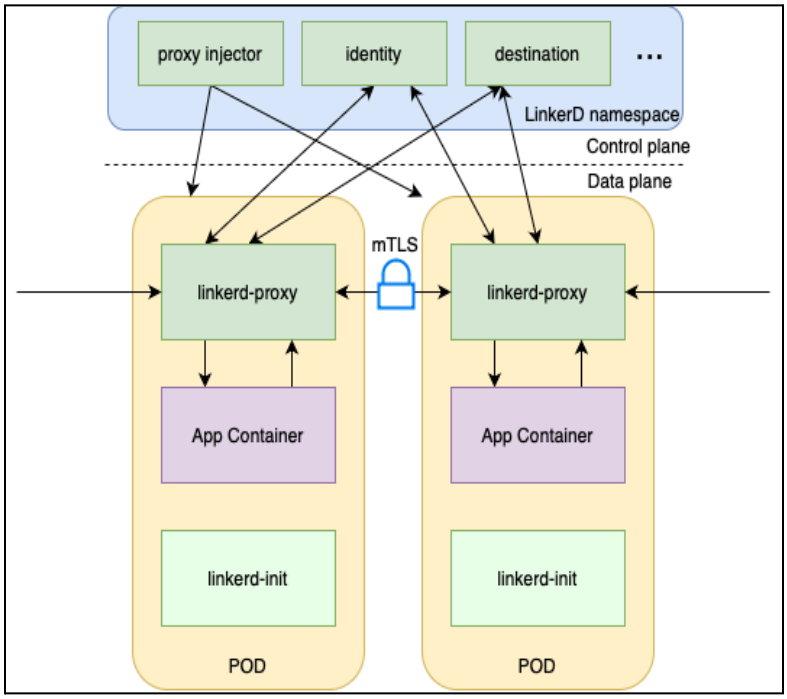
[Cloud-Fog-Edge architecture]

As depicted in the image above, the topology consists of five K3S clusters, each one deployed on a unique virtual machine. Each of these clusters has one node available for running workloads, whose available resources are predetermined by the VM that it is running on. Since an application can potentially be deployed in multiple locations, seamless communication between its different components must be guaranteed, to provide stability.

Linkerd's multicluster service mesh provides a solution for this problem. It is a sophisticated framework designed to facilitate seamless, secure, and efficient communication across multiple Kubernetes clusters. At its core, the multicluster service mesh extends the capabilities of Linkerd's single-cluster service mesh, allowing services in different Kubernetes clusters to communicate with each other as if they were in the same cluster. This is achieved through a

shared trust model and a unified control plane that orchestrates cross-cluster communication. The Linkerd multicluster setup includes components like service mirror controllers and gateway proxies, which collectively work to establish and manage connections between clusters. These connections are secured using mutual Transport Layer Security (mTLS), ensuring that inter-cluster communications are encrypted and authenticated.

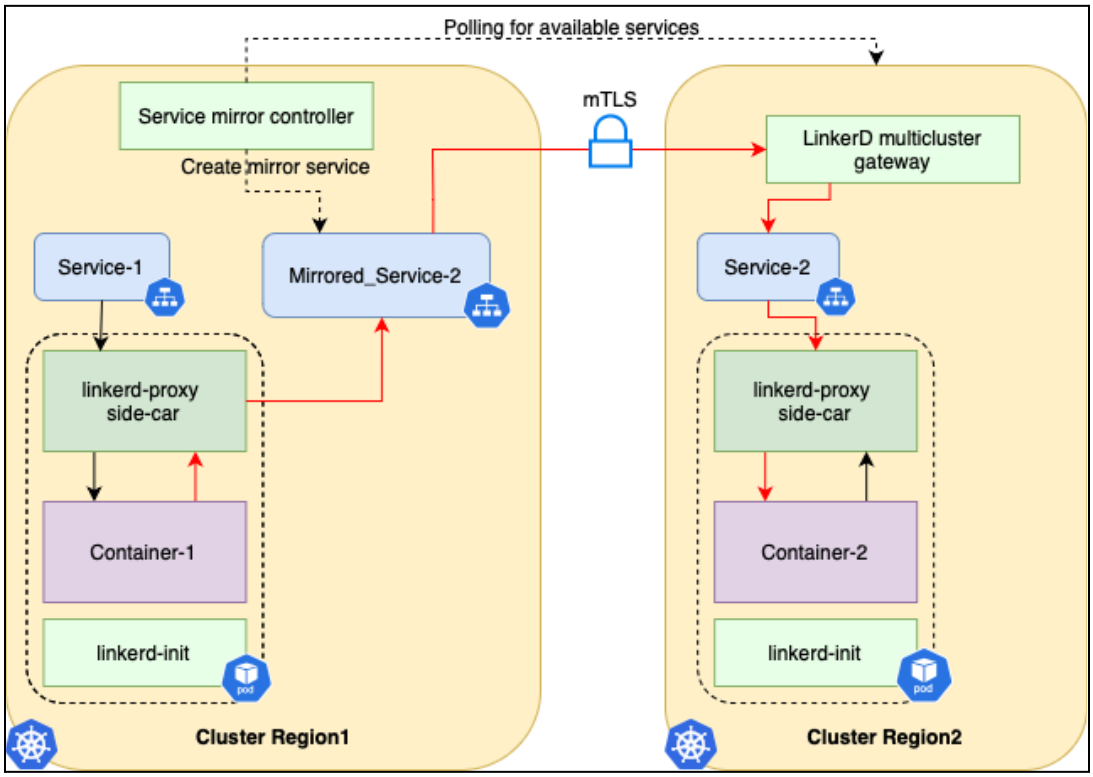
In practice, Linkerd's multicluster service mesh addresses common challenges in running applications in distributed environments, such as handling failover, replication, and cross-cluster service discovery. For instance, services in one cluster can seamlessly failover to services in another cluster, enhancing the overall resilience and availability of the system. Additionally, the multicluster architecture simplifies complex operational tasks, enabling a more straightforward approach to global observability, traffic management, and policy enforcement across clusters.
Linkerd comes equipped with a comprehensive observability plugin that provides detailed metrics and insights into the behavior of services within the mesh. This plugin automatically collects, aggregates, and displays metrics such as request volume, success rates, response times, and latency distributions for each service, using the injected proxy at each pod.
These metrics are instrumental in identifying and diagnosing issues, optimizing performance, and making informed decisions about scaling and resource allocation. Some well-known tools for metrics collection and visualization embedded are Grafana and Prometheus.



[LinkerD service-mesh schema]

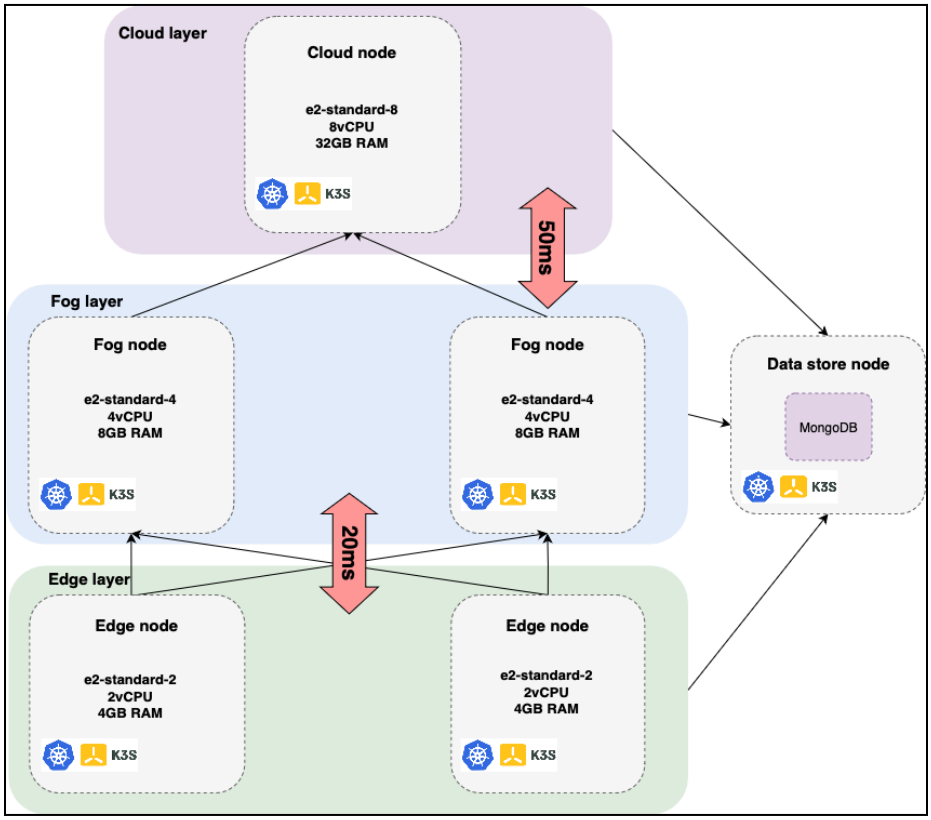
Initiating the LinkerD multicluster connection between two clusters involves deploying a "Gateway" pod in the target cluster. This pod is tasked with accepting requests from the source cluster and directing them to the appropriate service. Meanwhile, in the source cluster, a "Service mirror" component actively monitors the target cluster, identifying available services to replicate within the source cluster.

For instance, if microservice1 in the source cluster needs to interact with microservice2 in the target cluster, it will attempt to connect to service2. However, since service2 isn't present in the source cluster and instead its mirrored version exists, we set up a Traffic Split [14]. This configuration instructs all Linkerd-proxy sidecars to route requests to the mirrored version of service2 as required.



[LinkerD Multicluster setup between two k3s clusters]

In our topology, we leverage Google Cloud's platform for virtual machine provisioning, installing essential tools, and simulating a Cloud-Fog-Edge environment. As defined earlier, we aim to mimic a resource-limited setting by allocating fewer resources to machines further from the cloud. The resource allocation for these nodes is illustrated in the image below.



[Cloud-Fog-Edge topology in Google Cloud Platform]

A critical element of our topology is the introduction of artificial network delay. In real-world situations, computational nodes in such environments are often dispersed across various geographical locations, naturally resulting in network latency. However, since we're hosting our VMs in a cloud provider's data center, the inherent network delay is minimal.
We also exclude the "*Data Store*" node from the delays, as its used to host the database of the application.

To simulate this delay, we used a Linux tool known as Traffic Control (tc) [15]. Traffic Control is a robust utility designed for configuring the kernel's packet scheduler. It can be used to artificially induce packet loss, bandwidth restrictions, network delays, and other forms of traffic management.

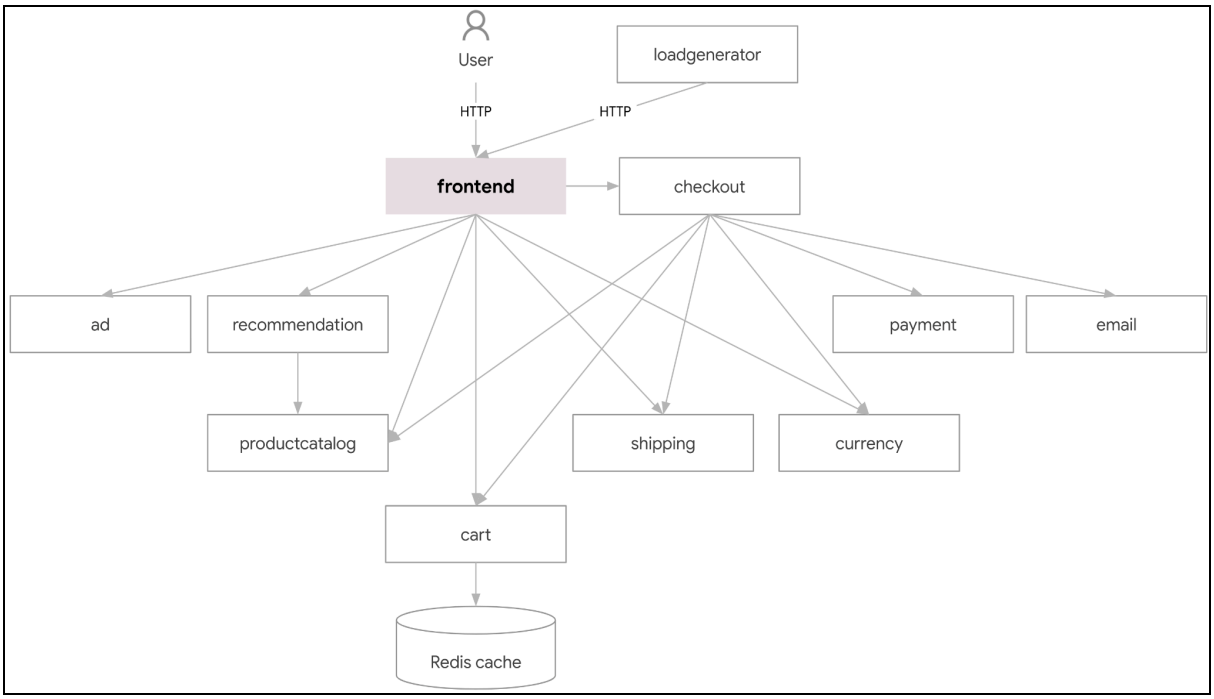
4. Application Service Placement

4.1. Applications

4.1.1. Online Boutique by Google

The Online Boutique [16] (*hereafter referred to as Eshop*) is a reference application developed by Google and made available as open-source. It serves as an educational tool for those who want to dive into the microservices world.
This application is used to illustrate various concepts such as scaling strategies and deployment models. Google uses this application to showcase the functionality of its tools, such as to illustrate its Kubernetes Engine (GKE) service functionality and more.

Structured as a collection of multiple microservices, the Online Boutique simulates a comprehensive e-commerce platform. These microservices, encompassing a range of functionalities like product catalog management, shopping cart operations, checkout processes, and payment handling, are independently designed. This independence facilitates enhanced flexibility and scalability of the application.
Communication among these services is efficiently handled by gRPC [17], a high-performance remote procedure call (RPC) framework known for facilitating effective communication in distributed systems.
Within the architecture of the application, each microservice is dedicated to a specific role, with inter-service communication managed through APIs, thereby ensuring a scalable and efficient e-commerce experience for users. As detailed on its official GitHub repository, the Online Boutique comprises 12 distinct microservices, each written in a different programming language. The figure below depicts the application's schema.



Employing different programming languages and technologies, each component was built to provide particular functionality, described below in detail:

- Frontend [Go]:
    - *Exposes an HTTP server to serve the website. Does not require signup/login and generates session IDs for all users automatically.*
- Cartservice [C#]
    - *Stores the items in the user's shopping cart in Redis and retrieves them.*
- Productcatalogservice [Go]
    - *Provides the list of products from a JSON file and the ability to search products and get individual products.*
- Currencyservice [*Node.js*]
    - *Converts one money amount to another currency. Uses real values fetched from the European Central Bank. It's the highest QPS service.*
- Paymentservice [*Node.js*]
    - *Charges the given credit card info (mock) with the given amount and returns a transaction ID.*
- Shippingservice [*Go*]
    - *Gives shipping cost estimates based on the shopping cart. Ships items to the given address.*
- Emailservice [*Python*]
    - *Sends users an order confirmation email (mock).*
- Checkoutservice [*Go*]
    - *Retrieves user cart, prepares orders and orchestrates the payment, shipping, and email notification.*

- Recommendationservice [*Python*]
    - *Recommends other products based on what's given in the cart.*
- Adservice [Java]
    - *Provides text ads based on given context words.*
- Loadgenerator [*Python/Locust*]
    - *Continuously sends requests imitating realistic user shopping flows to the frontend.*

4.1.2. MartianBank by Cisco

MartianBank, a project developed by Cisco, serves as an open-source demonstration of microservices architecture [18]. This application, demonstrating cloud-native tool capabilities, models an online banking system. It encapsulates essential features typical of such platforms, including functionalities that allow customers to access and manage their accounts, execute financial transactions, find ATMs, and submit loan applications [19].

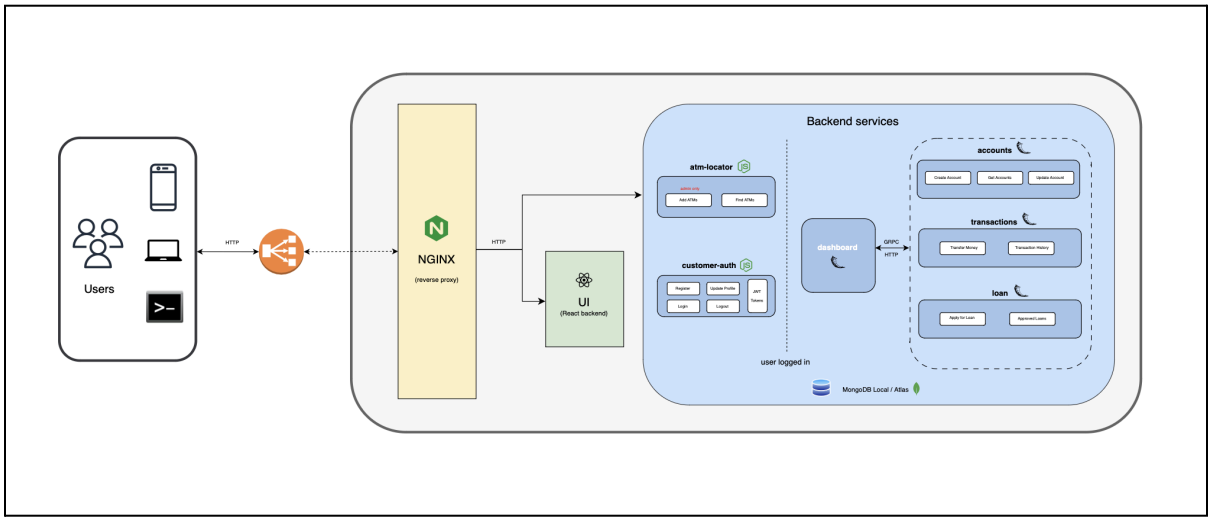
Constructed with a variety of languages and technologies, MartianBank is pre-configured for containerized deployment. It stands as a ready-to-use solution, particularly suited for implementation in Kubernetes environments. This makes it an ideal testbed for exploring fault tolerance strategies and experimenting with various other technological aspects, such as container orchestration, management, and deployment strategies validation in the Cloud environments.

The application's architecture consists of seven microservices, each capable of communicating through gRPC or HTTP protocols. These services, described in detail below, collectively contribute to the application’s functionality, showcasing a real-world implementation of a microservice-based system.

- UI [*React JS*]:
    - Illustrate the frontend page of the application and act as a router from the requests initialized by the user.
- Dashboard [*Python*]:
    - Built using the Flask framework, it implements the API that handles and routes all requests from the frontend to the targeted backend service.
- Atm-locator [*Node.js*]:
    - Provides the necessary information about the bank’s available ATMs, with an integrated map depicting the precise location.
- Customer-auth [*Node.js*]
    - This component handles user-related requests such as registration, authentication, information update, and user profile retrieval.
- Accounts [*Python*]:
    - Acts as a proxy to deliver all information for one or more users as well as the new user creation requests.

- Transactions [*Python*]:
    - All transaction-related requests are handled by this microservice, either for initiating a new bank transfer or for retrieving the history.
- Loan [*Python*]:
    - Applying for a new loan, reviewing the status of an approved loan or requesting the history of previous ones, everything is managed by Loan microservice.
- Nginx:
    - *Acts as a proxy for all requests arriving at the application.*

For our implementation, Nginx was replaced with the default K3S Ingress controller (Traefik[20]). This controller is a LoadBalancer Kubernetes service, which routes the incoming requests to the relevant pods.
Also, the *UI* and *Dashboard* modules have been integrated, with the *UI* offering the graphical interface of the application and the *Dashboard* managing and directing user requests to the relevant backend module. This composite module forms the client-side entry point of the application.



[MartianBank schema]

4.2. Application Benchmarking

When it comes to evaluating microservice application deployment strategies on Kubernetes environments, one of the key components is application benchmarking. Benchmarking allows for the analysis of the performance and behavior of the application under specific conditions. This can provide valuable insights into the strengths and weaknesses of different deployment strategies.

Advantages of application benchmarking include the ability to identify potential performance bottlenecks, understand resource utilization, and compare the effectiveness of different deployment strategies. By simulating real-world conditions, benchmarking can also help in predicting how the application will behave in a real-life production environment.

Overall, while application benchmarking offers valuable insights, it's important to carefully weigh its advantages and disadvantages when evaluating microservice application deployment strategies.

In our research, we employed Locust to generate artificial traffic for benchmarking, adhering to pre-established scenarios tailored to both applications. For the purpose of benchmarking, we deployed each application within a single-node K3S cluster, replicating a scenario akin to utilizing a Kubernetes solution offered by a Cloud provider, but without any constraints on resources.
Subsequently, we executed Locust scripts to initiate the load on the application. These scripts were comprehensively designed to simulate various traffic scenarios, effectively testing the full range of application functionalities and procedures.

Eshop locust scenario		
Request	*HTTP Type*	*Distribution of requests*
Browse homepage	GET	4.53%
Retrieve shopping cart	GET	13.04%
Add product to shopping cart	POST	13.03%
Proceed to checkout	POST	4.40%
View product	GET	56.44%
Change currency	POST	8.56%
Total		**100.00%**

MartianBank locust scenario		
Request	*HTTP Type*	*Distribution of requests*
Browse homepage	GET	9.09%
User login	POST	4.55%
Retrieve user profile	GET	15.91%
Update profile information	POST	4.55%
User logout	POST	4.55%
Create new user account	POST	2.28%
Retrieve all user's bank accounts	GET	6.82%
Retrieve particular bank account details	GET	6.82%
Retrieve all ATM locations	GET	13.5%
Retrieve particular ATM details	GET	9.09%
Create new bank account	POST	2.27%
Create new transfer	POST	9.8%
New loan application	POST	6.5%
Retrieve loan details	GET	4.29%
Total		**100.00%**

For initiating a benchmark test using Locust, specifying the total count of simulated users is essential. Each user in Locust represents an individual process that executes tasks defined in the testing scenario. In our research, we configured the number of concurrent users to be 150.

These testing scenarios are crucial for deriving important metrics that reflect the application's performance, which are later used for determining deployment strategies. In setting up our experiment, we leveraged the telemetry and monitoring features of the LinkerD service mesh. This enabled us to acquire various metrics, including service-wise average response latency, request rates per second, transport-level statistics, and resource usage for each pod.

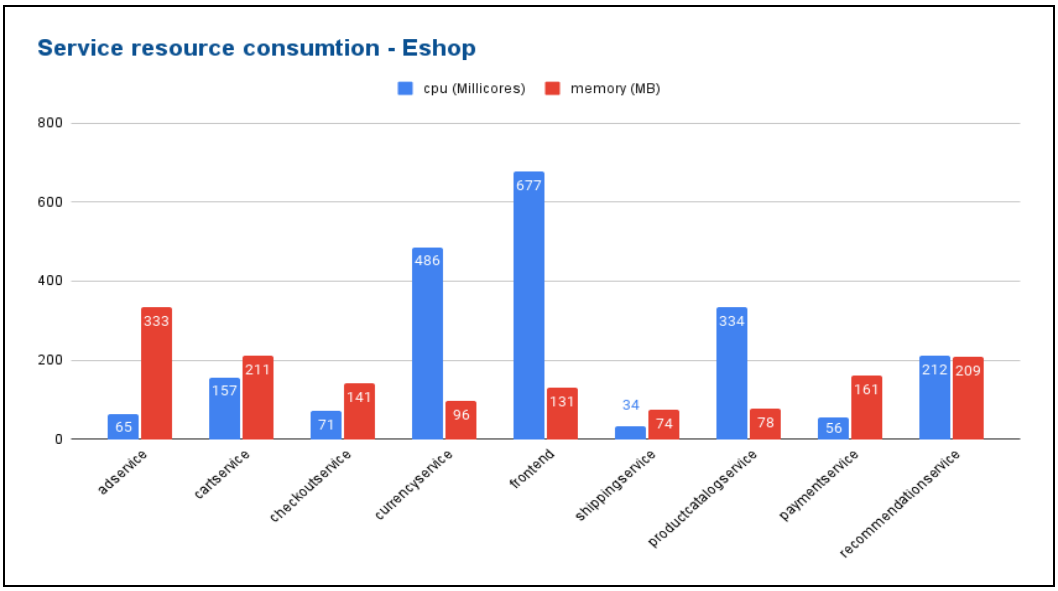
LinkerD integrates a Prometheus server, which is utilized for gathering these metrics through the PromQL query language.
As an example, the queries employed for gathering metrics for our algorithm are outlined below:

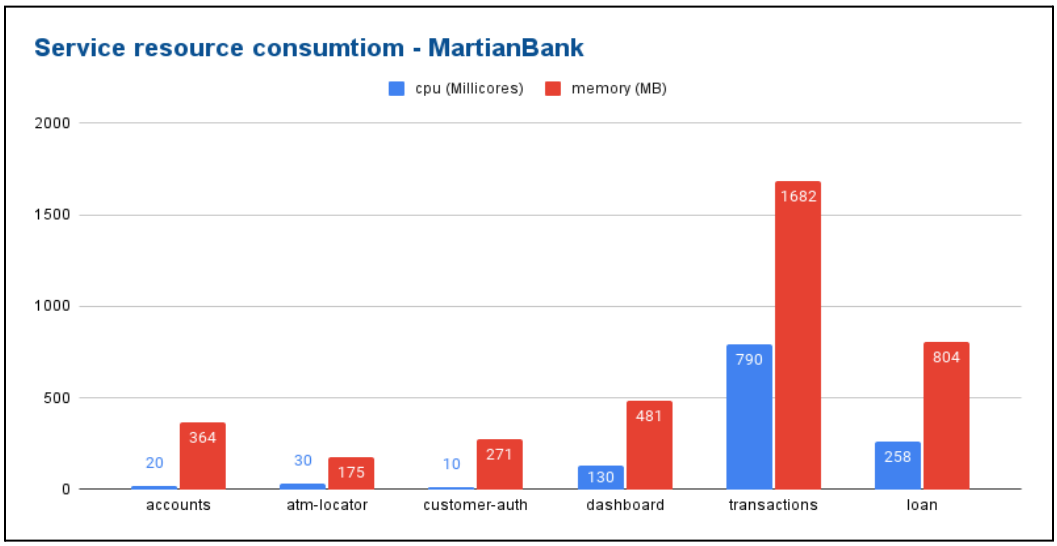
```
# Collect the average response time for each service
response_latency_ms_sum, response_latency_ms_count

# Collect the rate of R/W data traffic between two services
tcp_write_bytes_total, tcp_read_bytes_total

# Collect the rate of requests sent from service-to-service
request_total

# Collect the average CPU utilization of a service
container_cpu_usage_seconds_total

# Collect the average memory utilization of a service
container_memory_usage_bytes
```

We accumulated all relevant statistics during the 20-minute window of intensified application usage. Utilizing various inherent functions of Prometheus, we then computed the average values of these metrics.

Beginning by assessing the resource usage of each application service under simulated traffic conditions, we gather crucial data that aid in making informed decisions about service placement, especially in environments with limited resources. The following graphs illustrate the outcomes obtained from each application's benchmarking process.

CPU consumption is quantified in Milllicores, where, within Kubernetes, one CPU or vCPU core is equivalent to 1000 millicores. Memory usage, initially measured in bytes, has been converted to megabytes (MB) for clarity and ease of understanding.





Moreover, to gather diverse metrics to develop deployment strategies based on various criteria, we utilized PromQL queries as previously illustrated. For our experiments, we chose to focus on metrics such as the average number of requests per second (RPS), average data throughput in bytes, and average response latency per request between services. Additionally, we formulated a weighted affinity metric that combines 50% of the requests per second value and 50% of the average latency value between two services, for all service-to-service connections.

The formula used to calculate the weighted affinity between all microservices is:

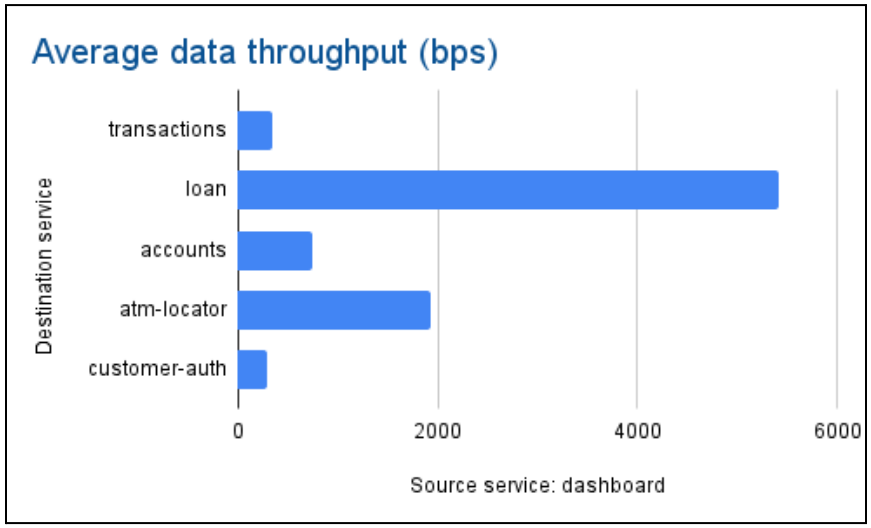
$$weighted_{affinity} = \frac{\sum_{i=1}^{2} w_i * X_i}{\sum_{i=1}^{2} w_i}$$

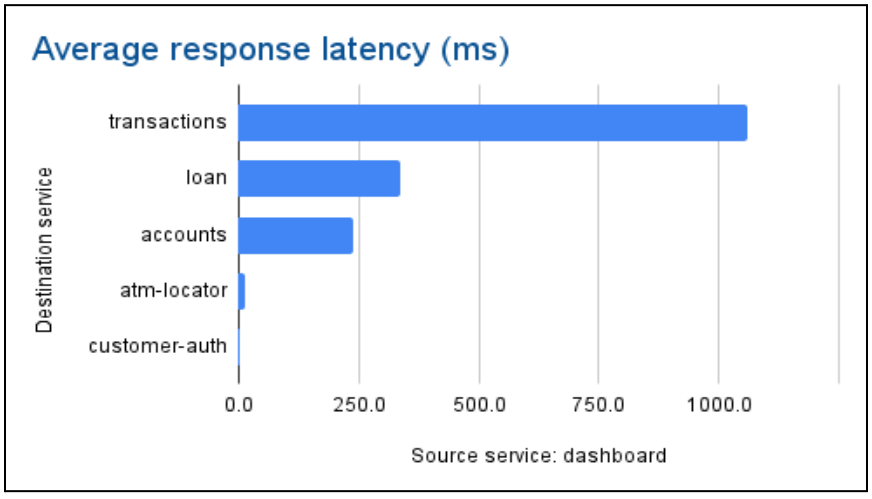
$$where\ X_1 = RPS(service \rightarrow service),\ X_2 = Avg_{latency}(service \rightarrow service)\ and\ w_{1,2} = 0.5$$

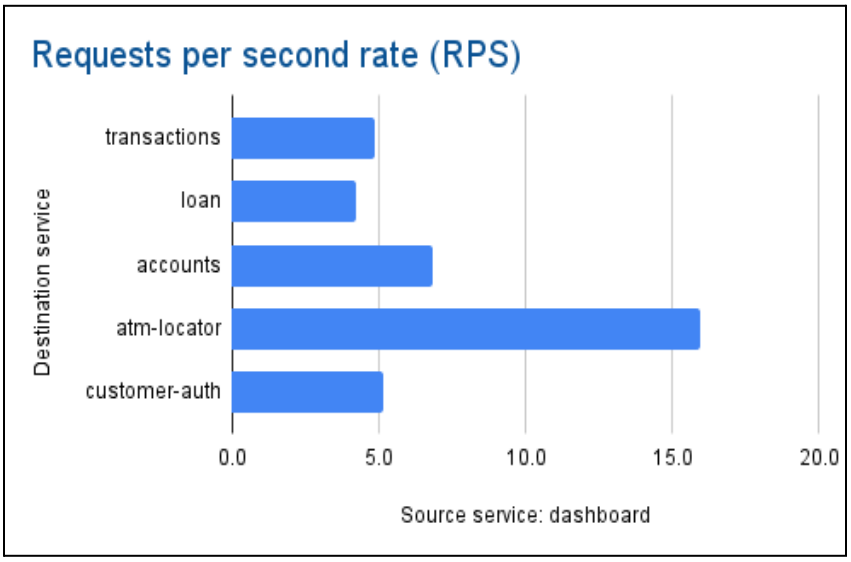
The affinities among these microservices are pivotal in the context of an application and its deployment strategy. They significantly influence the application's performance from the end-user's perspective, particularly within a distributed topology environment.

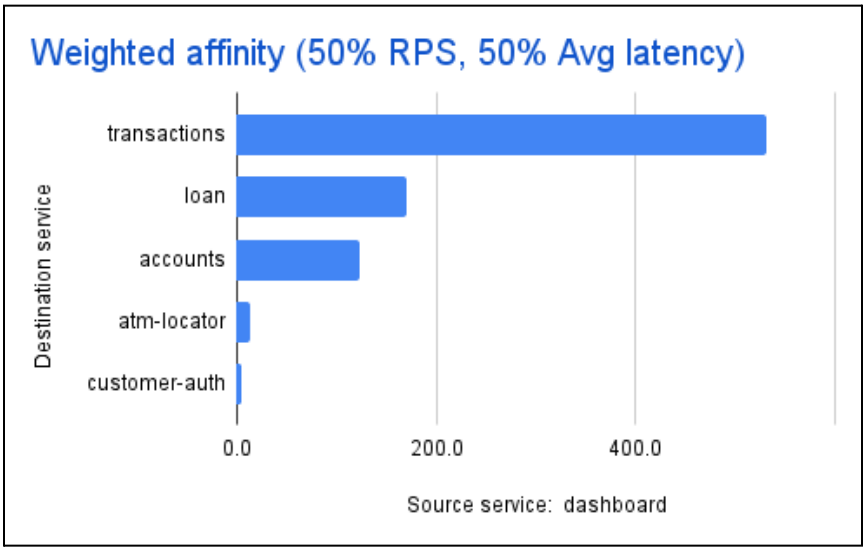
Analyzing the MartianBank application reveals insightful details about the behavior of its various components and their interactions. Notably, all requests initially go through the dashboard service, which then redirects them to the appropriate microservice, making the dashboard the primary source of all inter-service communication.

We observe that the connection with the *loan* microservice involves the most significant amount of data exchange, whereas the *transactions* microservice exhibits the highest response latency. Additionally, the *Requests Per Second* (RPS) affinity is influenced not just by the distribution pattern of the benchmark test but also by each microservice's latency. This is further impacted by the tool's design to generate requests with a random delay ranging from one to five seconds.



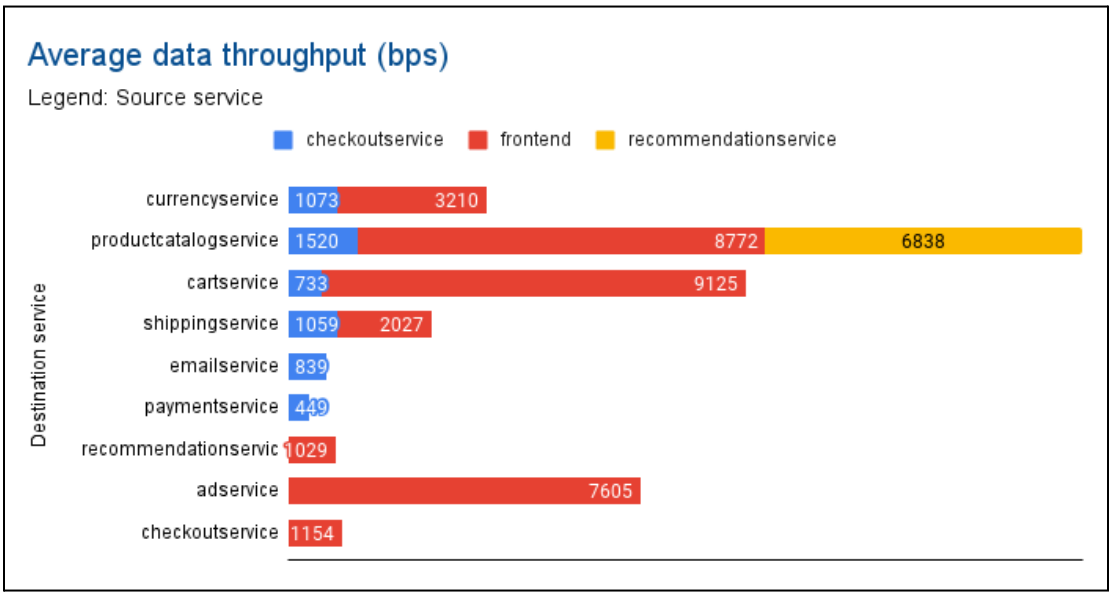


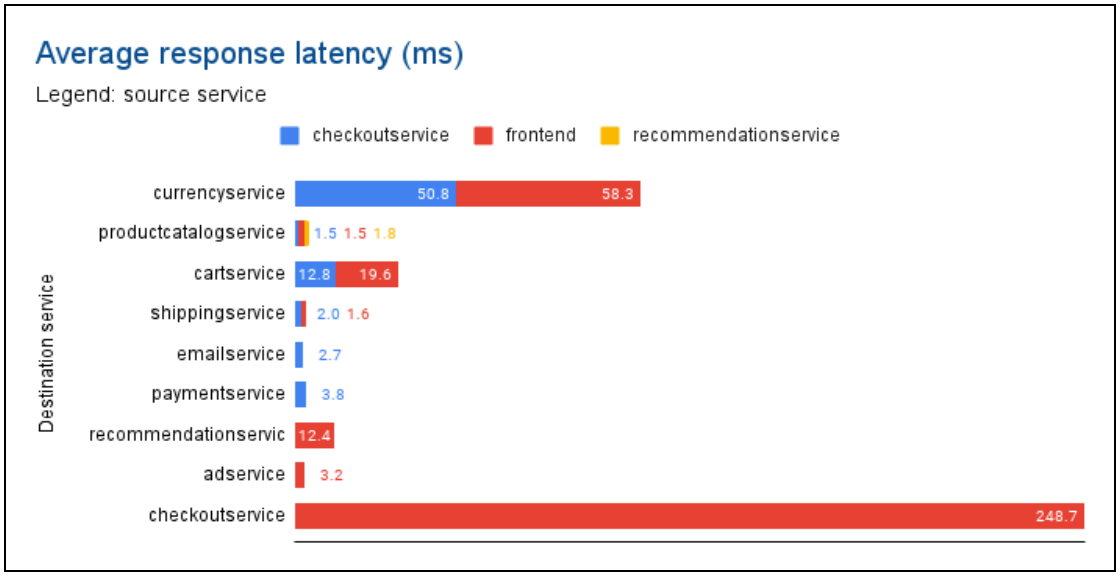

Requests per second rate (RPS)
transactions
loan
accounts
atm-locator
customer-auth
Destination service
0.0
5.0
10.0
15.0
20.0
Source service: dashboard

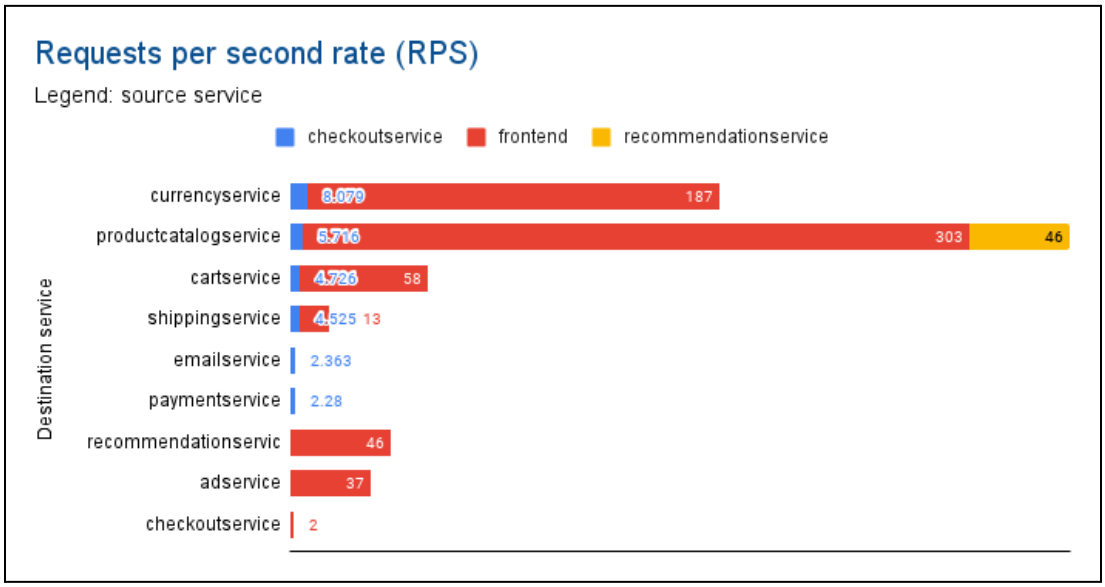

Weighted affinity (50% RPS, 50% Avg latency)
transactions
loan
accounts
atm-locator
customer-auth
Destination service
0.0
200.0
400.0
Source service: dashboard

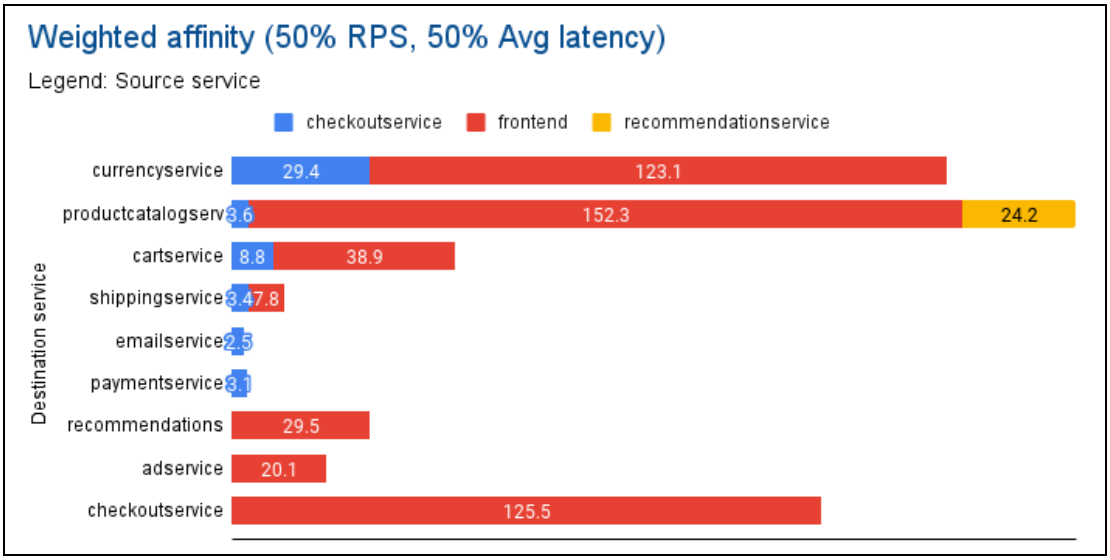
Progressing with the examination of Google's Online Boutique application, its intricate architecture becomes apparent. This application involves extensive inter-service communications, with the frontend service acting as the central hub for all requests.

Key observations from the analysis reveal that the productcatalog microservice is the most data-intensive, receiving the majority of requests, especially as users browse products. Additionally, in terms of response latency, the checkout microservice registers the highest delays, followed by the currency microservice. Notably, the currency microservice handles a significantly higher volume of requests on average.




Average response latency (ms)
Legend: source service
checkoutservice
frontend
recommendationservice
Destination service
currencyservice
productcatalogservice
cartservice
shippingservice
emailservice
paymentservice
recommendationservic
adservice
checkoutservice
50.8
58.3
1.5 1.5 1.8
12.8
19.6
2.0 1.6
2.7
3.8
12.4
3.2
248.7


Requests per second rate (RPS)
Legend: source service
checkoutservice
frontend
recommendationservice
Destination service
currencyservice
productcatalogservice
cartservice
shippingservice
emailservice
paymentservice
recommendationservic
adservice
checkoutservice
8.079
187
5.716
303
46
4.726
58
4.525 13
2.363
2.28
46
37
2



Following the metrics collection, the next section will detail how we can employ this information to develop and assess service placement strategies for both applications.

4.3. Service Placement Strategies

4.3.1. Application as weighted Directed Acyclic Graph

In this section, we explore the treatment of microservice-based application architectures as Directed Acyclic Graphs (DAGs)[21], highlighting the benefits of this approach, how we incorporate benchmark metrics as service-to-service affinities, and our graph traversal placement algorithm and its outcomes in the subsequent section.

Visualizing the relationships and dependencies among microservices through a DAG format is insightful. In such a graph, each microservice is a node, and their interactions or dependencies are the directed edges connecting these nodes. This representation is valuable for understanding data flow, request handling, and dependencies within the microservices architecture.

When it comes to placement algorithms, the DAG model of microservices is particularly advantageous. These algorithms are essential for efficiently deploying and scaling microservices in distributed systems. The DAG perspective allows for a comprehensive understanding of microservices and their interrelationships, enabling placement algorithms to

optimize aspects such as communication latency, bandwidth, and resource usage, resulting in more effective microservice placement.

Additionally, the DAG framework assists in identifying challenges in microservice placement, like preventing the co-location of services with incompatible resource demands or reducing network traffic among interconnected microservices.

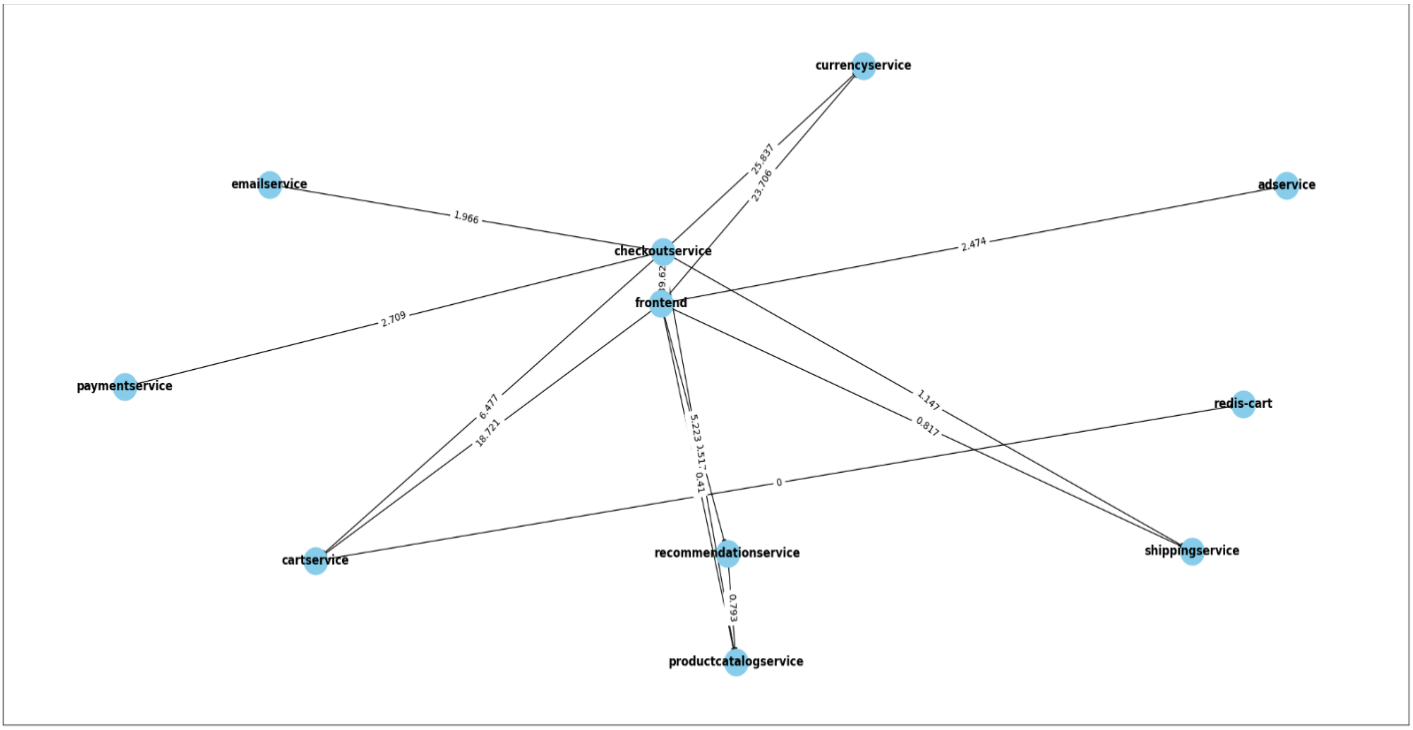
In our research, we utilized the Python library NetworkX [22], specifically designed to aid in network visualization and analysis. This library provides an extensive range of built-in functions and algorithms to address various aspects of network complexity analysis.

To construct the graph of the application, we first identified the connections between microservices using Prometheus. As previously mentioned, LinkerD's sidecar, integrated into each pod, provides numerous metrics, including communication links with other pods. Leveraging this data, we constructed the graph where each microservice represents a node. The nodes are interconnected with directed edges that reflect the communication metrics gathered, thereby mapping out the network of microservice interactions.

In the following Python code example, it can be seen that it is particularly straightforward to create graphs with this tool, using the embedded *DiGraph* function.

```
#Create_directed_graph.py - Example

#Import NetworkX library
import networkx as nx
#Initialize a Directed Graph
G = nx.DiGraph()
#Insert new nodes in the graph
G.add_node("service1")
G.add_node("service2")
#Create edges between nodes
G.add_edge("service1", "service2", weight=5)
```

In the figure beneath, we present a graph that depicts a DAG created using the NetworkX library and the metrics from the *Online Boutique* application.



4.3.2. Placement algorithm

This section details the implementation of our placement algorithm and the outcomes derived from the benchmark metrics acquired during application stress testing.

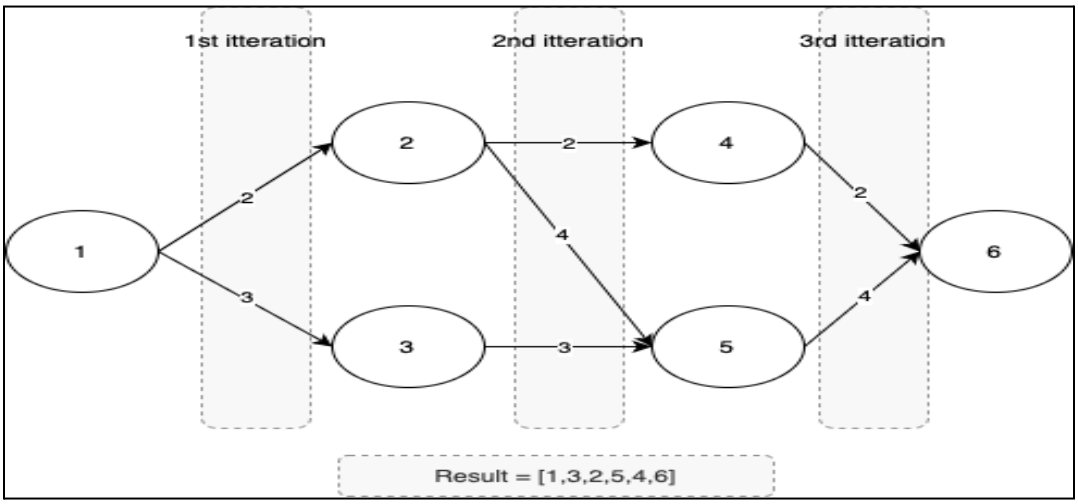
As previously outlined, the two primary steps in generating the algorithm's input data involve collecting crucial metrics from application benchmarking and constructing the Directed Acyclic Graph (DAG) representation of the applications. Utilizing the NetworkX library, we created a graph object that encapsulates all vital information in a single data structure. This facilitates easy access and navigation through the data, leveraging the established connections.

Our algorithm necessitates deploying and operating both applications within a Kubernetes environment, subjected to either real or simulated traffic, to gather accurate metrics. Upon initiating the algorithm, it prompts for user input, offering choices regarding the affinity to be applied between microservices and the available application namespaces. Following the user's input, a series of functions are activated to execute the process.

Subsequently, we introduce the core logic of the algorithm developed for generating a placement proposal, illustrated through pseudocode. As observed, the algorithm is structured into three essential parts, each responsible for a final reliable placement outcome.

```
# Pseudocode - Graph_traversal function

graph_traversal_order(application_graph, client_module):
  start_node = client_module;
  ordered_microservices = []; #Store the nodes during traversal, ordered by affinity weight
  #Initiallize the client_module as the first microservice for placement
  ordered_microservices.append(client_module);
  current_level_itteration.append(client_module);

  while not graph.remaining_nodes() == 0:
     #For each in the graph, starting from the root node, retrieve the successors and order
based on the edge weight. Then remove that node and continue with the rest until the last
node
      for node in current_level_itteration:
         get current_node.successors() && successor not in ordered_microservices;
         sorted_successors = sort_successors_by_edge_weight(ancestor_node);
         ordered_microservices.append(sorted_successors);
         current_level_itteration.append(sorted_successors);
         remove_current_ancestor_from_graph();

#This function returns a list of nodes ordered by the edge weight, during the graph traversal
    return  ordered_microservices;
```

Taking into account that the starting point of the graph is always the client module (UI) of the application, the graph traversal function proceeds by acquiring the successor nodes along with their respective edge weights to execute a descending order sort. In subsequent iterations, the function gathers successors from all nodes of the current level, repeating the sorting process, until every node is incorporated into the sorted list or until the end of the graph is reached.



[Graph traversal node sorting example]

```
# Pseudocode - place microservices to computational nodes

place_microservices(client_module, app.pod_resource_usage,
ordered_microservices):
# Initially a graph is created with nodes being the infrastructure computational nodes
 create_topology_graph();
 infrastructure_quota_availability(100%, 60%, 30%);

#Itterate through the topology graph, starting always from an edge node
 for each node in infrastructure_quota_availability:
   #Itterate throught the ordered microservice list
    for each microservice in ordered_microservices:
      # If the node has available resources, the microservice can be placed
       if (microservice.memory_usage <= node.available_memory
          && microservice.cpu_usage <= node.available_cpu):

          place_microservice_in_node(microservice, node);
       else
          continue;

    select_next_node;

 #The results is a microservice-to-node match
 return microservice_to_node_placement;
```

We have predefined our topology graph's characteristics, such that it can mimic a real resource constrained Cloud-Fog infrastructure. The algorithm applies the placement using this as an input.

```
# Pseudocode - placement algorithm
   topology = nx.DiGraph()
   # Add nodes to the DAG
   topology.add_node('cloud', cpu=16000, ram=64000)
   topology.add_node('fog1', cpu=2000, ram=2000)
   topology.add_node('fog2', cpu=2000, ram=2000)
   topology.add_node('edge1', cpu=1000, ram=1000)
   topology.add_node('edge2', cpu=1000, ram=1000)
   # Add edges to the DAG
   topology.add_edge('edge1', 'fog1')
   topology.add_edge('edge2', 'fog1')
   topology.add_edge('edge1', 'fog2')
   topology.add_edge('edge2', 'fog2')
   topology.add_edge('fog1', 'cloud')
   topology.add_edge('fog2', 'cloud')
```

The placement will be applied three times, each time on an infrastructure with predefined resource availability of 100%, 60% and 30% of the initial availability. These scenarios will provide more data on which affinity performs better under resource constrained environments.

Finally, the following function displays the primary operation of the placement algorithm, outlining the sequence of functions required to produce the final output. The algorithm operates in a continuous loop, quitting only when the user inputs an interrupt signal. Consequently, users have the flexibility to request placement recommendations from the algorithm, selecting different affinities from the available options.

```
# Pseudocode - placement algorithm
placement_algorithm():
    while True:
        # The user inputs details about the wanted affinity and app. namespace
        affinity, namespace = user_input(service_affinity,application_namespace);
        # The client_module is predefined for each application
        client_module = define_client_module(namespace);
        # Collect application metrics using PromQL queries
        metrics = gather_application_metrics(application_namespace);
        # Create the graph using the available metrics
        app_graph = create_microservices_graph(metrics, namespace);
        # Retrieve the placement order according to graph_traversal function
        ordered_microservices = graph_traversal_order(app_graph, client_module);
        # Return the node-to-microservice placement result
        return place_microservices(client_module,
metrics.pod_resource_usage,ordered_microservices);
```

5. Experimental results

This section analyzes the outcomes achieved from implementing the placement algorithm's suggestions, as detailed in the preceding chapter, followed by re-running the benchmark tests for both applications.

In each case, we executed application stress tests using the benchmark setup outlined in Chapter 4.2. This simulates a situation with 150 concurrent users per application, generating diverse requests. Additionally, we employed the 50% and 95% percentile metrics for response latency, as recorded by Locust, to assess the efficacy of each affinity in the placement algorithm.

Additionally, to provide another point of comparison for the placements, we established a scenario where the client module of each application is positioned on the Edge layer, while all other microservices remain in the Cloud. This approach will showcase the improvement in application performance that can be achieved by leveraging the lower layers of the topology for microservice placement.

Executing the placement algorithm using both applications and employing all four affinities, we have compiled the following suggested placements:

Google Online Boutique				
Resource availability	**Affinity**	**Edge**	**Fog**	**Cloud**
100%	**Requests per second**	Frontend, productcatalogservice, currencyservice, cartservice, recommendationservice, shippingservice, checkoutservice, paymentservice	Adservice, emailservice	--
	Average Response Latency	Frontend, productcatalogservice, currencyservice, cartservice, recommendationservice, shippingservice, checkoutservice, paymentservice	Adservice, emailservice	--
	Weighted affinity	Frontend, productcatalogservice, currencyservice, cartservice, recommendationservice, shippingservice, checkoutservice, paymentservice	Adservice, emailservice	--
	Average Throughput	Frontend, productcatalogservice, currencyservice, cartservice, recommendationservice, shippingservice, checkoutservice, paymentservice	Adservice, emailservice	--

In the above scenario, as the resources suffice to host most of the microservices in the Edge layer, there is no difference between each affinity used.
Following the next iteration, resource availability has been updated to 60%.

Google Online Boutique				
Resource availability	**Affinity**	**Edge**	**Fog**	**Cloud**
60%	**Requests per second**	Frontend, productcatalogservice, currencyservice, cartservice, shippingservice	Adservice, emailservice, recommendationservice,checkoutservice, paymentservice	--
	Average Response Latency	Frontend, currencyservice, cartservice, shippingservice, checkoutservice	Adservice, emailservice, recommendationservice,productcatalogservice, paymentservice	--
	Weighted affinity	Frontend, productcatalogservice, currencyservice, cartservice, checkoutservice,	Adservice, emailservice, recommendationservice, paymentservice, shippingservice	--
	Average Throughput	Frontend, productcatalogservice, currencyservice, recommendationservice, shippingservice,	Adservice, emailservice, cartservice, checkoutservice, paymentservice	--

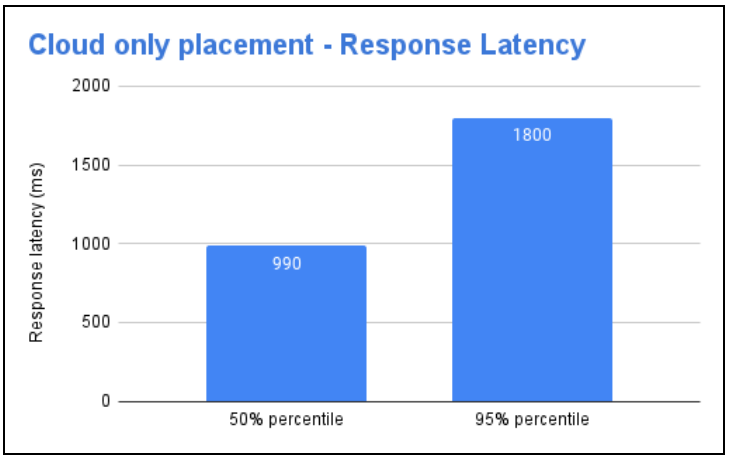
During this scenario, where the infrastructure resource availability is at 60%, we can observe that the algorithm is forced to allocate more microservices in the Fog layer. This will eventually increase the overall response latency of the application.

And finally in the worst-case scenario, where the available resources to deploy our application is at 30%, we can allocate less resources at Edge and Fog layers, increasing the performance costs of the application for the end user, in terms of average response latency.

Google Online Boutique				
Resource availability	**Affinity**	**Edge**	**Fog**	**Cloud**
30%	**Requests per second**	Frontend, productcatalogservice,	currencyservice, cartservice	Adservice, emailservice, recommendationservice ,checkoutservice, paymentservice, shippingservice
	Average Response Latency	Frontend, shippingservice, checkoutservice	currencyservice, cartservice	Adservice, emailservice, recommendationservice ,productcatalogservice, paymentservice
	Weighted affinity	Frontend, checkoutservice, shippingservice	productcatalogservice, currencyservice, paymentservice,	cartservice, Adservice, emailservice, recommendationservice
	Average Throughput	Frontend, productcatalogservice,	currencyservice, recommendationservice	shippingservice Adservice, emailservice, cartservice, checkoutservice, paymentservice

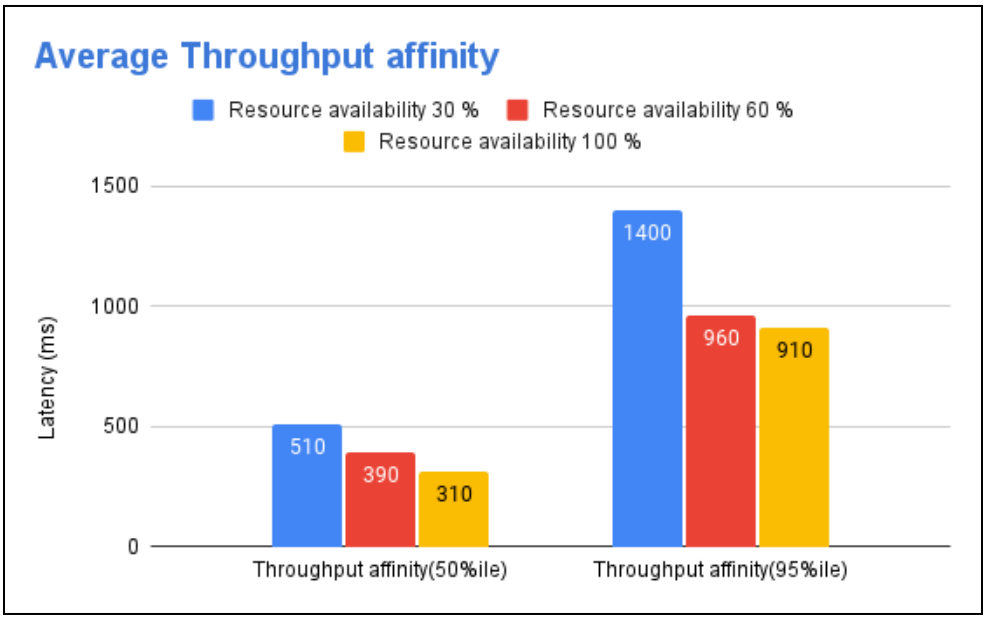
Following, we present the metrics obtained from benchmarking the application deployed using the above placement proposals.

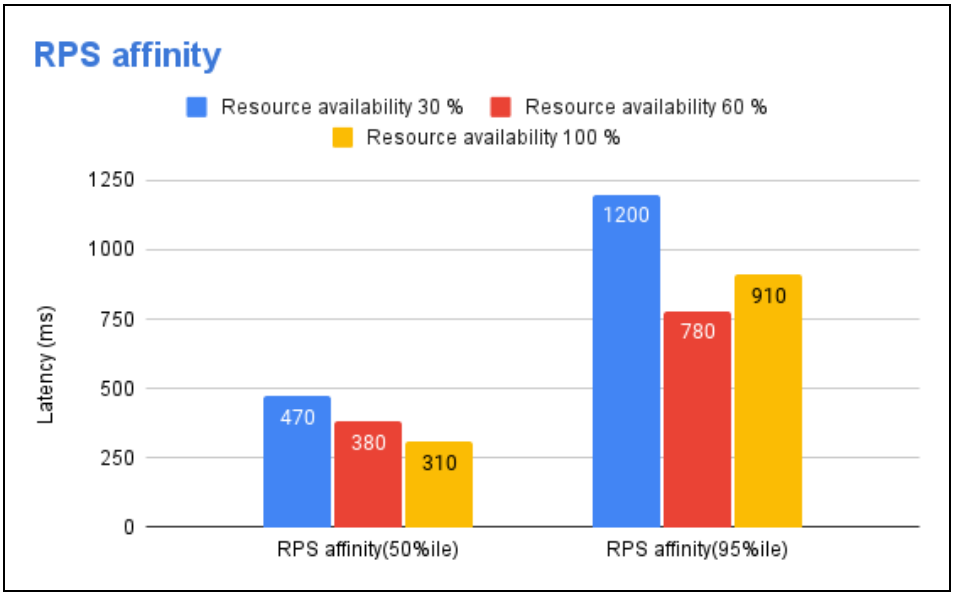
To provide another point of comparison for the placements, we established a scenario where the client module of each application is positioned on the Edge layer, while all other microservices remain in the Cloud. This approach will showcase the improvement in application performance that can be achieved by leveraging the lower layers of the topology for microservice placement.



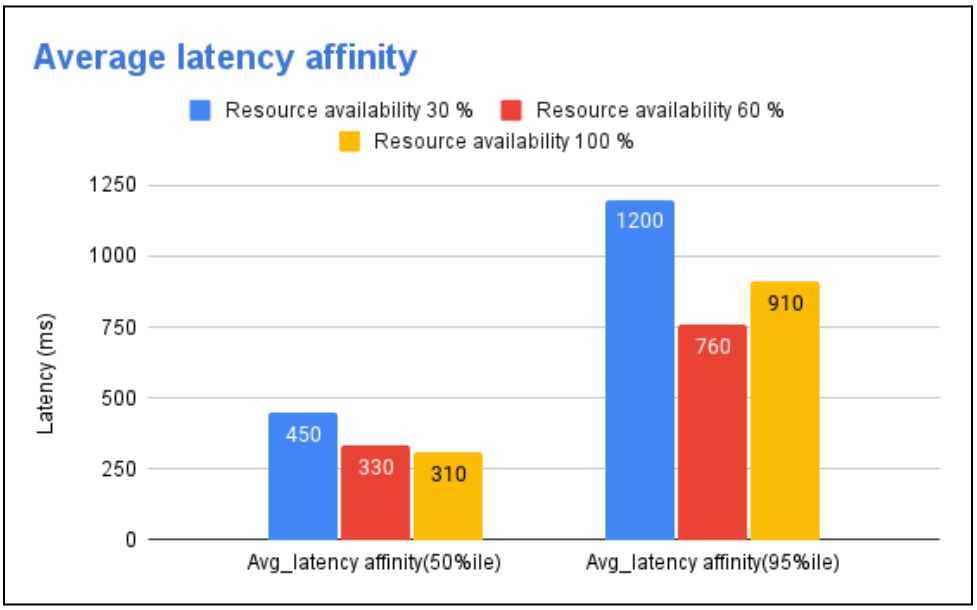
By evaluating the Online Boutique application, we anticipated no significant difference in overall latency across various affinities at 100% resource availability, and this was indeed observed. However, as resource availability decreases, response latency begins to vary.

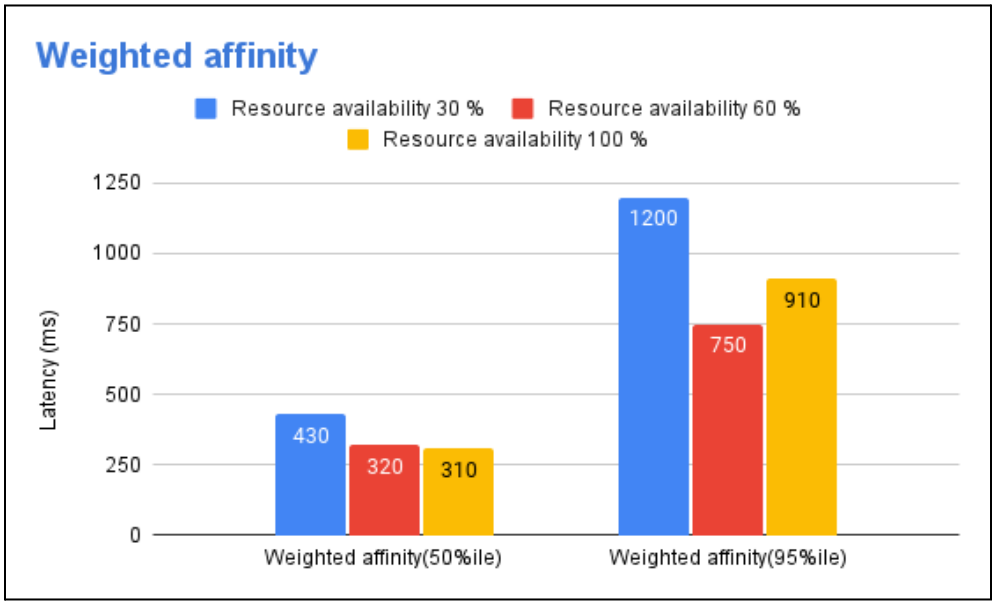
Notably, the Average Throughput affinity exhibits the highest latency when resource availability is reduced to 30%. Conversely, the Weighted affinity demonstrates the lowest latency, closely followed by the Average Latency and RPS affinities. This indicates that a combined approach of these latter two, tends to be the most effective for microservice placement across the topology.





Another significant observation from the two charts above is that at 60% resource availability, employing the Average Throughput affinity results in a reduced median response time (50th percentile) compared to using the RPS affinity. This occurs as the RPS affinity positions the *cartservice* on the Edge layer, in contrast to the Throughput affinity, which favors the *recommendation* service. Based on the benchmark metrics, the cartservice receives more requests per second, has a greater data throughput, and a higher response latency, compared to the recommendation service. This variance in service characteristics accounts for the difference in response times observed.

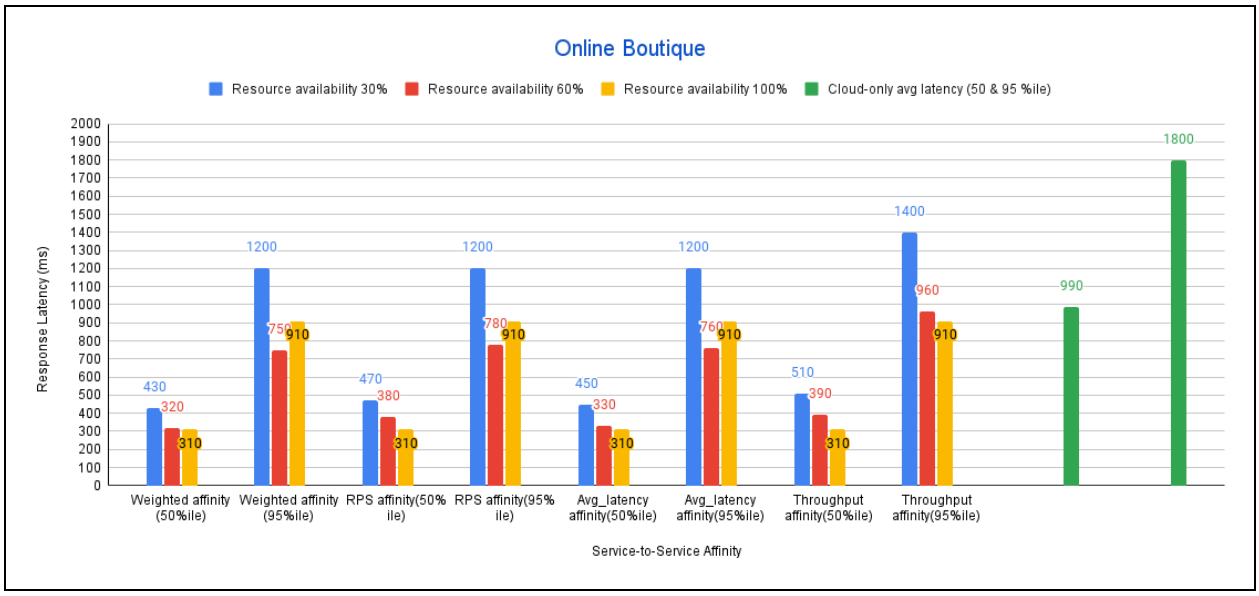




The outcomes align with expectations, given that the algorithm, when employing the Weighted affinity, prioritizes positioning the microservices that exhibit the highest response latency and those receiving the most of requests per second, specifically the *checkout* and *productcatalog* microservices accordingly.

To better assess each scenario, the subsequent chart consolidates all metrics into a single visual representation.

It's evident that shifting workloads closer to the Edge layer substantially lowers response latency, enhancing the end user's experience. However, the optimal strategy for each application is largely dictated by the nature of the traffic it generates, specifically on which services are more frequently used by users. Then, selecting the most suitable affinity based on the available resources in the designated infrastructure can further optimize the user experience with the application.



Furthermore, the results of the placement algorithm for the MartianBank application are outlined below. Given the varied resource demands of its microservices, this application shows distinct placement patterns. Notably, even when 100% resource availability is present, the algorithm utilizes the Cloud node, as the Edge and Fog layers are insufficient to fulfill the resource requirements of all the microservices.

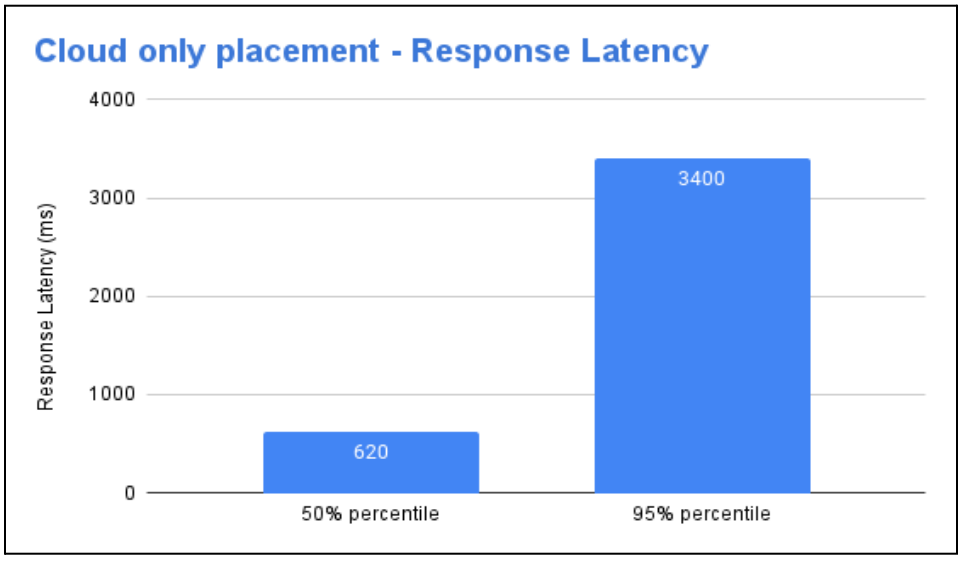
MartianBank				
Resource availability	**Affinity**	**Edge**	**Fog**	**Cloud**
100%	**Requests per second**	dashboard, atm-locator, accounts	customer-auth, transactions	loan
	Average Response Latency	dashboard, atm-locator, accounts	customer-auth, transactions	loan
	Weighted affinity	dashboard, atm-locator, accounts	customer-auth, transactions	loan
	Average Throughput	dashboard, atm-locator, accounts	customer-auth, loan	transactions

MartianBank				
Resource availability	**Affinity**	**Edge**	**Fog**	**Cloud**
60%	**Requests per second**	dashboard, atm-locator, customer-auth	accounts	transactions, loan
	Average Response Latency	dashboard, accounts	atm-locator, customer-auth	transactions, loan
	Weighted affinity	dashboard, accounts	atm-locator, customer-auth	transactions, loan
	Average Throughput	dashboard, atm-locator, customer-auth	accounts	transactions, loan

MartianBank				
Resource availability	**Affinity**	**Edge**	**Fog**	**Cloud**
30%	**Requests per second**	dashboard	atm-locator	Accounts, transactions, loan, customer-auth
	Average Response Latency	dashboard	accounts	transactions, loan, customer-auth, atm-locator
	Weighted affinity	dashboard	accounts	transactions, loan, customer-auth, atm-locator
	Average Throughput	dashboard	atm-locator	Accounts, transactions, loan, customer-auth

Similarly, when the resource availability in the environment decreases to just 30% of its initial capacity, we observe an increasing number of microservices being moved towards the Cloud. This change is anticipated to result in a significant rise in the application's response time for the end user.

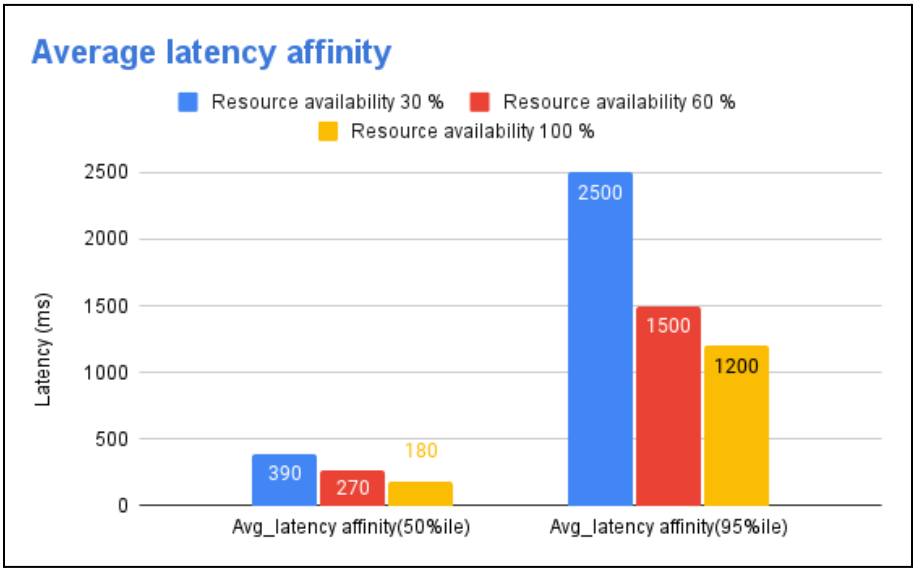
Moving forward to assessing the placement proposals for the MartianBank application, we deployed the microservices using both the algorithm's proposed placements and a baseline deployment with all components in the Cloud, except for the client module, which remained at the Edge.

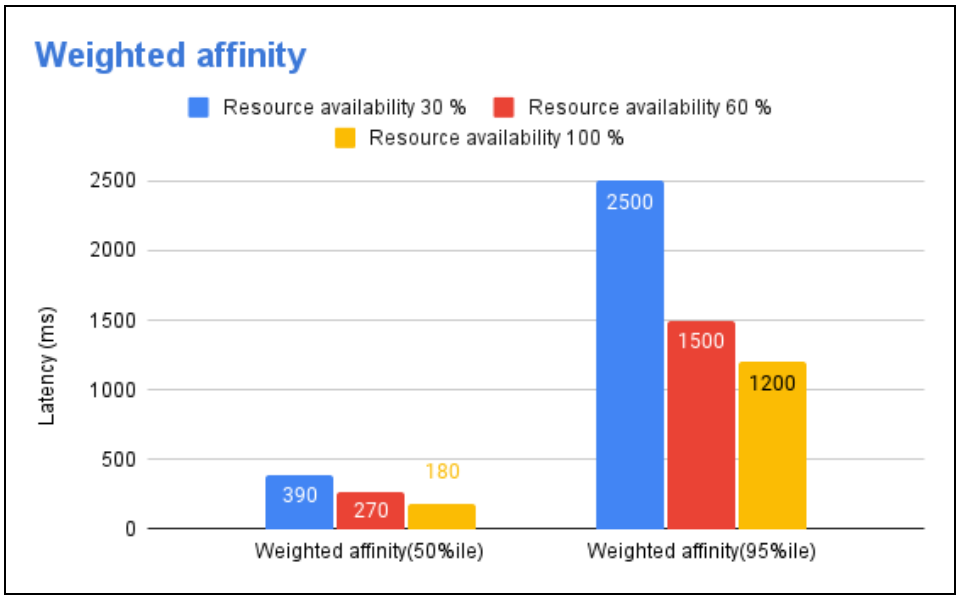


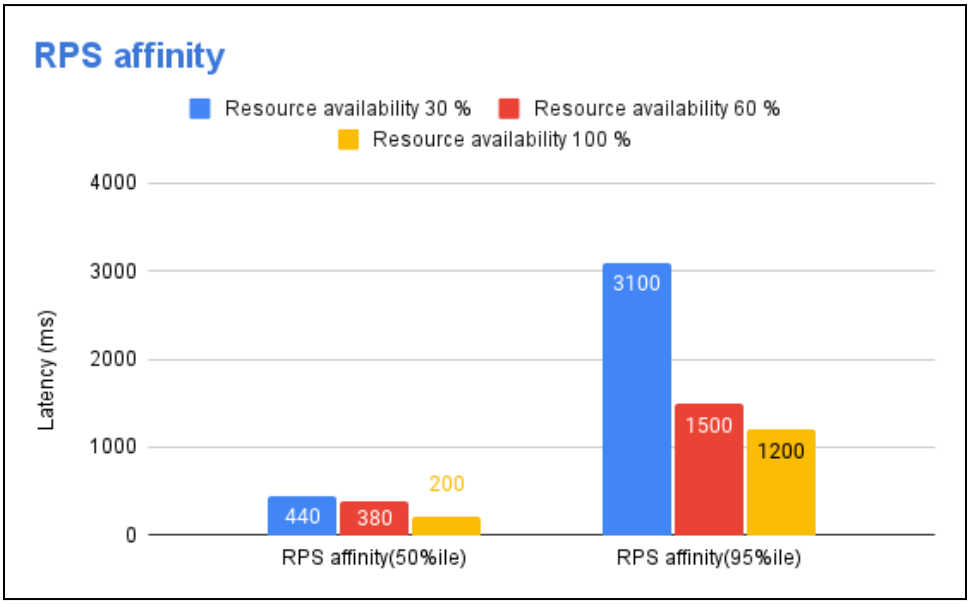
Unlike the previous application, with MartianBank the algorithm engages the Cloud layer even at 100% resource availability. This is due to the Fog-Edge layer's inability to meet the resource demands of the microservices.

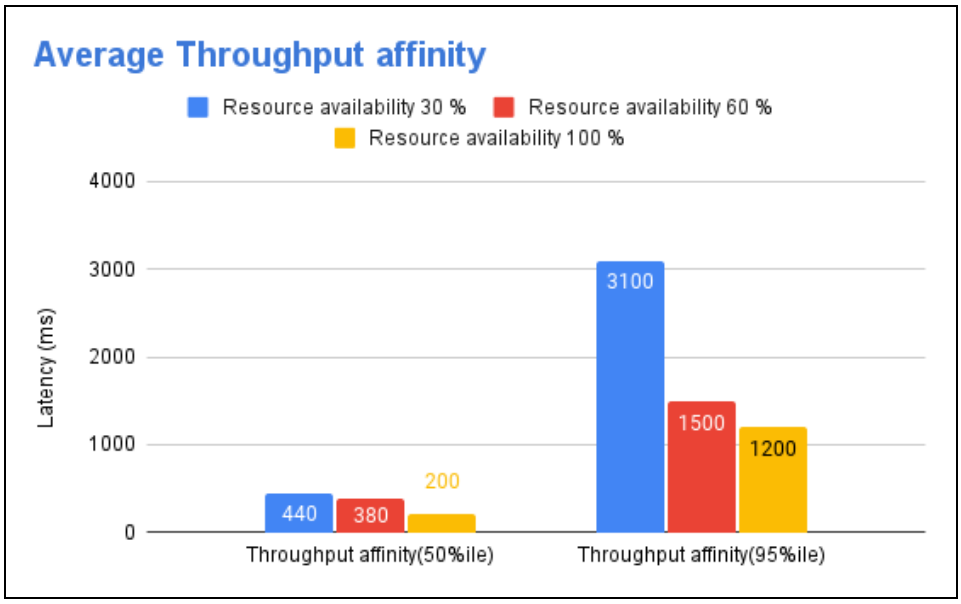
A notable distinction in this scenario is observed between the Average Throughput affinity placement and other placements. The latter tend to favor the *transactions* microservice, allocating it to the Fog layer, whereas the Throughput affinity opts for its placement in the Cloud. Given this service's relatively high response latency, its placement significantly impacts the application's overall latency at the 50th percentile, as seen in the charts below..

In the other scenarios, where resource availability is reduced to 60% and 30%, the placements show similarities, with Weighted and Average Latency affinities aligning closely, as do the RPS and Average Throughput affinities.



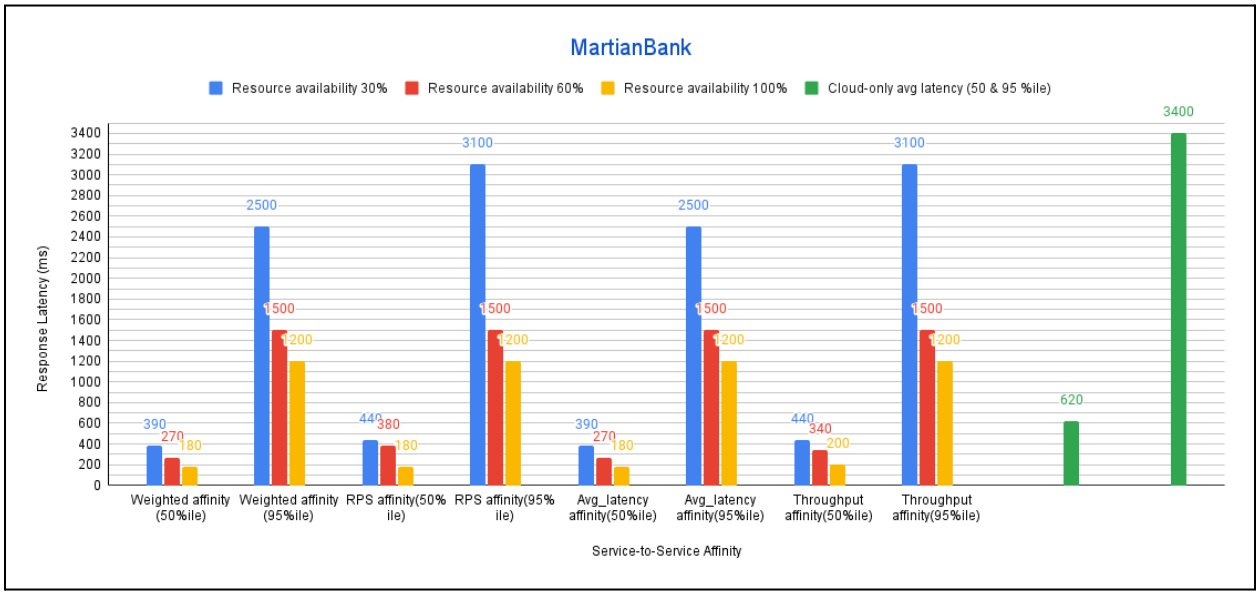

Weighted affinity
Resource availability 30 %
Resource availability 60 %
Resource availability 100 %
Latency (ms)
2500
2000
1500
1000
500
0
390
270
180
2500
1500
1200
Weighted affinity(50%ile)
Weighted affinity(95%ile)


RPS affinity
Resource availability 30 %
Resource availability 60 %
Resource availability 100 %
Latency (ms)
4000
3000
2000
1000
0
440
380
200
3100
1500
1200
RPS affinity(50%ile)
RPS affinity(95%ile)



Comparing to the findings from the Online Boutique application, the Weighted and Average Latency affinities again emerge as the most effective in the placement proposal algorithm. Although resource constraints may not always allow for hosting critical services at the Edge layer, prioritizing services that require more processing time can lead to an overall performance advantage for the application.
The results mentioned above are combined into a single chart for a clearer comparison of the outcomes of each affinity.

This visualization underscores the substantial improvement in application performance for the end user, achieved by transferring some of the workload to the Fog-Edge layers.



6. Conclusion and Future Work

In this concluding chapter, we sum up the journey of this thesis, outlining the addressed problem, our experimental methodology, and the results we achieved.

From the beginning, this research has focused on the intricate task of strategically positioning microservices in a distributed Cloud-Fog environment, aiming to optimize key performance metrics like response latency. We examined static placement scenarios within a topology designed to mirror a real-world setting of multiple Kubernetes clusters spread across various geographical locations.

Our experimental setup included five K3S nodes, a suitable lightweight Kubernetes solution for the resource-restricted Edge computing paradigm. Additionally, we utilized LinkerD, a robust service mesh tool equipped with Multicluster functionality, enabling seamless integration of application components across different clusters. LinkerD provided essential services like reliable discovery, secure communication, and detailed metrics on resource and network utilization.

Leveraging this setup, we put two applications under artificial loads using Locust as our benchmarking tool. The comprehensive metrics obtained from LinkerD were instrumental in the development of our algorithm.

This algorithm, built on the concept that a microservice-based application can be modeled as a Directed Acyclic Graph (DAG), assigned weights to the edges representing various microservice affinities, such as average data throughput, request rate, and response latency. The NetworkX Python library was employed to create the application's graph structure and our topology, with defined resource availability for each node.

By representing both applications and the topology as graphs, we navigated through them to strategically place microservices on each node, following the decision-making policy outlined in Chapter 4, Section 4.3.2.

We then deployed the microservices of each application according to the algorithm's placement suggestions for different resource availability scenarios and subjected them to benchmarking loads to evaluate overall performance from the end user's perspective. These results were also contrasted with a scenario where both applications were hosted in the Cloud, except for the client module on the Edge, highlighting the benefits of utilizing computational nodes near users for specific workloads.

As detailed in the Experimental Results Chapter, each affinity leads to a unique microservice-to-node mapping, emphasizing different services based on the selected affinity. We noted more pronounced differences in placement strategies under resource constraints. Notably, while Requests-per-Second and Average Latency emerged as critical affinities, their combination in the Weighted Affinity offers superior placement decisions for the applications.

Looking forward to the future of microservice placement in distributed Cloud-Fog environments, several ways for further research and development present themselves. Building upon the foundations laid in this thesis, the field can be expanded in multiple directions to enhance the efficiency, reliability, and scalability of microservice architectures in these complex environments.

- Dynamic Placement Strategies: An area for expansion is the exploration of dynamic microservice placement. Unlike the static placements examined in this study, real-world applications often face varying workloads and evolving network conditions. Future research could focus on developing algorithms that adapt in real-time to these fluctuations. By continuously adjusting the allocation of microservices in response to changing demands and network states, these dynamic strategies could maintain or even enhance performance and resource efficiency. Particularly, the possibility of an algorithm to perform on a variable infrastructure would offer a significant advantage. Having the option to add more nodes in the Fog layer K3S cluster, for instance, would allow for offloading more workloads, closer to data load.
  This would represent a significant step towards creating more performant Cloud-Fog computing environments.

- Machine Learning for Placement Decisions: Another promising direction is the integration of machine learning in making informed placement decisions. Utilizing historical data and predictive analytics, machine learning models could identify patterns and trends in application usage and network performance. These insights could then be used to anticipate future states and recommend optimal placement strategies proactively. Such a predictive approach could not only improve current performance but also help in planning and scaling infrastructure to meet anticipated future demands.

In addition to these technical advancements, it's crucial to consider the practical implementation aspects. This includes developing standardized protocols for dynamic placement, ensuring compatibility across different cloud and fog platforms, and addressing potential security implications. The integration of machine learning models also presents challenges in terms of data collection, processing, and model accuracy, which must be carefully addressed to ensure reliable and effective deployment strategies.

Overall, the extension of this research into dynamic placements and machine learning-driven strategies holds great potential. It could lead to significant advancements in the field of distributed computing, making Cloud-Fog environments more adaptive, efficient, and suited to the evolving needs of modern digital infrastructures.

7. Bibliography

[1] Cisco. [n. d.]. ”Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. White Paper. 2016. Available online:”. Fog Computing and the Internet of Things: Extend the ...ResearchGatehttps://www.researchgate.net › post › download

[2] A. Brogi and S. Forti, "QoS-Aware Deployment of IoT Applications Through the Fog," in *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1185-1192, Oct. 2017, https://doi.org/10.1109/JIOT.2017.2701408

[3] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. 2019. “Microservices-based IoT Application Placement within Heterogeneous and Resource Constrained Fog Computing Environments.” In Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'19). Association for Computing Machinery, New York, NY, USA, 71–81. https://doi.org/10.1145/3344341.3368800

[4] K. Velasquez, D. P. Abreu, M. Curado and E. Monteiro, "Service Placement for Latency Reduction in the Fog Using Application Profiles," in IEEE Access, vol. 9, pp. 80821-80834, 2021, https://doi.org/10.1109/ACCESS.2021.3085370

[5] Rajkumar Buyya; Satish Narayana Srirama, "Predictive Analysis to Support Fog Application Deployment," in Fog and Edge Computing: Principles and Paradigms, Wiley, 2019, pp.191-221, doi: 10.1002/9781119525080.ch9. keywords: {Quality of service;Cloud computing;Bandwidth;Hardware;Smart buildings;Internet of Things;Software}, https://github.com/di-unipi-socc/FogTorchPI

[6] Rodríguez, M A., & Buyya, R. (2018, November 11). Container-based cluster orchestration systems: A taxonomy and future directions. https://doi.org/10.1002/spe.2660

[7] Docker Engine https://docs.docker.com/engine/

[8] Kubernetes https://kubernetes.io

[9] K3S Lightweight Kubernetes https://k3s.io

[10] LinkerD Service Mesh https://linkerd.io/what-is-a-service-mesh/

[11] Locust https://locust.io

[12] Google Cloud Platform https://cloud.google.com

[13] Kubernetes Service https://kubernetes.io/docs/concepts/services-networking/service/

[14] LinkerD Traffic Split https://linkerd.io/2.15/features/traffic-split/

[15] tc - Linux manual page: https://man7.org/linux/man-pages/man8/tc.8.html

[16] Online Boutique - Microservice demo application by Google https://github.com/GoogleCloudPlatform/microservices-demo

[17] gRPC protocol: https://grpc.io

[18] Martian Bank - Microservices demo application https://github.com/cisco-open/martian-bank-demo

[19] Introducing MartianBank - a microservice demo application for cloud-native products: https://outshift.cisco.com/blog/martianbank-a-microservice-demo-application-for-cloud-native-products

[20] Traefik Ingress Controller: https://docs.k3s.io/networking

[21] Gupta H, Vahid Dastjerdi A, Ghosh SK, Buyya R. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments.” Softw Pract Exper. 2017;47:1275–1296. https://doi.org/10.1002/spe.2509

[22] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008
https://networkx.org/documentation/stable/index.html