Department of Electrical and Computer Engineering

Intelligent Systems Laboratory

Dynamic Service Placement in Kubernetes with Reinforcement Learning

Diploma Thesis of Vionis Georgios

Examination committee:

- Professor Euripides G.M. Petrakis (Supervisor)
- Associate Professor Vasilis Samoladas
- Assistant Professor Nikolaos Giatrakos

2025

Abstract

This thesis takes a deep dive into the space of Service Placement in Distributed Multi-Clustered Cloud environments. This work aims to reduce the operational cost of the system as well as the latency of using a Highly Scalable Reinforcement Learning model. For this model, the system follows a hierarchical directed acyclic graph structure with multiple tiers. The First tier makes up the central computational and resource infrastructure providing the most potential for scaling. Intermediate tiers are located physically closer to the outer-most tier meaning they have substantially lower propagation delay, they are however lacking when compared to the Central tier regarding compute and scaling capabilities. The aforementioned outside tier represents the final tier consisting of nodes with potentially low computational power and is the server through which clients connect to the system, whether that is IoT devices, smartphones, or any other potential consumer. All applications follow Micro-Service Architectures, meaning they are comprised of a multitude of (micro)services. This allows for moving these services across the tiers as needed to maximize performance and minimize cost and latency. As described earlier the inner-most tiers provide improved performance at the expense of increased latency. Therefore, moving these services to the intermediate or even the outer-most tier can be beneficial both cost wise and in terms of latency if the gain of reduced latency and network I/O outweighs the loss stemming from multiple deployments across many tiers for the same service. The proposed model: DynaQSP, aims to handle the placement and deletion of deployments across each tier utilizing a dynamic Q Reinforcement Learning model while also appropriately managing the traffic flow after each change. To collect metric information and manage functions and networking of applications a Service Mesh technology (Linkerd) will be utilized. DynaQSP collects this information and uses it to train our Reinforcement Learning model which in turn makes decisions regarding the management of all the services. Three realistic microservices-based applications, Google's "Online Boutique", "Bank of Anthos" and "TeaStore", were deployed in the previously described heterogeneous multicluster environment on the Google Cloud Platform with the purpose of evaluating the effectiveness of the proposed model. After considerable implementation-testing-evaluation cycles the model was fine-tuned and the system latency and cost was measured for different load scenarios and compared to default behavior as well as related work. The experimental results demonstrate significant improvement over both default behavior and related work in terms of both reduced cost and latency while ensuring uptime consistency. More Specifically Latency has been reduced by up to 80% and operational costs have been cut by as much as 44%. The employment of realistic applications in a heterogeneous multicluster environment ensures practicality and relevance to the evaluation, confirming the efficacy of the DynaQSP model in real-world cases.

Περίληψη

Η παρούσα διπλωματική εργασία εστιάζει στην περιοχή της Τοποθέτησης Υπηρεσιών σε Κατανεμημένα Πολυ-Συστοιχιακά Περιβάλλοντα Cloud. Στόχος της είναι η μείωση της καθυστέρησης (latency) των υπηρεσιών, καθώς και του λειτουργικού κόστους του συστήματος, μέσω ενός μοντέλου Μηχανικής Μάθησης Ενίσχυσης (Reinforcement Learning) με υψηλή επεκτασιμότητα (highly scalable). Το σύστημα ακολουθεί μια ιεραρχική δομή κατευθυνόμενου ακύκλου γράφου με Βαθμίδες. Η κεντρική βαθμίδα αποτελεί τον κεντρικό υπολογιστικό και πόρο-εντατικό πυρήνα του συστήματος, προσφέροντας τη μεγαλύτερη δυνατότητα κλιμάκωσης. Οι ενδιάμεσες βαθμίδες βρίσκεται σταδιακά ολοένα και πιο κοντά στη βαθμίδα επαφής με τον χρήστη, η οποία είναι και τελευταία, γεγονός που μειώνει σημαντικά την καθυστέρηση διάδοσης, αλλά υστερεί σε σχέση με την κεντική όσον αφορά τις υπολογιστικές και κλιμακούμενες δυνατότητες. Η εξωτερική βαθμίδα αντιπροσωπεύει το τελευταίο επίπεδο και πιθανώς αποτελείται από κόμβους χαμηλής υπολογιστικής ισχύος που λειτουργούν ως διακομιστές μέσω των οποίων οι χρήστες συνδέονται στο σύστημα, είτε πρόκειται για συσκευές IoT, smartphones, υπολογιστές ή οποιονδήποτε άλλο καταναλωτή υπηρεσιών. Όλες οι εφαρμογές ακολουθούν αρχιτεκτονικές Micro-Service, που σημαίνει ότι αποτελούνται από πολλαπλές (μικρο)υπηρεσίες. Αυτό επιτρέπει τη μεταφορά αυτών των υπηρεσιών μεταξύ των επιπέδων, όπως απαιτείται, για να μεγιστοποιηθεί η απόδοση και να ελαχιστοποιηθούν το κόστος και η καθυστέρηση. Όπως περιγράφηκε προηγουμένως, τα ανώτερα επίπεδα προσφέρουν βελτιωμένη απόδοση εις βάρος αυξημένης καθυστέρησης. Συνεπώς, η μετακίνηση αυτών των υπηρεσιών πιο κοντά στην εξωτερική βαθμίδα μπορεί να είναι ωφέλιμη τόσο από πλευράς κόστους όσο και καθυστέρησης, εάν το κέρδος από τη μείωση της καθυστέρησης και του δικτυακού I/O υπερβαίνει τις απώλειες από πολλαπλές αναπτύξεις στις διάφορες βαθμίδες για την ίδια υπηρεσία. Το προτεινόμενο μοντέλο: DynaQSP, στοχεύει στη διαχείριση της τοποθέτησης και διαγραφής αναπτύξεων σε κάθε επίπεδο χρησιμοποιώντας ένα δυναμικό μοντέλο Ενίσχυσης Q, ενώ παράλληλα διαχειρίζεται κατάλληλα τη ροή κίνησης μετά από κάθε αλλαγή. Για τη συλλογή πληροφοριών μεταβλητών, τη διαχείριση λειτουργιών και τη δικτύωση των εφαρμογών θα χρησιμοποιηθεί τεχνολογία Service Mesh (Linkerd). Το DynaQSP συλλέγει αυτές τις πληροφορίες και τις χρησιμοποιεί για την εκπαίδευση του μοντέλου Reinforcement Learning, το οποίο με τη σειρά του λαμβάνει αποφάσεις για τη διαχείριση όλων των υπηρεσιών. Τρεις ρεαλιστικές εφαρμογές βασισμένες σε μικρο-υπηρεσίες, το “Online Boutique” της Google, το “Bank of Anthos” και το “TeaStore”, εγκαταστήθηκαν στο προαναφερθέν ετερογενές πολυ-συστοιχιακό περιβάλλον στο Google Cloud Platform με σκοπό την αξιολόγηση της αποτελεσματικότητας του προτεινόμενου μοντέλου. Μετά από αρκετούς κύκλους υλοποίησης, δοκιμών και αξιολόγησης, το μοντέλο βελτιστοποιήθηκε και η καθυστέρηση και το κόστος του συστήματος μετρήθηκαν για διαφορετικά σενάρια φόρτου και συγκρίθηκαν με την προεπιλεγμένη συμπεριφορά και με σχετικές έρευνες. Τα πειραματικά αποτελέσματα δείχνουν σημαντική βελτίωση τόσο στο κόστος όσο και στην καθυστέρηση σε σχέση με την προεπιλεγμένη συμπεριφορά και τις σχετικές έρευνες, διασφαλίζοντας παράλληλα τη συνέπεια του uptime. Πιο συγκεκριμένα τα πειραματικά αποτελέσματα εμφανίζουν ελάττωση της καθυστέρησης του συστήματος έως και κατά 80% και του κόστους λειτουργίας κατά 44%. Η χρήση ρεαλιστικών εφαρμογών σε ένα ετερογενές πολυ-συστοιχιακό περιβάλλον εξασφαλίζει την πρακτικότητα και τη συνάφεια της αξιολόγησης, επιβεβαιώνοντας την αποτελεσματικότητα του μοντέλου DynaQSP σε πραγματικές περιπτώσεις.

Acknowledgements

I am deeply grateful for the continuous support and insightful guidance provided by my supervisor, Professor Euripides Petrakis, as well as the Intelligent Systems Laboratory team. Throughout this endeavor they have been there in times of need to assist with any obstacles I could not overcome. Their suggestions and overall guidance have helped shape this work into a more complete version.

I also must express my sincere gratitude to my colleagues (and friends) who have always been there for me, making this journey an enjoyable experience. Without them I would not have been able to see this to this end. Their advice and company are something I will forever be grateful for.

I dedicate this work to my friends and family for their unwavering support thought this journey.

Contents

Table of Figures

Table Of Algorithms

Table of Charts

Table of Tables

1 Introduction

1.1 Problem Definition

In recent years there has been a massive spike in the industry of systems moving from In-House to Cloud environments. This change has led to a big investment in finding ways of effectively and efficiently utilizing the benefits of these new technologies. As each providers' capabilities improve, so does the need for researching methods and practices through which said capabilities can be utilized while at the same time resolving any potentially undesired derivatives of the new platforms. In essence, research is called to find ways of reaping the full benefits such as flexibility, integrity and scalability while also avoiding the main concern which is higher operation costs.

A Cloud environment is mainly accessed through a Cloud Provider also known as Infrastructure Provider, such as Amazon Web Services, Google Cloud Platform, Microsoft Azure. The Cloud is essentially a diverse set of resources such as Virtual Machines, Containers, databases and serverless functions that can be easily provisioned and managed through APIs or Web Interfaces. This allows clients to allocate more of their resources to their actual system rather than wasting time and money on infrastructure management.

There is however a caveat, that being clients having second thoughts about migrating to Cloud environments. This stems from a fear of well-known issues such as vendor lock-in and increased costs. In recent years the three main providers have veered away from enforcing provider specific rules and have been heading towards a standardization of the Cloud industry. As a result, the main remaining issue is skepticism and apprehension due to potentially increased costs. This practically forces Cloud Providers to engineer new tools, methods and mechanisms to reliably reduce operation costs for any potential customer.

This leads to the more modern hierarchical approach to the distribution of clusters as the industry moves away from the initial monolithic Cloud structure and into more dynamic variations of architectures. This work proposes the Tier Based Directed Acyclic Graph Architecture whose methodology was described in the prologue. A similar approach is utilized by the well-known Cloud – Fog – Edge architecture. Figure 1.1 provides a concise but insightful representation of the difference between the more traditional Cloud – Fog – Edge and the proposed Tier Based Graph infrastructures and the behavior between each layer as well as the relationship of the components within a layer.

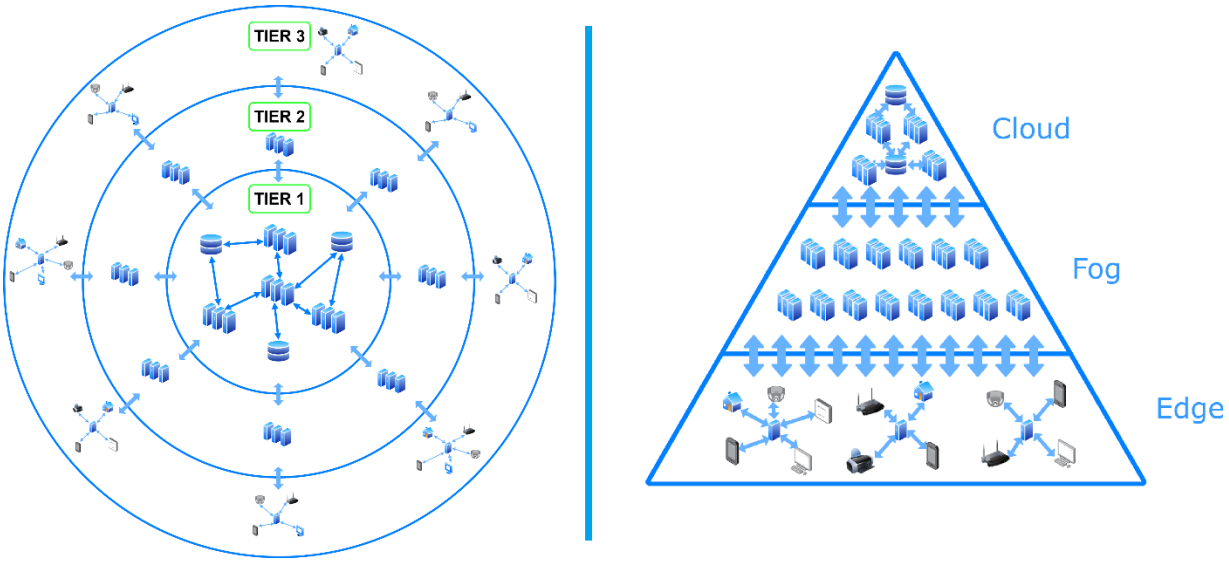

Figure 1.1 Infrastructure's Models

Along with the currently mainstream approach of centralized Cloud computing, a particular set of new challenges emerge. These issues stem primarily from the specific structuring of the existing hierarchy. In comparison the suggested Tier Based directed acyclic graph offers inherent benefits deriving from its more distributed approach.

The Central tier effectively represents Cloud computing and is essentially a centralized array of compute machines and databases that allows for processing and storing vast quantities of data. On the other hand, its centralized nature has a few downsides, such as reduced responsiveness, bandwidth limitations, and increased network latency.

Computing in the intermediate tiers aims to battle these ramifications by employing a more distributed network of nodes. By moving relatively powerful compute nodes closer to the client, higher responsiveness is achieved. The primary benefit of Intermediate Computing however is the vastly superior bandwidth allowing for greater coverage along an ever-increasing number of users. These effects can be further enhanced by combining this approach with Computing on the outer-most tier.

The outer most layer aims to be the interface of consumers with the network and as such is comprised of, a larger network of smaller compute machines that aim to reduce latency. Large amount of compute is not feasible on these nodes, serving primarily to collect and aggregate incoming requests. This paradigm achieves near perfect latency and responsiveness, while also offering enhanced privacy and security. Furthermore, efficient algorithms can utilize the small computational capability to offload work from nodes higher in the hierarchy, effectively reducing network cost and latency.

1.2 Scope of Thesis

This thesis aims to create a dynamic service placement strategy for services utilizing a Reinforcement learning model in order to reduce latency and cost while maintaining identical performance of all applications deployed on a tier based distributed directed acyclic graph infrastructure. The environment consists of multiple Kubernetes clusters, the CLI of which allows for easy management of each cluster. The proposed model's strategy focuses on reducing latency and cost through dynamic allocation of services across various nodes in the multi-clustered Kubernetes network. The implementation process involved the creation of a Reinforcement Learning model that in turn utilizes specialized tools for managing and monitoring the services within the distributed environment. Said model will consequently be integrated within the previously described multi-clustered distributed cloud computing infrastructure to provide a complete solution to the aforementioned issues.

With the purpose of addressing the challenges of the service placement problem in distributed architectures, this thesis proposes an innovative solution in the form of an adapted application called DynaQSP. DynaQSP is essentially a Reinforcement Learning model that utilizes state data gathered through Prometheus API in order to train itself, while adapting to changes in environment machine costs, network costs and network load, and in turn make decisions. More specifically the model takes into account metrics such as RAM and CPU usage, response latency, and network throughput. Utilizing these metrics appropriately, it aims to optimize its decision-making to maximize overall efficiency of the system architecture.

The general objective of this work is to revolutionize service placement by enabling hassle free efficient and effective management without the need to manually edit your service placement plan for any distributed multi cluster Micro-Service application. This will be done not only based on the state of each individual service but also based on the dependencies of said service to other services in the same or other clusters. This is achieved by considering the egress network throughput, and positively rewarding any reduction in said throughput derived from the model's decisions. For example, in a scenario where there is a deployment on the Central tier that gets mirrored all the way to the outer-most tier, the model will weigh the cost of adding an additional deployment on any of the intermediate of outer tiers against the reduction in cost from the avoided Egress network throughput and latency and make the most optimal decision. Through the creation of the DynaQSP model, this thesis strives to advance the broader field of cloud computing and service placement strategies for Micro-Service Architectures in distributed multi-cluster environments. The final goal is a solution that is customizable to the needs of diverse infrastructure frameworks and applications, reducing time spent setting up the system while also providing efficient automation of the service placement decision-making process. The significant benefit of this work is the fact that it does not require specifically architectures with a specific amount of levels and can, in fact, support any predefined directed acyclic hierarchical structure. We will however be using a Tier 1 – Tier 2 – Tier 3 environment for the purpose of this thesis to elucidate the inner workings of the DynaQSP model by taking advantage of its status as a more widely known architecture.

2 Background and related work

In this section, we provide an overview of the theoretical background that forms the foundation of this thesis. We delve into the underlying concepts and principles of the implementation of the model's service placement strategy. Additionally, we discuss the metric tools and agents that will be utilized and modified to support the implementation of these strategies.

2.1 Related Work

Service placement in distributed systems has attracted considerable attention from both academic and industrial communities, as it plays a critical role in optimizing how services are allocated. Over the years, a variety of algorithms and strategies have been proposed to enhance service placement decisions. In particular, research on applying Reinforcement Learning to service placement in Distributed Multi-cluster Cloud environments has yielded significant contributions, as demonstrated by the studies in [1], [2] and [3].These works provide a comprehensive overview of the field, highlighting diverse approaches and methodologies designed to address the challenges of effective service placement.

The most important mention should be given to the work by my colleague which set the foundations upon which this work was built. The goal of the previous work as seen in [4] was to reduce latency by implementing specific algorithms, including LFU and LRU adaptations that utilized resource usage and latency metrics to aid in decision-making. Similarly to this work again, [1] aimed to reduce latency by implementing a Q learning agent with sole purpose of reducing latency by monitoring the CPU usage of the Cluster.

Reinforcement Learning in dynamic environments can be quite complex at times and particularly so with distributed environments, especially when the outcome is non-deterministic and its stochastic process is unknown. Different implementations tend to focus on enhancing one or two aspects, often sacrificing performance in other areas.

The work seen in [5] proposes a distributed RL-based approach for dynamic service offloading in Fog Computing. The system optimizes execution time while ensuring QoS for IoT applications. It uses Asynchronous Proximal Policy Optimization [6] with multiple actor-learners running in parallel across fog servers to make adaptive service placement decisions. It leverages PPO clipping and V-trace techniques to improve convergence speed. The model optimizes scalability and latency compared to traditional centralized methods. Much like this work, [7] also utilizes PPO in comparison with A2C [8] in an attempt to reduce energy consumption and improve service availability.

Another similarly adaptive system is proposed in [9], namely FogReinForce, a Deep Q-Network based RL framework for value-driven service placement in fog computing. It prioritizes services based on Value-of-Information, ensuring that the most critical data is processed first. Unlike cost or latency-focused methods this approach maximizes end-user utility by dynamically adjusting placements based on real-time conditions. This framework is centralized and learns optimal placement policies through trial and error exploration much like this work does.

Another centralized implementation is utilized by [10]. Here a Deep Reinforcement Learning based dynamic service placement framework for Mobile Edge Computing is introduced. It aims to minimize service latency by optimizing service placement under user mobility constraints while considering cost efficiency. The model integrates Mixed-Integer Linear Programming [11] for structured optimization and a migration conflict resolution mechanism to avoid unnecessary service allocations.

In contrast, [12] employs a distributed approach that leverages neural networks to adapt to temporal patterns, focusing on Cloud, Fog, and Edge Computing environments with the goal of reducing execution costs. It utilizes the IMPALA framework (Importance Weighted Actor-Learner Architecture [13]) to train a Deep Reinforcement Learning strategy that efficiently manages Directed Acyclic Graph IoT applications by using parallel actor learners. These learners generate diverse experience trajectories, which enhance both convergence speed and adaptability.

Work	Infrastructure	Placement Approach	Platform	Application	Placement
[2]	Cloud-Fog-Edge	distributed	Simulation	Simulation	dynamic-RL
[4]	Cloud-Fog-Edge	distributed	K3s	iXen,Boutique	dynamic
[5]	Fog-Edge	distributed	Simulation DRL	Simulation	dynamic-RL
[9]	Fog	centralized	Simulation FogReinForce	Simulation	dynamic-RL
[10]	Edge	centralized	Simulation DSP-DRL	Simulation	dynamic-RL
[12]	Cloud-Fog-Edge	distributed	Simulation IMPALA	Simulation	dynamic-RL
[14]	Cloud-Edge	centralized	Simulation	Simulation	dynamic-RL
This Work	Tier Based Directed Acyclic	distributed	K3s	Boutique, Bank Of Anthos, Teastore	dynamic-RL

Table 2.1 Related Work Comparison Table

In this thesis, we focus on enhancing service placement performance and load balancing within a distributed multicluster Tier based environment. To achieve this, we utilize a Kubernetes distribution as our microservices orchestration platform. While incorporating Reinforcement Learning into such environments has become common, we tailor its capabilities to meet our specific needs and integrate it effectively into our system.

This work builds on the approach described in [4], which applied LFU and LRU principles for service placement, using it as a benchmark to evaluate the efficiency and effectiveness of our proposed algorithm. Our strategy for load balancing involves distributing incoming requests evenly across multiple clusters to prevent any single service from becoming overwhelmed and to ensure a balanced workload.

Unlike previous studies that predominantly used synthetic environments, we assess our methods using realistic, microservices-based applications, thereby providing more practical and insightful performance data. Ultimately, our goal is to reduce response times, maintain high Quality of Service (QoS) for end users, and deliver an enhanced user experience.

2.2 Infrastructure and Tools

2.2.1 Reinforcement Learning

Reinforcement Learning (RL) [15] is a branch of machine learning focused on training agents to make a sequence of decisions by interacting with an environment. The agent learns to achieve a specific goal by maximizing cumulative rewards obtained through its actions. Unlike supervised learning, where the model learns from a labeled dataset, reinforcement learning relies on trial-and-error exploration and feedback from the environment to refine its decision-making capabilities. This feedback is provided in the form of rewards or penalties based on the outcomes of the agent's actions.

The foundation of reinforcement learning lies in its ability to handle dynamic environments and problems where the optimal solution is not known in advance. It is particularly effective for tasks requiring sequential decision-making, where the consequences of actions are not immediately apparent and may affect future outcomes. Applications of reinforcement learning span diverse fields such as robotics, game playing, resource management, traffic control, and autonomous systems.

Core Principles of Reinforcement Learning

- **Agent**: The decision-maker that interacts with the environment.
- **Environment**: The external system with which the agent interacts, providing feedback in response to the agent's actions.
- **State**: A representation of the environment at a specific point in time.
- **Action**: A decision made by the agent that affects the state of the environment.
- **Reward**: A scalar feedback signal received by the agent for each action, indicating the immediate benefit or cost of that action.
- **Policy**: A strategy that maps states to actions, guiding the agent's behavior.
- **Value Function**: An estimate of the expected cumulative reward for a state or state-action pair, guiding the agent toward more rewarding outcomes.
- **Exploration vs. Exploitation**: The trade-off between trying new actions to discover their potential rewards (exploration) and leveraging known actions to maximize immediate rewards (exploitation).

In reinforcement learning, the agent starts with little or no knowledge about the environment and explores possible actions to learn which ones yield the most favorable outcomes. Through iterative interactions, the agent refines its policy based on the rewards and penalties received, eventually converging toward an optimal strategy.

The mathematical framework of reinforcement learning is often modeled as a Markov Decision Process (MDP) [16]. In this framework, the environment is represented as a set of states, actions, transition probabilities, and rewards. At each step, the agent observes the current state, selects an action, transitions to a new state, and receives a reward. The goal is to learn a policy that maximizes the cumulative reward over time, also known as the return.

Two main approaches to reinforcement learning are value-based methods and policy-based methods. Value-based methods, such as Q-learning, focus on learning the value function to evaluate the quality of state-action pairs. In contrast, policy-based methods directly optimize the policy by adjusting its parameters to maximize the expected reward. Hybrid approaches, such as Actor-Critic methods, combine the strengths of both techniques.

One of the significant challenges in reinforcement learning is the balance between exploration and exploitation. While exploration is essential to discover new strategies and learn about the environment, excessive exploration can delay convergence. On the other hand, focusing too much on exploitation may cause the agent to settle for suboptimal strategies. Techniques such as epsilon-greedy [17] and softmax [18] policies are often employed to manage this trade-off effectively.

Reinforcement learning has gained prominence due to its success in solving complex problems that are difficult to model or optimize using traditional methods. For instance, RL-powered agents have achieved human-level performance in games like chess, Go, and video games. Beyond games, RL is widely applied in areas such as autonomous vehicle navigation, resource allocation in data centers, dynamic pricing, and personalized recommendations.

2.2.1.1 Q Learning

Q-Learning [19] is a model-free reinforcement learning algorithm designed to help an agent learn an optimal policy for decision-making in a Markov Decision Process (MDP). By using trial-and-error interactions with the environment, the agent discovers the best actions to take in various states to maximize cumulative rewards over time. The algorithm is particularly powerful because it does not require prior knowledge of the environment's dynamics (such as transition probabilities or reward distributions). Instead, it relies solely on observing the rewards and state transitions that result from its actions.

At its core, Q-Learning revolves around a **Q-value** or **action-value function**, which estimates the expected cumulative reward of taking a specific action in a given state and then following the optimal policy thereafter. The algorithm iteratively updates these Q-values to refine the agent's understanding of the environment.

Core Principles of Q-Learning

- **States (S)**: Represent the set of all possible situations the agent can encounter.
- **Actions (A)**: Represent the choices available to the agent in each state.
- **Rewards (R)**: Provide feedback to the agent, indicating the immediate benefit or cost of an action.
- **Q-Table**: A table where each entry corresponds to a state-action pair and holds the Q-value (expected reward) for that pair.
- **Learning Rate (α)**: Controls how much weight is given to new information when updating Q-values.
- **Discount Factor (γ)**: Determines how much importance is placed on future rewards compared to immediate rewards.

- **Exploration vs. Exploitation**: Balances the need to try new actions to discover their potential (exploration) against using known actions to maximize rewards (exploitation).

The essence of Q-Learning lies in the **Q-value update rule**, which adjusts the value of a state-action pair based on observed rewards and the agent's estimate of future rewards. The update equation is given by:

$$Q^{new}(s,a) = (1-\alpha)Q^{cur}(s,a) + a(reward + \gamma Q(s',\alpha))$$

Here, the term Q(s,a) represents the current Q-value for the state-action pair.

The update rule ensures that Q-values are adjusted iteratively, moving closer to the true values that reflect the long-term reward potential of each state-action pair. Q-Learning starts with an initialized Q-table (often with arbitrary values). The agent then explores the environment by taking actions, observing rewards, and updating the Q-table based on its experiences. Over time, as the Q-values converge, the agent forms a policy that maps states to optimal actions.

Q-Learning is widely used in reinforcement learning because of its flexibility and simplicity. It can handle environments where the agent's actions do not always lead to deterministic outcomes, making it robust in stochastic settings. Moreover, Q-Learning can learn optimal policies without requiring a model of the environment, which is why it is categorized as a **model-free** algorithm.

2.2.2 Micro-Service Architecture

Microservices Architecture [20] is a software design approach where applications are built as a collection of small, autonomous services that communicate with each other over well-defined APIs. This architecture is a departure from traditional monolithic systems, where all components are tightly integrated into a single codebase. In a microservices-based system, each service is designed to handle a specific business capability, functioning independently and being loosely coupled with other services.

Microservices are particularly suited for large, complex applications that require scalability, agility, and frequent updates. By breaking down a system into smaller, manageable components, teams can develop, deploy, and scale each service independently. This results in a more adaptable and resilient system that can evolve alongside business requirements.

A defining characteristic of microservices is their decentralized nature. Unlike monolithic systems, where a single database might be shared among all components, microservices often adopt a "polyglot persistence" approach, where each service manages its own data. This autonomy allows services to use the most appropriate tools and technologies for their specific tasks, further enhancing flexibility.

Core Principles of Microservices Architecture

- **Single Responsibility**: Each service is designed to fulfill a single business capability, such as user authentication, order processing, or payment handling.
- **Autonomy**: Services are self-contained and operate independently. They can be developed, deployed, and scaled without impacting others.
- **Decentralized Data Management**: Each service owns its database, promoting autonomy and reducing dependencies.
- **Lightweight Communication**: Services interact using lightweight protocols, typically RESTful APIs, gRPC, or messaging queues.
- **Independently Deployable**: Services can be updated or replaced without redeploying the entire system.
- **Scalability**: Each service can be scaled independently based on its unique load requirements.
- **Resilience**: Systems are designed with fault tolerance in mind, ensuring that failures in one service do not cascade across the entire application.

In a microservices-based architecture, each service is deployed independently, often in containers or virtual machines. Tools like Docker and Kubernetes have become instrumental in managing microservices deployments by offering capabilities such as container orchestration, load balancing, and fault recovery. This infrastructure ensures that microservices are highly portable, enabling seamless deployment across various environments, from on-premises data centers to cloud platforms.

One of the critical benefits of microservices architecture is its ability to scale horizontally. Services experiencing increased demand can be scaled independently without affecting other parts of the system. For instance, if an e-commerce application faces high traffic during a sale, services handling payments or inventory can be scaled independently to accommodate the surge, leaving other services unaffected.

However, while microservices offer numerous advantages, they also introduce complexities. The distributed nature of microservices requires robust inter-service communication, monitoring, and error handling. Failures in a single service can lead to cascading effects if not managed correctly. The Model proposed in this article aims to aid in combatting such issues by providing automated management for service placement and network traffic splitting.

2.2.3 K3s

K3s [21] is a lightweight, certified Kubernetes [22] distribution created by Rancher Labs, specifically designed for resource-constrained environments, weak computing, and small-scale deployments. It simplifies the complexities of Kubernetes while retaining compatibility with its ecosystem. K3s achieves

this by removing non-essential components, streamlining installation, and reducing its resource footprint, making it a powerful choice for smaller scale work, development, and testing environments. Despite its lightweight nature, K3s adheres to the Kubernetes Certified Conformance Program, ensuring it supports standard Kubernetes APIs and tools.

Core Principles of K3s:

- **Lightweight:** Removes non-essential features, consolidates components, and optimizes for minimal resource consumption.
- **Simplified:** Reduces complexity in installation and operation with a single binary and minimal setup steps.
- **Portable:** Supports diverse hardware, including ARM devices and Client-focused architectures.
- **Compatible:** Maintains full Kubernetes API compatibility, enabling use with standard tools, Helm charts, and other Kubernetes resources.
- **Resilient:** Offers features like embedded HA setups and lightweight storage options to enhance reliability in various deployment scenarios.

2.2.3.1 Architecture of K3s

The architecture of K3s simplifies the traditional Kubernetes model while maintaining its core functionality. The control plane in K3s consolidates essential Kubernetes components, including the API server, scheduler, and controller manager, into a single lightweight binary. This streamlining reduces memory and CPU usage while retaining the full capabilities of Kubernetes. By default, K3s uses SQLite as its datastore, which is suitable for single-node or small clusters. For larger, high-availability configurations, external databases such as MySQL or PostgreSQL can be used to provide enhanced scalability and reliability.

Worker nodes in K3s use containerd, a lightweight container runtime, instead of Docker. This decision further reduces resource consumption without compromising container management capabilities. Networking and ingress are simplified through built-in add-ons, such as Flannel for networking and Traefik for ingress, which are pre-integrated and ready to use out of the box. This eliminates much of the manual configuration typically required in a Kubernetes deployment. The architecture is particularly well-suited for Tier 3 scenarios where devices may have limited connectivity and computing power.

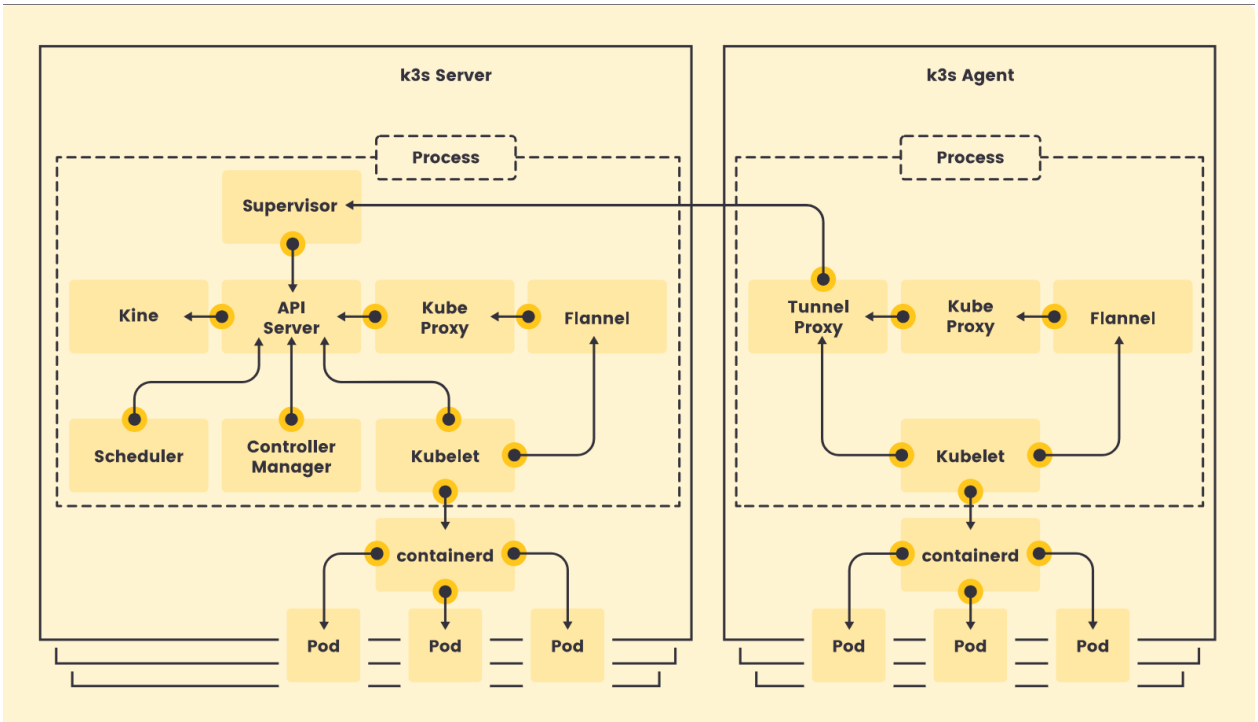

Figure 2.1 K3s Structure

2.2.3.2 Use Cases for K3s

K3s is particularly valuable in environments where traditional Kubernetes deployments would be too resource-intensive or complex. In Tier 3 computing scenarios, K3s enables efficient management of workloads on resource-constrained devices such as IoT gateways or industrial controllers. Its support for ARM architectures makes it ideal for deploying containerized applications on devices like Raspberry Pi, powering smart devices, or enabling IoT networks. For development and testing, K3s provides a lightweight Kubernetes environment that can be quickly installed on local machines or virtual machines, allowing developers to experiment with containerized applications without the overhead of a full Kubernetes cluster.

Small-scale production environments also benefit from K3s, where its minimal infrastructure requirements make it a cost-effective solution for teams deploying containerized applications. Additionally, K3s is well-suited for air-gapped environments, such as remote industrial sites or military installations, where network access is limited. Its simplified architecture and portability enable rapid deployment and maintenance in such challenging conditions. Accordingly, K3s can also be used for the development process to allow for smaller environments, thus smaller costs, which was the main driving factor for using it in this work.

2.2.4 Service Mesh

A Service Mesh [23] is an infrastructure layer that manages service-to-service communication in distributed systems, particularly microservices architectures. It provides a set of functionalities, such as service discovery, traffic control, observability, and security, without requiring changes to application code. By abstracting communication logic away from the application, service meshes enable developers to focus on building business functionality while ensuring consistent networking and operational policies.

In a microservices architecture, where dozens or hundreds of services communicate over dynamic, distributed networks, service meshes address challenges like managing traffic between services, monitoring performance, handling failures, and securing data in transit. This layer operates transparently, intercepting and managing all communication between services. Its primary goals are to enhance the reliability, scalability, and maintainability of applications.

Core Principles of a Service Mesh:

- **Abstraction:** Separates application logic from networking logic, allowing consistent communication policies across services.
- **Resilience:** Improves system reliability with features like retries, circuit breaking, and load balancing.
- **Observability:** Provides real-time visibility into traffic flows, latency, error rates, and other key metrics.

- **Security:** Implements features like encryption (e.g., mutual TLS) and authentication for secure communication.
- **Traffic Control:** Enables advanced traffic routing, such as canary deployments, traffic splitting, and failover strategies.

A few of the most popular Services meshes running on Kubernetes include Istio, Linkerd(v2) and Consul Connect. For this work, Linkerd(v2) will be selected due to its setup simplicity.

2.2.4.1 Linkerd

Linkerd [24] is a lightweight, open-source service mesh designed to enhance the observability, reliability, and security of service-to-service communication in cloud-native applications. Originally developed by Buoyant [25] and now governed by the Cloud Native Computing Foundation (CNCF) [26], Linkerd is optimized for simplicity, performance, and usability. By providing a transparent layer of networking features, it enables developers to manage complex, distributed microservices architectures without modifying application code.

Service meshes like Linkerd address common challenges in modern distributed systems, including observability, traffic control, fault tolerance, and security. Linkerd achieves this by injecting proxies alongside application services to manage and monitor all network communication between them. Unlike some other service meshes, Linkerd prioritizes lightweight design and operational simplicity, making it an accessible choice for teams adopting service meshes for the first time.

2.2.4.2 Linkerd's Architecture

The architecture of Linkerd [27] centers around a "data plane" and a "control plane," which work together to manage service communication and provide operational insights.

The **data plane** consists of lightweight proxies (referred to as Linkerd proxies) deployed as sidecars alongside each application instance. These proxies handle all service-to-service traffic, applying routing rules, collecting telemetry data, and implementing security policies like mTLS. Written in Rust, the proxies are designed to be highly performant and consume minimal system resources.

The **control plane** manages the configuration and monitoring of the data plane. It is responsible for distributing policies, collecting telemetry data from the proxies, and providing an interface for administrators to observe and control the mesh. The control plane runs as a set of Kubernetes-native components and integrates seamlessly with Kubernetes' native tools and APIs.

By separating the control plane from the data plane, Linkerd ensures a clear division of responsibilities, enabling high performance and scalability. This architecture also ensures transparency, as application services are unaware of the service mesh and require no changes to participate in it.

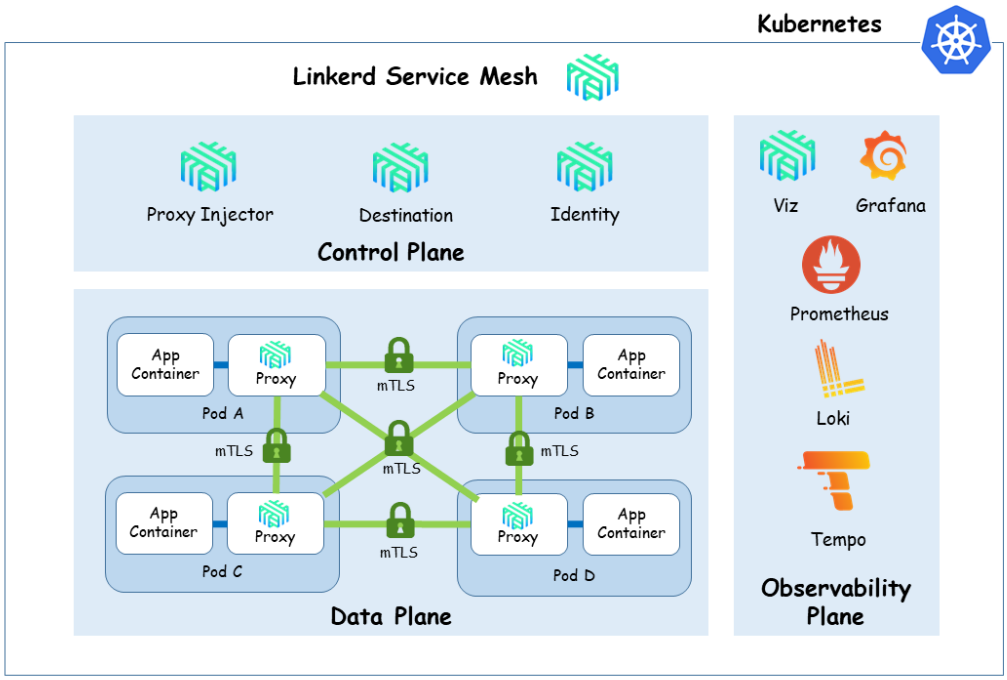

Figure 2.2 Linkerd Architecture

Linkerd Also provides a bunch of helpful Extensions for additional functionality. For this work, Linkerd SMI, Linkerd Viz, and Linker Multi-Cluster will be used.

2.2.4.3 Linkerd SMI

Linkerd's integration with SMI [28] allows it to act as a conformant implementation of the SMI standard. This means that Kubernetes users can utilize the SMI APIs to configure and manage service mesh functionality in Linkerd without relying on Linkerd-specific configurations.

Key Features of Linkerd SMI:

- **Traffic Split API:** Linkerd supports SMI's TrafficSplit API, which enables users to control traffic routing between multiple versions of a service. This feature is commonly used for rolling out canary deployments or implementing blue-green deployments.
- **Traffic Access Control:** Through the TrafficTarget API, Linkerd allows users to define which services are allowed to communicate with each other. This enables fine-grained access control policies for enhanced security.

- **Metrics and Observability:** Linkerd integrates with SMI's metrics API to provide telemetry data, such as request rates, success rates, and latency, in a standardized format. These metrics can be used with monitoring tools like Prometheus or dashboards like Grafana.
- **Protocol Support:** Linkerd's lightweight Rust-based proxy works seamlessly with SMI APIs, ensuring high performance while adhering to the standardized interfaces.

By supporting SMI, Linkerd makes it easier for organizations to adopt a service mesh without the need for custom configurations or mesh-specific integrations, reducing the learning curve for new users.

2.2.4.4 Linkerd Multi-cluster

The **Linkerd Multi-Cluster** [29] Extension is a powerful feature of Linkerd that enables seamless and secure communication between services running across multiple Kubernetes clusters. This extension is designed to address the growing need for distributed systems to operate across geographically dispersed clusters or hybrid environments. By providing a lightweight and user-friendly approach to multi-cluster connectivity, Linkerd simplifies the complexities of service-to-service communication in multi-cluster setups while ensuring security, observability, and reliability.

2.2.4.5 Linkerd VIZ

Linkerd Viz [30] is an extension of Linkerd that provides robust observability tools for monitoring and debugging microservices running in a Kubernetes cluster. It enhances the core functionality of Linkerd by offering out-of-the-box dashboards, visualizations, and metrics that help operators understand the behavior of their services and their interactions. With its focus on simplicity, usability, and performance, Linkerd Viz enables teams to gain actionable insights into their applications without requiring complex configurations or third-party tools.

By installing Linkerd Viz, users can visualize service health, track request latency, monitor success rates, and identify potential issues such as high error rates or bottlenecks. This observability is critical in distributed systems, where debugging and optimizing service-to-service communication can be challenging.

2.2.5 Locust

Locust [31] is an open-source load testing tool that allows developers and testers to simulate user behavior on web applications, APIs, or other systems. It is written in Python and provides a flexible and user-friendly framework for conducting performance testing and identifying bottlenecks in applications. With its ability to scale and run distributed tests across multiple machines, Locust is suitable for testing everything from single endpoints to large-scale systems.

Locust's primary advantage is its Python-based scripting approach, which allows users to define load testing scenarios using plain Python code. This enables highly customizable tests tailored to specific workflows, making it particularly effective for complex systems or non-standard protocols.

Locust simulates user behavior by creating "locusts" or "users," which are essentially virtual users that perform tasks defined in a Python script. These tasks represent the actions that real users would perform, such as making HTTP requests, logging in, or navigating through pages. Locust spawns a specified number of these virtual users and monitors their performance as they interact with the system under test.

3 System Topology and Methodology

The efficient placement and relocation of services within a distributed environment are deeply influenced by the intricate nature of the system's topology. In this section, we explore the architecture and design of the proposed multi-cluster Directed Acyclic Tier Based infrastructure, providing a detailed overview of its structural topology. By thoroughly examining the interconnections and hierarchical organization across various layers, we gain insights into the smooth flow of requests, the deployment and initialization of services, and the dynamic processes that enable adaptive service placement and migration. Furthermore, this analysis is not limited to 3 Tier systems; the principles and mechanisms discussed are applicable to any hierarchical, distributed, directed acyclic graph, multi-clustered architecture. Our objective is to demystify the inherent complexity of the system’s design and shed light on the diverse processes that underpin its functionality and efficiency.

3.1 Overview of System Topology

The topology of the system is pivotal for ensuring effective service placement and undeployment. It is structured into distinct tiers, each fulfilling a specific role within the overarching architecture. These tiers consist of Tier 1, Tier 2, and Tier 3, which are purposefully designed to address various aspects of request processing and service management. Requests traverse through clusters within these layers, following a defined path. If a service is not available in the current cluster along this path, the request is forwarded to a connected cluster in the next layer. To provide a visual understanding of the Tier 1-Tier 2-Tier 3 model and the interactions between these layers, Figure 3.1 illustrates the topology and interconnections within the system.

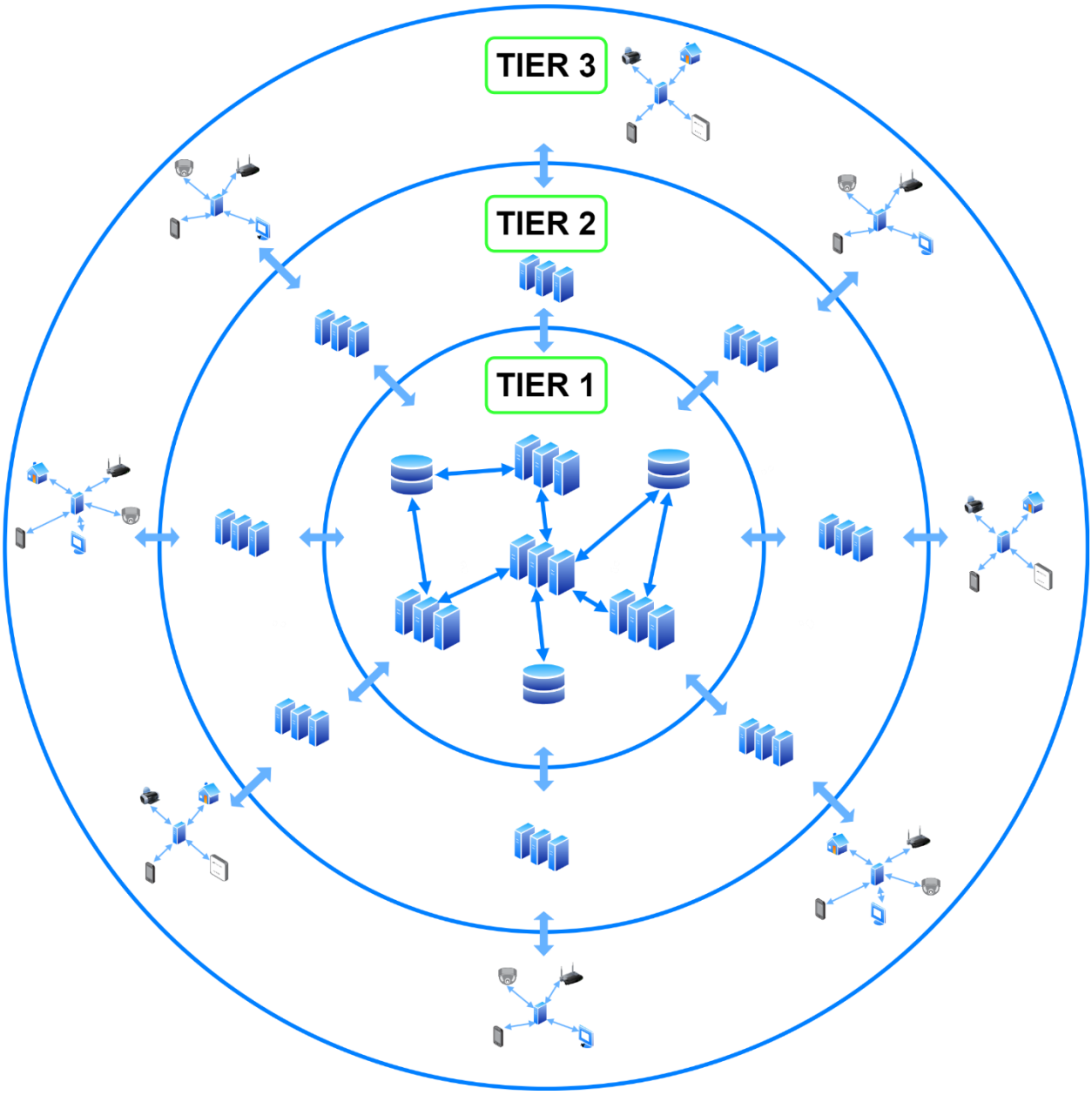

Figure 3.1 Topology Model

In the system's topology, the Tier 1 layer functions as the centralized infrastructure, typically located in data centers or provided by public cloud services. This layer delivers key benefits such as scalability, substantial computing power, and abundant resources, making it ideal for handling and processing complex workloads. Services deployed in the Tier 1 layer form the cornerstone of the system, serving as the primary entry point for processing incoming requests. By leveraging the capabilities of the Tier 1 layer, the system establishes a robust foundation for efficient request handling, enabling subsequent layers to effectively fulfill their roles within the overall architecture.

Positioned as a critical intermediary between the Tier 1 layer and the Tier 3 layer, the Tier 2 layer provides additional computing resources and services strategically located closer to the Tier 3. This placement enables efficient data processing and management for Tier 3 devices while addressing challenges such as latency and network congestion. By utilizing the capabilities of the Tier 2 layer, the system can effectively offload computational and storage demands from both the Tier 1 and Tier 3 layers. Its proximity to Tier 3 devices ensures faster data processing, reduced latency, and more responsive operations. In essence, the Tier 2 layer serves as an essential bridge in the system's topology, enhancing computational capacity and service availability closer to the Tier 3.

The Tier 3 layer forms the foundation of the system's topology, representing the tier nearest to end-user devices. While the Tier 3 layer is characterized by its limited computational resources, its strategic placement close to users provides significant benefits. By deploying services near end-user devices, the Tier 3 layer minimizes the round-trip time for requests, leading to faster response times and an improved user experience. This proximity is particularly advantageous for latency-sensitive applications and scenarios requiring real-time data processing.

3.2 Initial Service State

The system's topology employs a strategic approach to service deployment to optimize efficiency. At the outset, all services, apart from the user interfaces (Frontend), are hosted within the Tier 1 layer. This layer offers the robust computational resources, scalability, and high availability needed to support a wide array of services and applications. Serving as the centralized backbone of the system, the Tier 1 layer provides critical functionalities and ensures seamless access to data, forming the foundation for the system's operations.

After the deployment phase, services undergo an initialization process to prepare them for handling requests. This process is streamlined through the use of the Linkerd Service Mesh, which provides robust features to enhance service communication and management. A key element in this setup is Linkerd Multicluster, which plays a critical role in connecting the different layers of the system. Using said extension, mirrored services are created to facilitate seamless communication between the user interfaces deployed in the Tier 3 layer and the services hosted in the Tier 1 layer. These mirrored services act as a bridge, enabling efficient interaction and ensuring smooth access to the functionality and data within the Tier 1 layer. This design ensures effective communication and coordination between the Tier 3 and Tier 1 layers.

To route requests from the Tier 3 layer to the appropriate services, Linkerd's TrafficSplits mechanism is employed. This feature enables seamless traffic routing to mirrored services without the need for additional configuration at the traffic source. TrafficSplits utilize weight values assigned to each service, determining the proportion of traffic directed to each one.

Initially, this mechanism channels all requests from the Tier 3 layer through the Tier 2 layer and onward to the services in the Tier 1 layer. By leveraging this routing approach, requests originating from the user interfaces are efficiently directed to the corresponding services in the Tier 1 layer, ensuring optimal use of available resources and capabilities.

3.3 Service Placement Methodology

The methodology for service placement is a pivotal aspect of the system, directly influencing resource utilization, scalability, and responsiveness. The system employs the DynaQSP model, which is deployed across all clusters within the hierarchical structure. This model operates independently within each cluster, utilizing decentralized strategies to manage states and actions effectively. Instead of relying on a centralized service, decision-making is distributed, enabling each cluster to evaluate its own resources and workload. Placement decisions are guided by key factors such as request latency, network throughput, and resource availability (CPU and RAM). This approach ensures efficient use of computational resources and network bandwidth while maintaining system responsiveness and scalability.

The model evaluates the trade-offs between resource utilization costs, network expenses, and the latency benefits of deployment actions, selecting the option that achieves the most optimal outcome. By following this approach, it ensures minimal latency while simultaneously reducing costs. For instance, services with high network throughput are more likely to be deployed in lower hierarchical clusters, closer to the Tier 3, whereas services with high resource demands but low network throughput are better suited for exclusive deployment in cloud clusters. By continuously gathering metrics on the current system state and incorporating insights from previous states, the DynaQSP model dynamically adjusts service deployments and deletions to optimize performance and resource efficiency.

3.4 Traffic Splitting Methodology

The distributed deployment of DynaQSP enables services to be instantiated across multiple clusters, facilitating efficient load balancing and traffic redistribution. When a service under stress is identified, DynaQSP dynamically adjusts the weights in the TrafficSplit configuration, redirecting portions of the traffic to other connected clusters. This adaptive load balancing mechanism ensures the system can effectively manage fluctuating workloads while sustaining high performance.

Overall, DynaQSP's decentralized and adaptive design allows for continuous monitoring, dynamic traffic routing, and orchestration of service deployments and removals. This flexibility optimizes the utilization of resources, enhances system responsiveness, and ensures a reliable and robust service environment.

Let's analyze an example scenario to see the functionality of the Traffic Split algorithm. Suppose we have a service named adservice that is initially only deployed on the Tier 1 layer and mirror down to the Tiers 2 and 3. This would look something like so:

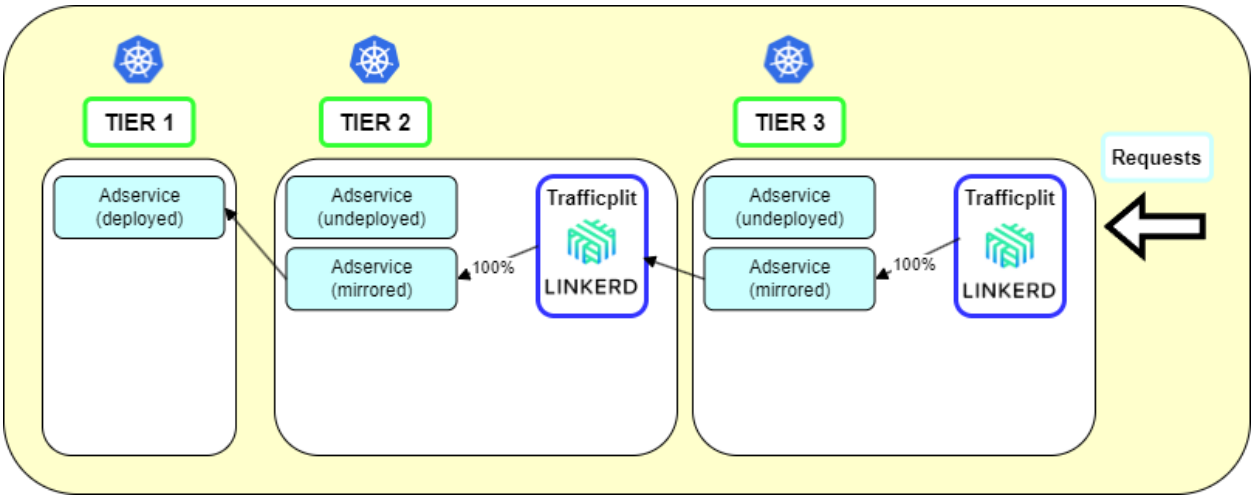

Figure 3.2 Traffic Split Initial State

After some time the Reinforcement learning Agent might decide it is optimal to deploy the adservice locally on the Tier 3 Cluster.

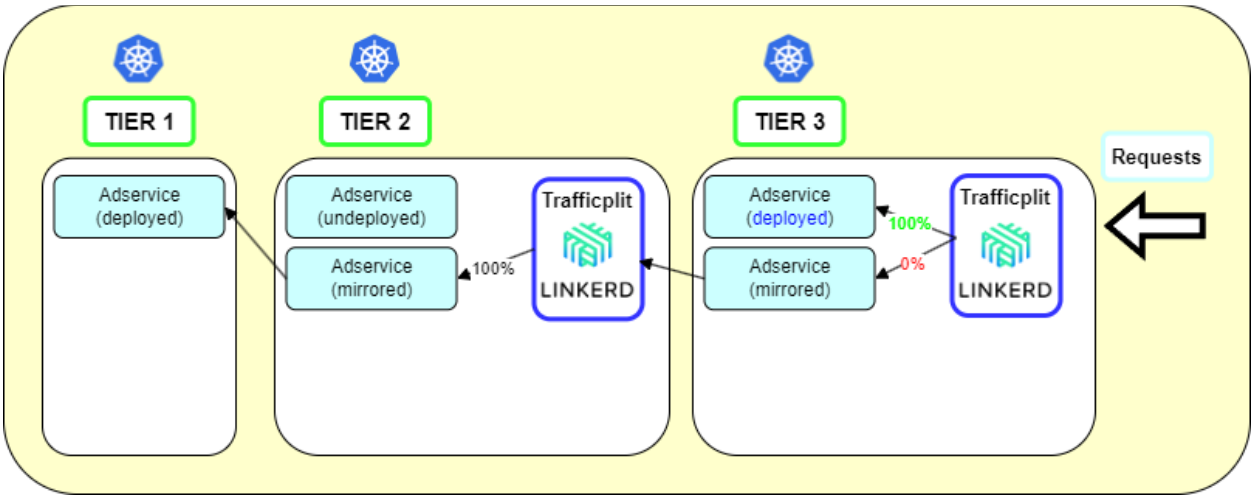

Figure 3.3 Traffic Split Service Deployment

However, this might lead to problems with higher loads as the Tier 3 Cluster does not have the same computational capabilities as the Tier 1 Cluster. This will lead to an overload of the service deployed and thus increase latency dramatically.

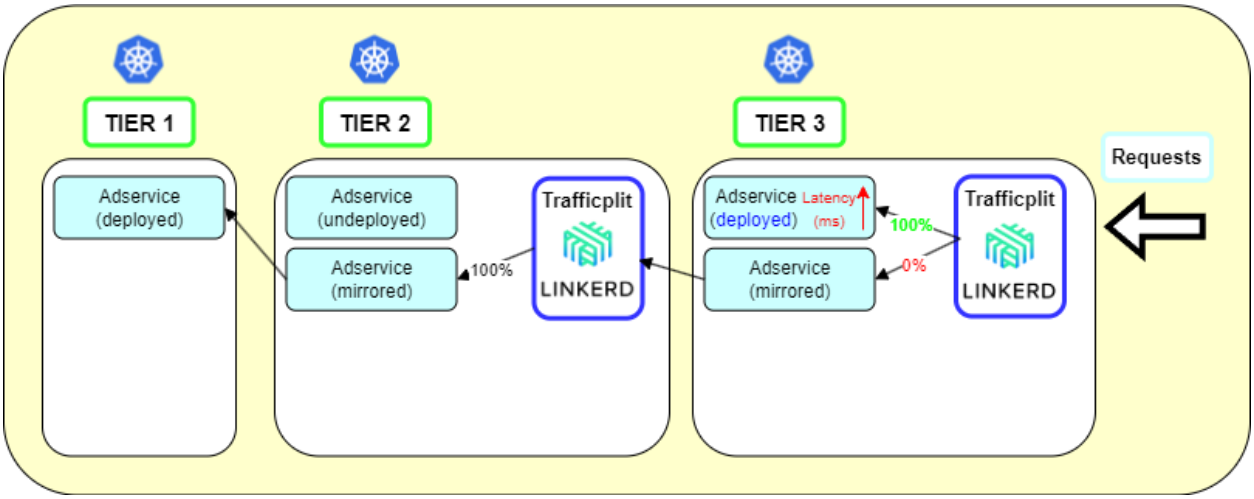

Figure 3.4 Traffic Split Service Overload

The algorithm will pick up on this sudden jump and instantly start directing more traffic to the mirrored service to cover any shortcoming of the local deployment until the baseline latency is restored.

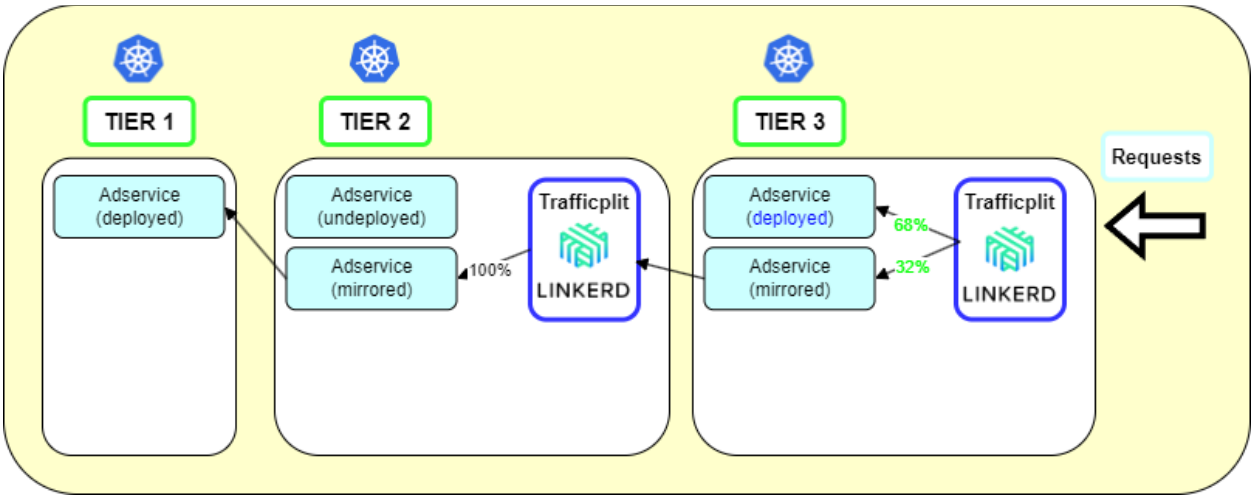

Figure 3.5 Traffic Split Service Rebalancing

The Design used in this work will be more complex and can be represented by what is shown below:

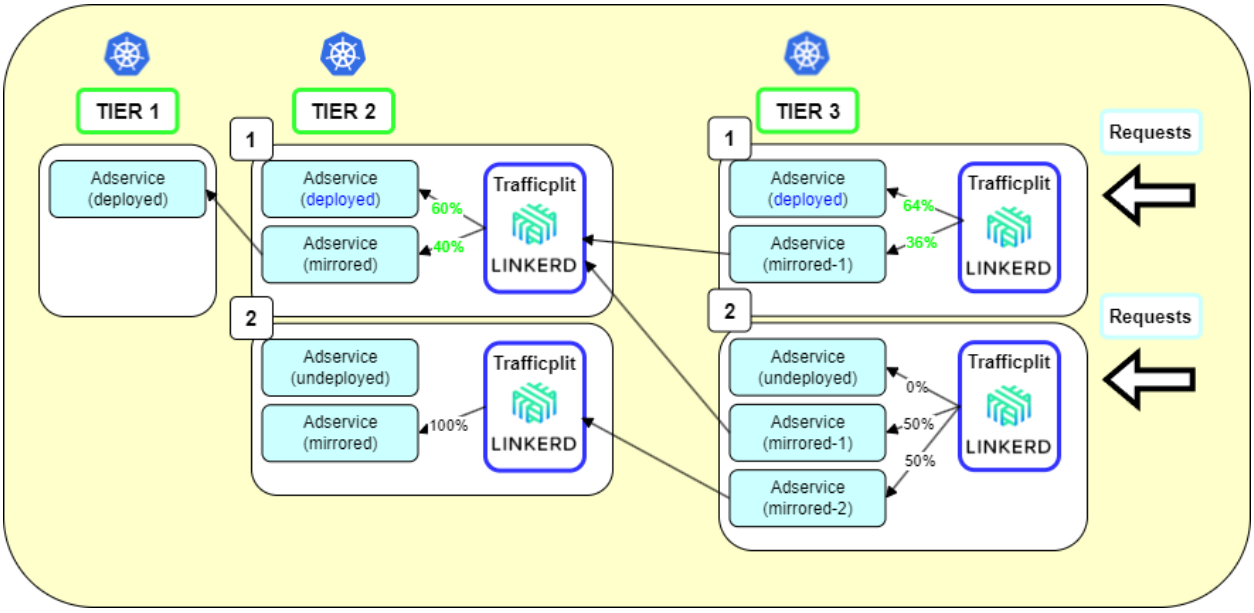

Figure 3.6 Traffic Split Complex

4 System Design

This chapter focuses on the implementation phase of the service placement strategy, detailing each cluster's inner working and the critical components necessary for its successful execution. We begin by defining the key metrics that underpin the placement and load-balancing strategies, providing the foundation for informed decision-making. Following this, we present and conduct an in-depth analysis of the algorithms that form the backbone of the implementation framework. The discussion then shifts to an examination of the cluster's architecture and the application developed, DynaQSP, which facilitates placement and load balancing within the system. Finally, we introduce the benchmark applications used to evaluate the proposed placement strategies, offering a comprehensive assessment of their effectiveness.

4.1 State Metrics

Performance metrics play a critical role in the design, monitoring, and evaluation of distributed systems, particularly for the proposed service placement management model. Key metrics commonly used to assess performance include Network Throughput, measured in terms of **Requests Per Second (RPS)** or **Data Transfer Rates** (Mbps/Gbps), and Response Latency, which reflects the time taken to process and respond to requests. These metrics will be collected using specialized tools and integrated into the proposed service placement strategies. By leveraging this data, the system can dynamically manage the deployment and removal of microservices across the hybrid Tier 1 – Tier 2 – Tier 3 infrastructure, ensuring that it consistently meets the desired performance objectives.

Network throughput metrics provide valuable insight into the overall load on a system's network. The **Requests Per Second (RPS)** metric measures the number of requests the system processes per second, reflecting its capacity to handle incoming and outgoing traffic. Complementing this, **Data Transfer Rates** offer a more granular view by quantifying the total number of bytes transferred into and out of the system each second. Unlike RPS, data transfer rates account for variations in request sizes, ensuring a more accurate representation of data flow without skewing estimations due to irregular request sizes. By combining these two metrics, it becomes possible to calculate the average size of each request, offering a comprehensive understanding of network activity.

In this work, both metrics are collected using the Prometheus API. Services with high network throughput are more likely to be deployed at lower levels of the hierarchy, such as in Tier 3 or Tier 2 clusters. This placement minimizes inter-level communication, reducing network costs and improving system efficiency. By strategically leveraging network throughput metrics, the system ensures optimal resource utilization and cost-effective service placement.

The **Response Latency** metric, often referred to as **Time-to-First-Byte (TTFB)**, measures the time taken for the first byte of a server's response to reach the client. This metric includes both the network travel time and the server processing time. In this work, the **95th percentile latency metric** is used to evaluate the performance of microservices. This metric reflects the response time below which 95% of the requests fall, focusing attention on the tail end of the latency distribution where delays can significantly impact the overall user experience.

The model employs this metric in a predictive manner, estimating the potential cost each latency value may impose on the system, based on an assumed **Service Level Agreement (SLA)** with a predefined baseline latency value. As the observed latency approaches or exceeds the **SLA**'s maximum acceptable value, a substantial cost is added, discouraging the model from decisions that would further increase response times. Consequently, services with higher latency are prioritized for deployment in clusters closer to the user, leveraging proximity to reduce latency and enhance overall performance. This metric is also collected and monitored using Prometheus.

4.2 System Architecture

In a single-node K3s [21] architecture, the cluster operates on a single virtual machine (VM) and orchestrates containerized applications using Kubernetes primitives. This cluster manages the deployment, scaling, and lifecycle of applications, ensuring they run efficiently. The resources available to the node are fixed and determined by the hardware specifications of the VM, as Kubernetes auto-scaling is not utilized in this setup.

The **K3s Control Plane** components are streamlined into a single process, simplifying the cluster's operation and management. This lightweight approach reduces overhead and makes it easier to manage the cluster compared to traditional Kubernetes setups. The **Data Plane**, responsible for running and managing containerized applications, consists of containerd (the container runtime) and the pods it controls. Similar to a standard Kubernetes node, the K3s node can host multiple pods, with each pod containing one or more containers that deliver microservices or application components.

Communication between containers is facilitated by **Kubernetes Services**, which provide a well-defined DNS name and port number for each service. These services expose a group of pods to either internal or external consumers by providing a stable network endpoint (IP address and port). This allows seamless communication between components within the cluster or with external entities, ensuring consistent access to the pods. The architecture of a single-node K3s cluster is depicted in Figure 4.1.

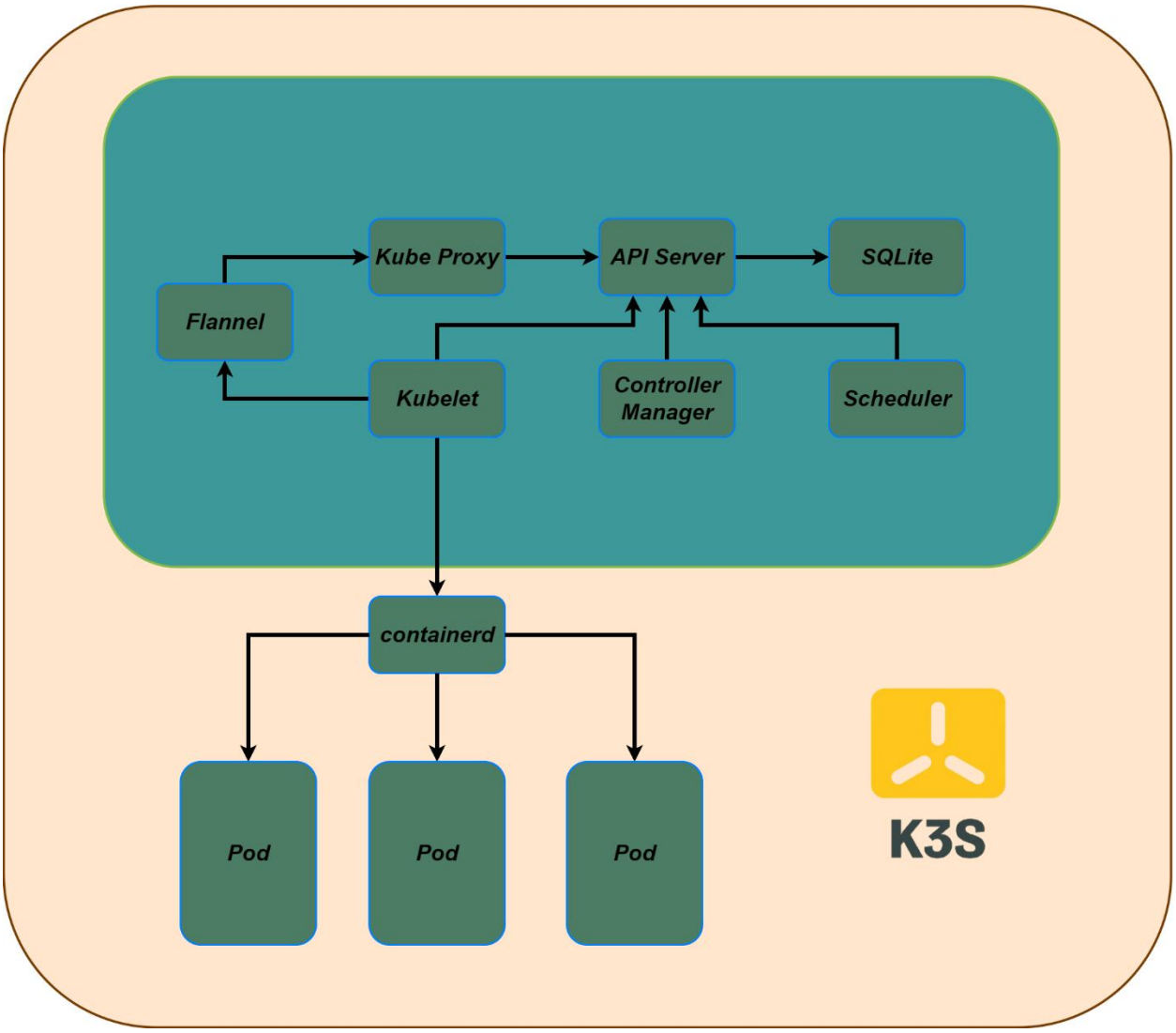

Figure 4.1 Single-Node K3s cluster Architecture

Deploying the Linkerd service mesh in the cluster introduces a sidecar proxy container alongside each pod in a Kubernetes deployment. This sidecar proxy enables the transparent interception and routing of network traffic for the associated application pod. The sidecar proxy handles all inbound and outbound traffic for the pod, including communication between other pods within the same namespace or across different namespaces. When a request is directed to a pod, the sidecar proxy intercepts it and determines whether the traffic should be routed to another pod within the cluster. This mechanism ensures secure, efficient, and transparent communication across services. Figure 4.2 illustrates a pod that has been injected with Linkerd's sidecar proxy.

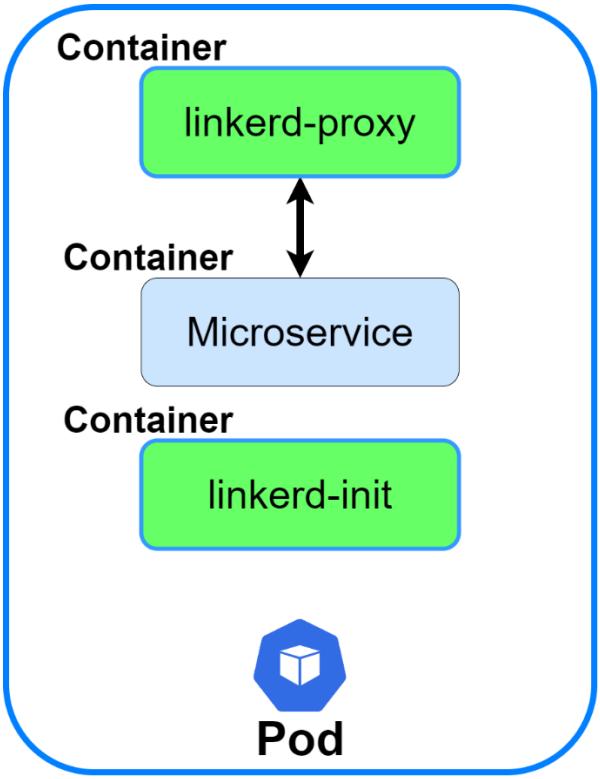

Figure 4.2 Linkerd Injected Pod

Linkerd monitors network traffic passing through its sidecar proxies to provide enhanced observability and security features. By installing the **Linkerd Viz** extension, users gain deeper insight into the service mesh, including the ability to visualize the service topology. This extension integrates with a built-in **Prometheus** instance, which collects metrics by querying the proxies. The extracted data can be visualized using the included **Grafana** instance, enabling detailed analysis of system performance and interactions.

In addition, **Linkerd SMI (Service Mesh Interface)** introduces the **TrafficSplit** specification, allowing dynamic control over traffic routing. With TrafficSplit, portions of traffic directed to a Kubernetes Service can be shifted to other destination services, enabling flexible traffic management strategies such as canary deployments or blue-green rollouts.

Installing the **Linkerd Multi-cluster** extension further extends the functionality of the service mesh to span multiple Kubernetes clusters. This enables seamless communication between microservices running across different clusters, ensuring consistent observability, security, and traffic management throughout the distributed system.

Figure 4.3 presents a comprehensive illustration of the Linkerd Service Mesh setup, showcasing its core installation and extensions, including Linkerd Viz and Linkerd Multi-cluster. The diagram highlights the segregation between the **Control Plane**, which manages the service mesh's

operations, and the **Data Plane**, which consists of meshed microservices. Each pod within the Data Plane, as well as components of the service mesh itself, is equipped with a **linkerd-proxy** and **linkerd-init**. This configuration allows Linkerd to monitor and observe both meshed microservices and its own operational elements, ensuring an efficient and secure service mesh deployment.

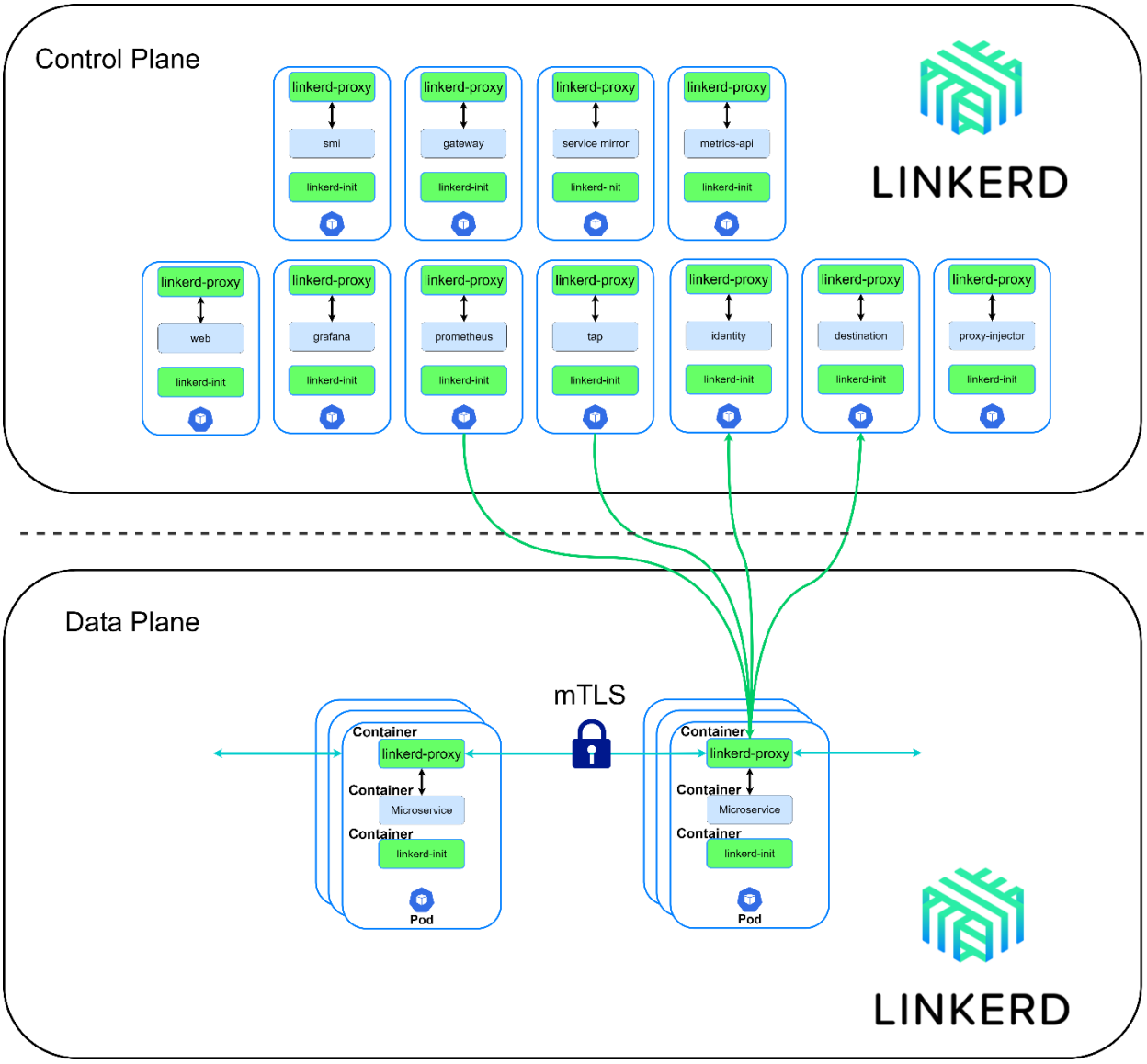

Figure 4.3 Linkerd Architecture

This thesis leverages the advanced capabilities of the Linkerd Service Mesh by employing multiple clusters configured as described earlier and interconnected using the Multi-cluster extension. This approach allows for seamless integration of microservices operating across different clusters, enhancing the flexibility and scalability of the system architecture.

4.3 DynaQSP

DynaQSP marks a major step forward in tackling the challenges of service placement within hybrid multicluster Tier 1-Tier 2-Tier 3 infrastructures. Developed in Python, DynaQSP takes advantage of the robustness and flexibility of two key libraries: the Kubernetes Python client and the Prometheus API client. These libraries serve as core components, enabling DynaQSP to interact seamlessly with the Kubernetes API and leverage the critical metrics provided by Prometheus.

The Kubernetes Python client is a key component of DynaQSP's functionality, enabling seamless interaction with the Kubernetes API. Through this client, DynaQSP can effectively engage with the underlying Kubernetes infrastructure, including managing various Kubernetes resources such as deployments, services, and traffic splits, as well as accessing detailed resource information. This integration empowers DynaQSP to make data-driven decisions regarding the deployment and removal of microservices based on real-time data and the system's operational requirements.

Simultaneously, DynaQSP leverages the Prometheus API client to establish a crucial connection to the Prometheus HTTP API. As a leading monitoring and alerting tool in cloud-native environments, Prometheus provides an extensive range of metrics. By utilizing the Prometheus API client, DynaQSP gains access to detailed system metrics, including data transfer rates, response latency, resource utilization, and other performance indicators. These metrics are integral to DynaQSP's decision-making processes, driving its strategies for optimal service deployment and management.

DynaQSP incorporates two key features: performance and cost-efficient service placement and dynamic traffic management. These capabilities ensure that services are optimally placed and that incoming traffic is effectively managed to maintain system efficiency and responsiveness.

For efficient service placement, DynaQSP employs a continuous learning and decision-making process that evaluates the current system state. It analyzes key metrics such as data transfer rates, response latency, and RAM usage to determine the optimal placement of services based on their importance, resource requirements, and operational costs, including machine and network costs. This process incorporates Kubernetes resource requests and limits for CPU and RAM, which are defined in the configuration files of the services. These configuration files also include machine and network cost parameters, enabling the model to calculate the running costs for each service. By taking these specifications into account, DynaQSP achieves near-optimal cost efficiency across all services within the cluster.

In addition to cost considerations, DynaQSP carefully assesses resource availability in the cluster to ensure that deployment decisions align with allocated resources. This approach

avoids resource contention and ensures that the system operates within its capacity while meeting service demands effectively.

Beyond service placement, DynaQSP also excels in traffic management, a critical aspect of load balancing and stress mitigation. The model dynamically splits incoming traffic between the deployed instances within the local cluster and mirrored services across other connected clusters. This intelligent traffic-splitting strategy helps balance the load across clusters and alleviates stress on individual services, ensuring consistent performance and system stability.

These features function on a periodic basis, continuously adapting to the evolving needs of the system to maintain optimal performance and efficient resource utilization. The core workflow and decision-making process of DynaQSP are outlined in Algorithm 4.1.

Algorithm 1: DynaQSP main process

1. *Load Cluster Configuration properties*
2. *Initialize all Objects*
3. *Establish Prometheus connection*
4. *Period <- 60s*
5. ***While** True:*
6. *Sleep(period)*
7. *performServicePlacement()*
8. *updateTrafficSplits()*
9. *cycle <- cycle + 1*
10. ***End While***

Algorithm 4.1 DynaQSP main loop

The algorithm outlined above captures the primary workflow of DynaQSP. Initially, the cluster configuration properties are loaded into the model, followed by the initialization of key objects, including Metrics, Utils, Scheduler, and RLAgent. A connection to Prometheus is then established, allowing the system to access and process real-time metrics. Once these preparatory steps are completed, DynaQSP enters its main operational loop, where the service placement function is executed every minute.

In summary, DynaQSP integrates the decision-making capabilities of a Reinforcement Learning Agent for service placement with advanced traffic-splitting techniques. This combination optimizes resource utilization and enhances service delivery. By collecting and analyzing metrics

such as data transfer rates, response latency, CPU and RAM usage, and Kubernetes resource requests and limits, as well as considering machine and network cost rates for each cluster, DynaQSP achieves cost-efficient and performance-driven deployment and removal of services. Additionally, the model ensures effective load balancing and stress management by dynamically splitting traffic among all available instances of a service across clusters.

These combined features position DynaQSP as a powerful tool for addressing the complexities of service placement in hybrid Tier 1-Tier 2-Tier 3 infrastructures. Its ability to enhance performance, scalability, and cost efficiency makes it an indispensable solution for managing services in distributed environments.

4.4 Service Placement Algorithm

In distributed systems, the strategic placement of microservices across varied infrastructures plays a critical role in determining overall performance. Therefore, implementing effective service placement strategies is essential to maintaining optimal system functionality. This thesis presents a Reinforcement Learning Agent designed to dynamically deploy and remove microservices within a hybrid distributed infrastructure. The Agent operates based on key metrics, including network throughput, response latency, and resource utilization, ensuring efficient resource management and enhanced system performance. The main flow of the service placement algorithm is shown below in Algorithm 4.2

Algorith 2: performServicePlacement()

```
1. metrics <- getPrometheusMetrics()
2. state <- getStateFromMetrics(metrics)
3. For service in services:
4.         action <- RLAgent.chooseAction(state)
5.         performAction(service, action)
6.         previousStates[service].push(state).subList(0, 5)
7.         updateQTable(previousStates)
8. End For
```

Algorithm 4.2 Service Placement Flow

4.4.1 Metric Data to State

Data on CPU usage, RAM usage, Response latency, Data Transfer Rates (In and Out) and Requests Per Second (In and Out) is collected through the Prometheus API. After acquiring the metric data, the model collects them in a map where each elements' key is the service name holding those metrics. It is important to note, that there is a separation between data for deployed and non-deployed services, as a cluster with a service that is deployed still contain the original linked network service created by Linkerd Multicluster and thus have to be aggregated. These can then be used at will for all the following functionalities. This process is shown in the Algorithm 4.3 found below.

Algorithm 3: getStateFromMetrics(metrics)

1. ***For*** *service* ***in*** *services:*
2. *states[service] <- serviceMetrics*
3. ***End For***

Algorithm 4.3 Metrics to State Collection

4.4.2 Select Action

The Q table holds all the values for any state action pair by storing an array of Q values for each action (**Deploy**, **Delete**, **DoNothing**) based on each state. This Q value effectively holds the practical cost of each action according to the state of the service when the action is taken. By choosing the action with the minimum value we can be sure that our decision incurs the minimum costs possible.

After collecting all metric data into a state map and before determining an action, the state data must be converted into an equivalent representation suitable for the state matrix that stores rewards for each possible state. To achieve efficient mapping and ensure the model converges to an optimal result within a reasonable timeframe, a partitioning scale of 1 to 10 has been selected. Metrics such as CPU usage, RAM usage, response latency, and network throughput are transformed from their raw values into this scale.

For CPU the mapping happens from a range of 0.5% to 50% utilization where 0.5% represents 1 and 50% represents 10. This can be achieved by first multiplying the percentage to scale it from 1 to 100 and take the square root of it.
Similarly for RAM usage, values from 10MB to 1GB are mapped once again to a scale from 1 to

10. This time we divide the metric by 10MB and take the root of the result. We then take the higher of the two values (CPU and RAM) and use that as the indicator for Resource Utilization.

The same approach can be applied to response latency and network throughput, as these metrics can vary significantly between services. For instance, one service might have an expected response latency of 5ms, while another could range between 500ms and 1000ms. Similarly, network throughput can differ drastically depending on the service.

To address this variability, a baseline value is defined as an environment property for both response latency and network throughput. Using this baseline, the metrics are mapped through appropriate functions and normalized onto the 1 to 10 scale, ensuring consistency across diverse services. For response latency, a root function has been chosen as the mapping function. This method effectively handles disparities; for example, a baseline of 5ms would map values of 5ms and 1000ms to approximately 1 and 10, respectively, aligning with the desired representation.

$$\frac{\sqrt{\text{latency\_actual}}}{10}$$

For network throughput, the mapping function employs the base-10 logarithm multiplied by 5 of the data rate divided by the baseline. For example, a service transmitting 10KB of data over the network in a minute (a low data rate) would map to a value of 1. Conversely, a high-throughput service, such as a streaming service sending over 1MB per minute, would map to a value of 10.

$$5 \times \log_{10}\left(\frac{\text{rate}}{10000} + 1\right)$$

Once these metrics are mapped, the model can accurately retrieve and update the cost associated with the equivalent state in the state matrix. If the Agent attempts an action that cannot be performed, no changes are made, and the action is skipped. This safeguard ensures

DynaQSP consistently makes optimal decisions based on its current state. The process described above is detailed in **Algorithms 4.4 and 4.5**.

Algorithm 4: chooseAction(state)

```
resources, latency, dataRate <- getMappedState(state)
If rand(0,1) < learningRate:
        action <- selectRandomAction()
Else
        action <- actions[min(QTable[resources][latency][dataRate])]
End If
If IsActionPossible(action):
        return action
Else:
        return actions.DoNothing
End If
```

Algorithm 4.5 Select Action Process

Algorithm 5: getMappedState(state)

```
# cpu_metric is the usage percentage in decimal form
# ram metric is the total ram utilized in bytes
cpuMapped <- int(clamp((state.cpuMetric*200)^(1/2),1, 10)
ramMapped <- int(clamp((state.ramMetric/(10^7))^(1/2),1, 10)
resourcesMapped <- max(cpuMapped,ramMapped)
# latency in ms
latencyMapped <- int(clamp(state.latency ^ (1/2), 1, 10))
# dataRate in total bytes transferred
dataRateMapped <- int(clamp(5*log(state.dataRate / 10000 + 1), 1, 10))
return resourcesMapped, latencyMapped, dataRateMapped
```

Algorithm 4.4 State Mapping

4.4.3 Update Q Table

After the model selects and executes an action, it updates the state matrix by referencing a matrix of arrays that store up to five of the most recent states for each service. This mechanism enables the model to retroactively adjust the costs of previous actions, as described in lines 6–9 of Algorithm 4.2 Service Placement Flow. The update function incorporates a modified Q-learning algorithm, designed to be more dynamic and reflective of the actual outcomes of its actions rather than relying solely on expected results. The Q-learning algorithm used in this process is expressed in the form that was described earlier in: Q Learning

By leveraging the knowledge of the previous five states, the model can propagate updates backward, adjusting the Q-values of older states while considering the outcomes of subsequent actions. This approach makes the decision-making process more future-aware, enabling the model to incorporate information that would otherwise remain uncertain or probabilistic. As a result, the system avoids suboptimal outcomes by aligning past state evaluations with the observed results of later actions. This process is depicted in **Algorithm 4.6** below.

Algorithm 6: updateQTable(previousStates)

```
1.  Q <- 0
2.  If len(previousStates) != RLAgent.PREVIOUS_STATES:
3.      return
4.  For i in range(len(previousStates)):
5.      currentState <- previousStates[i]
6.      previousState <- previousStates[i+1]
7.      prevMappedState <- getMappedState(previousState)
8.      If (i == 0):
9.          mappedState <- getMappedState(currentState)
10.         Q <- QTable[mappedState][0]
11.     Else:
12.         Q <- getReward(previous_state, state) + γ Q
13.     End If
14.     Qprev <- QTable[prevMappedState][0]
15.     n  <- QTable[prevMappedState][1]
16.     n <- n +1
17.     Q = (Qprev * (n-1) / n) + Q/n   # average Q over all previous values
18.     QTable[prevMappedState] <- [Q, n]
19. End For
```

Algorithm 4.6 Update Q Table

The most critical aspect of the Q-table update process involves calculating the reward (or in this case, the cost) of the action performed. To determine this cost, the model calculates the current running cost of the service based on the per-hour cost of the virtual machine (VM) hosting the service, multiplied by the service's resource usage. Additionally, network costs are factored in by multiplying the cost per GB of network usage by the service's network throughput.

To account for the impact of latency, a representative cost is included, which scales proportionally higher as latency increases. This ensures that latency penalties are incorporated into the overall cost calculation, encouraging decisions that minimize latency while balancing other costs. To balance the contribution of actual costs and the impact of latency-related costs, a multiplier, referred to as **MULT**, was determined through an iterative process of trial and error. The full calculation process is detailed in **Algorithm 4.7**, shown below.

Algorithm 7: getCost(previousState, currentState)

```
1. If isDeployed:
2.     return MACHINE_COST * MULT * currentState.deployment.resources +
3.         NETWORK_COST * MULT * currentState.service.network_load +
4.         ((previousState.latency – currentState.latency)/base_latency) ^ (1/2)
5. Else If
6.     return NETWORK_COST * MULT * currentState.service.network_load +
7.         ((previousState.latency – currentState.latency)/base_latency) ^ (1/2)
8. End If
```

Algorithm 4.7 Cost Calculation Function

4.5 Load Balancing Algorithm

In a distributed system, efficient traffic management across various infrastructures is essential for achieving load balancing and mitigating stress on services. This thesis introduces an algorithm that dynamically manages traffic by intelligently splitting it between deployed instances within the local cluster and mirrored services across other clusters. The algorithm considers the resource limits specified in the service configurations. By leveraging this information, **DynaQSP** calculates a weight that ensures traffic is distributed in a way that optimizes resource utilization and prevents overloading any individual service. The process for calculating these weights is detailed in **Algorithms 4.8** and **4.9**.

Algorithm 8: updateTrafficSplits(services, states, isDeploy, isDelete)

1. ***For*** *service* ***in*** *services:*
2. *split* ***<-*** *kubectl.getTrafficSplit()*
3. *updateSplit(split, services, states[service], isDeploy, isDelete)*
4. *Kubectl.setTrafficSplit(split)*
5. ***End For***

Algorithm 4.8 Update Traffic Split part 1

Algorithm 9: updateSplit(split, services, states, isDeploy, isDelete)

```
1.  If isDeploy:                        #direct all traffic to deployment
2.      split.deploy.weight <- 1000
3.      For parent in split.directParents:
4.          parent.weight <- 0
5.      End For
6.  Else If isDelete:                   #direct all traffic to parents
7.      split.deploy.weight <- 0
8.      For parent in split.directParents:
9.          parent.weight <- 1000/len(split.directParents)
10.     End For
11. Else:
12.     If !isdeployed(service):
13.         Return
14.     latency_lows[service] <- min(latency_lows[service], states[service].latency)
15.     If states[service].latency > latency_lows[service]:
16.         decreaseDeploymentWeight()
17.         increaseParentWeight()
18.     Else:
19.         increaseDeploymentWeight()
20.         decreaseParentWeight()
21.     End If
```

Algorithm 4.9 Update Traffic Split part 2

In these algorithms, the percentage changes in response latency for each service are monitored and used to calculate appropriate adjustments to traffic split weights across all instances of the service, whether mirrored or locally deployed. Initially, after deploying a service, all traffic is directed toward the in-cluster deployment. This approach minimizes costs by reducing unnecessary propagation to higher-tier clusters in the hierarchy. Conversely, when a service is removed, all traffic is redirected away from the cluster, as it can no longer handle incoming requests.

For other scenarios, the system imposes a cap on the rate of change for traffic adjustments per cycle. Specifically, changes in traffic weights toward the in-cluster deployment are limited to a maximum of 25% per cycle. This cap prevents abrupt changes that could result from overreacting to minor spikes in latency, which may not be significant enough to warrant a drastic response.

By capping adjustments and allowing gradual changes, the system achieves smoother traffic redirection. This method ensures that the system operates efficiently, avoids resource wastage, and maintains stability while optimizing performance and minimizing latency.

4.6 Benchmark Applications

This study utilizes three distinct applications to evaluate and validate the proposed service placement strategies: Google's **Online Boutique** [32] and **Bank Of Anthos** [33] and Descartes Research's **TeaStore** [34]. These applications were selected for their versatility, complexity, and ability to represent modern cloud-based systems effectively. Both applications are deployed across multiple K3s clusters, integrated with the Linkerd Service Mesh and **DynaQSP**, the application developed specifically for this research. DynaQSP is tasked with executing decisions aligned with each evaluated placement strategy, enabling a thorough assessment of its effectiveness in real-world scenarios.

4.6.1 Google's Online Boutique

The **Online Boutique** is an open-source reference application developed by Google, designed to introduce developers to the concepts of microservices. This application allows users to experiment with a variety of cloud-native technologies, including scaling strategies, fault tolerance mechanisms, and deployment models. It serves as a demonstration platform for technologies such as Kubernetes/GKE, Istio, Stackdriver, and gRPC, as highlighted by Google.

The Online Boutique consists of multiple microservices that collectively provide a complete e-commerce experience. These microservices handle a range of functionalities, including a product catalog, shopping cart, checkout, payment processing, and more. Each service is independently designed, which enhances its scalability and flexibility. Communication between these microservices is facilitated using **gRPC**, a high-performance remote procedure call (RPC)

framework. In this architecture, gRPC APIs enable efficient communication between distributed systems, ensuring seamless interaction and scalability across the platform.

According to the official GitHub repository, the application comprises 12 microservices, each written in different programming languages. A detailed breakdown of its architecture is as follows:

- **Frontend**: Written in Go, this microservice provides an HTTP server to host the website. It does not require user authentication and automatically generates session IDs for all users.
- **Cart Service**: Written in C#, it manages the user's shopping cart by storing and retrieving items from a Redis database.
- **Redis Cart**: An in-cluster Redis database used by the Cart Service to store cart data.
- **Product Catalog Service**: Developed in Go, it delivers a list of products from a JSON file and allows users to search for or retrieve individual products.
- **Currency Service**: Written in Node.js, it converts monetary values between currencies, fetching real exchange rates from the European Central Bank. This service handles the highest number of queries per second.
- **Payment Service**: Also developed in Node.js, it simulates payment processing by charging credit card information (mock) and returning transaction IDs.
- **Shipping Service**: Written in Go, it provides shipping cost estimates based on the user's shopping cart and mock shipping to specified addresses.
- **Email Service**: Developed in Python, it sends order confirmation emails (mock).
- **Checkout Service**: Written in Go, it retrieves the user's cart, processes orders, and orchestrates payment, shipping, and email notification services.
- **Recommendation Service**: Developed in Python, it suggests additional products based on the contents of the user's cart.
- **Ad Service**: Written in Java, it provides contextual text ads based on given keywords.
- **Load Generator**: Created in Python, this microservice continuously sends requests to simulate realistic user shopping activities and test the application.

During the evaluation process, the **Load Generator** microservice is excluded, as its purpose is solely to generate continuous traffic for testing the application.

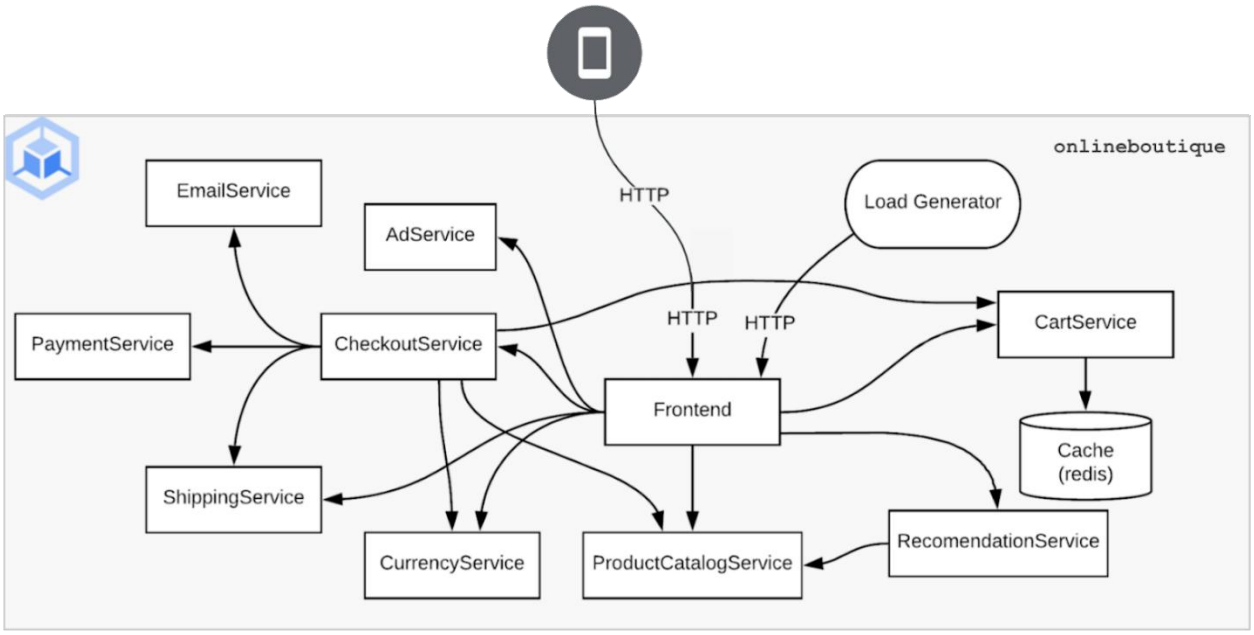

Figure 4.4 Google's Online Boutique Architecture

The Google Online Boutique stands out as a robust reference application for understanding microservices architecture, container orchestration, scaling, and resource management. Its versatile and complex design makes it an excellent tool for evaluating cloud-based systems. The inclusion of multiple services, each with unique functionality and varying demands, provides a reliable and representative benchmark for research and development. By utilizing this application, developers and researchers can validate their findings while gaining practical insights into building resilient, scalable, and efficient cloud-native systems.

4.6.2 Bank Of Anthos

The **Bank of Anthos** is an open-source, web-based application developed by Google to simulate a bank's payment processing network. It allows users to create artificial bank accounts and perform transactions, serving as a practical tool for demonstrating the modernization of enterprise applications using Google Cloud technologies.

The architecture of Bank of Anthos is composed of multiple microservices, each responsible for specific functionalities within the banking simulation. This design promotes flexibility and scalability, allowing developers to explore various aspects of cloud-native application development. The key microservices include:

- **Frontend**: Developed in Python, this service exposes an HTTP server to serve the website, providing users with interfaces for login, signup, and account management.

- **Ledger Writer**: Implemented in Java, it accepts and validates incoming transactions before recording them in the ledger database.
- **Balance Reader**: Also written in Java, this service offers an efficient, readable cache of user balances as retrieved from the ledger database.
- **Transaction History**: A Java-based service that provides a readable cache of past transactions, facilitating quick access to transaction records.
- **Ledger Database**: Utilizes PostgreSQL to maintain a ledger of all transactions, with options to pre-populate data for demonstration purposes.
- **User Service**: Written in Python, it manages user accounts and authentication, issuing JSON Web Tokens (JWTs) for secure communication between services.
- **Contacts**: A Python service that stores lists of other accounts associated with a user, aiding in functionalities like the "Send Payment" and "Deposit" forms.
- **Accounts Database**: Employs PostgreSQL to store user accounts and related data, with capabilities to pre-populate with demo users.
- **Load Generator**: Developed using Python and Locust, this component continuously sends requests to the frontend, simulating realistic user interactions to test the application's performance.

By leveraging the Bank of Anthos, developers can gain hands-on experience with microservices architecture, container orchestration, and the deployment of scalable applications on Google Cloud. The application's modular design and comprehensive documentation make it an excellent resource for learning and experimentation in cloud-native development.

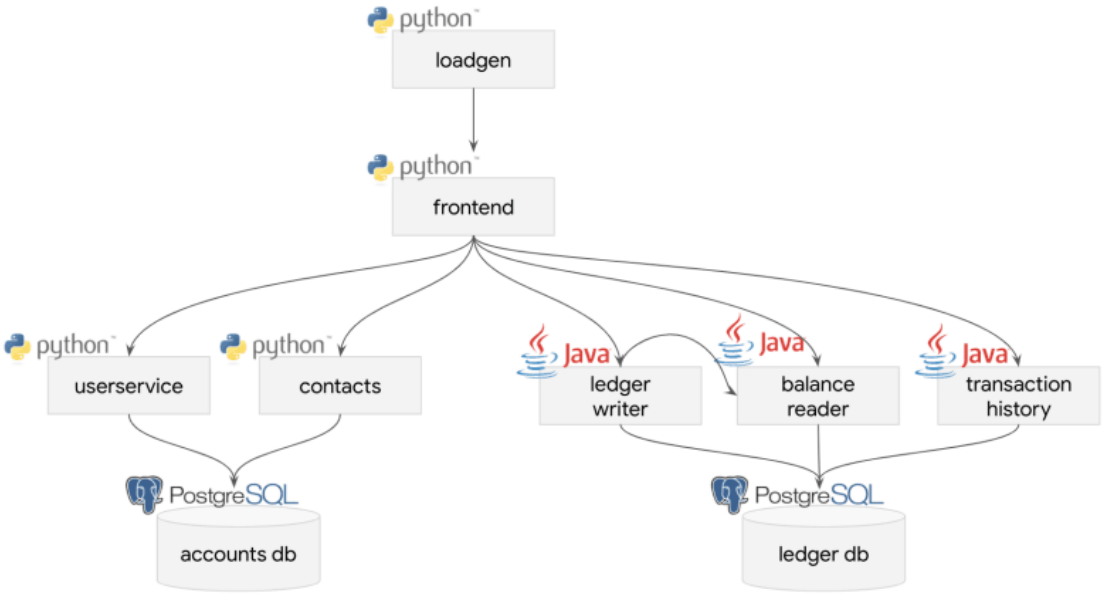

Figure 4.5 Bank Of Anthos Architecture

4.6.3 Tea Store

The **TeaStore** is an open-source microservice reference application designed to emulate a basic web store, serving as a benchmarking framework for researchers and practitioners in the field of distributed systems. Developed to facilitate the study of performance modeling, resource management, and scalability in microservice architectures, TeaStore offers a modern and representative technology stack.

The architecture of TeaStore comprises five distinct microservices, each responsible for specific functionalities within the online store. This design promotes flexibility and scalability, allowing users to explore various aspects of microservice-based application development. The key microservices include:

- **WebUI**: Acts as the entry point for users, managing the user interface and interacting with other services to retrieve and display data. It compiles and serves Java Server Pages (JSPs) featuring categories, products, recommendations, and images. The WebUI also performs preliminary validation on user inputs before communicating with the Persistence service.
- **Image Provider**: Handles the serving and resizing of product images in various sizes for the WebUI. It utilizes a caching mechanism to optimize performance, resizing and caching images as necessary. A least-frequently-used caching strategy is employed to reduce resource demand, with response times varying depending on whether an image is already cached or requires resizing.
- **Authentication**: Manages user login and session data, employing BCrypt for login verification and SHA-512 hashes for session validation. Session data encompasses shopping cart contents, login status, and order history. The service remains stateless, as all session data is transmitted to the client.
- **Persistence**: Serves as the layer atop the database, facilitating data retrieval and storage for other services. On startup, it populates the database with initial data and provides interfaces for the Image Provider and Recommender services to access necessary information.
- **Recommender**: Generates product recommendations based on user behavior and order history. Upon startup, it connects to the Persistence service to train its recommendation model using existing data.

These microservices communicate primarily through RESTful APIs, with service discovery and load balancing managed via a custom service registry and the Netflix Ribbon client-side load balancer. This setup allows for dynamic addition and removal of service instances during runtime, supporting scalability and fault tolerance.

TeaStore's modular architecture and comprehensive documentation make it an excellent resource for evaluating performance modeling and resource management techniques in microservice environments. Its design facilitates the study of various deployment and configuration scenarios, providing valuable insights into the behavior of distributed applications.

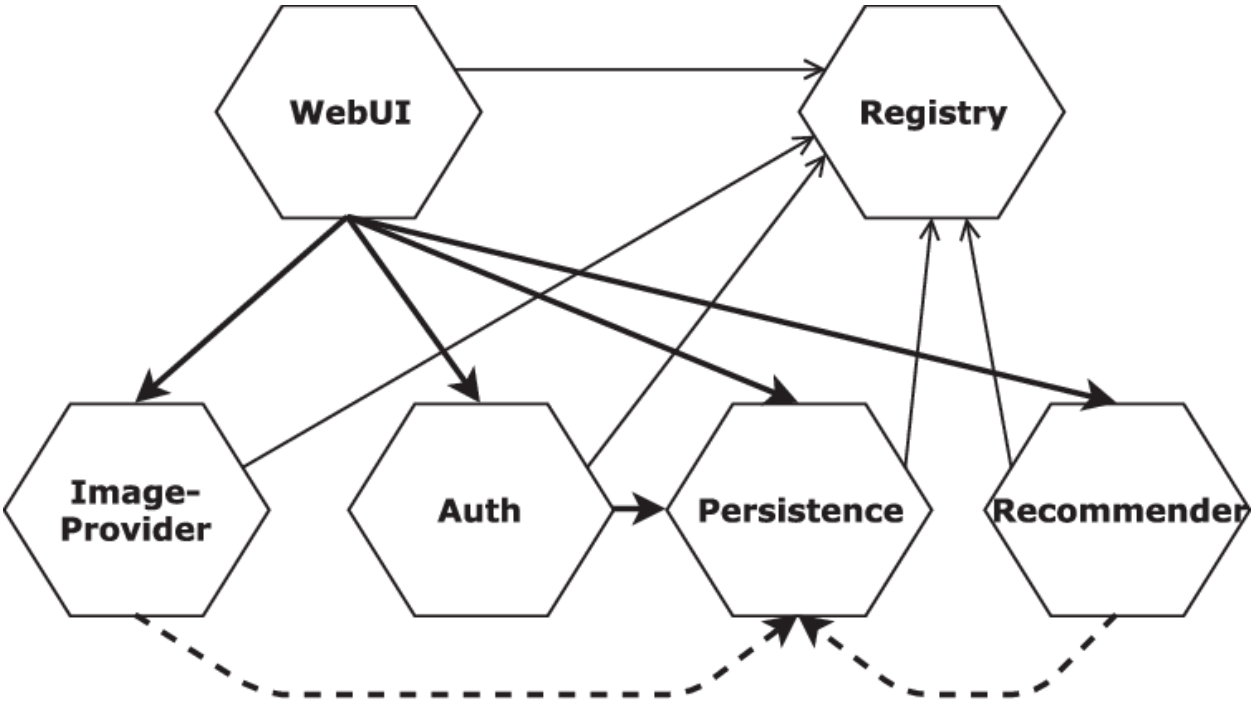

Figure 4.6 TeaStore Architecture

5 Experimental Results

This chapter presents a comprehensive assessment of the DynaQSP agent's ability to manage service placement in hybrid, multicluster, distributed directed acyclic infrastructures. The evaluation is conducted by examining the test environment, the service request plan, and the resulting data, including operational expenses, latency metrics, and the distribution of services across clusters under various load scenarios. The findings demonstrate how the Service Placement Reinforcement Learning Agent effectively streamlines service placement, lowers costs, and improves overall system performance, thereby enhancing the user experience.

5.1 Infrastructure

For the experiments, several Virtual Machines (VMs) on the Google Cloud Platform (GCP) [35] are employed to build a multi-cluster Kubernetes environment. This setup includes five VMs located in different regions, each operating as a virtual instance on GCP, which ensures scalability and flexibility for hosting applications and services.

Each VM runs K3s, a lightweight Kubernetes distribution, as a single-node cluster. K3s is selected because of its low resource usage and straightforward installation process, making it well suited for fast deployments during testing smaller scale cloud environments. **Table 5.1** outlines the main characteristics of these clusters.

Cluster Attributes	Values
VM Boot Image	debian-cloud/debian-12
Location Type	Zonal
K3s version	v1.22.16+k3s1
Horizontal Autoscaling	Disabled
Boot Disk	50GB

Table 5.1 Cluster Technical Characteristics

To replicate a Hierarchical Tier Based environment based on acyclic directed graph architecture, each cluster in our multi-cluster architecture is intentionally configured with unique attributes, such as CPU capacity, RAM, machine type, and placement across various zones, to mirror the diverse conditions found in cloud, fog, and edge computing. Each cluster represents a separate tier with distinct resource capacities and geographical locations, enabling us to effectively capture the inherent complexities and challenges of service management in these varied environments. **Table 5.2** provides a comprehensive summary of each cluster's unique characteristics.

Cluster Attributes	Cluster C1	Clusters C2 & C3	Clusters C4 & C5
Zone	europe-west2-a	europe-west3-a/b	europe-west6-a/b
Machine Type	e2-standard-8	e2-standard-4	e2-standard-4
vCPU	8	4	4
RAM (GB)	32	16	16

Table 5.2 Cluster Physical Characteristics

To create a cohesive and hierarchical structure among the clusters, we first install the Linkerd Service Mesh and its relevant extensions on every cluster within the multi-cluster architecture. The Linked Multi-cluster extension is particularly beneficial, as it enables us to connect clusters in a specific hierarchical order, forming a three-tier pyramid. The top tier (Cluster C1) represents the Central Tier, the middle tier (Clusters C2 and C3) embodies the Intermediate Tiers, and the bottom tier (Clusters C4 and C5) constitutes the User Interface layer. **Figure 5.1** provides a detailed visualization of this hierarchical arrangement and the interconnections between the clusters.

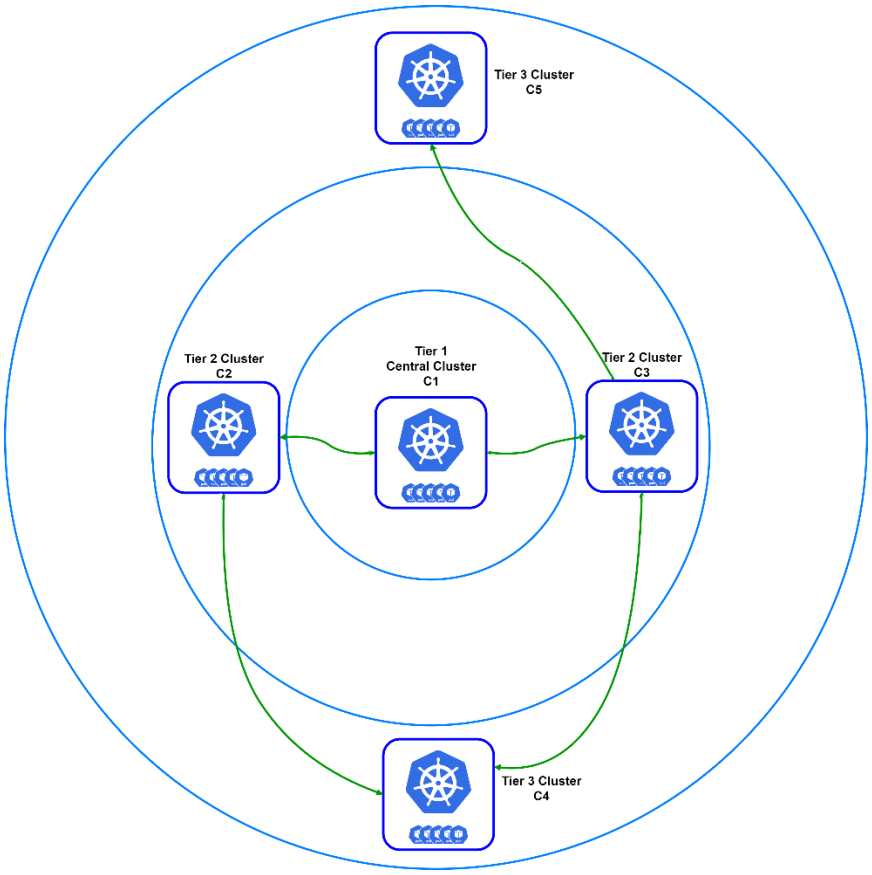

Figure 5.1 Cluster Distribution Diagram

After configuring the infrastructure, we proceeded to initialize the benchmark applications. As described in the Benchmark Applications section, our experiments utilized three microservice-based applications: Google's Online Boutique (11 microservices), Descartes Research's TeaStore (7 microservices) and Google's Bank of Anthos (8 microservices), totaling 26 microservices.

For application initialization, most microservices were deployed in the Cloud cluster to take advantage of its higher resource capacity and centralized nature. However, we took a different approach for the frontend components. The front-end microservices (both the user interface and the web application) were deployed in both clusters of the User Interface layer, as expected.

In the final phase of our experimental setup, the DynaQSP application was deployed on every cluster except the Central cluster, where its deployment was not necessary. During this stage, the frontend services were prepared to handle incoming traffic, and DynaQSP executed the required deployments and managed TrafficSplit operations across clusters.

Importantly, we ensured that certain microservices were not duplicated across clusters to avoid potential data inconsistencies. Specifically, the database services for each application (including Redis Cart, AccountsDB, LedgerDB, and TeaStoreDB) were intentionally confined to a single deployment. Similarly, the front-end services were not replicated across multiple clusters, preserving the integrity and consistency of the user interface.

5.2 Request Load Generation

This section describes the load testing process used to evaluate the performance of our benchmark application, Google's Online Boutique, Descartes Research's TeaStore, and Google's Bank of Anthos, under varying workload intensities. We employed Locust, a robust tool well-suited for simulating realistic user scenarios, to subject these applications to different load conditions. This allowed us to gain valuable insights into how each algorithm makes placement decisions and to assess the overall system performance.

We detail the load testing methodology, which includes the types of requests generated, the duration of each test, and the distribution of loads across clusters. The experimental results form the foundation for analyzing the system's efficiency in managing service placement within the described infrastructure.

To generate synthetic workloads, we implemented Python scripts using Locust that simulate user behavior and simultaneously swarm all benchmark applications with multiple concurrent users. Our strategy involved dynamically varying the number of concurrent users over time based on a periodic distribution. Specifically, the test plan ensured that requests were evenly distributed across the outermost tier clusters where the application frontends reside. These requests were selected probabilistically from a pool of available types to accurately replicate real user scenarios.

Figure 5.2 displays the user density distribution over time. **Tables 5.3 through 5.5** provide a detailed overview of all the request types used to simulate the application workflows and interactions among the microservices.

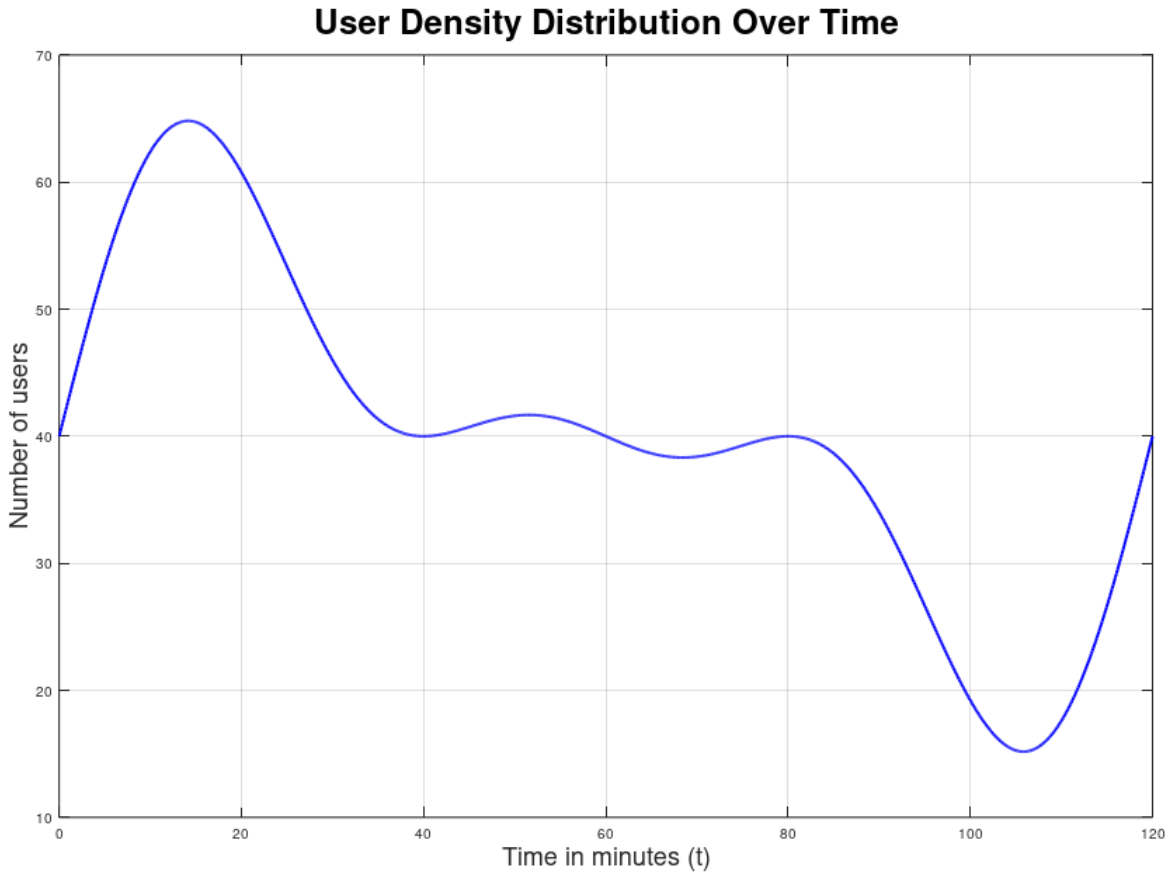

Figure 5.2 User Density Distribution

The User Density Distribution depicted above was scaled accordingly with each applications scaling capabilities, for example, Google's Online Boutique displayed higher scaling capabilities and was about to support an increased amount of requests therefore it was loaded significantly more that the other 2 applications.

Description	Type	Distribution
Navigate to Home Page	**GET**	**5.3%**
Browse Products	**GET**	**52.6%**
Add to Cart	**POST**	**10.5**
Set Currency	**POST**	**10.5%**
View Cart Contents	**GET**	**15.8%**
Checkout	**POST**	**5.3%**

Table 5.3 Google's Online Boutique Request Load Specifications

Description	Type	Distribution
Navigate to Home Page	GET	6.3%
Navigate to Login Page	GET	6.3%
Login	POST	6.3%
Browse Categories	GET	21.8%
Browse Products	GET	21.8%
Add product to cart	POST	21.8%
Checkout	POST	3.1%
Visit Profile	GET	6.3%
Logout	POST	6.3%

Table 5.4 Descartes Research's TeaStore Request Load Specifications

Description	Type	Distribution
Authenticated Tasks	--	50%
Login	POST	13.9%
View Index	GET	27.8%
View Home	GET	27.8%
Deposit Amount	POST	13.9%
Make Payment	POST	13.9%
Logout	POST	2.8%
Unauthenticated Tasks	--	50%
View Login	GET	45.5%
View Signup	GET	45.5%
Signup	POST	9.1%

Table 5.5 Google's Bank of Anthos Request Load Specification

To assess the performance of each placement strategy, we conducted a series of tests following the procedures detailed in Tables 5.3 to 5.5. These tests enabled us to record hourly operational costs, as well as measure both the average and the 95th percentile response latencies during load testing—with and without the DynaQSP agent. Additionally, we collected data on the services deployed relative to the current load intensity. This analysis provides valuable insights into the overall effectiveness of the placement strategies and enhances our understanding of the traffic splitting capabilities of DynaQSP.

5.3 Results

In our setup, we deployed five distinct single-node K3s clusters distributed across various zones. Our service placement tool, DynaQSP, leverages the Kubernetes API along with Prometheus to collect key metrics—including current CPU and RAM utilization, response latency, and network throughput—from each cluster. The placement routine is executed at regular intervals to continuously optimize resource allocation. To test the performance of each microservice placement strategy, we exposed the applications to synthetic workloads that stimulated network communication between the microservices. In the following sections, we detail the results of our reinforcement learning agent's placement decisions during different workload stages, including corresponding operational costs and latency figures measured before and after deploying the agent. These findings provide valuable insights into the performance and efficiency of our agent's decision-making process for service placement and traffic splitting in a Distributed Multi-cluster Tier Based environment.

5.3.1 Default Microservice Placement

For a start we deploy all the microservices in the Central Tier with the exceptions of the frontend applications and apply the workload mentioned above. We observe the system behavior and gather the required metrics, such as CPU and RAM utilization, Network Throughput as well as response latency, as our goal is to minimize operational costs while also ensuring acceptable or even improved response times. **Chart 5.1 – Chart 5.3** display the initial response latencies and **Table 5.6** contains a breakdown of the operational costs. **Table 5.7** shows the initial deployment distribution for all our microservices.

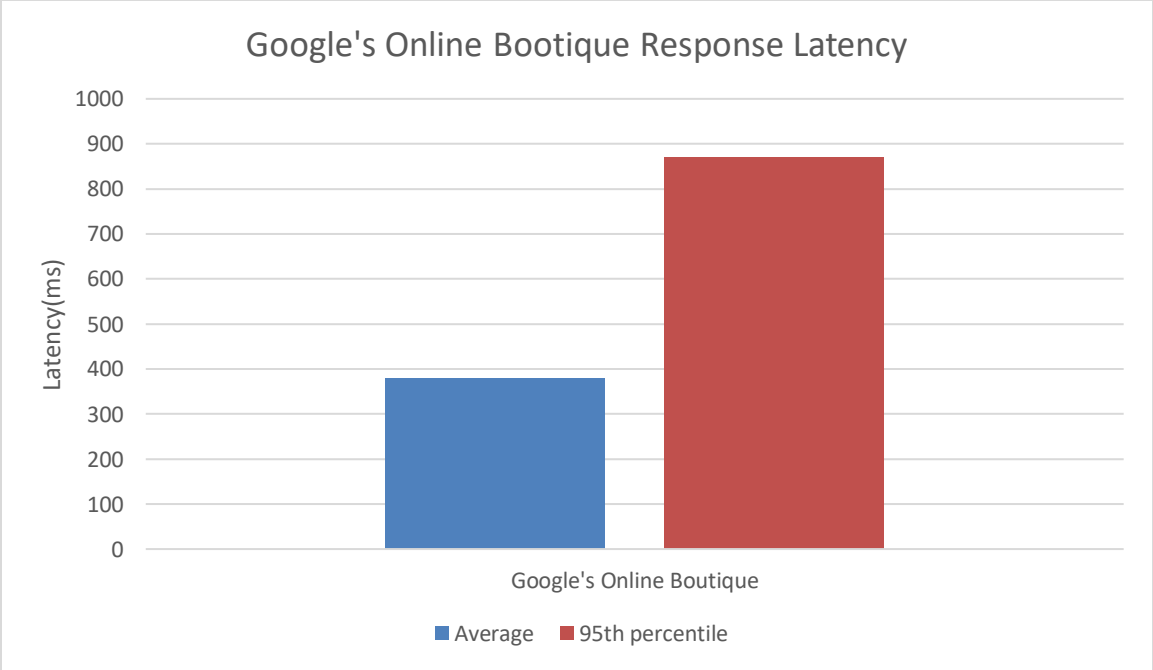

Chart 5.1 Google's Online Boutique Response Latency without DynaQSP

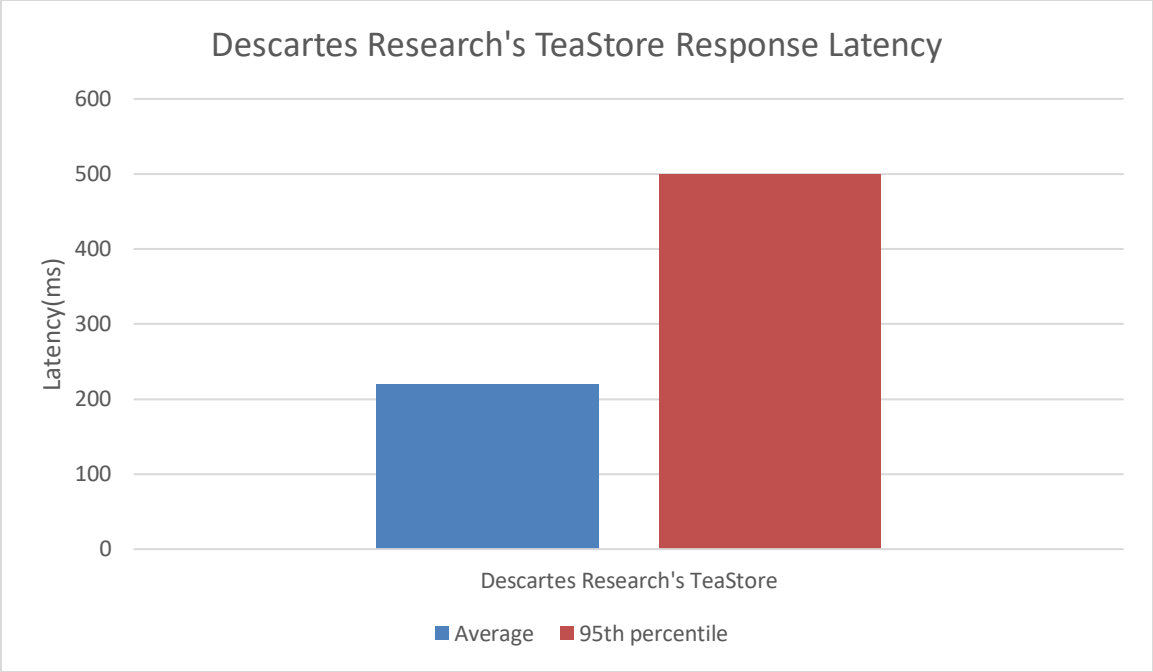

Chart 5.2 Descartes Research's TeaStore Response Latency without DynaQSP

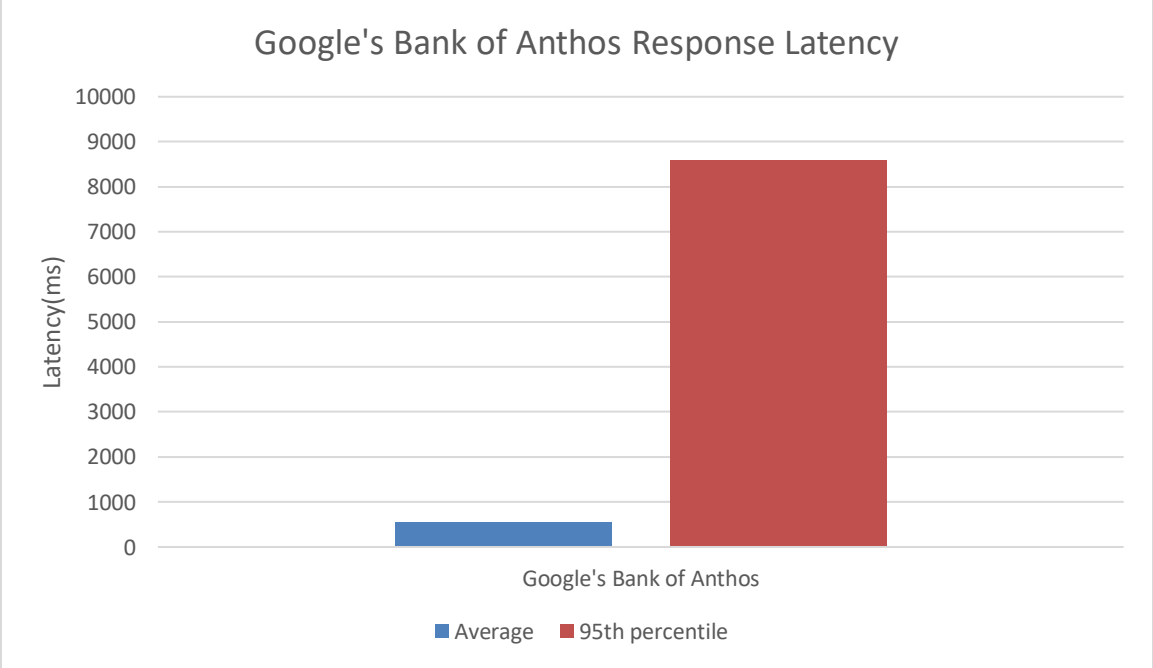

Chart 5.3 Google's Bank of Anthos Response Latency without DynaQSP

To compute the costs presented below, we used Google's Official Cloud Pricing dataset [36]. In our analysis, CPU usage is measured on a per-core basis—meaning that an average usage of 100% indicates one core was fully utilized during the test, while values exceeding 100% represent the use of additional cores. For network usage, we based our calculations on Google's Official Pricing dataset [37] and assumed an average cost of €0.05.

Per	CPU Usage (%)	RAM Usage (MB)	Network Usage (MB)	CPU cost (€)	RAM Cost (€)	Network Cost (€)
Cluster C1	130%	6668	7986	0.045	0.026	0.399
Cluster C2	0%	0	3801	--	--	0.190
Cluster C3	0%	0	8540	--	--	0.427
Cluster C4	45%	1298	3068	0.015	0.005	0.153
Cluster C5	45%	1300	6628	0.015	0.005	0.331
Total Cost	**1.611**					

Table 5.6 Operation Cost without DynaQSP

It is evident that network expenses constitute the largest portion of operational costs, so prioritizing reductions in these areas is likely to yield the most significant benefits.

	Cluster C1	Cluster C2	Cluster C3	Cluster C4	Cluster C5
deployments	accountsdb adservice balancereader cartservice checkoutservice contacts currencyservice emailservice ledgerdb ledgerwriter paymentservice productcatalogservice recommendationservice rediscart shippingservice teastoreauth teastoredb teastoreimage teastorepersistence teastorerecommender teastoreregistry transactionhistory userservice	--	--	frontend frontendbank teastorewebui	frontend frontendbank teastorewebui

Table 5.7 Service Placement without DynaQSP

5.3.2 DynaQSP Microservices Placement

After deploying our Reinforcement Learning Agent and allowing it to adapt to the environment, we gather the same information as before. We are expecting to see primarily a reduction in operation expenses (particularly in network costs) and at the least a maintenance if not an improvement to the response latency. In the Figures and Tables below, we can see the respective data that was displayed previously.

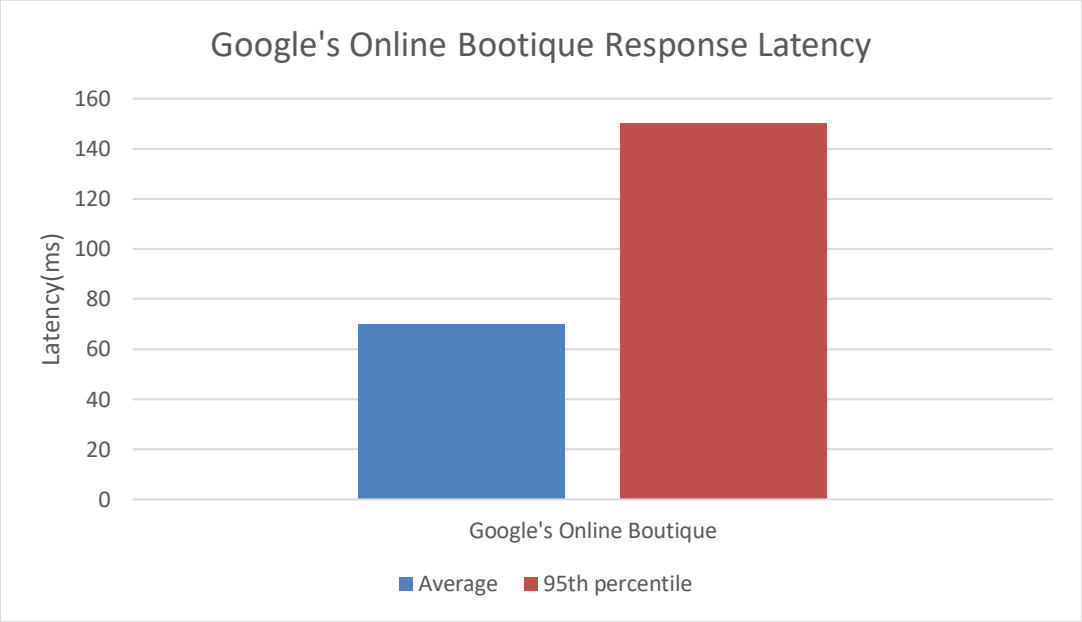

Chart 5.4 Google's Online Boutique Response Latency with DynaQSP

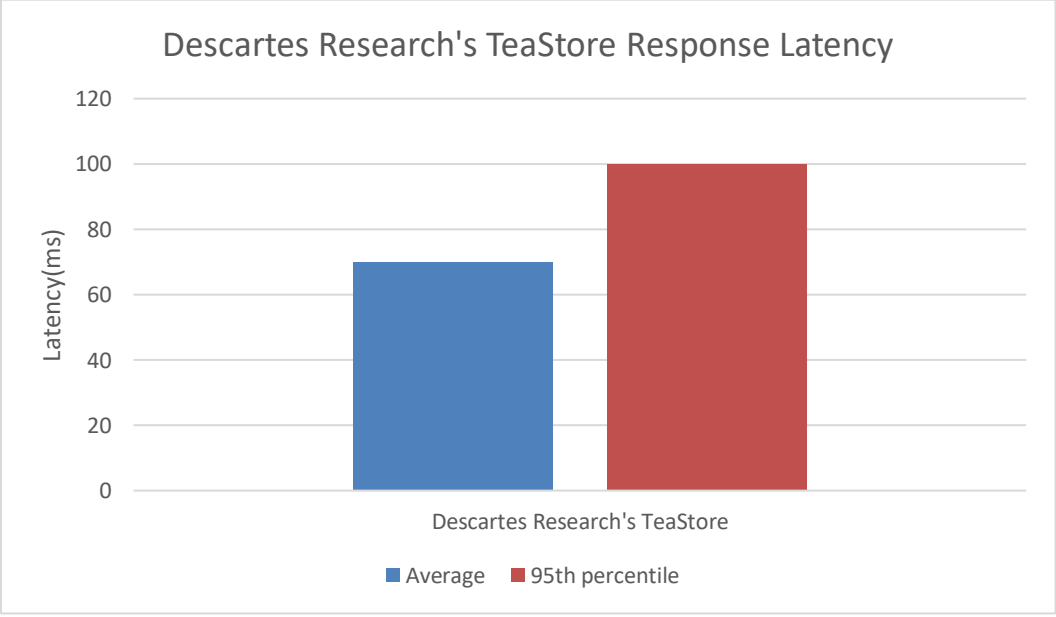

Chart 5.5 Descartes Research's TeaStore Response Latency with DynaQSP

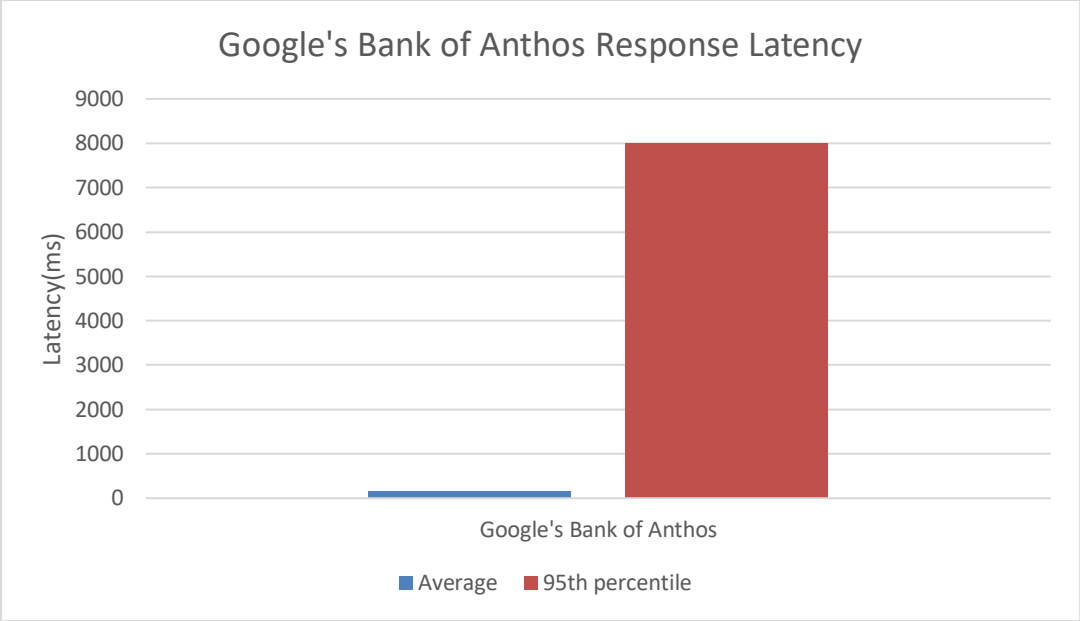

Chart 5.6 Google's Bank of Anthos Response Latency with DynaQSP

It is Important to note that this stability in the 95th percentile latency value in the Bank of Anthos application is stemming from the nature of the user-service micro-service that inherently takes a long time to respond due to its extensive communication with the database.

Per	CPU Usage (%)	RAM Usage (MB)	Network Usage (MB)	CPU cost (€)	RAM Cost (€)	Network Cost (€)
Cluster C1	68%	6945	31	0.023	0.027	--
Cluster C2	28%	4506	11	0.009	0.017	--
Cluster C3	26%	4709	11	0.009	0.018	--
Cluster C4	154%	8463	5136	0.015	0.005	0.256
Cluster C5	109%	5891	7820	0.015	0.005	0.391
Total Cost	**0.905**					

Table 5.8 Operation Cost with DynaQSP

When looking at the deployment decisions for the DynaQSP model we will ignore the already existing and unaltered deployments in the Central C1 Cluster.

	Cluster C2	Cluster C3	Cluster C4	Cluster C5
deployments	adservice balancereader transactionhistory teastorepersistence recommendationservice userservice	adservice balancereader cartservice checkoutservice contacts currencyservice emailservice ledgerwriter paymentservice productcatalogservice recommendationservice shippingservice teastoreauth teastoreimage teastorepersistence teastorerecommender teastoreregistry transactionhistory	frontend frontendbank teastorewebui balancereader checkoutservice recommendationservice teastorepersistence cartservice contacts teastoreauth	frontend frontendbank teastorewebui balancereader checkoutservice recommendationservice teastorepersistence cartservice teastoreauth ledgerwriter

Table 5.9 Service Placement with DynaQSP - High Load

	Cluster C2	Cluster C3	Cluster C4	Cluster C5
deployments	adservice balancereader transactionhistory userservice	adservice balancereader cartservice checkoutservice contacts currencyservice emailservice ledgerwriter paymentservice productcatalogservice recommendationservice shippingservice teastoreauth teastoreimage teastorepersistence teastorerecommender teastoreregistry transactionhistory	frontend frontendbank teastorewebui recommendationservice teastorepersistence cartservice teastoreauth	frontend frontendbank teastorewebui checkoutservice recommendationservice teastorepersistence cartservice teastoreauth

Table 5.10 Service Placement with DynaQSP - Low Load

5.3.3 Comparisons

Now that we have Collected all the required data from our tests it is time to compare the results. In **Table 5.11** we can see the improvement margins for the response latency.

Application	Percentile	Default	DynaQSP	
Google's Online Boutique	50th	380	70	81%
	95th	870	150	83%
Descartes Research's TeaStore	50th	220	70	68%
	95th	500	100	80%
Google's Bank of Anthos	50th	530	150	72%
	95th	8600	8000	7%

Table 5.11 Latency Improvements

Regarding operation costs, the overall reduction is calculated to be a significant **44%!!!**

5.4 Discussion

Analyzing the performance of the DynaQSP model provides valuable insights into its inner workings and environmental awareness. This understanding allows the model to adapt to system changes and implement adjustments where necessary. By discerning when the system is under stress versus when it is lightly loaded, it effectively reduces operational costs and response latency. Over time, it gains experience and refines its decision-making process, ultimately converging on a near-optimal policy. Perhaps most notably, its scalability improves as the number of microservices increases, enabling faster and more accurate information gathering.

All this wouldn't be possible without the Traffic Splitting algorithm that runs in parallel to the Reinforcement Learning Agent. By distributing incoming requests efficiently across multiple clusters, DynaQSP achieves load balancing and prevents any single service from becoming overwhelmed. This traffic-splitting capability enhances resource utilization by evenly allocating workloads and mitigating bottlenecks. In addition, it improves response times and user experience by effectively managing traffic. The feature adds an extra layer of flexibility and scalability, enabling the system to handle high volumes of requests while maintaining stability. Consequently, integrating traffic splitting into the service placement decision-making process can further boost the overall performance and responsiveness of the distributed system.

In conclusion the combined effect of these two algorithms produces significant benefits for both latency and operating costs. Reducing latency by 80%+ and operating costs by 44% is, to say the least, satisfactory.

6 Conclusion and future work

In this final chapter, we summarize the contents and findings of this Thesis and outline future work focused on refining the service placement problem and exploring advanced optimization strategies for distributed systems.

The primary goal of this study was to reduce both operational costs and response latency in a distributed, multi-cluster, multi-level environment with a directed acyclic graph architecture. This was achieved by strategically placing services and dynamically distributing traffic in response to changing system demands.

To validate our approach, we developed a comprehensive testing framework using Google Cloud Platform, Kubernetes for orchestration, and Linkerd as a service mesh. Multiple clusters with varying specifications and geographical locations were configured and interconnected to mirror the described tier-based, distributed DAG architecture.

After designing and implementing the system, we conducted tests using three microservice-based applications, Google's Online Boutique, Descartes Research's TeaStore, and Google's Bank of Anthos, to simulate real-world workloads. We collected data on system resource usage, response latency, and network traffic, comparing system performance with and without the deployment of the DynaQSP model. Ultimately, the results demonstrated significant improvements in both response times and operating costs.

While this thesis offers significant insights and optimization strategies for service placement and traffic splitting, several opportunities for future research remain.

One promising improvement is to develop a replacement-centric version of the model that compares multiple deployment options rather than evaluating a single deployment in isolation.

Additionally, enhancing the traffic splitting algorithm by incorporating reinforcement learning or another machine learning approach could improve its performance. Instead of merely reallocating resources based on predetermined metrics, such an enhancement would consider the actual impact of each action, leading to more optimal outcomes.

In the current implementation, each instance of the DynaQSP application is deployed separately within each cluster, and it only has visibility into the state of its own cluster. Future work could enhance the overall effectiveness of the placement strategies by enabling communication between DynaQSP instances across clusters. Sharing system-wide state information would provide a more holistic view of the entire distributed environment, facilitating coordinated decision-making and further optimization of service placement.

Lastly, while this thesis focuses on single-node clusters within the multicluster environment, future work should explore placement strategies for environments where each cluster comprises multiple nodes. In such settings, minimizing end-user latency remains critical, but the efficient assignment of microservices to nodes becomes equally important. Effective node-level placement can optimize resource utilization,

enhance load balancing, and reduce costs, while also decreasing network traffic by maximizing the use of available resources.

7 Bibliography

[1] C. T. Joseph, J. P. Martin, K. Chandrasekaran and A. Kandasamy, "Fuzzy Reinforcement Learning based Microservice Allocation in Cloud Computing Environments. (2019, October 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/8929586," 2019.

[2] A. F. O. a. J. Santos, "Reinforcement Learning-Driven Service Placement in 6G Networks across the Compute Continuum," 2024 20th International Conference on Network and Service Management (CNSM), Prague, Czech Republic, 2024, pp. 1-9, doi: 10.23919/CNSM62983.2024.10814365, 2024.

[3] "Aysun Aslan, Gülce Bal, Cenk Toker(2020)Dynamic Resource Management in Next Generation Networks based on Deep Q Learning. (2020, October 5). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/9302152".

[4] "Prountzos, A., & Petrakis, E. G. M. (2024). DeFog: Dynamic micro-service placement in hybrid cloud–fog–edge infrastructures. International Journal of Web and Grid Services, 20(3), 266–291. https://doi.org/10.1504/IJWGS.2024.139802".

[5] M. Goudarzi, M. A. Rodriguez, M. Sarvi and R. Buyya, "μ μ-DDRL: A QoS-Aware Distributed Deep Reinforcement Learning Technique for Service Offloading in Fog Computing Environments," in IEEE Transactions on Services Computing, vol. 17, no. 1, pp. 47-59, Jan.-Feb. 2024, doi: 10.1109/TSC.2023.3332308, 2024.

[6] "Wikipedia contributors. (2025, February 10). Proximal policy optimization. Wikipedia. https://en.wikipedia.org/wiki/Proximal_policy_optimization".

[7] "Santos, G. L., Endo, P. T., Lynn, T., & Sadok, D. (2022). A reinforcement learning-based approach for availability-aware service function chain placement in large-scale networks. Future Generation Computer Systems, 136, 93–109. https://doi.org/10.1016/j.f".

[8] N. 2. Dixitaniket (2024, "Advantage Actor-Critic (A2C) algorithm explained and implemented in PyTorch," Medium. https://medium.com/@dixitaniket76/advantage-actor-critic-a2c-algorithm-explained-and-implemented-in-pytorch-dc3354b60b50.

[9] F. Poltronieri, M. Tortonesi, C. Stefanelli and N. Suri, "Reinforcement Learning for value-based Placement of Fog Services," 2021 IFIP/IEEE International Symposium on Integrated Network Management (IM), Bordeaux, France, 2021, pp. 466-472., 2021.

[10] S. Lu, J. Wu, J. Shi, P. Lu, J. Fang and H. Liu, "A Dynamic Service Placement Based on Deep Reinforcement Learning in Mobile Edge Computing," Network. 2022; 2(1):106-122. https://doi.org/10.3390/network2010008, 2022.

[11] "Mixed-Integer Linear Programming (MILP) Algorithms. (n.d.). https://www.mathworks.com/help/optim/ug/mixed-integer-linear-programming-algorithms.html".

[12] M. Goudarzi, M. Palaniswami and R. Buyya, "A Distributed Deep Reinforcement Learning Technique for Application Placement in Edge and Fog Computing Environments," in IEEE Transactions on Mobile Computing, vol. 22, no. 5, pp. 2491-2505, 1 May 2023, doi: 10.1109/TMC.2021.3123165, 2023.

[13] "Papers with Code - IMPALA Explained. (n.d.). https://paperswithcode.com/method/impala".

[14] Y. Hao, M. Chen, H. Gharavi, Y. Zhang and K. Hwang, "Deep Reinforcement Learning for Edge Service Placement in Softwarized Industrial Cyber-Physical System," in IEEE Transactions on Industrial Informatics, vol. 17, no. 8, pp. 5552-5561, Aug. 2021, doi: 10.1109/TII.2020.3041713.

[15] "GeeksforGeeks. (2024, September 4). Reinforcement learning. GeeksforGeeks. https://www.geeksforgeeks.org/what-is-reinforcement-learning/".

[16] "Wikipedia contributors. (2025b, February 11). Markov decision process. Wikipedia. https://en.wikipedia.org/wiki/Markov_decision_process".

[17] "GeeksforGeeks. (2023, January 10). EpsilonGreedy Algorithm in Reinforcement learning. GeeksforGeeks. https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/".

[18] "Wikipedia contributors. (2025c, February 14). Softmax function. Wikipedia. https://en.wikipedia.org/wiki/Softmax_function".

[19] "GeeksforGeeks. (2025, February 12). QLearning. GeeksforGeeks. https://www.geeksforgeeks.org/q-learning-in-python/".

[20] "Wikipedia contributors. (2025a, February 8). Microservices. Wikipedia. https://en.wikipedia.org/wiki/Microservices".

[21] "K3s. (n.d.). https://k3s.io/".

[22] "Production-Grade Container orchestration. (n.d.). Kubernetes. https://kubernetes.io/".

[23] "What is a service mesh? (n.d.). https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh".

[24] "The world’s most advanced service mesh. (n.d.). Linkerd. https://linkerd.io/".

[25] "Buoyant | The next generation of networking. (n.d.). Buoyant. All of the Service Mesh. None of the

Service Mess. https://buoyant.io/".

[26] "CNCF. (n.d.). Cloud Native Computing Foundation. https://www.cncf.io/".

[27] "Architecture. (n.d.). Linkerd. https://linkerd.io/2-edge/reference/architecture/".

[28] "Linkerd. (n.d.). linkerd-smi/charts/linkerd-smi at main · linkerd/linkerd-smi. GitHub. https://github.com/linkerd/linkerd-smi/tree/main/charts/linkerd-smi".

[29] "Linkerd. (n.d.-a). linkerd2/multicluster/charts/linkerd-multicluster at main · linkerd/linkerd2. GitHub. https://github.com/linkerd/linkerd2/tree/main/multicluster/charts/linkerd-multicluster".

[30] "Linkerd. (n.d.-b). linkerd2/viz/charts/linkerd-viz at main · linkerd/linkerd2. GitHub. https://github.com/linkerd/linkerd2/tree/main/viz/charts/linkerd-viz".

[31] "Locust.io. An open source load testing tool. https://locust.io/".

[32] GoogleCloudPlatform, "Online Boutique," GitHub - GoogleCloudPlatform/microservices-demo: Sample cloud-first application with 10 microservices showcasing Kubernetes, Istio, and gRPC GitHub. https://github.com/GoogleCloudPlatform/microservices-demo.

[33] GoogleCloudPlatform, "Bank Of Anthos," GitHub - GoogleCloudPlatform/bank-of-anthos: Retail banking sample application showcasing Kubernetes and Google Cloud. GitHub. https://github.com/GoogleCloudPlatform/bank-of-anthos.

[34] DescartesResearch, "TeaStore," GitHub - DescartesResearch/TeaStore: A micro-service reference test application for model extraction, cloud management, energy efficiency, power prediction, single- and multi-tier auto-scaling. GitHub. https://github.com/DescartesResearch/TeaStore.

[35] "Cloud Computing Services | Google Cloud. (n.d.). Google Cloud. https://cloud.google.com/".

[36] "Google Cloud pricing List for CPU and Memory Usage. (n.d.). [Dataset]. https://cloud.google.com/compute/all-pricing?hl=en".

[37] "Google Cloud pricing List for Network Usage. (n.d.). [Dataset]. https://cloud.google.com/vpc/network-pricing?hl=en".