

Quality-driven request routing and adaptive monitoring for services over the computing continuum

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ignjat Karanovic, BSc

Matrikelnummer 01529940

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dr. Pantelis Frangoudis

Wien, 5. Februar 2026

Ignjat Karanovic

Pantelis Frangoudis



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Quality-driven request routing and adaptive monitoring for services over the computing continuum

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Ignjat Karanovic, BSc

Registration Number 01529940

to the Faculty of Informatics

at the TU Wien

Advisor: Dr. Pantelis Frangoudis

Vienna, February 5, 2026

Ignjat Karanovic

Pantelis Frangoudis



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Ignjat Karanovic, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 5. Februar 2026

Ignjat Karanovic



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I would like to thank my supervisor, Pantelis Frangoudis, for his guidance, valuable feedback, and continuous support throughout this thesis. His work and direction were fundamental in shaping the overall structure, methodology, and focus of the thesis, and his comments significantly improved both its technical depth and clarity.

I would also like to thank Ivan Cilic for his valuable support and insights regarding the design and practical use of the QEdgeProxy system. His guidance on deployment procedures, architectural structure, and installation workflow was instrumental in understanding and extending the original framework. I am also grateful for making QEdgeProxy available as an open-source project, which enabled in-depth analysis, experimentation, and research in this thesis.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Eine effiziente Verteilung von Anfragen in Edge- und Cloud-Computing-Umgebungen ist entscheidend für die Aufrechterhaltung einer optimalen Systemleistung. Traditionelle Load-Balancing-Ansätze basieren häufig auf statischen Konfigurationen und passen sich nicht an veränderliche Bedingungen wie schwankende Netzwerklatenzen, verfügbare Ressourcen oder unterschiedliche Workload-Verteilungen an. Dies führt zu einer ineffizienten Ressourcennutzung und einer Verschlechterung der Quality of Service (QoS). Diese Arbeit geht über bestehende statische und rein latenzorientierte Load-Balancing-Ansätze hinaus und führt den *Adaptive Score-based Routing Balancer (ASRB)* ein, einen dynamischen, scorebasierten Mechanismus zur Anfrageweiterleitung für Kubernetes-basierte Service-Deployments. ASRB berücksichtigt gemeinsam Informationen auf Infrastrukturebene, Messungen der Antwortzeit sowie anwendungsspezifische Qualitätsindikatoren für die Verarbeitung von Machine-Learning-(ML-)Workloads. Konkret wird als anwendungsspezifischer Indikator die Vorhersagegenauigkeit der eingesetzten ML-Service-Instanzen herangezogen, die gemeinsam mit der Latenz zu einem gewichteten Gesamtscore kombiniert wird. Dieser Score wird von Anfrage-Routing-Proxys genutzt, die *nativ und nahtlos* in Kubernetes integriert sind, um zur Laufzeit die geeignetste Service-Instanz auszuwählen. Für fundierte Routing-Entscheidungen sind aktuelle Informationen über den Infrastruktur- und Servicezustand erforderlich. Die Erfassung solcher Informationen erfordert jedoch das Monitoring einer Vielzahl von Laufzeitmetriken über mehrere Systemschichten hinweg, was bei großskaligen Service-Deployments zu erheblichem Netzwerk- und Verarbeitungsaufwand führen kann. Um dieser Herausforderung zu begegnen, wird ein adaptiver Monitoring-Ansatz eingeführt. Anstatt alle Knoten gleichmäßig zu überwachen, konzentriert der Proxy seine Monitoring-Aktivitäten auf Knoten und Instanzen, die mit höherer Wahrscheinlichkeit die aktuellen QoS-Ziele erfüllen, wodurch der Overhead reduziert wird, ohne die Genauigkeit der Routing-Entscheidungen zu beeinträchtigen. ASRB wird ohne Änderungen an Kubernetes implementiert und lässt sich daher einfach in bestehenden Cluster-Umgebungen deployen und betreiben. Umfangreiche Experimente in einer Kubernetes-basierten Umgebung zeigen, dass ASRB – abhängig von seiner Konfiguration – statische und latenzorientierte State-of-the-Art-Mechanismen hinsichtlich der Antwortzeit übertreffen, höhere Genauigkeit erzielen kann, wenn diese stärker gewichtet wird, oder flexible, vom Betreiber steuerbare Trade-offs ermöglicht. Gleichzeitig weist ASRB geringere Fehlerraten, höhere Reaktionsfähigkeit gegenüber Änderungen der Betriebsbedingungen sowie deutlich reduzierten Monitoring-Overhead auf.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Efficient request distribution in edge and cloud computing environments is crucial for maintaining optimal system performance. Traditional load-balancing approaches often rely on static configurations, failing to adapt to changing conditions such as varying network latency, resource availability, and workload distribution. This leads to inefficient resource utilization and degraded Quality of Service (QoS). This thesis goes beyond existing static and latency-oriented load balancing approaches by introducing the *Adaptive Score-based Routing Balancer (ASRB)*, a dynamic, score-based request routing mechanism designed for Kubernetes-based service deployments. ASRB jointly considers infrastructure-level information, response time measurements, and application-level quality indicators, with a particular focus on serving Machine Learning (ML) workloads. Specifically, and given this focus, the application-level indicator considered in this work is the prediction accuracy of the deployed ML service instances, which is combined with latency into a single weighted score. This score is used by request-routing proxies *natively and seamlessly* integrated with Kubernetes to select the most suitable service instance to serve each request at runtime. For effective routing decisions, up-to-date information is required on the infrastructure and service state. However, collecting such information involves monitoring a wide range of runtime metrics across multiple system layers, which can introduce significant traffic and processing overhead for large-scale of service deployments. To address this challenge, we introduce an adaptive monitoring process. Instead of probing all nodes uniformly, the proxy concentrates its monitoring efforts on nodes and instances more likely to satisfy the current QoS objectives, thereby reducing overhead while preserving accurate routing decisions. ASRB is implemented without requiring any modifications to Kubernetes, which makes it straightforward to deploy and operate in existing cluster environments. Extensive testbed experiments conducted in a Kubernetes-based setup show that, depending on its configuration, ASRB can outperform static and latency-oriented state-of-the-art mechanisms in terms of response time, achieve higher accuracy when this matters more, or strike favorable trade-offs in a flexible and operator-controllable way. At the same time, it does so with reduced failure rates, higher responsiveness to changes in the operating environment, and significantly lower monitoring overhead.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	ix
Abstract	xi
Contents	xiii
1 Introduction	1
1.1 Motivation	2
1.2 Problem Statement	3
1.3 Objectives and Research Question	4
1.4 Contributions	5
1.5 Thesis Structure	5
2 Background and Related Work	7
2.1 From Cloud Computing to a Cloud–Edge–IoT Continuum	7
2.2 Kubernetes and Container Orchestration	8
2.3 Machine Learning in Edge Environments	9
2.4 Load Balancing and Request Routing	11
2.5 QoS in Distributed Systems	14
2.6 Summary and Research Gap	15
3 System Design	17
3.1 Architecture Overview	17
3.2 Latency vs. Accuracy Tradeoff	20
3.3 Monitoring Mechanism Design	22
3.4 Score-Based Routing Algorithm	24
3.5 System Extensibility	28
4 Implementation	29
4.1 Codebase Overview	29
4.2 Pod Scoring and Metadata Handling	34
4.3 Routing Logic and Request Handling	35
4.4 Node Labeling and Label-Driven Monitoring	39
4.5 Deployment Architecture	41
	xiii

4.6	Challenges and Implementation Notes	43
5	Evaluation	47
5.1	Experimental Setup	47
5.2	Performance Metrics	51
5.3	Evaluation Results	54
5.4	Discussion of Results	68
6	Discussion	73
6.1	Strengths and Contributions	73
6.2	Generalizability	74
6.3	Limitations	76
7	Conclusion and Future Work	79
7.1	Summary of Findings	79
7.2	Future Research Directions	80
	Overview of Generative AI Tools Used	83
	List of Figures	85
	List of Tables	85
	List of Algorithms	87
	Bibliography	89

Introduction

In an era where cloud and edge computing technologies are becoming increasingly integral to modern applications, the need for intelligent, adaptive, and efficient service and infrastructure management is more pressing than ever. Modern services are increasingly deployed across a computing continuum spanning IoT devices, edge nodes, and cloud data centers, resulting in a highly heterogeneous and geographically distributed execution environment [DPD22, ADJJ⁺24]. As services scale to meet the demands of latency-sensitive and compute-intensive workloads, such as those found in IoT, AR/VR, and AI-driven applications, ensuring consistently high Quality of Service (QoS) presents a significant challenge. This has led to the emergence of modular and containerized service architectures, with Docker and Kubernetes standing out as the dominant platforms for container orchestration and workload management across distributed systems. Several existing approaches focus on latency-centric request routing and load balancing in distributed systems, including proxy-based solutions designed for Kubernetes environments [CJZ⁺24].

These technologies aim to abstract the complexity of distributed deployments by orchestrating service containers in a way that minimizes perceived latency, essentially providing end users with the illusion that services are running “just around the corner.” To achieve this level of responsiveness, services must be deployed across a heterogeneous, often large-scale infrastructure comprising both cloud and edge nodes. However, maintaining service quality across such an environment is non-trivial due to inherent trade-offs between key performance indicators, as well as the highly heterogeneous, variable and dynamic nature of the infrastructure, particularly towards its edge [RLF⁺20, GvKJ⁺25].

This distributed execution model is commonly referred to as the *computing continuum*, which emphasizes the tight integration of IoT devices, edge nodes, and cloud data centers into a unified and highly dynamic system [DPD22, ADJJ⁺24, BFS⁺25]. Within this paradigm, the increasing prevalence of AI-driven applications has led to the emergence of *edge intelligence*, where machine learning models are executed closer to data sources

in order to reduce latency, preserve bandwidth, and enable context-aware decision making [DZF⁺20].

Quality of Service in distributed systems is multifaceted. For example, minimizing network latency is critical for responsiveness, yet edge nodes often have limited computational capacity compared to centralized cloud data centers. In contrast, achieving high inference accuracy in AI applications may require complex models that demand more time and resources, thus increasing latency. Other relevant factors include resource availability, energy efficiency, service availability, and system scalability. These competing objectives require intelligent request routing strategies that can dynamically adapt to varying infrastructure conditions and application-level QoS constraints.

In this thesis, we investigate how to manage QoS-aware request routing for containerized services deployed across a distributed computing continuum. We propose a dynamic routing framework that evaluates real-time metrics, such as latency, model performance, and resource utilization, to make informed decisions about where to route each service request. As a concrete use case, we focus on an image recognition service employing different neural network models (e.g., MobileNet and ResNet) to illustrate trade-offs between inference speed and classification accuracy. The proposed framework is implemented and evaluated in a Kubernetes-based testbed under dynamic conditions, demonstrating its effectiveness in improving request latency, QoS compliance, and routing behavior compared to baseline load-balancing approaches.

1.1 Motivation

The motivation behind this thesis lies in the need to move beyond static request routing strategies in distributed systems to more dynamic and intelligent mechanisms that respond to real-time infrastructure and application conditions. In modern service deployments, especially in edge and cloud environments, requests are routed to one of many distributed service instances. Traditionally, these routing decisions are based on static rules or limited metrics, such as round-robin selection or basic load balancing. Although simple to implement, such approaches do not take into account dynamic factors such as network latency, system load, or application-specific quality requirements, resulting in suboptimal performance and poor QoS.

This thesis is motivated by the observation that, if we implement a monitoring mechanism capable of collecting up-to-date performance and resource metrics from all active service nodes, and we define clear QoS priorities based on user expectations (e.g., low latency, high accuracy), then it becomes possible to evaluate trade-offs at runtime and adapt routing decisions accordingly. For instance, if a user values fast response time over result precision, a lightweight model on a nearby edge node may be preferred. Conversely, if accuracy is the priority, a more powerful but distant cloud node running a high-accuracy model may be more suitable.

By making context-sensitive and QoS-aware routing decisions, the system aims to improve

the perceived quality of the service for end users. Moreover, the system accounts for the fact that some nodes may be inherently unable to satisfy certain QoS thresholds, such as latency or ML inference accuracy, under given environmental conditions. This can be due to factors such as unfavorable network placement or the deployment of ML model variants that cannot meet target accuracy levels. Identifying such cases enables the system to adapt its monitoring behavior, for example by reducing monitoring intensity for low-quality nodes, while prioritizing routing towards service instances that are more likely to meet QoS objectives. This potential for intelligent trade-off management and adaptive routing in heterogeneous environments constitutes a central motivation for the work presented in this thesis.

Finally, from a system engineering perspective, this thesis aims to provide solutions that can be applied in a straightforward manner within the standard orchestration framework for containerized applications, namely Kubernetes and its edge-centric derivatives. To this end, the design and implementation of the proposed mechanisms rely exclusively on native Kubernetes facilities and avoid any modifications to the Kubernetes core. This design choice is intended to lower the adoption barrier and ensure compatibility with existing Kubernetes deployments.

1.2 Problem Statement

Despite significant advances in cloud and edge computing, current request routing mechanisms often rely on static or narrowly scoped decision rules that fail to account for the dynamic nature of distributed environments. These approaches typically optimize for a single objective, such as minimizing latency or balancing load, without considering application-specific performance requirements or the diverse capabilities of the service nodes. In edge deployments, where resource constraints and infrastructure volatility are common, static methods often fail to adapt to rapid changes. This highlights the need for lightweight adaptive solutions that remain robust in real-world conditions.

This becomes especially problematic in AI-driven applications, where different service instances may offer varying trade-offs between accuracy and computational efficiency. For example, lightweight models may respond faster but deliver lower-quality results, while more accurate models may introduce additional processing delays. Routing mechanisms that do not account for such trade-offs risk degrading user experience or overloading critical nodes.

The core problem addressed in this thesis is how to design a request routing framework that dynamically selects service instances based on multiple QoS parameters, such as latency, model accuracy, and resource utilization, while adapting in real-time to changes in the system. The solution must operate efficiently at scale and integrate with standard orchestration platforms such as Kubernetes. Furthermore, monitoring strategies that provide sufficient observability without introducing excessive overhead must be included, particularly in resource-constrained edge environments.

1.3 Objectives and Research Question

The overarching objective of this thesis is to explore whether dynamic QoS-aware request routing can provide meaningful improvements over static routing strategies in distributed service deployments spanning the edge-to-cloud continuum. In current systems, routing decisions are often made using predefined rules or simple load-balancing heuristics, which fail to account for the dynamic nature of real-world workloads, network variability, and heterogeneous resource capabilities. Such limitations can lead to inefficiencies, degraded user experience, and under-use of the available infrastructure.

This thesis seeks to design, implement, and evaluate a routing framework that makes decisions based on real-time monitoring data from service nodes and application-specific QoS preferences. The system dynamically selects the optimal service instance for each request by considering factors such as network latency, node load, model inference time, and expected accuracy. The intention is to shift from a one-size-fits-all static routing model to a flexible, context-aware approach that adapts to changing conditions.

Another important objective is to evaluate the practicality and scalability of the proposed approach in realistic deployment scenarios. Dynamic routing introduces overhead through increased monitoring, decision complexity, and potential coordination between routing components. Therefore, the thesis also aims to determine whether the performance benefits expressed in terms of reduced latency, improved inference quality, and better SLO fulfillment justify the cost and complexity of implementing and maintaining such a system. The evaluation in this thesis focuses on a small-scale Kubernetes testbed, using controlled workloads to assess performance and overhead under realistic conditions. While the cluster size is limited, the experiments capture the relative performance differences between routing strategies. These percentage-based improvements reflect trends that would also appear in larger deployments, even though large-scale simulations are outside the scope of this work. In summary, the work carried out in this thesis aims to answer the following central research question:

Research Question: Can a dynamic, QoS-driven request-routing framework guided by real-time infrastructure and application-level metrics significantly outperform static routing methods in terms of latency, accuracy, service-level compliance, and monitoring overhead? In particular, does the adaptive monitoring mechanism reduce API request load and monitoring payload volume? Finally, are the resulting improvements sufficient to justify the additional complexity and overhead in larger scale deployments?

Ultimately, the research aims to provide actionable insights into the trade-offs between routing complexity, system overhead, and QoS gains, contributing to a broader understanding of adaptive workload management in distributed computing environments.

1.4 Contributions

The main contribution of this thesis is the design and implementation of the *Adaptive Score-based Routing Balancer (ASRB)*, a dynamic, QoS-aware request-routing algorithm. ASRB builds on the QEdgeProxy load balancer [CJZ⁺24], a latency-oriented request routing scheme for edge-centric Kubernetes deployments, but redesigns its routing and monitoring logic. Unlike traditional static routing methods, this algorithm enables adaptive decision-making by combining real-time latency measurements, model-specific accuracy, and resource awareness.

In addition, the thesis introduces an adaptive monitoring mechanism that selectively adjusts the intensity of infrastructure monitoring based on the likelihood that nodes meet QoS thresholds. This approach significantly reduces monitoring overhead in large-scale deployments while maintaining sufficient observability for accurate routing decisions.

Together, these components provide a flexible and extensible solution capable of adapting to diverse application requirements and varying infrastructure conditions.

1.5 Thesis Structure

This thesis is organized into the following chapters:

Chapter 1 – Introduction: Provides an overview of the research context, motivation, problem statement, objectives, and key contributions.

Chapter 2 – Background and Related Work: Reviews relevant concepts in cloud and edge computing, Kubernetes, load balancing techniques, and existing QoS-aware routing approaches.

Chapter 3 – System Design: Describes the architecture of the proposed dynamic routing solution, including the scoring mechanism and adaptive monitoring strategy.

Chapter 4 – Implementation: Details how ASRB is implemented and integrates with Kubernetes, describing the internal workings of its system components, the introduced data structures, and its routing and monitoring logic.

Chapter 5 – Evaluation: Presents the experimental setup and results, comparing the proposed solution to a number of request routing approaches from the state of the art.

Chapter 6 – Discussion: Analyzes the implications of the results, discusses the strengths and limitations of the approach, and highlights areas for improvement.

Chapter 7 – Conclusion and Future Work: Summarizes the thesis outcomes and outlines directions for further research and system enhancements.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Background and Related Work

This chapter provides a foundational background and an overview of related work relevant to this thesis. It begins by discussing the evolution and interplay between edge and cloud computing, followed by an overview of container orchestration with Kubernetes. The role of machine learning in edge environments is examined, with an emphasis on inference trade-offs. We then explore traditional and modern load balancing strategies and Quality of Service considerations in distributed systems. The chapter concludes with a summary that identifies gaps in the current literature that this thesis aims to address.

2.1 From Cloud Computing to a Cloud–Edge–IoT Continuum

Cloud computing refers to the delivery of computing resources, such as storage, processing power, and applications, over the Internet, allowing users to access services remotely rather than hosting them locally. According to the National Institute of Standards and Technology (NIST), cloud computing is “*a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources*” [MG11]. Instead of maintaining dedicated on-premises servers, organizations can rely on remote data centers provided by platforms such as AWS, Azure, or Google Cloud. This model enables scalability, cost efficiency, and ease of maintenance.

From the perspective of a cloud user, cloud computing alleviates the need to maintain on-premises infrastructure by enabling centralized management, elastic resource scaling, and high service availability. At the same time, organizations benefit from simplified maintenance and the ability to access services globally without direct involvement in infrastructure operation.

However, current and next-generation data-driven IoT applications increasingly expose the limitations of a purely cloud-centric service engineering and delivery paradigm.

Emerging AI-driven IoT applications, such as autonomous mobility, augmented and extended reality, and industrial control systems, operate under stringent latency requirements that necessitate placing computation close to where it is needed. Hosting service components exclusively in centralized cloud data centers may therefore be prohibitive due to increased network latency caused by physical distance. Moreover, while cloud computing reduces management overhead, it also implies reduced direct control over the underlying infrastructure. Privacy constraints and regulatory requirements further limit the applicability of cloud-based processing for sensitive data. These limitations, together with technological advances at the device and network edge, such as mobile AI accelerators and high-speed connectivity, have led to the emergence of edge computing as a complementary paradigm [ADJJ⁺24, DPD22].

To mitigate some of these limitations, edge computing emerged as a complementary paradigm. Edge computing moves data processing and storage closer to end users—either on-site or in regional edge nodes. This significantly reduces round-trip latency and improves responsiveness, especially for latency-sensitive applications such as real-time video analytics, autonomous vehicles, and industrial automation [Sat17]. Cloud and edge computing often coexist in a computing continuum, where services are strategically distributed to balance latency, performance, and resource efficiency [TGD25].

2.2 Kubernetes and Container Orchestration

As modern applications become increasingly complex, the need for efficient, scalable, and portable deployment mechanisms has become essential. This shift gave rise to the use of containers, which are lightweight, portable software units that package an application together with its dependencies, libraries, and configuration, ensuring that it runs consistently across different environments [CI20, PBSJ19]. Unlike traditional virtual machines, containers are more resource-efficient, start faster, and are better suited to microservices architectures that decompose systems into loosely coupled, independently deployable components [QSDE24].

The containerization movement was largely popularized by the emergence of Docker in 2013, which simplified the creation and management of containers. While Docker provided the tooling to package and run containers, it did not address the operational complexity involved in deploying containers at scale across clusters of machines. As applications moved into production, new challenges emerged: containers needed to be scheduled, scaled, monitored, restarted upon failure, and connected to one another securely and efficiently.

To address this need, the concept of container orchestration emerged [Kha17]. Container orchestration refers to the automated management of container lifecycles, including deployment, scaling, networking, and availability. Several orchestration platforms were developed over time, including Docker Swarm, Apache Mesos, and Kubernetes—the latter quickly becoming the industry standard due to its flexibility, strong open-source

community, and backing by Google, which originally developed it based on their internal Borg system [Clo23].

Kubernetes, first released in 2014 and donated to the Cloud Native Computing Foundation (CNCF), is a powerful open-source platform designed to automate the deployment, scaling, and management of containerized applications. It abstracts away the underlying infrastructure and allows developers to declare the desired state of their system, which Kubernetes then ensures is met through its control loops. Kubernetes manages clusters of machines (nodes), deploys containerized applications onto them as pods, and provides services such as service discovery, load balancing, rolling updates, and self-healing.

Furthermore, Kubernetes plays a central role in enabling edge computing, especially through lightweight distributions like k3s, which reduce the resource overhead needed to run Kubernetes on constrained devices [Fou23]. This compatibility makes it feasible to deploy Kubernetes clusters that span from small, decentralized edge nodes to powerful cloud data centers.

In the context of this thesis, Kubernetes is a foundational component, as the proposed dynamic request routing algorithm is built on top of an existing Kubernetes-based load balancer, namely QEdgeProxy [CJZ⁺24]. Kubernetes provides the abstraction and operational mechanisms needed to manage the distributed service instances that make up the edge-cloud deployment. This involves mechanisms to dynamically deploy, monitor, and scale application components, so that these instances, whether running on cloud VMs, local edge devices, or Raspberry Pis, are orchestrated ensuring consistency and observability across the infrastructure. At the same time, Kubernetes exposes APIs that the routing layer (QEdgeProxy and ASRB, devised in this thesis) can query to retrieve real-time cluster state and service metadata, including pod health and resource usage information. This interaction is crucial to enable real-time routing decisions based on latency, model accuracy, and infrastructure state. ASRB uses QEdgeProxy as its code base, but redesigns its routing and monitoring logic to support multi-dimensional QoS-aware decision making at a reduced monitoring cost. Notably, like QEdgeProxy, ASRB is implemented as a Kubernetes-native component that runs on each node, making this approach compatible with existing Kubernetes deployments without modifications.

In summary, Kubernetes is not only a tool for operational convenience; it is an enabler of the computing continuum. By providing a consistent orchestration layer across heterogeneous environments, Kubernetes empowers the design of intelligent systems that are location-aware, failure-resilient, and capable of responding to real-time performance metrics. This orchestration backbone underpins the dynamic routing framework proposed in this thesis.

2.3 Machine Learning in Edge Environments

The growing demand for real-time, intelligent applications has driven the integration of machine learning (ML) into edge computing environments. This convergence of edge

computing and machine learning is commonly referred to as *edge intelligence*, which focuses on the execution of ML inference near data sources in a resource-aware manner to meet latency, bandwidth, and privacy requirements [DZF⁺20]. From smart cameras and industrial automation to health monitoring and augmented reality, many modern systems now rely on ML models for tasks such as classification, detection, and prediction. Performing inference directly at the edge—close to where data is generated—enables faster response times, reduced bandwidth usage, and improved data privacy. However, deploying ML models in edge environments introduces significant challenges, particularly due to the limited computational resources and energy constraints of edge devices. For example, resource limitations and device diversity challenge vertical and horizontal scaling, ML model placement algorithms face complex optimization landscapes, while the distribution of functionally equivalent (but potentially with varying inference quality) ML model instances over nodes with varying capabilities complicates ML inference request routing.

A core trade-off in edge-based ML is between accuracy and efficiency. While powerful models can offer high accuracy, they often require substantial memory, compute, and energy, resources that edge nodes like Raspberry Pis or mobile devices cannot always provide. However, lightweight models can operate efficiently on such devices, but may deliver lower prediction quality. Therefore, selecting the appropriate model at runtime based on the current state of the infrastructure and the application-level QoS requirements is a key part of dynamic routing in distributed intelligent systems [DZF⁺20].

This thesis uses image classification as a representative ML workload to demonstrate how dynamic QoS-aware request routing can balance latency and model performance. In particular, it evaluates two widely adopted convolutional neural network (CNN) architectures: MobileNet and ResNet50. These models serve as practical examples of the latency-accuracy trade-off present in ML deployment decisions.

MobileNet is designed for lightweight inference on resource-constrained devices. It achieves reasonable accuracy with a small model footprint and reduced computational overhead [HZC⁺17], making it suitable for real-time edge scenarios where latency and efficiency are prioritized.

ResNet50, by contrast, is a deeper and more complex model capable of achieving higher classification accuracy [HZRS16]. However, it requires significantly more processing power and inference time, making it better suited to nodes with stronger computational capabilities or to applications where accuracy is prioritized over speed.

The selection between these two models illustrates the broader architectural decisions that edge systems must make. By incorporating runtime awareness of each model's characteristics and the current state of the infrastructure, the dynamic routing system proposed in this thesis can make informed request routing decisions for ML workloads. In particular, it enables routing requests to service instances that are more likely to meet both latency and accuracy requirements under current conditions.

The performance characteristics of MobileNet and ResNet50 have been well-documented

and are consistent with their intended use cases in edge and cloud scenarios [HZC⁺17, HZRS16]. These models not only serve as benchmark architectures in this thesis but also reflect real-world deployment challenges for intelligent edge systems.

The qualitative comparison is independent of a specific hardware platform and is intended to reflect general architectural trade-offs rather than absolute performance on a particular edge device.

Table 2.1: Qualitative comparison of MobileNet-V3 and ResNet-50

Feature	MobileNet-V3	ResNet-50
Model size	Smaller (~ 21 MB)	Larger (~ 100 MB)
Inference latency	Lower	Higher
Accuracy	Lower	Higher
Edge suitability	High	Limited
Use-case focus	Real-time	Accuracy-critical

Table 2.1 provides a qualitative comparison of the two models considered in this work. The comparison is based on the architectural characteristics and intended deployment scenarios of MobileNet-V3 and ResNet-50, as documented in the original model publications [HZC⁺17, HZRS16].¹

More recent and accurate models such as Vision Transformers (ViT) [DBK⁺21] or Swin Transformers [LLC⁺] exist, but a systematic comparison of different architectures is outside the scope of this thesis, which focuses on the latency-accuracy trade-off.

2.4 Load Balancing and Request Routing

In distributed systems, particularly those spanning heterogeneous cloud and edge infrastructures, effective load balancing is critical for ensuring performance, reliability, and efficient resource utilization. Load balancing refers to the process of distributing incoming service requests across multiple service instances or nodes in a manner that prevents overloading any single component while maintaining acceptable levels of QoS.

Traditionally, load balancing strategies have relied on simple, stateless heuristics such as round-robin, least connections, or random selection. These techniques are lightweight and easy to implement but do not account for real-time infrastructure conditions or application-specific requirements. For example, they may direct traffic to geographically distant or already overloaded nodes, resulting in increased latency and degraded service quality [FP19].

To address these limitations, more advanced state-aware or adaptive routing methods have been proposed. These techniques incorporate runtime metrics such as CPU usage,

¹The reported model size values correspond to the PyTorch implementation of MobileNet-V3-Large and Resnet-50, serialized using the `torch.save()` function. The corresponding values for MobileNet-V3-Small and Resnet-152 were measured to be approximately 9.8 MB and 230 MB, respectively.

memory load, or request latency into the routing decision-making process. However, most still focus primarily on infrastructure-level indicators and fail to account for application-layer characteristics such as inference time, model accuracy, or energy consumption. These factors are especially relevant in AI-driven and edge computing scenarios [NPK22].

A key distinction also exists between static and dynamic routing approaches. Static routing relies on predefined rules or configurations that remain fixed regardless of changes in the system. While this provides predictability, it often leads to inefficiencies in dynamic environments where network conditions, resource availability, and workload profiles are constantly shifting. In contrast, dynamic routing adapts in real time, using continuous monitoring to respond to changing infrastructure and application conditions, making it more suitable for volatile edge-cloud deployments.

In this thesis, we refer to routing strategies such as round-robin, random selection, and proximity-based routing with fixed or infrequently updated latency estimates as static [FP19, CJZ⁺24]. While some proximity-aware approaches incorporate latency measurements, these are often refreshed at coarse time scales and do not adapt per request. In contrast, ASRB performs dynamic routing by continuously updating latency estimates, incorporating application-level metrics, and adapting routing decisions at runtime.

In Kubernetes-based environments, load balancing is typically handled by the built-in component kube-proxy, which distributes requests using basic round-robin or random strategies at the service level [Clo23]. Although sufficient for simple deployments, this approach lacks flexibility and performance awareness. More advanced solutions like service meshes (e.g., Istio²) offer customizable routing rules and observability but often introduce operational overhead through sidecar proxies and centralized control planes, making them less practical for lightweight or resource-constrained edge deployments [ZSX⁺23].

Routing decisions become even more critical in QoS-aware systems, where service-level objectives (SLOs) must be met for each request. In such systems, it is not enough to route traffic to any available node; the selected node must also be able to satisfy specific constraints such as maximum response time or minimum inference accuracy. This introduces the need for routing mechanisms that can account for both system-level and application-level metrics.

2.4.1 CPU Load-Aware Scheduling: DWRR and ProxyDWRR

Beyond latency- and proximity-based strategies, several works focus on CPU load-aware scheduling at the edge. A representative example is the adaptation of Deficit Weighted Round Robin (DWRR) and its proxy-based variant ProxyDWRR for Kubernetes clusters [WRY⁺22]. In these schemes, flows or requests are assigned to servers according to weighted quanta that are dynamically adjusted based on current CPU utilization. The key idea is to extend classic DWRR with load feedback so that heavily utilized servers receive fewer new requests, while lightly loaded ones are favored.

²<https://istio.io/latest/docs/>

ProxyDWRR introduces an intermediary proxy node that aggregates CPU-load information and applies DWRR-based scheduling on behalf of backend servers. This reduces per-node state and allows centralised control over CPU-aware request distribution. Experimental results show that ProxyDWRR reduces response times under high load and improves resource utilisation compared to naive round-robin policies [WRY⁺22].

Conceptually, these approaches share similarities with ASRB’s resource-awareness: both use CPU load to avoid overloaded nodes. However, DWRR-based methods treat CPU load as the primary (or sole) decision factor and typically ignore model-specific QoS attributes such as accuracy or per-model latency. In contrast, ASRB combines latency, model accuracy, and resource metrics into a unified score and propagates overload information via Kubernetes labels. Unlike centralised ProxyDWRR scheduling, ASRB uses decentralised proxies that consume shared label state, enabling richer QoS trade-offs (e.g., latency vs. accuracy) at the cost of a more complex scoring pipeline. However, this complexity primarily relates to the monitoring process; the additional overhead introduced by routing decisions on the “hot path” (i.e., per request) is negligible, as confirmed by our evaluation results. This contrasts with approaches based on Deep Reinforcement Learning (Section 2.4.2), which, aside from state management and training overheads, unavoidably require neural network execution for each request.

2.4.2 Reinforcement Learning–Based Load Balancing

Another line of research uses reinforcement learning (RL) to learn load-balancing or routing policies directly from experience, instead of hand-crafting scoring functions. Representative examples include deep-RL schedulers for Kubernetes that model the load balancer as an agent, the cluster state as the environment, and routing decisions as actions with rewards based on latency, throughput, or SLA compliance [SWDTS24]. Similar approaches use reinforcement learning for latency-aware request routing in fog and edge computing environments [KFDS23, TW20]. Through exploration and reward feedback, RL-based systems can discover complex policies that capture multi-dimensional trade-offs and non-linear system dynamics.

RL approaches are particularly attractive in highly dynamic or non-stationary environments where analytical models are difficult to derive. Recent work shows that RL-based load balancers outperform static heuristics and classical algorithms such as round robin or least-connections under bursty or heterogeneous workloads [SWDTS24]. However, practical deployment faces several challenges: training requires large data or simulations, reward design is non-trivial, and online exploration raises concerns about system stability and SLA violations. Importantly, it is sometimes unclear how specific RL methods can handle changes in state space structure as a result of changes in a deployment and the underlying infrastructure (e.g., nodes joining or leaving), which may happen often in practical computing continuum settings.

Compared to these approaches, ASRB adopts a simpler and more transparent design. It uses an interpretable scoring function that combines latency and model accuracy via a

tunable parameter λ , making operator intent explicit and avoiding the need for offline or online training. Nevertheless, ideas from RL, such as adapting parameters based on long-term QoS feedback, could complement ASRB in future iterations.

2.5 QoS in Distributed Systems

QoS is a fundamental concern in distributed systems, as it directly impacts user satisfaction, system responsiveness, and operational efficiency. QoS refers to the system's ability to meet predefined performance targets related to key metrics such as latency, throughput, availability, and—in the context of AI applications—accuracy. These targets are often formalized as Service Level Objectives (SLOs), which define the acceptable thresholds for service behavior.

SLOs are increasingly used not only as specification mechanisms but also as first-class inputs to scheduling and resource management decisions. For example, the Polaris scheduler introduces SLO- and topology-aware microservice scheduling at the edge, explicitly using latency and performance objectives to guide placement and scaling decisions across heterogeneous infrastructures [PNM⁺22].

In most distributed systems, SLOs serve as operational expectations [NMP⁺20]. As illustrative examples, interactive applications could require end-to-end response times under 100 milliseconds, while inference services might target a minimum classification accuracy of 75% to meet usability standards. Failing to meet such objectives can degrade user experience.

In cloud-centric deployments, maintaining QoS is typically addressed through autoscaling, overprovisioning, and traffic shaping, often within relatively homogeneous infrastructures. However, in edge-cloud continuum environments, these challenges are amplified [DPD22, ADJJ⁺24]. Edge nodes operate under resource constraints (limited CPU, memory, and power) and face fluctuating network conditions. Additionally, the heterogeneity of devices and dynamic workload distribution introduce further complexity in maintaining consistent QoS.

To address these issues, QoS-aware systems must make context-sensitive decisions that account for both infrastructure and application requirements. For example, selecting an edge node with minimal latency might be beneficial, but if that node hosts a lower-accuracy model, the decision might conflict with an application's accuracy constraint. This introduces trade-offs that must be intelligently managed by the routing and orchestration mechanisms.

In AI-powered systems, QoS expands beyond infrastructure metrics to include model-level characteristics, such as inference time, prediction accuracy, and model size. The selection between models like MobileNet and ResNet50, for instance, inherently represents a trade-off between latency and precision: MobileNet offering faster but less accurate predictions, and ResNet50 providing higher accuracy at the cost of increased inference time.

2.6 Summary and Research Gap

The integration of cloud and edge computing has enabled the deployment of distributed, latency-sensitive services at scale. Container orchestration platforms like Kubernetes have further simplified the management of such services across heterogeneous infrastructures. However, as shown in the reviewed literature, many existing systems fall short in delivering true QoS awareness, especially in environments characterized by resource heterogeneity, fluctuating network conditions, and AI-driven workloads.

Traditional load balancing techniques such as round-robin or least-connections, commonly used in Kubernetes via components like `kube-proxy`, are stateless and agnostic to runtime conditions [Clo23]. While simple and scalable, these approaches do not consider real-time infrastructure state or application-specific requirements, leading to potential QoS violations in dynamic environments.

Recent research has introduced more advanced strategies. Proxy-mity by Fahs and Pierre [FP19] proposes proximity-aware routing based on network latency, offering significant performance improvements over static routing. Similarly, Resource Adaptive Proxy (RAP) by Nguyen et al. [NPK22] incorporates resource metrics into routing decisions. However, both systems are still limited in scope: they focus on infrastructure-level metrics (e.g., latency, CPU load) and do not account for application-layer QoS metrics, such as inference accuracy or model-specific performance trade-offs, which are critical in AI-centric services.

The QEdgeProxy framework [CJZ⁺24] takes a notable step forward by introducing a Kubernetes-native, latency-aware request-routing proxy for IoT services. While it efficiently balances load and meets basic latency-based QoS goals, it remains limited to latency-only optimization and does not address deeper trade-offs between latency and model performance. Moreover, its monitoring strategy applies uniform intensity across the cluster, which can lead to scalability issues in large deployments.

From the reviewed work, we identify the following key research gaps:

- Most current request routing solutions do not jointly consider both infrastructure-level and application-level QoS parameters.
- There is a lack of dynamic routing mechanisms that account for inference accuracy, which is critical in ML-based services.
- Existing monitoring strategies are often non-adaptive, leading to unnecessary overhead in large-scale or resource-constrained edge deployments.
- Many advanced routing solutions (e.g., Istio, service meshes) introduce high overhead, making them unsuitable for lightweight edge computing scenarios.

This thesis addresses these gaps with a dynamic, QoS-driven routing mechanism that integrates both infrastructure metrics (e.g., latency, CPU load) and model-aware char-

2. BACKGROUND AND RELATED WORK

acteristics (e.g., accuracy). It further introduces an adaptive monitoring strategy that adjusts its intensity based on the likelihood of a node meeting QoS thresholds, improving scalability and efficiency in edge-cloud environments.

System Design

This chapter presents the design of a dynamic, QoS-aware request-routing system tailored for distributed edge-cloud environments. The system addresses the limitations of traditional static or infrastructure-only routing mechanisms by integrating application-level performance metrics, such as model inference accuracy, when enabled by the routing configuration. It operates on top of a Kubernetes-managed infrastructure and introduces a modular scoring-based routing algorithm that balances latency-accuracy trade-offs in real time.

The design emphasizes scalability, lightweight deployment, and adaptability to diverse runtime conditions, with special focus on resource-constrained edge environments. In addition to the core routing logic, this chapter describes the architectural layout, the motivation behind latency-accuracy balancing, and the adaptive monitoring approach used to minimize overhead while maintaining accurate routing decisions.

Early design drafts explored the idea of a “dynamic perimeter,” where only a subset of promising nodes would be monitored more intensively. During implementation, this idea was simplified into the final label-driven adaptive monitoring strategy, which is lightweight, practical, and fully sufficient for guiding the routing decisions in ASRB.

3.1 Architecture Overview

This section presents the architecture of the proposed system for dynamic, QoS-aware request routing in distributed environments that span both cloud and edge infrastructure. The system is designed to operate within a Kubernetes-managed cluster and aims to make intelligent routing decisions based on both infrastructure-level and application-level metrics. These include real-time latency measurements, model inference accuracy, and resource utilization, enabling the system to adaptively select the most suitable service instance for each incoming request.

The system is implemented as a lightweight Kubernetes-native proxy component that runs on each node and operates in close coordination with the Kubernetes control plane. While the architecture draws technical inspiration from existing Kubernetes-based routing solutions, its design is centered around the novel integration of scoring-based decision logic, adaptive monitoring mechanisms, and ML model-aware routing, all tailored for scalable and latency-sensitive deployment scenarios.

3.1.1 System Components

The architecture consists of five key components:

1. **Client Devices** Clients initiate service requests (typically HTTP-based) originating from IoT devices, edge sensors, or user applications. These clients are unaware of the internal service topology and connect to a local proxy deployed on the same or a nearby node.
2. **Node-Local Routing Proxy** The routing proxy is deployed on each node as a DaemonSet, a Kubernetes construct that ensures exactly one instance of a pod runs on every node in the cluster. This deployment choice enables node-local request handling, reduces additional network hops, and avoids a centralized routing bottleneck. Alternative deployment options, such as a centralized proxy or a replicated deployment behind a service, were considered but would either introduce additional latency or create single points of congestion.

The routing proxy is responsible for some key tasks: (i) scoring of service pods (see below) based on the latency and model accuracy they offer, (ii) dynamic pod discovery and health checking, (iii) instrumenting the adaptive monitoring of nodes, and (iv) fallback request routing when necessary, which allows the proxy to select alternative service instances when the initially preferred target is unavailable or does not satisfy current QoS constraints.

3. **Service Pods** Each pod exposes a model inference service (e.g., image classification). Pods may run different models that are functionally equivalent, meaning they handle the same task and accept the same input, but exhibit different performance characteristics and resource requirements. Typical examples include MobileNet and ResNet50 for image classification, which differ in terms of inference latency, accuracy, and computational demand. These metadata are used for score calculation and QoS-aware service selection.
4. **Kubernetes Control Plane** Provides pod discovery, label and annotation access, and resource state. The routing proxy interfaces with the Kubernetes API to retrieve lists of eligible pods and updates internal caches via informers.
5. **Monitoring Subsystem** The system gathers both passive (CPU/memory from Kubernetes Metrics API) and active (latency via HTTP probes) data. A key

design contribution is the *adaptive monitoring strategy*, which reduces overhead by focusing on nodes and pods more likely to satisfy QoS thresholds.

3.1.2 Deployment Model

The system runs within a Kubernetes cluster that may span cloud VMs and edge devices (e.g., Raspberry Pi, Jetson Nano). Each node hosts a single routing proxy instance, which makes local, stateless routing decisions per request. This design minimizes inter-node coordination and ensures low overhead, making it suitable for latency-sensitive and resource-constrained environments.

Pod discovery, score computation, and monitoring are executed asynchronously. The architecture avoids global coordination, relying instead on continuously updated local views of the system. Metadata such as pod labels and annotations are used to filter and score service instances dynamically.

3.1.3 Proxy Deployment via DaemonSet

Each node in the cluster runs exactly one ASRB proxy instance, deployed as a DaemonSet. This ensures that routing always happens locally, in line with the following key premises:

- Requests originating on a node are routed by its own proxy.
- Proxies never forward traffic to each other.
- No global coordinator or centralized routing layer is required.

This design minimizes latency, avoids single points of failure, and reduces cross-node traffic. The routing proxy is deployed using a Kubernetes DaemonSet, whose configuration specifies environment variables controlling routing behavior, provides access to the Kubernetes API through a mounted configuration file, and exposes network ports for handling client requests as well as for exporting monitoring data. Monitoring information is made available through a metrics endpoint, which is consumed by Prometheus [RV15], a widely used monitoring system in Kubernetes-based environments.

Because each proxy instance maintains its own local caches of pods, latency, and node metrics, the system remains fully decentralized from a routing logic perspective. Kubernetes acts as the shared source of truth for state synchronization through labels and informers.

3.1.4 Request Flow Overview

The request handling workflow of the routing proxy can be summarized as follows. Upon receiving a client request, the proxy executes a sequence of steps to select an appropriate service instance and update its routing state.

1. A client sends a request to the local proxy.
2. The proxy retrieves all candidate pods for the service.
3. Each pod is scored based on current latency and, when enabled, model metadata (accuracy).
4. The best-scoring pod is selected; the request is forwarded.
5. Post-response, the proxy updates latency records and assigns Kubernetes node labels, which are lightweight key–value metadata attached to nodes. In our system, nodes are labeled as *good*, *bad*, or *overloaded*, where *good* and *bad* reflect the aggregated score derived from latency and model accuracy, and *overloaded* indicates excessive CPU or memory utilization. These labels allow proxies to share node state and guide subsequent routing decisions.
6. If no healthy pods are found, a fallback strategy (e.g., random or nearest-node selection) is applied.

This unified monitoring and labeling mechanism enables ASRB to make accurate, lightweight, and scalable routing decisions. Latency and resource usage feed into labels, labels drive monitoring frequency, and refreshed measurements feed back into the scoring and routing logic, forming a lightweight closed-loop mechanism.

3.2 Latency vs. Accuracy Tradeoff

In intelligent service deployments, particularly those involving real-time machine learning inference, routing decisions must balance two competing QoS dimensions: **response latency** and **inference accuracy**. This balance is especially important in edge-cloud environments, where both resource constraints and user expectations vary across applications.

In the context of this thesis, latency refers to the response time, that is, the total time required for a request to be transmitted, processed, and for the response to be returned to the client. It therefore includes both network and processing delay components. Applications such as augmented reality, autonomous systems, and interactive analytics require consistently low response times to remain usable. In contrast, inference accuracy reflects the quality or correctness of the model’s output, which is critical in domains such as medical diagnostics, industrial monitoring, and safety-critical automation.

In practice, achieving higher accuracy often incurs additional computational cost. Deep neural networks such as high-depth ResNet variants deliver stronger classification performance but require more processing time compared to lightweight architectures like MobileNet, or pruned and quantized model variants that reduce computational demand at the cost of some accuracy [HMD16]. Although cloud nodes may offer sufficient computational capacity for such heavy models, the additional network delay may offset these

benefits for latency-sensitive applications. Conversely, edge nodes typically offer lower network latency but may either host lightweight models with reduced accuracy or, if capacity allows, run more complex models with increased computation latency due to resource limitations, thereby offsetting proximity benefits.

In general, edge deployments benefit from reduced network latency due to physical proximity, but often rely on resource-constrained hardware, which increases inference time for complex models such as ResNet50. Cloud deployments, in contrast, offer substantially higher computational capacity, reducing inference latency, but incur additional network delay due to geographical distance. As a result, the end-to-end latency observed by a client depends on the interplay between network conditions and model execution time, leading to different trade-offs across deployment types.

Given these inherent trade-offs, routing decisions cannot be made solely on the basis of latency or resource availability. A model that is closer to the client may respond faster but deliver lower-quality results, while a more distant model may achieve better accuracy at the cost of increased delay. This makes traditional routing strategies insufficient for QoS-aware machine learning services.

To address this challenge, ASRB introduces a **weighted scoring mechanism** that quantifies both latency and model accuracy within a unified QoS metric. This score is computed dynamically for every request and enables the system to select the service instance that best reflects the application’s desired balance between responsiveness and inference quality. The scoring design allows applications to explicitly prioritize latency, accuracy, or any intermediate trade-off, providing a flexible foundation for QoS-driven routing across heterogeneous edge–cloud deployments.

3.2.1 Scoring Function Design

The core routing decision in ASRB is based on a weighted scoring mechanism that combines latency and model accuracy into a unified QoS metric. For each pod, the proxy collects the most recent latency measurement and retrieves the model’s accuracy annotation. The balance between these two factors is determined by the *latency importance weight* (LIW) parameter ($0 \leq \lambda \leq 1$), which is supplied with each request.

The scoring function is defined as:

$$S_p = \lambda \cdot \left(1 - \min \left(\frac{L_p}{1000}, 1 \right) \right) + (1 - \lambda) \cdot A_p \quad (3.1)$$

where:

- λ controls how strongly routing favors low latency,
- L_p is the most recent round-trip time measurement for pod p ,
- A_p is a static model-specific accuracy value (e.g., 0.70 for MobileNet, 0.78 for ResNet50).

Latency is transformed into a normalized latency score in $[0, 1]$ via $1 - \min(L_p/1000, 1)$, so that it is comparable to accuracy values and larger latencies do not dominate the score. The cap at 1000 ms bounds the influence of outliers while preserving the relative ordering among typical latency values.

This formulation ensures that both faster response times and higher inference accuracy contribute positively to the final selection score. By tuning λ , applications can prioritize responsiveness, accuracy, or any balance in between. For instance, in interactive user-facing applications, a higher λ (e.g., 0.8) prioritizes responsiveness, while in classification-critical systems, a lower λ (e.g., 0.3) increases the weight of model performance in the selection process.

3.2.2 Operational Implications

By embedding this trade-off directly into the routing logic, the system can route latency-sensitive requests to nearby service instances running lightweight models. At the same time, it can direct accuracy-sensitive requests to more powerful but potentially more distant nodes. The routing decisions dynamically adapt to changing conditions, such as network congestion or pod overload, while maintaining a flexible and tunable balance between response time and inference accuracy.

This scoring mechanism allows the system to remain QoS-compliant across a wide range of operating conditions without hardcoding model preferences or static thresholds. It forms the core decision engine of the proposed routing system and enables per-request optimization in dynamic, distributed infrastructures.

3.3 Monitoring Mechanism Design

Static, fixed-interval monitoring may waste resources by probing nodes unnecessarily when conditions are stable, or miss important changes when the update frequency is too low under volatile conditions.

To avoid these extremes, monitoring must be both *selective* and *responsive*. The system should allocate more observation effort to uncertain or critical nodes and reduce monitoring for consistently stable or obviously unsuitable ones.

3.3.1 Design Principles

The adaptive monitoring strategy operates under the following core principles:

- **Label-driven focus:** Nodes labeled `good` or `overloaded` receive higher monitoring frequency because they are more likely to be selected for routing.
- **Score-aware adaptation:** Monitoring frequency is increased for nodes with volatile scores or borderline performance (e.g., latency spikes, occasional failures).

- **Back-off on stability:** Nodes that demonstrate consistent performance over a defined window are monitored less frequently to conserve resources.
- **Exploratory sampling:** Nodes labeled `bad` are still refreshed periodically, but at the lowest frequency, allowing the system to detect recovery without generating unnecessary overhead.

3.3.2 An adaptive active-passive monitoring strategy

The monitoring mechanism combines both passive and active techniques. Passive monitoring relies on metrics exposed by Kubernetes, obtained through the Metrics API or via Prometheus, and includes indicators such as CPU utilization, memory usage, and pod health status. When latency information is missing or outdated, the proxy performs latency approximation by issuing a lightweight request to the pod's `/echo` endpoint to estimate the current round-trip time. These active probes are triggered adaptively and only when required, rather than on every request.

When a real latency measurement becomes available after a period of approximation, the cached latency value is not replaced immediately. Instead, the estimate is updated via linear blending between the previous value and the new measurement, controlled by `LAT_WEIGHT`. This prevents abrupt score changes while keeping the update mechanism simple.

Latency approximation is regulated by a cooldown mechanism. After an approximation cycle is triggered for a service, further approximation attempts are suppressed for a configurable interval (`QOS_COOLDOWN_S`) to avoid excessive probing. During this period, cached latency estimates are reused, and new real measurements are incorporated gradually using smoothing parameters (`LAT_WEIGHT`, `LAT_APPR_WEIGHT`) when they become available. This design limits monitoring overhead while still allowing latency estimates to adapt to changing conditions.

Each proxy instance maintains a local cache of pod information and recently observed latencies. Nodes labeled as `bad` are refreshed less frequently, while `good` nodes receive more frequent updates. This feedback mechanism keeps monitoring overhead low while ensuring that routing decisions rely on up-to-date information.

This adaptive strategy offers multiple advantages:

Reduced overhead: Limits unnecessary data collection, preserving CPU cycles and bandwidth, which is particularly important on resource-constrained edge devices.

Timely reaction to relevant changes: Prioritizes updates for nodes that are likely routing candidates, ensuring fresh data where it matters while avoiding unnecessary probing elsewhere.

Improved scalability: Enables the system to operate efficiently in large clusters without centralized coordination or uniform polling rates.

QoS awareness: Aligns monitoring effort with the system’s current routing logic and QoS objectives.

3.4 Score-Based Routing Algorithm

At the core of the proposed system lies the Adaptive Score-based Routing Balancer (ASRB), a dynamic, QoS-aware routing algorithm responsible for selecting the most suitable service instance for each incoming request. Unlike static or heuristic-based approaches, this algorithm integrates both infrastructure-level metrics (e.g., latency) and application-level metrics (e.g., model accuracy) to compute a context-sensitive score. The selected target is the pod with the highest score that satisfies the current QoS constraints.

3.4.1 Routing Objective

The goal of the algorithm is to maximize perceived QoS for each user request. This involves:

- Minimizing end-to-end latency,
- Maximizing inference quality (accuracy),
- Avoiding overloaded or degraded nodes,
- Respecting custom preferences via the λ weight parameter.

Routing is executed locally at each proxy instance in a stateless, per-request manner. The algorithm relies on the latest score and latency data available for all candidate pods.

3.4.2 Algorithm Steps

The routing process for each incoming request proceeds as follows:

1. **Service Discovery:** Retrieve all pods for the requested service from the Kubernetes API and keep only those in Ready state. For each pod, determine the label of its hosting node (good, overloaded, or bad) as maintained by ASRB.
2. **Metadata Extraction:** For each remaining pod, obtain the most recent latency estimate for its host (from the local cache or via latency approximation if the value is stale) and read the model accuracy from pod annotations.
3. **Scoring:** Compute the QoS score for each pod using the weighted scoring function. The score is computed as defined in Equation 3.1, where λ is the latency importance weight supplied with the request.

4. **Primary Candidate Filtering:** Select pods running on nodes labeled `good` and discard those whose score falls below a configured QoS minimum. If at least one such pod remains, continue with this set.

5. **Overloaded Fallback:** If no suitable `good`-node pods are available, repeat the same score-based filtering for pods on nodes labeled `overloaded` and treat them as a secondary candidate set.

6. **Last-Resort Fallback:** If neither `good` nor `overloaded` nodes provide a pod with acceptable score, ASRB may optionally fall back to pods on `bad` nodes or to a simple strategy such as random selection among healthy pods.

7. **Ranking and Selection:** From the final candidate set (preferably `good`, otherwise `overloaded`, and only then `bad`), sort pods by descending score and forward the request to the highest-scoring pod.

8. **Feedback:** After the response is received, update the latency measurement for the selected pod's host and recompute the best pod score per node. Based on this score and resource usage, ASRB updates the node's label (`good`, `overloaded`, or `bad`) to influence future routing and monitoring decisions.

The routing process for each incoming request is summarized in Algorithm 3.1 and proceeds as follows.

Input: Incoming request r , latency weight λ

Output: Selected pod p^*

- 1 **(1) Service Discovery:** Retrieve all Ready pods for service r .
- 2 **(2) Pod Categorization:** a) collect recent latency L_p from cache, b) collect score S_p from cache, c) classify pods into: *bestPods*: healthy and not overloaded, and *overloadedPods*: healthy but above resource limits.
- 3 **(3) Latency Refresh (optional):** If too few pods have valid latency data, the proxy triggers an asynchronous latency-refresh routine to update stale or missing latency information, as summarized in Algorithm 3.2.

Algorithm 3.2: Asynchronous latency refresh routine

Input: Set of candidate pods \mathcal{P}

Output: Updated latency cache entries and recomputed scores

- 1 **foreach** $p \in \mathcal{P}$ **do**
 - 2 **if** *latency for p is missing or outdated* **then**
 - 3 send lightweight request to p at `/echo`;
 - 4 measure round-trip time L_p ;
 - 5 update latency cache for p with L_p ;
 - 6 recompute score S_p using L_p and cached accuracy A_p ;
 - 7 store updated S_p in the score cache;
 - 8 **end**
 - 9 **end**
-

- 10 **(4) Score Computation:** For each candidate pod p :

$$S_p = \lambda \cdot (1 - \min(L_p/1000, 1)) + (1 - \lambda) \cdot A_p$$

where A_p is the model accuracy from annotations.

- 11 **(5) Ranking:** Select the highest-scoring pod within:

1. *bestPods*
2. else *overloadedPods*
3. else *random from all healthy pods*

- (6) Forwarding:** Forward request r to the selected pod p^* .

- (7) Feedback Update:** a) measure RTT, b) call `SetLatencyAndScore(p*)`, c) update node QoS label (good/bad).

Note: The overloaded label is not set in this feedback step; it is maintained separately by the resource-monitoring component based on CPU and memory thresholds.

3.4.3 Integration with Monitoring

The routing algorithm interacts with the monitoring subsystem in a lightweight manner. Latency information is maintained through two mechanisms: feedback updates collected after each request and on-demand approximation when cached values become stale. Resource metrics (CPU and memory) are refreshed periodically and used to label nodes as `overloaded` when predefined thresholds are exceeded. These labels are then used in two ways: they steer the monitoring process by refreshing `overloaded` nodes more frequently than `good` and `bad` nodes, and they support routing by treating pods on `overloaded` nodes as a fallback option that is considered only when no suitable pods on `good` nodes are available.

This loose coupling ensures that routing decisions benefit from timely latency and resource information without requiring a complex or high-overhead monitoring pipeline.

Table 3.1 summarizes the key differences between traditional static routing strategies and the dynamic QoS-aware routing approach proposed in this thesis.

Table 3.1: Comparison of Static vs. Dynamic Routing Strategies

Aspect	Static Routing	Dynamic Routing (This Work)
Routing Logic	Predefined or heuristic-based (e.g., round-robin, least connections)	Score-based selection using real-time metrics
Metrics Used	Infrastructure-only (latency or load)	Infrastructure + application-level (latency + model accuracy)
Adaptivity	None; does not respond to runtime changes	Continuously adapts to changing latency and resource state
Monitoring Scope	Uniform across all nodes	Periodic resource refresh + on-demand latency approximation
Overhead	Low, but may route poorly under stress	Slightly higher due to scoring and latency updates, but still lightweight
QoS Awareness	None or indirect	Integrated into routing through weighted latency-accuracy score
Suitability for Edge	Limited; blind to resource constraints or accuracy trade-offs	High; considers node load (via overloaded labels) and latency-accuracy trade-offs while remaining lightweight

3.5 System Extensibility

While this thesis demonstrates the proposed routing framework using an image classification workload with two neural network models (MobileNet and ResNet50), the system is intentionally designed to be modular and extensible. Its architecture and routing logic can be adapted to support a wide range of applications beyond image recognition.

3.5.1 Generalization to Other ML Applications

The proposed routing framework is not limited to image classification workloads. In general, any machine learning service that exposes measurable performance and quality metrics, such as inference latency, accuracy, energy consumption, or error rates, can be supported by extending the scoring logic. While the current implementation focuses on latency and accuracy, the modular design allows additional metrics to be incorporated through moderate changes to metadata extraction and score computation. Examples of diverse AI/ML tasks that could be served by our approach, as well as their metrics of interest, follow:

- **Natural Language Processing (NLP):** Selecting between lightweight BERT variants and full Transformer models based on input length and response time requirements [DCLT19].
- **Speech Recognition:** Balancing real-time decoding latency with word error rate (WER) for audio processing on mobile or embedded devices [PCPK15].
- **Video Analytics:** Dynamically routing object detection or scene classification tasks to edge or cloud nodes depending on frame resolution or real-time constraints [Joc20].

In each case, the scoring function could be adjusted by replacing or extending the accuracy term to represent domain-specific quality metrics (e.g., F1 score, BLEU score, WER, or frame rate).

3.5.2 Beyond AI: Supporting QoS in Generic Services

The same routing principles can be conceptually applied to non-ML services where Quality of Service varies across instances. For example:

- **Web Services:** Selecting API backends based on real-time response latency and availability.
- **Data Processing Pipelines:** Routing ingestion to the most responsive or least-loaded node based on current queue depth or throughput.
- **IoT Workloads:** Prioritizing edge compute units based on energy budget, connectivity, or hardware health.

Implementation

This chapter provides a detailed description of the implementation of the ASRB. The implementation builds on QEdgeProxy, but replaces its latency-only routing with a score-based mechanism and extends the monitoring subsystem with adaptive probing and QoS-driven update behavior.

ASRB is written in Go and operates directly on Kubernetes-managed infrastructure. It builds upon the lightweight architecture introduced in Chapter 3 and translates it into a fully functional system deployed via Kubernetes primitives such as DaemonSets, services, and annotated pods. This chapter describes the structure of the codebase, key data models, integration with Kubernetes APIs, scoring logic, monitoring components, and deployment methodology. Challenges encountered during development and solutions adopted are also discussed.

4.1 Codebase Overview

The implementation of the QoS-aware routing system is written in Go and designed to operate natively within a Kubernetes-managed edge-cloud environment. The system is based on the QEdgeProxy proxy implementation, which was extensively modified with new routing logic, adaptive monitoring, and real-time pod selection mechanisms. Each node in the cluster runs one instance of the proxy via a DaemonSet, ensuring decentralized, local decision-making.

This section presents the core files and components of the implementation, including their responsibilities, internal data structures, and runtime behavior. It also outlines how the system integrates with Kubernetes to support service discovery, metadata access, and metric collection.

4.1.1 Codebase Structure and Components

The project is organized around four key Go files:

- `main.go`
Launches the proxy on each node, sets up HTTP routing, and initializes the balancer and Kubernetes client. It handles incoming client requests and forwards them to the selected backend pod.
- `client.go`
Implements the Kubernetes client interface. It maintains an up-to-date cache of pods per service using informers.¹ The module also coordinates node-level resource metric collection, adaptive refresh cycles, pod cache maintenance, and node labeling based on aggregated scores.
- `balancer.go`
Encapsulates the routing logic. It filters healthy pods, computes and compares per-pod scores, and selects a target pod for each request. Latency approximation and score updates are performed based on recent observations, as described in Section 3.3 and Algorithm 3.2. The module enforces QoS requirements by jointly considering latency and model accuracy during routing, as detailed in Section 3.4.
- `model.go`
Defines the core data structures:
 - `PodInfo`: stores pod metadata including namespace, name, pod IP, host IP, the service target port, model type, model accuracy, and the current QoS score.
 - `PodState`: tracks per-pod health information, including the timestamp of the last request, a service health flag, and a failure counter used to exclude failing pods.
 - `NodeMetrics`: stores per-node latency and the current node label (`good`, `bad`, or `overloaded`).
 - `PingCache`: caches recent latency measurements together with their timestamp.
 - `PodInfoCache`: holds the list of pods per service and their service annotations.
 - `MaintainerData`: stores per-service lifecycle state for the pod cache and informer, including the last request timestamp and a stop channel used to terminate the service-specific informer when the service becomes inactive.

¹In Kubernetes, informers are client-side components that watch API objects (e.g., Pods or Nodes), cache their state locally, and receive incremental updates via event notifications. This avoids repeated polling of the API server and enables efficient, event-driven state tracking.

- `metrics.go`
Defines a set of Prometheus counters used to measure overhead. These counters track the number of Kubernetes API calls and the total payload volume generated by dynamic monitoring and scoring operations. This file does not perform resource collection itself; it only exposes instrumentation hooks used in `client.go`.

These files are organized under a flat package structure and use the Go module system. Configuration is partially driven by environment variables defined in the `DaemonSet` manifest and injected into each container at runtime, while other routing and scoring behaviors are implemented directly in code.

4.1.2 Environment Variables

ASRB exposes several configuration parameters via environment variables. These variables control smoothing behaviour, approximation frequency, latency validity, resource thresholds, and node label update cycles. They are defined in the `DaemonSet` manifest and injected into each proxy instance.

- `LAT_WEIGHT`
Weight controlling how strongly newly observed real latency samples influence the cached latency estimate, as described in Section 3.3.
- `LAT_APPR_WEIGHT`
Weight controlling how approximated latency values are smoothed during updates, as described in Section 3.3.
- `COOLDOWN_BASE_DURATION_S`
Base timeout duration applied to unhealthy pods before they can be retried.
- `PING_TIMEOUT_S`
Timeout for the `/echo` latency probe.
- `QOS_PERC`
Minimum percentage of pods that must satisfy the QoS score requirement before approximation is triggered.
- `QOS_COOLDOWN_S`
Minimum cooldown interval between consecutive latency approximation cycles, as described in Section 3.3.
- `PERIODIC_APPROX_TIMER`
Forces a periodic latency re-approximation every X seconds.
- `REAL_DATA_VALID_S`
Duration for which real request latency remains valid before approximation is required.

- `BEST_OF`
Number of top-scoring pods to randomly choose from during selection.
- `NODE_METRICS_CACHE_TIME_S`
How often CPU and memory metrics are refreshed for each node.
- `MAX_RES_USAGE`
CPU or RAM threshold above which a node becomes labeled overloaded.
- `SCORE_UPDATER`
How often aggregated pod scores are computed and node labels are patched.
- `LABEL_PATCH_COOLDOWN`
Minimum time required between two label updates for the same node.

4.1.3 Service Discovery and Pod Metadata

The proxy uses Kubernetes informers (via `client-go`) to subscribe to pod lifecycle events. When a pod is added, updated, or deleted, the proxy updates its internal podCache, which stores all currently eligible pods for each service.

Each pod provides application-specific metadata through Kubernetes annotations, such as:

```
annotations:  
  modelType: "mobilenet"  
  modelAccuracy: "0.7"
```

During informer updates, these annotations are parsed and stored inside the `PodInfo` structure:

```
type PodInfo struct {  
    Namespace      string  
    Name           string  
    IP             string  
    HostIP        string  
    ServiceTargetPort string  
    Model          string  
    ModelAccuracy  float64  
    Score          float64  
}
```

If the `modelAccuracy` annotation is missing or invalid, a default value of 0.0 is assigned. The accuracy value directly contributes to the QoS score, while service-level annotations such as `maxLatency` are used to filter pods during routing.

4.1.4 Scoring and Pod Selection

When a request arrives at the proxy (port 9090), ASRB performs the following steps:

1. Read the `latencyImportanceWeight` from the HTTP header.
2. Retrieve all `Ready` pods for the requested service using Kubernetes label selectors.
3. For each pod, obtain the latest latency estimate for its host from the local cache; if the value is stale or missing, trigger an asynchronous latency approximation to `/echo`.
4. Read the pod's accuracy value from annotations.
5. Compute a QoS score for each pod:

$$S_p = \lambda \cdot (1 - \min(L_p/1000, 1)) + (1 - \lambda) \cdot A_p$$

6. Group pods into *best*, *overloaded*, and *fallback* sets based on their node's QoS label.
7. Apply the QoS minimum threshold to remove pods whose score is below QoS_{\min} .
8. Select the highest-scoring pod, preferring: *bestPods* \rightarrow *overloadedPods* \rightarrow *random from all healthy pods*.
9. Forward the request to the selected pod.
10. Record the RTT, update host latency, and recompute the best pod score per node to update the node's label (good, overloaded, or bad).

The proxy uses the highest-scoring pod on each node to update dynamic node labels, which then drive monitoring frequency and fallback behavior.

4.1.5 Deployment Architecture

The system is deployed using Kubernetes manifests that define:

- A `DaemonSet` to deploy one proxy per node.
- A `Service` for exposing the proxy on port 30090 (`NodePort`).
- `Secret Volumes` for mounting Kubernetes config.
- Per-service `DaemonSets` for each model (MobileNet, ResNet).

Each image recognition pod exposes a classification endpoint on port 5000. The routing proxy is responsible for selecting which of these instances to forward requests to based on real-time conditions.

This architecture and modular file structure enable the system to remain lightweight, fully decentralized, and QoS-aware, while leveraging native Kubernetes components for observability and scaling.

4.2 Pod Scoring and Metadata Handling

The system assigns a dynamic QoS score to each pod based on both its observed latency and its annotated model accuracy. These scores are used at runtime to rank service instances and select the most suitable pod for forwarding incoming client requests. This section describes the implementation of the scoring function, metadata extraction from Kubernetes annotations, internal data structures, and fallback mechanisms used to ensure routing robustness.

4.2.1 Scoring Function

The scoring logic of ASRB is implemented in `balancer.go`. The score for each candidate pod is computed using the scoring function defined in Equation 3.1.

4.2.2 Data Structures for Score and State Tracking

ASRB relies on a set of lightweight data structures defined in `model.go` to maintain per pod and per node state. These structures were introduced earlier in the codebase overview and are reused throughout the scoring, routing, and monitoring logic (see Section 4.1).

4.2.3 Fallback Handling and Routing Robustness

To ensure stable behavior under missing data, node recovery, or transient failures, ASRB incorporates several fallback and robustness mechanisms that guarantee continuous service even when ideal routing conditions are not met.

- **Failed Pod Handling:** If a pod times out or returns an error, it is marked as temporarily unhealthy via `SetReqFailed()`. Such pods are excluded from selection until the timeout window expires.
- **Missing or Stale Latency:** Pods without valid latency are excluded from scoring until fresh values are assigned. When too few pods have valid measurements, the proxy triggers an asynchronous approximation cycle to restore RTT information.
- **Missing Accuracy Metadata:** Pods without a valid `modelAccuracy` annotation receive a default accuracy of `0.0`, ensuring that routing remains stable even when metadata is incomplete.

- **Node Recovery Handling:** Nodes with missing latency history trigger an active approximation request. Until a valid RTT is obtained, their score contribution is very low, making them unlikely routing candidates.
- **Tiered Fallback Selection:** If no high-scoring pods are available, ASRB falls back in order:
 1. pods on overloaded nodes,
 2. as a last resort, any remaining healthy pods.

This ensures that a valid backend is always selected when possible.

- **Latency Refresh Under Volatility:** If latency values become stale, inconsistent, or overly sparse, a targeted approximation cycle refreshes data without requiring full-cluster probing.

These mechanisms prevent routing failures, reduce the chance of selecting inappropriate pods, and allow ASRB to remain resilient during topology changes, pod rollouts, or temporary node instability.

Node-level decisions based on aggregated pod scores are discussed in Section 4.4.

4.3 Routing Logic and Request Handling

This section describes how the proxy handles incoming client requests and performs dynamic pod selection based on real-time scoring. The routing logic is implemented primarily in `main.go` and `balancer.go`, with supporting functionality in `client.go`. The system operates as a reverse proxy that selects the optimal backend pod for each request based on a weighted score that combines latency and, when the latency weight $\lambda < 1$, model accuracy. When $\lambda = 1$, routing becomes purely latency-based, matching a pure latency-based behavior.

4.3.1 Request Flow Overview

Each request from the client is received on the proxy's exposed port and processed in the following steps:

1. The HTTP handler in `main.go` parses the request and extracts the `latencyImportanceWeight` from the headers.
2. The proxy extracts the service name from the request host (e.g., `image-recognition`).
3. It calls `ChoosePod()` from `balancer.go`, passing the namespace, service name, and the supplied λ value.

4. `ChoosePod()` filters pods by readiness, latency validity, and resource constraints, computes QoS scores, and returns the selected pod's IP, HostIP, and target port.
5. The proxy forwards the request to the selected pod.
6. After receiving the response, it measures the round-trip time and updates latency and score information.

4.3.2 Entry Point and Forwarding Logic

The routing process begins in the `reverseProxyHandler()` function defined in `main.go`:

- It reads the `latencyImportanceWeight` HTTP header and parses it to a float.
- It passes this value to `getOriginServer()`, which internally calls the balancer's `ChoosePod()` function.
- Once the target pod is selected, it updates the request fields to forward it to the chosen pod's IP and port.

If forwarding fails (e.g., due to a timeout or network error), the proxy logs the failure and calls `SetReqFailed()`, which marks the corresponding pod entry as temporarily unhealthy.

4.3.3 Pod Selection Logic

The method `ChoosePod()` in `balancer.go` implements the score-based routing procedure described in Section 3.4. It retrieves candidate pods from the local cache, filters unhealthy instances, and ranks eligible pods according to their current QoS score. Pods hosted on nodes labeled as `overloaded` are considered only as a fallback when no suitable candidates on good nodes are available.

If latency information is missing or insufficient, `ChoosePod()` triggers a background update via `ApproximateLatencyAndScore()` before completing the selection. When no suitable pod can be identified, the method falls back to a local pod or a random healthy instance.

4.3.4 Request Feedback and Learning

After a request is completed:

- The round-trip time is recorded
- The latency for the host is updated via `SetLatency()`

- The score is recalculated for all pods running on the same node.
- The best score among those pods is sent to `UpdateNodeScoreLabel()` in `client.go`

This feedback loop continuously adapts routing to reflect network conditions and model performance.

4.3.5 Echo Endpoint for Probing

A secondary endpoint (`/echo`) is exposed in `main.go` for the proxy to ping pods without invoking the full model. It returns the query string in JSON format and is used to measure RTT during active probing:

```
http://<hostIP>:30090/echo?param1=value1
```

This endpoint enables latency approximation without interfering with model workloads.

4.3.6 Passive Monitoring of Resource Usage

ASRB periodically collects CPU and memory usage for each node via the Kubernetes Metrics API. This task is implemented in `client.go`, where the proxy refreshes node-level resource data every `NODE_METRICS_CACHE_TIME_S` seconds. During each refresh cycle, the system:

1. Retrieves the latest CPU and RAM usage for all nodes.
2. Compares these values with the configured overload threshold `MAX_RES_USAGE`.
3. Assigns the label `overloaded` to nodes exceeding the limit.
4. Leaves nodes below the threshold eligible to be labeled `good` or `bad` based on their QoS score.

Resource usage is used exclusively to determine whether a node is considered overloaded. No resource predictions or trend analysis are performed, keeping the monitoring mechanism lightweight. Since overloaded nodes are more likely to degrade latency or cause failures, ASRB refreshes their metrics more frequently than those of nodes labeled `bad`, avoiding unnecessary overhead for nodes unlikely to satisfy QoS constraints.

4.3.7 Active Latency Measurement

Latency measurement is carried out through a combination of request-feedback signals and active probing. Each proxy maintains a local cache of latencies that stores the most recent round-trip time observed for each node. The system updates latency in two ways:

Feedback-Based Latency Updates

After each successful client request, the proxy measures the round-trip time to the selected pod. This value is recorded through `SetLatency()`, updating both the cached latency for the host node and the pod's score. Feedback-based updates allow ASRB to learn real conditions directly from production requests. To avoid using stale information, each latency entry remains valid only for `REAL_DATA_VALID_S` seconds.

On-Demand Latency Approximation

When latency data becomes stale or insufficient, ASRB triggers an approximation cycle. This process, implemented in `balancer.go` via `ApproximateLatencyAndScore()`, sends a lightweight HTTP request to each pod's `/echo` endpoint. The probe avoids running the full model, returns a minimal JSON payload, and provides a reliable measurement of network RTT.

Approximation is controlled by the following parameters:

- `PING_TIMEOUT_S`: maximum waiting time for a probe,
- `QOS_PERC`: minimum percentage of pods that must satisfy QoS before approximation is skipped,
- `QOS_COOLDOWN_S`: cooldown between approximation cycles,
- `PERIODIC_APPROX_TIMER`: forced periodic refresh under stable conditions.

These parameters prevent excessive probing while ensuring timely recovery of missing latency data, especially when new pods appear or network conditions change abruptly.

4.3.8 Latency Smoothing and Validity Control

Raw latency samples may fluctuate due to transient network conditions. To avoid unstable routing decisions, ASRB applies latency smoothing using linear blending rather than maintaining a full history of measurements.

When a new latency measurement L_{new} becomes available, the cached latency value L_{old} is updated as:

$$L \leftarrow w \cdot L_{\text{new}} + (1 - w) \cdot L_{\text{old}},$$

where $w \in [0, 1]$ is a configurable smoothing weight. The parameter `LAT_WEIGHT` is used when incorporating real latency measurements, while `LAT_APPR_WEIGHT` is applied to approximated values obtained via active probing.

This approach provides basic smoothing without maintaining a time series or exponential decay state, keeping the monitoring mechanism lightweight and well-suited for resource-constrained edge environments.

4.3.9 Integrated View for Routing

At runtime, the scoring logic depends on the most recent latency available. Nodes with accurate, fresh latency data receive more stable scores and are more likely candidates for routing. Conversely, nodes lacking valid latency are penalized until an approximation cycle updates their values.

Combined, the passive and active monitoring systems provide ASRB with a timely, efficient, and robust view of cluster performance. This integrated approach ensures that routing decisions react quickly to changes in latency, resource utilization, and pod availability.

4.4 Node Labeling and Label-Driven Monitoring

This section describes how ASRB assigns and updates node labels (`good`, `overloaded`, `bad`) based on observed latency and resource usage, and how these labels are used to steer monitoring frequency and routing behavior.

4.4.1 Node Label Types

ASRB derives node labels from the monitoring and latency measurements described in the previous section. Instead of using individual metric values directly during routing, the system aggregates latency, accuracy, and resource usage information into a small set of node-level labels. These labels summarize each node's current ability to serve requests and are subsequently used to influence routing priority and monitoring frequency.

Each node maintains exactly one QoS label:

- **good** — resource usage is acceptable and the node's best pod score meets QoS expectations.
- **overloaded** — CPU or memory exceeds `MAX_RES_USAGE`; routing is allowed but deprioritized.
- **bad** — the node is healthy but its score is too low or its latency data is outdated.

Labels are written directly to the Kubernetes API by the proxy. No external controller is required.

4.4.2 Score-Based Label Assignment

After every request, the proxy recomputes the best QoS score for all pods on the same node and calls `UpdateNodeScoreLabel()`. The label update is based on:

1. the node's highest current pod score,

2. the configured QoS threshold,
3. an enforced cooldown (`LABEL_PATCH_COOLDOWN`) that prevents rapid relabeling by ensuring that a node cannot change its label again until a fixed time interval has elapsed since the previous label update.

Nodes labeled `overloaded` ignore score updates until their resource usage drops below the threshold.

4.4.3 Label Effects on Routing

Node labels determine the order in which pods are considered during routing:

1. `good` nodes — primary candidates for selection,
2. `overloaded` nodes — secondary fallback set,
3. `bad` nodes — last-resort fallback.

This three-tier structure allows ASRB to remain QoS-aware without complex heuristics.

4.4.4 Label Effects on Monitoring Frequency

Monitoring intensity is fully determined by labels:

- `overloaded` nodes are refreshed most frequently to detect recovery,
- `good` nodes are refreshed at moderate intervals,
- `bad` nodes are refreshed least often to minimize overhead.

Thus, labels act as a simple form of adaptive monitoring: the most relevant nodes maintain the freshest data, while low-potential nodes incur minimal cost.

4.4.5 Consistency Across Proxies

Since each node runs its own proxy, multiple instances may update labels in parallel. ASRB ensures consistency through:

- Kubernetes acting as the shared source of truth,
- patch-based updates instead of full object rewrites, meaning that only the affected label fields are modified via Kubernetes patch operations rather than re-submitting the entire node specification,
- cooldown enforcement to avoid update storms.

These labels provide a coarse summary of each node's current suitability for serving requests and are used to guide routing and monitoring decisions.

4.5 Deployment Architecture

The ASRB system is deployed as a fully Kubernetes-native component, designed to operate efficiently across heterogeneous edge–cloud environments. Its architecture builds directly on Kubernetes primitives such as DaemonSets, Services, node labels, and pod annotations. No external controllers or auxiliary services are required—the entire system runs in a decentralized manner, with each node-local proxy making independent routing and monitoring decisions based on its local cache and Kubernetes metadata.

4.5.1 Proxy Deployment via DaemonSet

The ASRB proxy is deployed as a Kubernetes `DaemonSet`, following the deployment model introduced in Chapter 3. In the implementation, this ensures that one proxy instance is scheduled on each node and initialized with the required runtime configuration.

Concrete `DaemonSet` parameters, including environment variables, API access, and exposed ports, follow directly from the design described in Chapter 3 and are not repeated here.

4.5.2 Service Exposure and Traffic Flow

Each application (e.g., MobileNet or ResNet) is deployed via its own Kubernetes `Deployment` or `DaemonSet`, exposing a service port (commonly 5000). ASRB proxies do not rely on Kubernetes Service load balancing; instead, they:

1. retrieve all pods for a given service using label selectors,
2. extract metadata and annotations (model type, accuracy),
3. score and select one target pod per request.

External requests enter the cluster through the ASRB node port (30090), handled by the node’s proxy. From there, the proxy performs QoS-aware selection and forwards traffic directly to the chosen pod via HTTP.

4.5.3 Metadata Sources and Local Caching

ASRB relies on three sources of metadata:

- **Pod cache:** maintained by Kubernetes informers in `client.go`, storing pod IPs, ports, and annotations.
- **Latency cache:** per-host RTT values updated via real requests or approximation probes.

- **Node metrics cache:** CPU and memory retrieved from the Metrics API at configurable intervals.

All monitoring and scoring operations use these caches. No proxy ever performs cluster-wide scans; informers and cached state ensure low overhead even for large deployments.

4.5.4 Node Labels as Architectural Connective Tissue

Dynamic node labels (`good`, `bad`, `overloaded`) serve as the key point of integration between monitoring and routing:

Node labels (i) guide routing prioritization in `ChoosePod()`, (ii) steer monitoring frequency in `client.go`, (iii) are propagated via the Kubernetes API, and (iv) are observed by all proxies through the shared API server.

Thus, the Kubernetes control plane acts as a lightweight state-sharing mechanism without requiring any centralized ASRB logic.

4.5.5 Scoring and Label Updater Background Processes

Each proxy spawns independent background goroutines:

- a **metrics refresher** that periodically updates CPU and memory,
- a **score updater** that recalculates the highest per-node pod score,
- a **label patcher** that writes updated labels to Kubernetes if the cooldown allows,
- a **latency approximator** triggered on demand or via periodic timers.

These goroutines run asynchronously, ensuring that request latency is not blocked by monitoring tasks. All state is maintained in local memory.

4.5.6 Deployment on Heterogeneous Infrastructure

ASRB supports mixed deployment topologies that span cloud virtual machines, GPU-accelerated edge nodes (e.g., Jetson), low-power single-board computers (e.g., Raspberry Pi), and other resource-constrained devices running lightweight Kubernetes distributions such as k3s.

Proxies operate identically across all node types. Differences in compute capacity naturally emerge through:

- resource usage readings (label `overloaded`),
- latency conditions (score impact),
- model deployment choices (e.g. ResNet on cloud, MobileNet on edge).

4.5.7 Observability and Monitoring Integration

Each proxy exposes a Prometheus metrics endpoint (`/metrics` on port 2112). It reports:

- number of API calls caused by monitoring and scoring,
- payload volume,
- latency approximation counts,
- request processing statistics.

This provides full visibility into the overhead introduced by ASRB and is used directly in evaluation.

4.5.8 Summary

The deployment architecture is fully decentralized, Kubernetes-native, and optimized for heterogeneous edge–cloud systems. It integrates pod discovery, adaptive monitoring, routing, and node labeling without introducing heavy controllers or centralized logic. By relying on DaemonSets, informers, and lightweight background processes, ASRB maintains low overhead while ensuring real-time QoS-aware routing across diverse infrastructures.

4.6 Challenges and Implementation Notes

The development of ASRB required extensive modifications of the original QEdgeProxy codebase. While the architectural design is conceptually simple, several practical challenges arose during implementation. These challenges were closely tied to Kubernetes behavior, Go concurrency, latency measurement accuracy, and maintaining system stability under dynamic conditions. This section summarizes the most relevant issues and the solutions adopted in ASRB.

4.6.1 Handling Incomplete or Stale Latency Data

A primary challenge was ensuring robust pod selection when latency information was missing, outdated, or inconsistent across nodes. In early versions of the system, routing decisions were sometimes skewed by stale or incomplete data: newly added pods initially lacked latency measurements and could receive misleading scores, nodes with little or no recent traffic retained outdated RTT values, and intermittently failing pods occasionally produced measurements that did not reflect their true performance. ASRB resolves these issues by: (i) enforcing a strict validity window (`REAL_DATA_VALID_S`), (ii) triggering approximation only when necessary (QoS violation or missing data), and (iii) excluding pods in timeout via `isServiceInTimeout()` until recovery. This ensures that routing decisions are never based on stale or misleading values.

4.6.2 Asynchronous Operations and Concurrency Pitfalls

ASRB uses multiple background goroutines for periodic resource refresh, node label updating, latency approximation, and informer-driven cache updates, which introduced several race conditions during development, including (i) latency cache updates overlapping with score computation, (ii) pod list changes occurring mid-calculation, and (iii) concurrent label patch operations issued by different proxies.

The final implementation uses fine-grained mutexes around shared maps and strictly enforces cooldown windows (`LABEL_PATCH_COOLDOWN`) to avoid update storms.

4.6.3 Avoiding Load Concentration and Unintended Pod Stickiness

Latency smoothing initially caused a “stickiness” effect: once a pod accumulated enough good measurements, it kept being selected even when conditions changed. This happened because real samples dominated approximated ones, and stale values remained valid for too long. To address this issue, we applied a number of mitigation actions. First, we introduced separate weights for real vs. approximated samples. Second, latency measurements expire after a bounded validity period; once stale, approximation is permitted, allowing the system to refresh latency estimates when no recent real measurement is available. Third, we incorporate node labels, so resource overload can break stickiness. Together, these prevent ASRB from locking onto a single node.

4.6.4 Node Label Oscillation and Stability

Frequent oscillation between good and bad labels was a recurring issue. Small score fluctuations (e.g., temporary RTT spikes) caused frequent relabeling, which in turn triggered repeated Kubernetes API calls. The system now ensures stability via three mechanisms. First, a minimum cooldown is enforced before node labels may change. Second, score thresholds introduce a hysteresis-like effect that prevents frequent label oscillations. Third, the `overloaded` label is handled separately and is driven exclusively by resource usage. Together, these mechanisms improve stability and reduce monitoring overhead.

4.6.5 Kubernetes-Specific Behaviour

Several challenges were tied to Kubernetes runtime behavior, including informers delivering out-of-order events, nodes being temporarily removed or not fully initialized, pods transitioning through intermediate states (e.g., `ContainerCreating`), and the Metrics API occasionally returning zero or partial data.

ASRB addresses these issues by ignoring pods that are not in the Ready state, validating Metrics API responses before use, retrying API calls where appropriate, and falling back to approximated latency when pod readiness fluctuates.

4.6.6 Impact of Resource Constraints on Edge Nodes

Edge nodes such as Raspberry Pi or Jetson devices may exhibit slower Metrics API responses due to limited CPU resources and contention between monitoring components and application workloads [CBJ⁺23]. They may also experience temporary CPU saturation during inference and return inconsistent memory usage values.

The `overloaded` label proved essential in preventing routing to nodes under stress. Without it, the latency-only score often selected nodes that responded fast temporarily but soon degraded.

4.6.7 Testing in Dynamic Scenarios

Dynamic scenarios (node add/remove, reboot, overload injection) revealed several subtle issues: New nodes were sometimes labeled `bad` due to missing scores, recovered nodes occasionally took too long to re-enter routing decisions, and latency approximation was sometimes triggered too aggressively when the topology changed. These issues were addressed by initializing new nodes as `good` to allow early scoring, reducing the wait time before recalculating scores for recovered nodes, and allowing QoS-percent checks to suppress unnecessary approximation cycles.

4.6.8 Prometheus Integration and Overhead Tracking

Instrumenting monitoring overhead required careful separation of “dynamic monitoring” vs “scoring” API calls. To ensure correct accounting, ASRB increments Prometheus counters in exactly two places (monitoring and scoring), logs statistics only at request boundaries via `LogDynamicMonitoringStats()`, and avoids double-counting events triggered by informers. This distinction was crucial for the evaluation chapter, where overhead reduction is a key result.

4.6.9 Summary

The challenges encountered during ASRB development centered around stale or missing latency, concurrency management, Kubernetes behavior, and maintaining stable node labels. By addressing these through smoothing, cooldowns, cached state, and clear separation of duties across background goroutines, the system achieves reliable, lightweight, and scalable QoS-aware routing in real-world edge-cloud conditions.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Evaluation

This chapter evaluates the performance of the ASRB using a continuous experiment consisting of 1200 requests. The experiment is divided into six phases (T1–T6), each representing a different runtime condition: initial system state, node arrivals, node reboots, pod deployments, overload injection, and recovery. Some stages reflect steady, unchanged topologies, while others introduce dynamic changes that stress ASRB’s responsiveness and adaptive monitoring.

Across all phases, ASRB is compared against several baseline routing strategies, including round-robin, QEdgeProxy’s original latency-only routing, Proxy-Mity (proximity-based), and accuracy-biased variants of ASRB (using $\lambda = 1$, $\lambda = 0.5$, and $\lambda = 0$), as well as ASRB configurations with different score update intervals. This enables a comprehensive comparison of latency behavior, accuracy trade-offs, routing stability across T1–T6, fairness, and monitoring efficiency.

The evaluation focuses on key indicators such as end-to-end latency, tail latency, accuracy distribution, λ -sweep trade-offs, fairness (Jain’s Index), dynamic behavior across phases T1–T6, and the monitoring overhead generated by each routing method. The following sections describe the experimental setup, the metrics used, and the results observed throughout the six-phase experiment.

5.1 Experimental Setup

The evaluation is performed on a Kubernetes-based testbed running QEdgeProxy and the proposed Adaptive Score-based Routing Balancer (ASRB). The goal is to observe routing behavior under realistic edge-cloud conditions while the system undergoes controlled topology changes, overload events, and recovery periods. Our results are derived from a continuous experiment consisting of 1200 sequential requests, divided into six phases (T1–T6). Each phase modifies the cluster state to test ASRB’s adaptiveness.

5.1.1 Cluster Topology

The testbed consists of heterogeneous nodes running k3s:

- **Master node:** cloud-hosted controller node running the Kubernetes control plane and, in our deployment, also hosting inference pods executing both MobileNet and ResNet models.
- **Worker1 and Worker2:** stable edge nodes hosting MobileNet and ResNet inference pods.
- **Worker3:** local on-premise edge device; initially without a pod (T1), receives a MobileNet pod in T4, and becomes overloaded in T5.
- **Worker4:** very-near edge node hosting only a MobileNet model; added in T2, rebooted in T3, and overloaded in T5.

Each node runs an ASRB/QEdgeProxy instance via a `DaemonSet`. Each model pod exposes an image-recognition service. MobileNet and ResNet50 represent accuracy–latency extremes.

5.1.2 Experiment Phases (T1–T6)

The 1200 requests are divided into equal segments across six stages:

- **T1 – Initial topology:** Workers 1–3 active; Worker3 has no model pod.
- **T2 – Node arrival:** Worker4 joins and becomes available.
- **T3 – Reboot and recovery:** Worker4 becomes unavailable, then recovers.
- **T4 – Pod deployment:** Worker3 receives a MobileNet pod.
- **T5 – Overload:** Workers 3 and 4 are intentionally overloaded (CPU > 85%, latency injection). To simulate overload during T5, two temporary stress pods were deployed on Worker3 and Worker4. Each pod executed CPU and memory load (`stress -cpu 4 -vm 2 -vm-bytes 1G`) for approximately 100s and was then deleted. This reliably pushed the nodes above 85% CPU utilization, triggering the overload condition used in the experiment.
- **T6 – Recovery:** Overload removed; nodes return to normal behavior.

These stages provide both steady-state periods (T1, T2, T4, T6) and dynamic events (T3, T5), enabling analysis of ASRB’s stability and responsiveness.

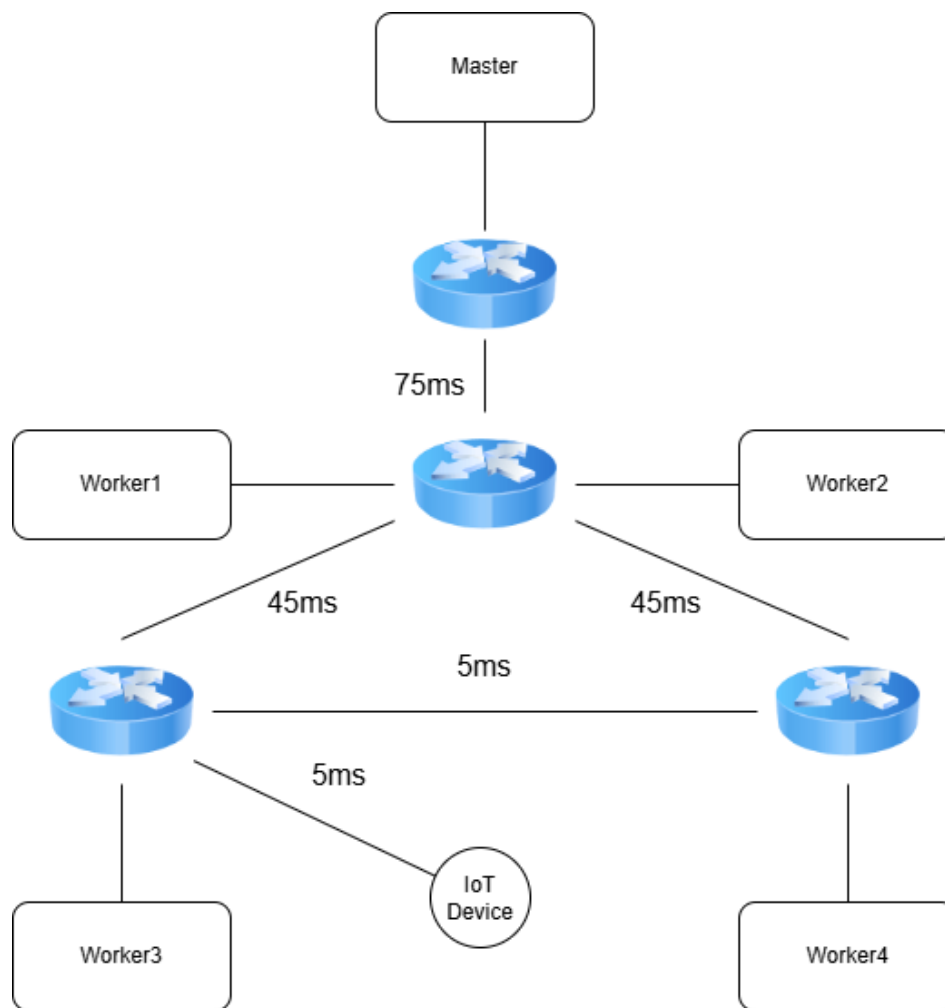


Figure 5.1: Cluster topology used in the evaluation. Worker3 is the closest node to the client device.

5.1.3 Routing Algorithms Compared

The following routing strategies are evaluated:

- **ASRB** with latency-accuracy weight $\lambda \in \{1, 0.5, 0\}$,
- **ASRB ScoreUpdater variants:** 5 s and 60 s intervals,
- **QEdgeProxy baseline:** original latency-only routing,
- **Proxy-Mity:** proximity-based routing ($\alpha = 1$ and $\alpha = 0.8$),
- **Round Robin.**

This set covers latency-centric, accuracy-centric, proximity-based, and static baselines.

5.1.4 Workload Characteristics

The evaluation consists of 1200 sequential HTTP inference requests using an identical image payload for all algorithms. Requests are issued by an external client machine via a NodePort service on port 30090, and all routing algorithms process the same request sequence to ensure uniform comparison.

5.1.5 Monitoring Overhead Scenarios

Monitoring overhead is collected using Prometheus counters exposed by each proxy instance:

- DynamicMonitoring API calls,
- Scoring API calls,
- Monitoring payload bytes (dynamic + scoring),
- Total monitoring overhead.

These metrics are formally defined later in Section 5.2.5. To analyze monitoring overhead and isolate the impact of different update mechanisms, we track these metrics and evaluate the following monitoring configurations. Each setup modifies either metric refresh frequency, score propagation frequency, or both, while keeping the routing logic unchanged.

ASRB minimal API calls This configuration represents the lowest-overhead adaptive setup. Both node-metric refreshes and score propagation are performed infrequently. Monitoring is largely cache-driven, and score updates are delayed, minimizing Kubernetes API interactions. This variant demonstrates how much monitoring traffic ASRB can save when adaptiveness is applied conservatively.

ASRB fastest → most API calls This configuration represents the upper bound of monitoring activity. Node metrics are refreshed frequently and score updates are propagated aggressively. It shows the maximum overhead ASRB can generate when prioritizing rapid detection of overloads and fast propagation of score changes.

ScoreUpdater 5 s and Monitoring 60 s In this setup, node metrics are refreshed conservatively, but score aggregation and label propagation are performed frequently. This isolates the cost of score updates themselves and highlights how faster score propagation improves responsiveness while increasing scoring-related API traffic.

ScoreUpdater 60 s and Monitoring 15 s This configuration refreshes node metrics frequently but propagates scores infrequently. It isolates the impact of aggressive resource

monitoring while keeping scoring overhead low, illustrating how metric freshness alone affects monitoring payload and overload detection timing.

Original Monitoring 15 s This baseline reflects the original QEdgeProxy behavior with frequent static monitoring. All nodes are polled periodically regardless of relevance, representing a worst-case static monitoring strategy with no adaptiveness or score-based filtering.

Original Monitoring 60 s This baseline uses a longer static monitoring interval. It demonstrates how reducing the polling frequency lowers overhead in the original design.

5.2 Performance Metrics

To evaluate the effectiveness of ASRB and compare it with baseline routing strategies, a set of quantitative metrics is collected during the 1200-request experiment. These metrics capture latency characteristics, inference quality, routing behavior, fairness, and the monitoring overhead introduced by adaptive score updates.

5.2.1 Latency Metrics

- **End-to-end latency:** time from sending a request to receiving the model output.
- **Latency distribution:** median, quartiles, and outliers visualized using boxplots.
- **Tail latency:** different percentiles extracted via CDF curves.
- **Latency stability:** evolution of latency across T1–T6, shown through per-request heartbeat plots.

Latency is measured client-side and propagated to ASRB via `SetLatency()` for score updates.

5.2.2 Accuracy Metrics

- **Model accuracy distribution:** proportion of requests served by MobileNet ($\approx 70\%$) and ResNet50 ($\approx 78\%$).
- **Latency–accuracy trade-off:** evaluated through the λ -sweep experiments ($\lambda = 1, 0.5, 0$).

Accuracy values come from model annotations and directly influence ASRB’s scoring function. For testing purposes, the accuracy values used by ASRB were fixed to 0.5 for MobileNet and 0.8 for ResNet50. These values approximate their relative accuracy levels and were kept constant across all experiments to ensure reproducible scoring behaviour.

5.2.3 Routing Behavior Metrics

- **Dominant node per request:** the sequence of selected nodes across T1–T6.
- **Routing stability:** frequency of node switches in response to topology changes.
- **Per-stage node distribution:** share of requests handled by each node in every phase.

These metrics quantify how routing decisions adapt to node arrivals, reboots, pod deployments, overloads, and recovery.

5.2.4 Fairness Metrics

Our fairness metric is **Jain’s Fairness Index** [JCH84], given by the following expression:

$$J = \frac{(\sum_i x_i)^2}{n \cdot \sum_i x_i^2},$$

where x_i denotes the number of requests served by node i and n is the total number of nodes.

Jain’s Fairness Index quantifies how evenly load is distributed across participating nodes. The metric is bounded between 0 and 1, where values close to 1 indicate near-perfect fairness (i.e., all nodes serve a similar number of requests), while lower values reflect increasing imbalance and concentration of load on a subset of nodes. In this evaluation, the index is used to distinguish intentional load concentration driven by QoS optimization from uniform load distribution.

5.2.5 Monitoring Overhead Metrics

Monitoring overhead is quantified using Prometheus counters exposed by each proxy instance.

`MonitoringAPICalls` Counts the number of static monitoring requests issued by the original QEdgeProxy. Each call corresponds to a periodic polling of node-level metrics for all nodes in the cluster, independent of their relevance for routing.

`MonitoringPayloadBytes` Measures the total payload size in bytes transferred by static monitoring in the original QEdgeProxy. Since the baseline performs no scoring or label propagation, this value fully represents the monitoring overhead of the system. In contrast, for ASRB the total monitoring overhead is computed as the sum of `DynamicMonitoringPayloadBytes` and `ScoringPayload`.

`DynamicMonitoringAPICalls` Counts the number of adaptive monitoring requests issued by ASRB to retrieve CPU and memory metric from nodes. Unlike the baseline, these calls are label-driven: nodes labeled differently have different refresh frequencies.

`DynamicMonitoringPayloadBytes` Measures the total payload size in bytes transferred by adaptive resource monitoring in ASRB. It is the sum of payloads generated by all dynamic monitoring API calls and depends on both the refresh frequency and the subset of nodes selected based on their labels.

`ScoringAPICalls` Counts Kubernetes API patch operations used by ASRB to propagate aggregated pod scores and update node labels. These calls are not present in the original QEdgeProxy and reflect the cost of score-based coordination via node labels.

`ScoringPayload` Measures the payload size in bytes of score propagation and node-label updates performed by ASRB through the Kubernetes API.

These values quantify the additional cost introduced by ASRB’s adaptive scoring and monitoring logic.

5.2.6 Data Collection and Visualization

During the 1200-request experiment, the following measurements and plots are produced, to provide a comprehensive view of routing performance across the dimensions of latency, accuracy, stability, fairness, and operational overhead. All evaluation results presented in the following sections are derived directly from these measurements.

- **Latency distribution (boxplots):** visualizes median, spread, and outliers of end-to-end request latency.
- **Tail latency (CDF curves):** shows high-percentile latency behavior (e.g., 50th, 70th and 95th percentiles).
- **Accuracy distribution (bar charts):** shows the average accuracy achieved by each routing strategy.
- **λ -sweep latency–accuracy trade-off:** evaluation in which the `LatencyImportanceWeight` parameter (λ) is varied across multiple values. Setting $\lambda = 1$ makes routing purely latency-driven, while $\lambda = 0$ makes routing purely accuracy-driven; intermediate values represent a controlled trade-off.
- **Per-T routing heartbeat (dominant node per request):** a request-by-request timeline indicating which node served each request across phases T1–T6. This plot highlights routing stability and adaptation speed, revealing how quickly an algorithm reacts to node arrivals, reboots, overloads, and recoveries.
- **Jain’s fairness index:** quantifies how evenly requests are distributed across nodes.
- **Monitoring overhead tables:** summarize API call counts and payload volumes for monitoring and scoring.

5.3 Evaluation Results

This section presents the results of the six-phase experiment (T1–T6) and compares the behavior of ASRB against baseline routing algorithms. The analysis covers latency, accuracy, dynamic adaptiveness, routing fairness, and monitoring overhead.

5.3.1 Latency Distribution

Figure 5.2 illustrates clear differences between routing strategies.

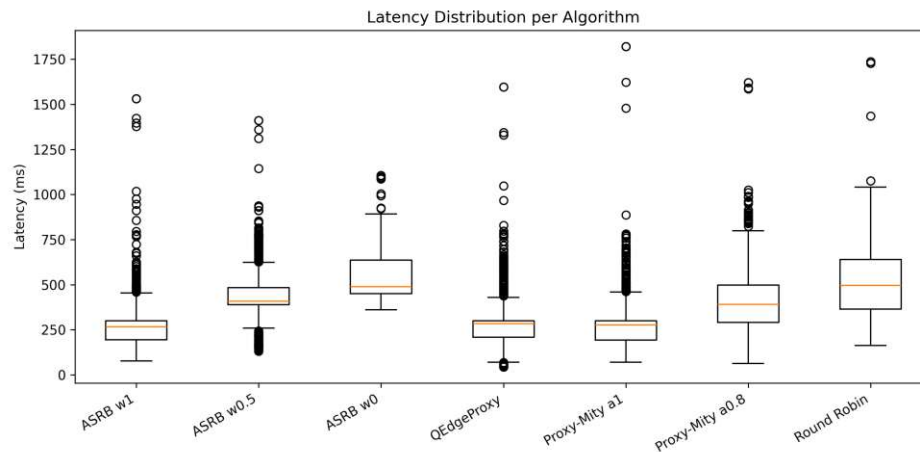


Figure 5.2: Latency distribution across all routing algorithms.

ASRB with $\lambda = 1$ achieves not only the lowest median latency but also a small interquartile range, indicating a highly stable latency profile.

Although ASRB ($\lambda = 1$), QEdgeProxy, and Proxy-Mity are all latency-oriented, they differ in how latency-based decisions are maintained under dynamic conditions. Proxy-Mity relies on plain latency ranking without explicit node-state tracking, overload persistence, or recovery gating. Consequently, it reacts only to instantaneous latency values and does not handle transient degradation or pod lifecycle events.

The original QEdgeProxy incorporates latency feedback and resource checks, but applies them in a purely instantaneous manner. Overload decisions are based on momentary CPU or memory thresholds and are not persisted through explicit node-state labels or cooldown periods. In addition, pods may be treated as routable before networking information (e.g., PodIP, HostIP) is fully initialized, which leads to failed requests during node joins or reboots.

ASRB ($\lambda = 1$) preserves a latency-centric objective but stabilizes routing through explicit node labeling, cooldown logic, and stricter pod admission semantics. Under dynamic conditions, these mechanisms reduce latency variance and extreme outliers by preventing routing to temporarily degraded or not-yet-ready nodes. While the overall

latency differences between the latency-oriented approaches remain small, ASRB achieves slightly more robust behavior. At the same time, the label-driven design enables selective monitoring refresh, allowing ASRB to maintain this stability while reducing monitoring traffic compared to the original QEdgeProxy.

ASRB ($\lambda = 0.5$) shows a noticeably higher median latency and increased variability compared to the pure latency-oriented variant. While it clearly outperforms Round Robin, its latency performance is comparable to Proxy-Mity ($\alpha = 0.8$), illustrating the trade-off introduced by balancing accuracy and latency, similar in spirit to how Proxy-Mity ($\alpha = 0.8$) trades latency for better load balancing.

In contrast, ASRB ($\lambda = 0$) exhibits a markedly higher median latency and an expanded upper tail, reflecting the slower inference time of the ResNet50 model. While its interquartile range is not the widest among all algorithms, the consistently elevated latency values highlight the performance cost of prioritizing accuracy over latency.

QEdgeProxy exhibits a compact interquartile range, indicating stable latency for the majority of requests. However, compared to ASRB ($\lambda = 1$) it shows a larger number of extreme outliers, revealing weaker tail behavior and reduced robustness during dynamic conditions such as node joins, reboots, or overload events.

Finally, Round Robin produces the worst overall latency distribution. Its interquartile range is the widest among all algorithms, and it exhibits several extreme latency outliers. This behavior underscores the limitations of static routing approaches in heterogeneous edge-cloud environments, where ignoring latency leads to unstable and inefficient request routing.

Result #1: Overall, the boxplot reveals that ASRB, particularly with $\lambda = 1$, delivers better latency behavior than both adaptive baselines (QEdgeProxy, Proxy-Mity) and static routing (Round Robin), while also allowing accuracy-aware trade-offs when $\lambda < 1$.

5.3.2 Tail Latency

Figure 5.3 provides a detailed comparison of the cumulative latency distribution across all routing strategies.

The plot shows the empirical cumulative distribution function (CDF) of request latencies. For a given latency value on the x-axis, the corresponding y-value indicates the fraction of requests that completed with latency less than or equal to that value. Curves positioned further to the left represent lower latency, while the slope of each curve reflects how concentrated or spread the latency values are across requests. Differences toward the right end of the plot highlight tail latency behavior, corresponding to the slowest fraction of requests.

Across the latency-oriented configurations, the CDF curves for ASRB ($\lambda = 1$), QEdgeProxy, and Proxy-Mity ($\alpha = 1$) largely overlap over most of the distribution, indicating

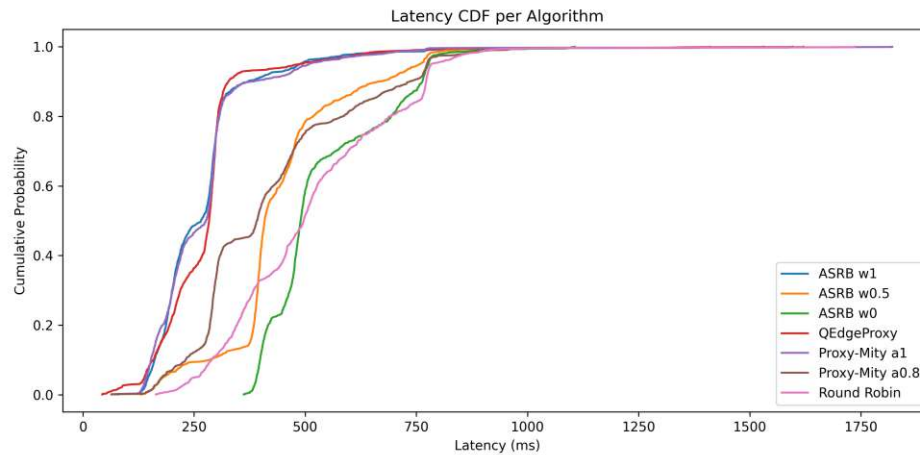


Figure 5.3: Tail latency (CDF) comparison.

that their overall latency behavior is very similar. Small deviations become visible only when examining specific percentile ranges more closely.

In the mid-range of the distribution (approximately the 20th–50th percentiles), QEdgeProxy reaches around 250 ms slightly earlier than ASRB ($\lambda = 1$) and Proxy-Mity ($\alpha = 1$) and shows higher latency in this range. Beyond this region, the curves converge, and toward the upper tail (around the 90th percentile), QEdgeProxy slightly surpasses both, exhibiting marginally better extreme tail behavior.

Result #2: Overall, the CDF results show that ASRB ($\lambda = 1$), QEdgeProxy, and Proxy-Mity ($\alpha = 1$) exhibit very similar latency distributions. The curves largely overlap across both median and high-percentile ranges, indicating comparable performance without pronounced differences in tail latency.

As expected, Round Robin and ASRB ($\lambda = 0$) exhibit the weakest latency performance. ASRB ($\lambda = 0$) performs worst overall, which is consistent with its design goal of maximizing accuracy while ignoring latency. Intermediate configurations, such as ASRB ($\lambda = 0.5$) and Proxy-Mity ($\alpha = 0.8$), achieve middle-ground performance. In the lower percentile range (approximately 10–30th percentile), ASRB ($\lambda = 0.5$) even shows slightly higher latency than Round Robin, reflecting the trade-off introduced by balancing accuracy and latency rather than optimizing purely for response time.

5.3.3 Accuracy Distribution

Figure 5.4 shows the average accuracy achieved by each routing strategy. Since the system does not measure model accuracy at runtime, accuracy is treated as a fixed, annotation-based property. In this experiment, MobileNet is assigned a constant accuracy of 0.50

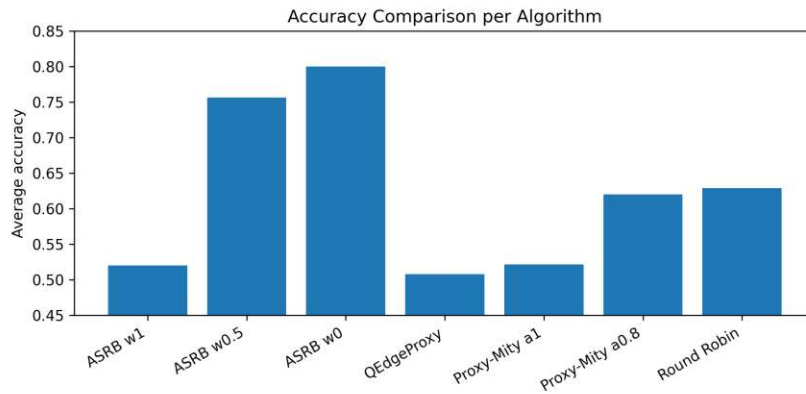


Figure 5.4: Accuracy distribution across routing strategies.

and ResNet50 a constant accuracy of 0.80. These values are used directly in the scoring function but are never validated or re-estimated during execution. Therefore, differences across algorithms reflect how frequently each method routes requests to ResNet50 rather than differences in model behaviour.

ASRB ($\lambda = 0$) achieves the highest average accuracy, ResNet50 baseline of 0.80. With no latency preference, the routing logic is completely accuracy driven, causing all requests to be sent to the ResNet50 pod whenever it is available. This represents the upper bound of achievable accuracy in the experiment.

ASRB ($\lambda = 0.5$) achieves the second-highest accuracy. It selects ResNet50 frequently, but still routes to MobileNet during latency spikes or transient overloads. This behavior demonstrates that the weighted scoring mechanism yields a smooth latency–accuracy trade-off: Increasing λ moves the route toward higher-accuracy models without eliminating responsiveness.

ASRB ($\lambda = 1$) shows one of the lowest accuracy among the ASRB variants. With a pure latency objective, the proxy favors MobileNet almost exclusively, since its inference time is substantially shorter. The resulting precision (just above 0.50) quantifies the cost of prioritizing latency over model quality.

The baseline algorithms, namely QEdgeProxy and Proxy-Mity, cluster near 0.50–0.55. Because they disregard model-level metadata, they only route to ResNet50 when it is temporarily faster or when topology events make it the shortest network path. Their accuracy results are therefore incidental rather than QoS-driven.

Round Robin achieves slightly higher accuracy than the baselines. By evenly distributing requests across all pods, it naturally produces a fixed mixture of MobileNet and ResNet50 outputs. However, this behavior is still not QoS-aware and lies far below the accuracy achievable by the controlled ASRB configurations.

Result #3: Overall, the accuracy comparison highlights the core advantage of ASRB: it is the only routing approach capable of intentionally and predictably steering accuracy through the λ parameter. The baselines exhibit no such control, and their results depend entirely on incidental runtime conditions.

5.3.4 Latency–Accuracy Trade-off

Figures 5.5 and 5.6 present the results of the λ -sweep experiment, where the latency–accuracy weight is varied across the three evaluated settings ($\lambda = 0, 0.5, \text{ and } 1$). Since model accuracy is not measured at runtime but treated as a static, annotation-based property (MobileNet: 0.50, ResNet50: 0.80), the observed accuracy depends solely on how frequently each model is selected. The two plots therefore directly reveal how strongly the ASRB scoring function steers routing decisions toward either latency or model quality.

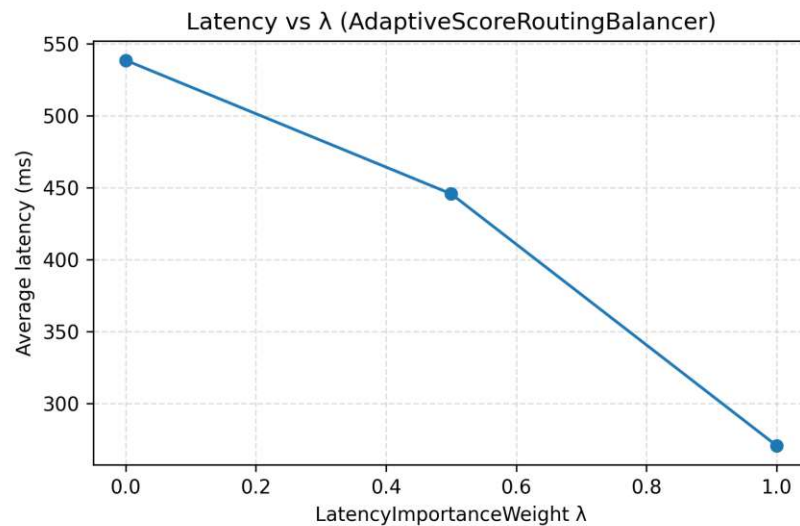
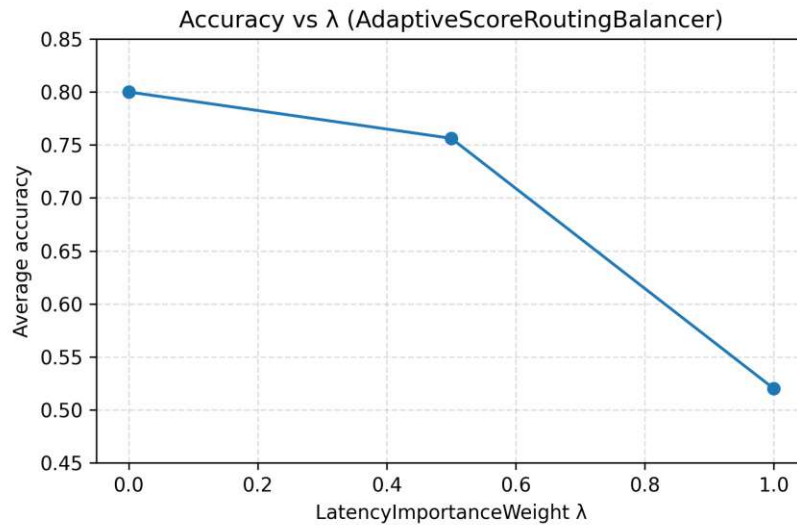


Figure 5.5: Latency vs. λ for ASRB.

The latency plot (Figure 5.5) shows a steep monotonic decrease as λ increases. For $\lambda = 0$, routing is fully accuracy-driven and thus we are sure that it selects ResNet50 almost exclusively, resulting in an average latency above 520 ms. At $\lambda = 0.5$, we see how the proxy still prefers ResNet50 in many situations, but switches to MobileNet when latency increases during overload or topology transitions. This mixed strategy reduces latency to around 440 ms. Finally, at $\lambda = 1$, routing becomes purely latency-oriented, producing the lowest average latency (below 300 ms). The monotonic decrease indicates that increasing λ consistently shifts routing toward lower-latency pods, producing a clear and predictable ordering of latency outcomes across the evaluated settings.

The accuracy plot (Figure 5.6) shows the inverse trend. With $\lambda = 0$, accuracy matches the fixed ResNet50 annotation (≈ 0.80), since the system routes almost exclusively to

Figure 5.6: Accuracy vs. λ for ASRB.

the high-accuracy model. As λ increases to 0.5, accuracy decreases only moderately, reflecting a balanced use of both models. However, at $\lambda = 1$, the accuracy drops sharply to approximately 0.52, which corresponds almost exactly to always using MobileNet. This confirms that the scoring mechanism correctly converges to MobileNet-dominated routing under pure latency optimisation.

Taken together, the two curves form a clear monotonic trade-off: lower latency implies lower accuracy, and vice versa. The observed behaviour across the evaluated λ values indicates that ASRB responds predictably and proportionally to changes in QoS priorities, allowing operators to tune service behaviour along a continuous latency–accuracy spectrum.

Result #4: This experiment demonstrates that ASRB fulfills its core design goal: it provides a configurable, predictable QoS trade-off mechanism, allowing operators to favour either speed (e.g., preview or interactive applications) or quality (e.g., safety-critical classification tasks) via a single, intuitive control parameter.

5.3.5 Dynamic Behavior Across T1–T6

Figures 5.7–5.10 show the request-by-request latency evolution over the entire 1200-request experiment for different routing schemes. These heartbeat plots provide insight into how each routing algorithm responds to topology changes, overload periods, and recovery events. Unlike boxplots or CDFs, these curves expose transient behaviours, reaction timing, and stability during each phase.

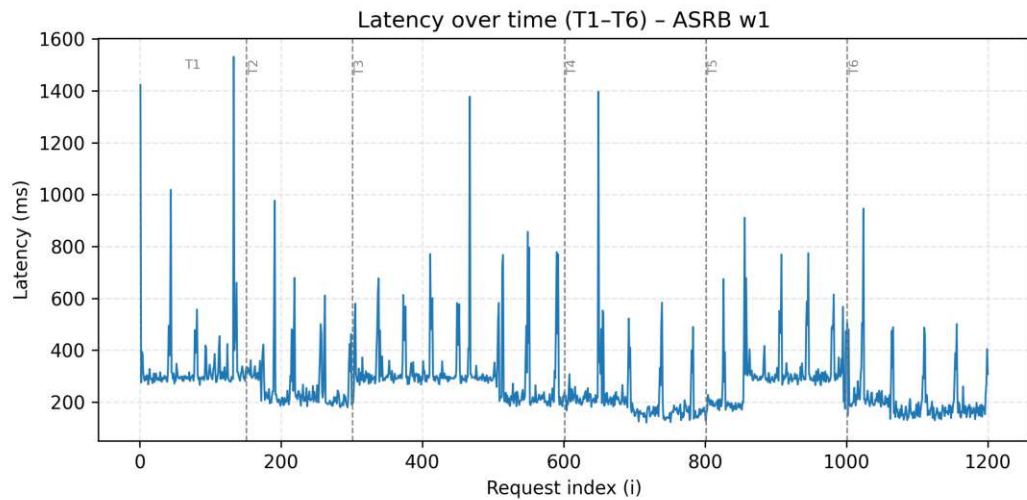


Figure 5.7: Dynamic latency across T1–T6 for ASRB ($\lambda = 1$).

ASRB ($\lambda = 1$). ASRB with pure latency preference achieves one of the most stable profiles across all phases. During T1, it consistently chooses the fastest nodes (Worker3 or Worker1), resulting in low baseline latency with only minor spikes caused by transient network fluctuations. When Worker4 joins in T2, ASRB incorporates it almost immediately, reflected by a small dip in average latency. The reboot event in T3 is also handled quickly: the latency spikes are short, and the system reroutes within a few requests. In T5, ASRB detects the overload on Worker3 and Worker4 early and shifts traffic toward stable nodes, preventing the long clusters of high latency visible in the baselines. T6 returns to a clean and stable profile.

QEdgeProxy. QEdgeProxy shows similar trends in stable phases but reacts significantly slower to dynamic events. In T3, latency spikes are not only higher but also occur for a longer stretch, indicating delayed detection of the rebooted node. During overload (T5), the proxy continues sending requests to degraded nodes for noticeably longer than ASRB, resulting in large high-latency clusters. This confirms that QEdgeProxy’s periodic, static monitoring is insufficient for rapid adaptation in dynamic edge environments.

Proxy-Mity ($\alpha = 1$). Although Proxy-Mity maintains latency values comparable to ASRB ($\lambda = 1$) during the overload phase T5 in this experiment, the routing decisions are still predominantly directed toward the overloaded node due to its proximity, as shown in Table 5.1. In a real-world deployment with stronger contention, variable background load, or longer overload duration, such behavior would likely result in significantly higher latency spikes than those observed in this controlled setup. This indicates that the experimental scenario may underestimate the impact of sustained overload on proximity-only routing.

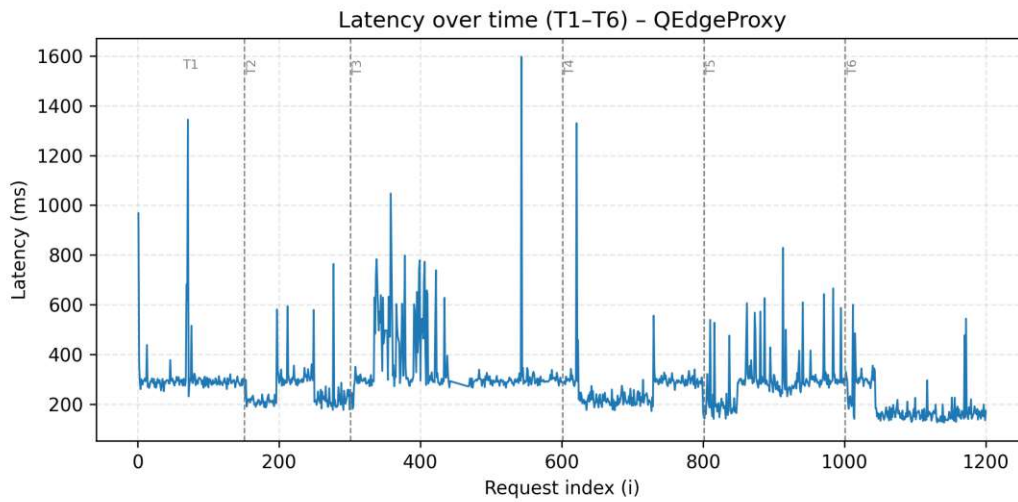
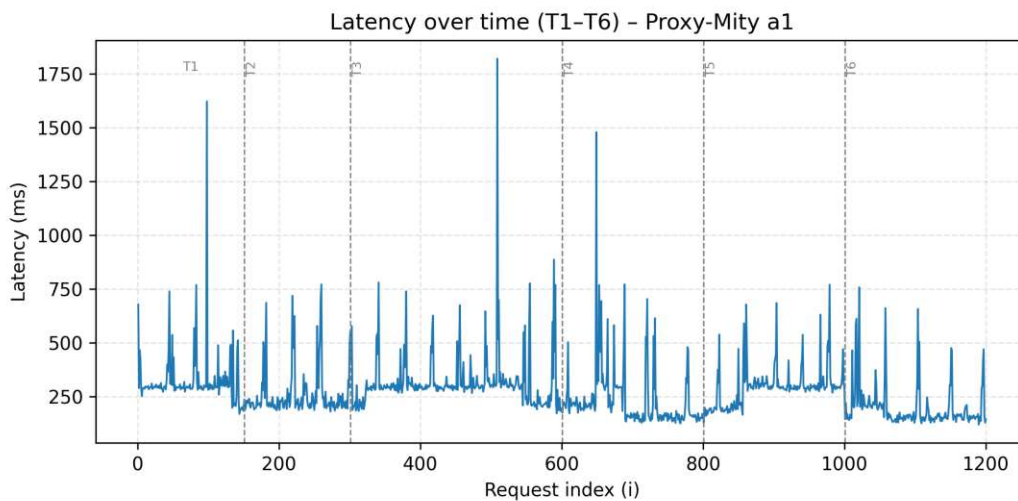


Figure 5.8: Dynamic latency for QEdgeProxy baseline.

Figure 5.9: Dynamic latency for Proxy-Mity ($\alpha = 1$).

Round Robin. Round Robin exhibits the worst dynamic performance. The heartbeat curve shows no phase-dependent adaptation, confirming that RR is entirely blind to topology changes, reboots, and overload. Latency in T5 degrades dramatically because RR continues to send requests to overloaded nodes with equal probability. Even in stable phases, variance remains high, reflecting repeated routing to slower nodes such as ResNet50 on distant workers.

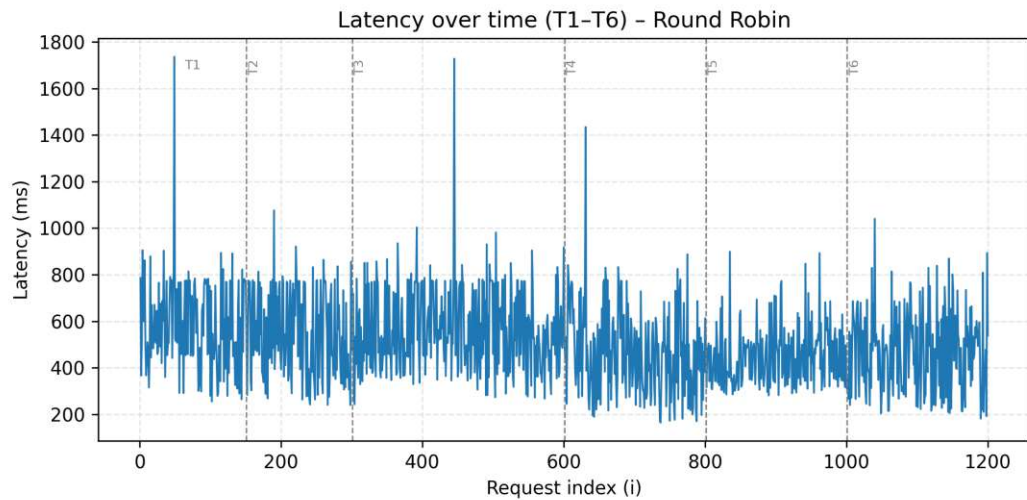


Figure 5.10: Dynamic latency for Round Robin.

Result #5: The heartbeat plots illustrate that ASRB is the only routing method that combines low baseline latency, fast reaction time, and resilience to overload. QEdgeProxy adapts slower, Proxy-Mity fails to avoid overloaded workers despite good proximity-based performance in stable phases, and Round Robin remains unaffected by system state. These results confirm that ASRB’s adaptive monitoring and scoring mechanisms are essential for stable performance in dynamic computing-continuum environments.

5.3.6 Request Failures and Initialization Race Conditions

During phases T2 (node arrival) and T3 (node reboot), nodes become visible to the cluster before their pods have fully initialized. This creates a short race window in which a pod object exists, but its PodIP is not yet assigned. Forwarding a request to such a half-initialized pod results in an immediate failure.

In the experiment, QEdgeProxy forwarded traffic to these pods as soon as they appeared in the API state, leading to **36 failed requests** during T2 and T3. Because the metadata was incomplete, the proxy could not associate failures with a specific node and therefore did not apply cooldown.

ASRB, in contrast, admits pods into its internal cache only after both PodIP and HostIP are present, and uses failure feedback to apply cooldown immediately. As a result, ASRB experienced only **2 transient failures** across the same phases. The node was then removed from routing until fully initialized.

Result #6: These results show that ASRB not only improves latency and monitoring efficiency but also significantly reduces failure rates during node join–reboot transitions.

5.3.7 Routing Distribution per Phase

The per-phase routing distribution in Table 5.1 shows how each algorithm shifts its dominant node across the stages T1–T6.

Table 5.1: Per-phase dominant node per algorithm (T1–T6).

Algorithm	Stage	Dominant Node	Dominant Count	Total Requests	Share
ASRB w1	T1	Worker1	120	149	0.81
ASRB w1	T2	Worker4	108	150	0.72
ASRB w1	T3	Worker1	152	300	0.51
ASRB w1	T4	Worker3	97	199	0.49
ASRB w1	T5	Worker1	84	200	0.42
ASRB w1	T6	Worker3	120	200	0.60
QEdgeProxy	T1	Worker1	144	150	0.96
QEdgeProxy	T2	Worker4	96	150	0.64
QEdgeProxy	T3	Worker1	224	264	0.85
QEdgeProxy	T4	Worker4	108	200	0.54
QEdgeProxy	T5	Worker4	85	200	0.43
QEdgeProxy	T6	Worker4	80	200	0.40
Proxy-Mity a1	T1	Worker1	101	150	0.67
Proxy-Mity a1	T2	Worker4	128	150	0.85
Proxy-Mity a1	T3	Worker1	124	299	0.41
Proxy-Mity a1	T4	Worker3	98	199	0.49
Proxy-Mity a1	T5	Worker4	112	200	0.56
Proxy-Mity a1	T6	Worker3	133	200	0.67
Round Robin	T1	Master	48	150	0.32
Round Robin	T2	Worker2	49	150	0.33
Round Robin	T3	Worker1	109	299	0.36
Round Robin	T4	Worker1	53	200	0.27
Round Robin	T5	Worker2	49	200	0.25
Round Robin	T6	Master	53	200	0.27

ASRB ($\lambda = 1$) consistently selects the lowest-latency node and adapts quickly when new nodes appear or recover, resulting in clear dominance transitions in T2, T3, and T4. During overload (T5), ASRB reduces traffic to Worker3 and Worker4, confirming that the scoring mechanism correctly penalizes degraded nodes.

QEdgeProxy exhibits similar dominant nodes in the stable phases, but reacts more slowly to dynamic events. Its dominance during T5 persists for longer on degraded nodes,

indicating delayed adaptation due to static monitoring intervals.

Proxy-Mity ($\alpha = 1$) primarily selects the nearest node, which explains its strong preference for Worker3 and Worker4 across multiple phases. However, this proximity bias causes it to stick to overloaded nodes in T5 more than ASRB, which lowers overall robustness.

Round Robin distributes requests evenly by design, producing a much flatter dominance profile with no strong preference for any node. As a result, it frequently routes to less suitable nodes, including the master and overloaded workers, which aligns with the high latency variance observed earlier.

Result #7: Overall, the distribution confirms that ASRB is the only method that consistently aligns dominant-node choices with real-time system conditions, while the baselines either react slowly (QEdgeProxy), follow static heuristics (Proxy-Mity), or ignore system state entirely (Round Robin).

5.3.8 Fairness Analysis

Jain’s Fairness Index in Table 5.2 quantifies how evenly each algorithm distributes load across the five available nodes.

Table 5.2: Load distribution and Jain’s Fairness Index per algorithm.

Algorithm	Master	Worker1	Worker2	Worker3	Worker4	J
ASRB ($\lambda = 1$)	40	382	172	224	380	0.772
ASRB ($\lambda = 0.5$)	42	721	316	77	44	0.458
ASRB ($\lambda = 0$)	88	0	1112	0	0	0.231
QEdgeProxy	19	538	100	130	377	0.630
Proxy-Mity ($\alpha = 1$)	42	335	224	239	358	0.821
Proxy-Mity ($\alpha = 0.8$)	301	359	337	73	129	0.810
Round Robin	315	345	327	71	141	0.821

Round Robin together with Proxy-Mity achieve the highest fairness. ASRB ($\lambda = 1$) also scores relatively high, because of good dynamic testbed scenario adaptation. QEdgeProxy shows moderate fairness, also showing a level of adaptation in driving towards low-latency pods in our experiment. ASRB ($\lambda = 0.5$) becomes skewed due to its combined preference for latency and high-accuracy ResNet50 placements. Finally, ASRB ($\lambda = 0$) achieves the lowest fairness, as it routes almost exclusively to the single ResNet50 node.

Result #8: These results confirm that lower fairness is not a deficiency but a direct consequence of QoS-driven optimisation: the more aggressively latency or accuracy is prioritised, the more load naturally concentrates on the nodes best matching those objectives.

5.3.9 Monitoring Overhead

The monitoring overhead is measured using the following Prometheus counters integrated into each proxy instance:

- `dynamic_monitoring_api_calls`,
- `dynamic_monitoring_payload_sum`,
- `scoring_api_calls`,
- `scoring_payload_sum`.

These counters quantify both the rate and the volume of monitoring traffic generated by each algorithm. Table 5.3 summarizes the results for all configurations.

A key observation is that **ASRB significantly reduces monitoring traffic compared to the original QEdgeProxy**. Although the number of monitoring API calls is higher, the monitoring overhead of the baseline is dominated by the much larger payload size per call, since static monitoring repeatedly queries all workers regardless of their relevance or state.

The baseline QEdgeProxy generates **3 288 010 Bytes** with a 15s monitoring cache and **1 069 393 Bytes** with a 60s cache during the experiment. In contrast, ASRB configurations consistently come with reduced overheads, as reported below:

- **ASRB minimal (cache 60 s, score 60 s):** 634 581 Bytes
(**40.7% reduction** vs. Original cache 60 s; **80.7% reduction** vs. Original cache 15 s)
- **ASRB fastest (cache 15 s, score 5 s):** 1 012 015 Bytes
(**5.4% reduction** vs. Original cache 60 s; **69.2% reduction** vs. Original cache 15 s)
- **ASRB score 5 s / monitoring 60 s:** 762 204 Bytes
(**28.7% reduction** vs. Original cache 60 s; **76.8% reduction** vs. Original cache 15 s)
- **ASRB score 60 s / monitoring 15 s:** 891 267 Bytes
(**16.7% reduction** vs. Original cache 60 s; **72.9% reduction** vs. Original cache 15 s)

These totals include both dynamic monitoring and score-update traffic. Importantly, **score updates are not required by the original QEdgeProxy**; they are introduced by ASRB's adaptive scoring mechanism. If we isolate *only dynamic monitoring* and exclude score-update payloads, the reduction becomes more pronounced.

- **QEdgeProxy static monitoring (15 s):** 3 288 010 B
- **QEdgeProxy static monitoring (60 s):** 1 069 393 B
- **ASRB dynamic monitoring (15 s):** 676 418 B
- **ASRB dynamic monitoring (60 s):** 399 260 B

This corresponds to a **79.4% reduction** in pure monitoring overhead for the 15 s cache and a **62.7% reduction** for the 60 s cache. Overall, ASRB reduces baseline monitoring traffic by approximately **63%–79%**, depending on the cache interval.

Even though the absolute numbers in this experiment are modest (a five node cluster), the relative reductions are already large, ranging from **63% to 79%**. This shows that ASRB yields substantial savings not only at scale but also in small deployments. In larger clusters with dozens or hundreds of nodes where static monitoring traffic grows linearly with cluster size the impact becomes even more pronounced. Reducing monitoring traffic by more than **60% per node** directly translates into significant bandwidth savings and lower control-plane load, helping to avoid contention with application traffic.

These results show that, despite introducing additional score updates, ASRB keeps the *total* monitoring overhead below that of the original QEdgeProxy across all tested configurations. Moreover, in systems where scoring is performed locally or where model-quality metadata is cached rather than propagated through the Kubernetes API, the dynamic monitoring component alone can achieve even larger reductions in control-plane traffic.

Result #9: Overall, the results demonstrate that ASRB’s adaptive-monitoring design not only improves latency and responsiveness but also substantially reduces system-level overhead. This makes ASRB suitable for both small and large-scale edge deployments, where control-plane traffic must be kept low to avoid contention with application workloads.

Table 5.3: Monitoring overhead: API calls and payload bytes per configuration.

Algorithm	Mon.	Score	Total	Mon. B	Score B	Total B
QEdgeProxy (static, cache 15 s)	136	0	136	3,288,010	0	3,288,010
QEdgeProxy (static, cache 60 s)	40	0	40	1,069,393	0	1,069,393
ASRB minimal (cache 60 s, score 60 s)	48	24	72	399,260	235,321	634,581
ASRB fastest (cache 15 s, score 5 s)	52	33	85	684,031	327,984	1,012,015
ASRB score 5 s / monitoring 60 s	46	35	81	421,448	340,756	762,204
ASRB score 60 s / monitoring 15 s	51	22	73	676,418	214,849	891,267

5.3.10 ScoreUpdater Interval: 5 s vs. 60 s

The ScoreUpdater interval affects only the scoring-related monitoring traffic, i.e., `scoring_api_calls` and `scoring_payload_sum`. A shorter interval (5 s) performs more frequent score updates, increasing scoring traffic, while the longer interval (60 s) reduces this overhead at the cost of slightly less stable latency.

From the measured values, the following figures can be derived:

- ASRB ScoreUpdater 5 s: average latency 284.49 ms, max 1783.61 ms,
- ASRB ScoreUpdater 60 s: average latency 297.45 ms, max 1824.17 ms.

The scoring-related monitoring overhead is:

- **5 s interval:** 35 scoring API calls, 340 756 Bytes scoring payload,
- **60 s interval:** 22 scoring API calls, 214 849 Bytes scoring payload.

This yields a reduction of:

$$340\,756 - 214\,849 = 125\,907 \text{ Bytes}$$

which corresponds to a **36.9% decrease in scoring-related payload**, along with a reduction from 35 to 22 score-update API calls.

Latency differences remain modest: the 60 s interval increases average latency by only about **4.5%**. This indicates that the update interval mainly influences how often long-term score adjustments propagate to the Kubernetes API, while short-term adaptation continues to be driven by direct latency measurements.

Result #10: Overall, the 5 s interval provides slightly better latency stability, whereas the 60 s interval reduces scoring-related overhead by approximately 37% with minimal performance impact.

5.3.11 Effect of Node-Metrics Cache Time

In addition to the ScoreUpdater interval, ASRB relies on a node-metrics cache whose freshness is controlled via `NODE_METRICS_CACHE_TIME_S`. To evaluate the impact of this parameter on overload detection, two stress pods are deployed on `Worker3` and `Worker4` during T5. These pods generate continuous CPU and memory load for approximately 100 s before terminating.

With a **short cache time** (15 s), ASRB detects the overload after roughly **55 requests** and immediately reduces traffic to the stressed nodes. Because the cache is refreshed

frequently, the system also clears the overloaded label soon after the stress pods terminate, allowing the nodes to re-enter the candidate set without delay.

With a **long cache time** (60 s), overload detection is significantly delayed: the overloaded label is applied only after about **180 requests**, at which point the stress phase is nearly over. Moreover, the stale cache retains the overloaded label for some time *after* the nodes have recovered, causing ASRB to avoid healthy nodes based on outdated metrics.

Result #11: These results show that the node-metrics cache time is critical for timely overload detection and recovery. A short cache provides fast reaction to both degradation and restoration, while a long cache leads to late detection during T5 and unnecessary avoidance of nodes that have already returned to normal operation.

5.3.12 Summary

The evaluation shows that ASRB delivers latency performance comparable to the fastest baselines while providing substantially greater flexibility and adaptiveness. Although the latency improvements over QEdgeProxy and Proxy-Mity are moderate, ASRB consistently matches or slightly exceeds them while incorporating model-quality awareness and label-driven monitoring. The system reacts quickly to node arrivals, reboots, pod deployments, and overload conditions, resulting in more stable behaviour across all dynamic phases.

Fairness outcomes follow naturally from ASRB's objectives, concentrating load when optimizing for latency or accuracy, whereas round robin distributes requests uniformly. Monitoring overhead remains low, with ASRB producing substantially less traffic than the original QEdgeProxy despite maintaining a more responsive monitoring strategy.

Overall, the results confirm that ASRB offers a balanced combination of competitive latency, adaptive routing, and efficient monitoring, making it a strong fit for dynamic and heterogeneous edge-cloud environments.

5.4 Discussion of Results

The evaluation demonstrates that ASRB achieves its primary goals: reducing latency, improving accuracy under configurable policies, reacting robustly to dynamic cluster changes, and maintaining low monitoring overhead. The following paragraphs summarize the main insights.

5.4.1 Latency and Tail Behavior

ASRB with $\lambda = 1$ consistently provides the lowest median latency and the most compact distribution. Its tail latency is substantially improved compared to QEdgeProxy and Proxy-Mity, particularly during T3 (reboot) and T5 (overload), where baselines suffer

from slow adaptation. These results confirm that latency-sensitive scoring enables ASRB to prioritize healthy and nearby nodes more effectively than pure proximity routing.

5.4.2 Accuracy and Trade-offs

The λ -sweep experiments validate that accuracy and latency form a monotonic and predictable accuracy–latency trade-off. ASRB ($\lambda = 0$) achieves the highest accuracy by selecting ResNet50 almost exclusively, while $\lambda = 1$ delivers optimal latency through MobileNet. The intermediate setting $\lambda = 0.5$ provides a balanced compromise. This shows that ASRB’s design allows operators to tune service behavior precisely according to application needs (e.g., fast preview vs. high-quality inference).

5.4.3 Dynamic Adaptiveness

Across the six-phase experiment (T1–T6), ASRB reacts faster and more consistently to topology events than all baselines. Node arrivals (T2), reboots (T3), and pod deployments (T4) are integrated into routing decisions within a few dozen requests, while overload conditions (T5) trigger early avoidance of congested nodes. Baselines without adaptive monitoring or scoring react slower and often continue routing to degraded nodes. The heartbeat plots confirm that ASRB maintains stable routing even under sudden changes.

5.4.4 Fairness and Load Balance

Fairness results reflect the intended design: ASRB ($\lambda = 1$) is deliberately biased toward the lowest-latency nodes, while ASRB ($\lambda = 0.5$) spreads load more evenly. Proxy-Mity and QEdgeProxy achieve moderate fairness, and Round Robin remains the only perfectly balanced method. These results indicate that fairness is not an error but a policy choice controlled by λ . Applications prioritizing robustness or energy distribution can choose a more balanced configuration.

5.4.5 Monitoring Overhead

A central design concern for adaptive monitoring is network overhead. Despite combining dynamic monitoring with score updates, ASRB reduces total monitoring overhead by approximately **17–41%** compared to the original QEdgeProxy when equivalent cache intervals are used, and by **63–79%** when only pure monitoring traffic is considered. Although absolute byte volumes remain modest in a five-node cluster, the relative savings are already substantial and scale linearly with cluster size. In larger deployments, where static monitoring traffic grows proportionally with the number of nodes, these reductions become highly significant. Furthermore, if score calculations were performed locally or cached instead of propagated through the Kubernetes API, ASRB would yield even greater monitoring savings.

5.4.6 Impact of Node-Metrics Cache Time

The responsiveness of overload detection depends strongly on the freshness of cached node metrics. With a short cache time (15s), ASRB detects CPU- and memory-induced overload within roughly 55 requests in T5, while the stress containers are still running. With a long cache time (60s), detection is delayed to around 180 requests, at which point the overload is already ending, and the stale cache also causes ASRB to avoid nodes that have already recovered. These results show that short cache intervals are essential for timely reaction to overload events, especially in dynamic edge environments where nodes may degrade or recover quickly.

5.4.7 Overload Handling Limitations of QEdgeProxy

An additional observation concerns overload behaviour in the original QEdgeProxy. While the baseline implementation detects overload conditions, it does not apply a persistent or well-defined “overloaded” label to nodes. As a result, overload handling is inconsistent: even during the T5 stress phase, QEdgeProxy occasionally routed requests to nodes that were already above 85% CPU utilization.

These fluctuations occurred because the baseline relies solely on instantaneous metric-based decisions without enforcing a cooldown or exclusion period. Consequently, an overloaded node may briefly appear healthy due to transient metric variations and is selected again, only to become overloaded once more. This leads to unstable routing behaviour and unnecessary latency spikes during overload periods.

ASRB avoids this issue through explicit overloaded labeling and cooldown logic, ensuring that nodes exceeding resource thresholds are consistently avoided until their metrics indicate recovery. This results in far more stable routing compared to the baseline approach.

5.4.8 Robustness to New-Node and Reboot Race Conditions

The behaviour observed in T2 and T3 highlights a fundamental robustness difference between QEdgeProxy and ASRB under dynamic cluster conditions. When a node joins or reboots, Kubernetes often exposes the pod object before its networking information is fully initialized. QEdgeProxy immediately treats such pods as routable, which makes it vulnerable to short “initialization gaps” in which a pod exists in the API but cannot yet receive traffic. This explains the high number of failed requests observed for the baseline.

ASRB avoids this problem through stricter admission semantics: a pod is not considered eligible until both `PodIP` and `HostIP` are populated, and any forwarding failure triggers immediate cooldown of the responsible node. As a result, ASRB experienced only two transient failures in the same scenarios, both quickly isolated.

These findings show that ASRB not only routes more effectively under normal load but also behaves predictably during disruptive events such as node joins, reboots, or rapidly changing cluster membership. This robustness is especially important in edge systems,

where nodes frequently appear and disappear due to mobility, intermittent connectivity, or limited resources.

5.4.9 Overall Assessment

Overall, ASRB demonstrates that adaptive, score-based routing with configurable λ can be applied effectively in dynamic edge environments without incurring excessive monitoring overhead.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Discussion

This chapter reflects on the results presented in the previous section and analyzes the broader implications of the Adaptive Score-based Routing Balancer. While the evaluation demonstrates clear benefits in terms of latency, accuracy trade-offs, responsiveness to dynamic conditions, and reduced monitoring overhead, it is equally important to interpret these outcomes in the context of system design choices, experimental constraints, and real-world deployment considerations.

The discussion is structured around the main strengths of the approach, its limitations, and the degree to which the findings can generalize beyond the specific testbed and workload used in this thesis. Together, these insights provide a foundation for understanding the practical relevance of ASRB and for identifying promising directions for future work.

6.1 Strengths and Contributions

The evaluation highlights several key strengths of ASRB and confirms that the system meets the design goals defined earlier.

6.1.1 QoS-Aware Routing with Tunable Trade-offs

ASRB introduces a unified scoring mechanism that combines latency and model accuracy into a single, tunable QoS metric. Through the parameter λ , applications can explicitly control the relative importance of responsiveness versus inference quality, enabling configurable routing behaviour.

6.1.2 Improved Latency Stability Under Dynamic Conditions

ASRB improves median and tail latency under dynamic conditions by stabilizing routing decisions through node labeling, cooldown logic, and stricter pod-admission rules, avoiding

degraded or not-yet-ready nodes earlier than latency-only baselines. In particular, explicit overloaded labels prevent premature re-selection of stressed nodes, reducing latency spikes during reboots and overload phases.

6.1.3 Reduced Monitoring Overhead

A key contribution of ASRB is its label-oriented adaptive monitoring strategy. By selectively refreshing metrics based on node labels and caching, ASRB reduces Kubernetes API calls and monitoring payloads while preserving timely state information for routing decisions. This design is particularly relevant for large-scale or bandwidth-constrained edge deployments, where static, unconditional monitoring becomes costly.

6.1.4 Lightweight, Fully Decentralized Design

ASRB operates without a central controller. Each proxy makes independent routing decisions using local caches, Kubernetes informers, and node labels. This decentralized design avoids bottlenecks, simplifies deployment, and remains fully compatible with standard Kubernetes mechanisms, making it well suited for heterogeneous edge-cloud environments.

Overall, ASRB contributes a lightweight and practical approach to QoS-aware routing, consolidating the key strengths observed in the evaluation.

6.2 Generalizability

ASRB's design is generic and does not depend on any model-specific or cluster-specific assumptions, making it applicable to a broad range of edge-cloud deployments. Since ASRB already achieves latency performance that stands shoulder to shoulder with established baselines such as QEdgeProxy and Proxy-Mity, while offering greater flexibility through QoS scoring and label-driven monitoring, the approach generalizes well to other heterogeneous setups, different hardware tiers, and multi-model workloads. The mechanisms for overload detection, failure handling, and dynamic adaptation remain valid as long as nodes expose basic health and latency feedback, enabling ASRB to be deployed in both small edge clusters and larger distributed infrastructures.

6.2.1 Applicability to Larger Clusters

Although the experiments were conducted on a small five-node k3s deployment, the core mechanisms of ASRB—local scoring, label-driven monitoring, and decentralized decision making—scale naturally to larger environments. Monitoring overhead grows with the number of nodes, but the relative reduction achieved by dynamic monitoring (approximately 17–41% in total overhead and 63–79% for pure monitoring traffic compared to QEdgeProxy) should hold or improve at scale, especially because static scraping, i.e., periodic unconditional polling of all nodes regardless of state changes, becomes increasingly

expensive in larger clusters. However, very large deployments may require additional coordination or backoff logic to avoid simultaneous label updates from multiple proxies.

6.2.2 Applicability Across Different Edge–Cloud Hierarchies

ASRB’s routing logic builds on generic Kubernetes abstractions, namely latency feedback, pod readiness, accuracy annotations, and node labels, without relying on any deployment-specific assumptions. Because these signals are available in standard Kubernetes clusters, the approach generalizes naturally across multi-tier edge hierarchies, heterogeneous hardware platforms (ARM, x86), mixed cloud–near-edge deployments, and geographically distributed mini-clusters. Environments with heterogeneous latency and variable resource availability benefit especially from ASRB’s adaptive scoring, which allows it to stand shoulder to shoulder with proximity-based baselines in raw latency performance while offering greater flexibility through QoS-aware routing and label-driven monitoring.

6.2.3 Workload Generalization

Although evaluated with image-classification models (MobileNet and ResNet50), ASRB’s mechanisms generalize to workloads where QoS depends on measurable performance characteristics. These may include (i) inference services with multiple model variants, (ii) video-analytics or stream-processing pipelines, (iii) sensor-fusion or time-series prediction systems, (iv) CPU- or GPU-intensive microservices, and (v) non-ML workloads with defined service-level objectives.

It should be noted, however, that applying ASRB to domains where “accuracy” is not a meaningful QoS attribute does not work fully out of the box. In such scenarios, the QoS dimension represented by accuracy must be replaced or reinterpreted based on the needs of the target application. This typically requires some engineering adjustments, such as defining an alternative metric, supplying the corresponding metadata, and ensuring that the monitoring infrastructure can observe it.

While these adaptations are necessary, the general architecture of ASRB remains applicable: the framework is designed to incorporate different QoS signals and can be extended to support multi-objective routing once suitable metrics are available.

6.2.4 Generalization to Variable Traffic Patterns

The experiment used a uniform request stream, but ASRB’s design does not assume fixed arrival patterns. Because routing decisions are made per request and rely on latency feedback, ASRB is expected to perform similarly under bursty or intermittent traffic, periodic request patterns, heterogeneous client loads, and multi-client access through the same proxy. However, scenarios involving extremely high request rates may require batching, queue-length awareness, or rate limiting to prevent oscillations.

6.2.5 Robustness in Realistic Failure Scenarios

The system proved robust to controlled events such as node arrivals, reboots, and overloads. These results suggest good generalization to common real-world conditions, including rolling updates, transient resource spikes, pod restarts, or short-lived network disruptions. More severe failures, such as long network partitions, multi-node outages, or heavy cross-tenant interference, were not evaluated and may affect resilience differently.

6.2.6 Dependence on Kubernetes Ecosystem

ASRB relies heavily on Kubernetes abstractions such as pods, labels, DaemonSets, and the Metrics API. Generalization to non-Kubernetes environments would require: reimplementing (i) the monitoring interface, (ii) the label assignment logic, and (iii) pod-level readiness and health checks. Within the Kubernetes ecosystem, including k3s, microk8s, and managed cloud Kubernetes, the approach generalizes easily with minimal adaptation.

6.2.7 Summary

Overall, the architectural principles behind ASRB transfer well to a broad spectrum of Kubernetes-based edge deployments, heterogeneous model types, and dynamic execution contexts. Although absolute performance will vary with cluster scale, workload characteristics, and network layout, the core advantages of adaptive QoS-aware routing, fast reaction to changing conditions, and lower monitoring overhead are expected to hold across diverse real-world environments.

6.3 Limitations

While ASRB demonstrates strong performance across multiple evaluation dimensions, several limitations remain and point toward areas for future improvement.

6.3.1 Limited Scale of the Testbed

The evaluation was performed on a small five-node k3s cluster. Although relative improvements are expected to generalize, absolute performance characteristics, especially monitoring overhead and latency variation, may differ in larger clusters or geographically distributed edge deployments. The system was not evaluated under large-scale topologies, multi-region layouts, or high request concurrency.

6.3.2 Reliance on Kubernetes Metrics Accuracy

ASRB depends on Kubernetes Metrics API for CPU and memory usage. These values may be delayed, coarse-grained, or unavailable on certain lightweight edge nodes. Inaccurate or stale metrics directly influence the correctness of the `overloaded` label, potentially

delaying overload detection. The system does not implement fallback heuristics when metrics are missing or inconsistent.

6.3.3 Simplified Resource Model

The current implementation considers only coarse CPU and memory usage for overload detection. Other factors, such as GPU load, thermal throttling, network contention, or I/O pressure, are not integrated. Nodes may therefore appear healthy despite significant performance degradation in non-CPU bottlenecks.

6.3.4 Latency Estimation Limitations

Latency is inferred using two mechanisms: feedback from real requests and lightweight `/echo` probing. Although effective, both approaches introduce limitations:

- `/echo` does not capture full end-to-end inference delay (model execution time).
- Network fluctuations may still cause transient instability in scoring.
- Approximation accuracy depends on proxy-to-node connectivity and transient load spikes.

The system does not incorporate confidence intervals or statistical filtering beyond smoothing.

6.3.5 Static Model Accuracy Values

Model accuracy is treated as a static annotation rather than a dynamic metric. In real-world workloads, model performance may depend on input distribution, model drift, or fine-tuning stages. ASRB does not evaluate per-request accuracy or integrate more complex QoS indicators such as confidence scores or model calibration metrics.

6.3.6 Simplified Scoring Function

The scoring formula uses a linear combination of normalized latency and accuracy. This provides interpretability but does not capture diminishing returns, non-linear QoS preferences, task-specific utility curves, or request-type heterogeneity. More advanced multi-objective utility models were not explored.

6.3.7 Sensitivity to Parameter Configuration

ASRB exposes numerous configuration parameters (e.g., cache times, smoothing weights, thresholds). While flexible, this increases the risk of suboptimal tuning. Incorrect values may cause stale data usage, oscillating labels, or unnecessary probing. Automated tuning or adaptive configuration is not implemented.

6.3.8 No Cross-Proxy Coordination

The decentralized design improves scalability but also limits coordination. Each proxy maintains its own local view of the cluster, which can lead to temporary inconsistencies, delayed convergence on new labels, and duplicate or conflicting Kubernetes patch requests. Stronger cross-node synchronization might further improve responsiveness at scale.

6.3.9 Limited Workload Diversity in Evaluation

The evaluation workload consists of repeated inference requests using a single static image. While this isolates routing behavior, it does not reflect fluctuating input sizes, mixed workloads, variable arrival patterns, or bursty traffic. Real-world services may exhibit higher temporal variance that could stress the system differently.

6.3.10 Lack of Energy-Aware Routing

The system does not consider energy usage or battery constraints on edge devices. For mobile or energy-sensitive deployments, accuracy–latency trade-offs alone may be insufficient.

6.3.11 Absence of Network-Layer Awareness

ASRB does not incorporate network topology, bandwidth, or link-quality information beyond latency. Packet loss, jitter, or multi-hop routing paths could meaningfully affect QoS in wide-area or hierarchical edge networks.

6.3.12 Single-Request Optimization

Routing decisions are made on a per-request basis without considering flow affinity, session stickiness, batching opportunities, or queue length at pods. These factors may influence system behavior under high load or when inter-request dependencies exist.

6.3.13 Evaluation Constraints

Certain behaviors, such as pod churn, container restarts, network partitions, and multi-tenant interference, were not systematically tested. The evaluation testbed also used relatively stable hardware without aggressive resource contention or background workloads.

6.3.14 Summary

Despite these limitations, the experimental results show that ASRB is robust and effective within the tested scope. Nevertheless, its performance and general applicability can be further improved by addressing the identified constraints.

Conclusion and Future Work

This chapter summarizes the main findings of the thesis and reflects on the practical implications of the Adaptive Score-based Routing Balancer. The work demonstrates that QoS-aware routing based on combined latency and accuracy scoring can significantly improve the performance and stability of edge-based inference services, while also reducing monitoring overhead through adaptive mechanisms. At the same time, several opportunities remain to extend and generalize the approach. The following sections present the core conclusions of the research and outline directions for future development.

7.1 Summary of Findings

The research presented in this thesis shows that ASRB provides clear benefits over existing routing strategies for edge-based inference services. By combining latency measurements, model accuracy annotations, and adaptive monitoring into a single scoring and decision-making framework, ASRB consistently delivers lower median latency, improved tail performance, and predictable accuracy–latency trade-offs controlled through the parameter λ .

The six-phase experiment (T1–T6) demonstrates that ASRB reacts quickly to dynamic cluster events such as node arrivals, reboots, pod deployments, and overload conditions. Its label-driven monitoring and strict pod-selection rules allow the system to avoid degraded nodes earlier than baseline methods, resulting in more stable routing behavior and fewer performance spikes.

A key contribution is the reduction of monitoring overhead. Compared to the original QEdgeProxy, ASRB lowers total monitoring traffic by approximately 17–41%, and pure monitoring traffic by 63–79%, while achieving better routing decisions. This is made possible by selectively refreshing metrics based on node labels and maintaining lightweight local caches of latency and score data.

Overall, the evaluation confirms that ASRB is an effective, decentralized, and scalable approach for QoS-aware request routing in heterogeneous edge–cloud environments. The system improves performance, reduces overhead, and adapts reliably to changing runtime conditions without requiring central coordination.

7.2 Future Research Directions

The results of this thesis highlight several promising directions for extending ASRB and its adaptive monitoring mechanisms.

7.2.1 Adaptive Monitoring Based on Traffic Intensity

The current monitoring strategy adapts based on node labels but does not consider traffic volume. In real deployments, request load strongly correlates with the likelihood of overload or congestion. When traffic is high, nodes may degrade rapidly, making frequent resource checks desirable. When traffic is low, the cluster state typically changes more slowly, allowing monitoring intervals to be reduced. An adaptive scheme that adjusts monitoring frequency according to observed request rates could further reduce overhead while improving responsiveness during peak periods.

7.2.2 Learning-Based Extensions

A natural extension of ASRB is to replace or augment its hand-crafted scoring function with a learning-based controller. Reinforcement learning-based load balancers [SWDTS24] demonstrate that policies can be optimised directly for end-to-end QoS metrics such as tail latency or SLA violations.

In the context of ASRB, an RL agent could dynamically adapt parameters such as λ , cache times, or monitoring frequencies based on observed traffic patterns and cluster conditions, while still relying on the existing label and monitoring infrastructure. This would preserve the operational advantages of the current design while enabling more nuanced and workload-specific routing strategies.

7.2.3 Richer QoS Dimensions

Future work may extend the scoring function with additional QoS dimensions such as energy usage, GPU utilization, per-request model confidence, output quality metrics, or monetary cost. Integrating multiple objectives, potentially through non-linear scoring or utility-based optimization, would make ASRB applicable to a broader range of services.

7.2.4 Dynamic Accuracy Estimation

Model accuracy is treated as static metadata in the current implementation. Incorporating runtime confidence scores, drift detection, or online accuracy estimation would enable routing decisions that react to changes in input distribution or model behavior.

7.2.5 Improved Latency Estimation

Latency probing could be enhanced through more sophisticated filtering or by integrating jitter, packet loss, and multi-hop network information. Confidence intervals, stability windows, or weighted history could improve decision stability in noisy environments.

7.2.6 Cross-Proxy Coordination

ASRB currently operates fully independently on each node. Although this simplifies design, future variants may benefit from lightweight coordination mechanisms to share aggregated latency statistics, confirm label updates, or avoid redundant Kubernetes patches during rapid topology changes.

7.2.7 Interaction with other orchestration dimensions

This thesis addressed the request routing problem in relative isolation from other orchestration problems, such as service placement and resource allocation. However, interesting problems and opportunities emerge when its interplay with these orchestration dimensions is considered. Combining ASRB with Horizontal Pod Autoscalers, KEDA (Kubernetes-based Event Driven Autoscaling),¹ or container admission policies could create a closed-loop system in which routing, scaling, and scheduling co-adapt based on shared QoS objectives.

7.2.8 Summary

Overall, future research can evolve ASRB into a more adaptive and more expressive framework capable of coordinating QoS-aware routing across larger and more diverse edge deployments. The adaptive monitoring principles developed in this thesis form a solid basis for these extensions, offering a clear path toward richer QoS models, dynamic workload integration, and scalable multi-node coordination.

¹<https://keda.sh>



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Overview of Generative AI Tools Used

The author used AI-assisted tools (OpenAI ChatGPT) for text refinement and for receiving suggestions on structure and phrasing. All technical content, analysis, implementation, and final formulations were created and verified independently by the author.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

5.1	Cluster topology used in the evaluation. Worker3 is the closest node to the client device.	49
5.2	Latency distribution across all routing algorithms.	54
5.3	Tail latency (CDF) comparison.	56
5.4	Accuracy distribution across routing strategies.	57
5.5	Latency vs. λ for ASRB.	58
5.6	Accuracy vs. λ for ASRB.	59
5.7	Dynamic latency across T1–T6 for ASRB ($\lambda = 1$).	60
5.8	Dynamic latency for QEdgeProxy baseline.	61
5.9	Dynamic latency for Proxy-Mity ($\alpha = 1$).	61
5.10	Dynamic latency for Round Robin.	62

List of Tables

2.1	Qualitative comparison of MobileNet-V3 and ResNet-50	11
3.1	Comparison of Static vs. Dynamic Routing Strategies	27
5.1	Per-phase dominant node per algorithm (T1–T6).	63
5.2	Load distribution and Jain’s Fairness Index per algorithm.	64
5.3	Monitoring overhead: API calls and payload bytes per configuration.	66



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

3.1	ASRB Score-Based Request Routing Algorithm	26
3.2	Asynchronous latency refresh routine	26



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

- [ADJJ⁺24] Auday Al-Dulaimy, Matthijs Jansen, Bjarne Johansson, Animesh Trivedi, Alexandru Iosup, Mohammad Ashjaei, Antonino Galletta, Dragi Kimovski, Radu Prodan, Konstantinos Tserpes, et al. The computing continuum: From iot to the cloud. *Internet of Things*, 27:101272, 2024.
- [BFS⁺25] Luiz F. Bittencourt, Roberto Rodrigues Filho, Josef Spillner, Filip De Turck, José Santos, Nelson L. S. da Fonseca, Omer F. Rana, Manish Parashar, and Ian T. Foster. The computing continuum: Past, present, and future. *Comput. Sci. Rev.*, 58:100782, 2025.
- [CBJ⁺23] Breno G. S. Costa, Abhik Banerjee, Prem Prakash Jayaraman, Leonardo R. Carvalho, João Bachiega Jr., and Aleteia Araujo. Achieving observability on fog computing with the use of open-source tools. In *Proc. 20th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2023)*, 2023.
- [CI20] Emiliano Casalicchio and Stefano Iannucci. The state-of-the-art in container technologies: Application, orchestration and security. *Concurr. Comput. Pract. Exp.*, 32(17), 2020.
- [CJZ⁺24] Ivan Cilic, Valentin Jukanovic, Ivana Podnar Zarko, Pantelis A. Frangoudis, and Schahram Dustdar. Qedgeproxy: QoS-aware load balancing for IoT services in the computing continuum. In *Proc. IEEE EDGE*, 2024.
- [Clo23] Cloud Native Computing Foundation. Kubernetes. <https://kubernetes.io>, 2023. <https://kubernetes.io>.
- [DBK⁺21] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proc. 9th International Conference on Learning Representations (ICLR)*, 2021.
- [DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186, 2019.

- [DPD22] Schahram Dustdar, Victor Casamayor Pujol, and Praveen Kumar Donta. On distributed computing continuum systems. *IEEE Transactions on Knowledge and Data Engineering*, 35(4):4092–4105, 2022.
- [DZF⁺20] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y Zomaya. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020.
- [Fou23] Cloud Native Computing Foundation. K3s - Lightweight Kubernetes. <https://docs.k3s.io>, 2023. <https://docs.k3s.io>.
- [FP19] Ali J. Fahs and Guillaume Pierre. Proximity-aware traffic routing in distributed fog computing platforms. In *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 410–419. IEEE, 2019.
- [GvKJ⁺25] Panagiotis Giannakopoulos, Bart van Knippenberg, Kishor Chandra Joshi, Nicola Calabretta, and George Exarchakos. Key metrics for monitoring performance variability in edge computing applications. *EURASIP J. Wirel. Commun. Netw.*, 2025(1):38, 2025.
- [HMD16] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*, 2016.
- [HZC⁺17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
- [HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [JCH84] Rajendra K. Jain, Dah-Ming Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *DEC Research Report TR-301*, 1984.
- [Joc20] Glenn et al. Jocher. YOLOv5 by Ultralytics. <https://github.com/ultralytics/yolov5>, 2020. Accessed: 2024-12-10.
- [KFDS23] Vasileios Karagiannis, Pantelis A. Frangoudis, Schahram Dustdar, and Stefan Schulte. Context-aware routing in fog computing systems. *IEEE Trans. Cloud Comput.*, 11(1):532–549, 2023.
- [Kha17] Asif Khan. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Comput.*, 4(5):42–48, 2017.

- [LLC⁺] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows.
- [MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report NIST Special Publication 800-145, National Institute of Standards and Technology, 2011.
- [NMP⁺20] Stefan Nastic, Andrea Morichetta, Thomas W. Puztai, Schahram Dustdar, Xiaoning Ding, Deepak Vij, and Ying Xiong. SLOC: service level objectives for next generation cloud computing. *IEEE Internet Comput.*, 24(3):39–50, 2020.
- [NPK22] Quang-Minh Nguyen, Linh-An Phan, and Taehong Kim. Load-balancing of kubernetes-based edge computing infrastructure using resource adaptive proxy. *Sensors*, 22(8):2869, 2022.
- [PBSJ19] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Trans. Cloud Comput.*, 7(3):677–692, 2019.
- [PCPK15] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *Proc. ICASSP*, pages 5206–5210, 2015.
- [PNM⁺22] Thomas Puztai, Stefan Nastic, Andrea Morichetta, Víctor Casamayor Pujol, Philipp Raith, Schahram Dustdar, Deepak Vij, Ying Xiong, and Zhaobo Zhang. Polaris scheduler: Slo-and topology-aware microservices scheduling at the edge. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pages 61–70. IEEE, 2022.
- [QSDE24] Lamees M. Al Qassem, Thanos Stouraitis, Ernesto Damiani, and Ibrahim M. Elfadel. Containerized microservices: A survey of resource management frameworks. *IEEE Trans. Netw. Serv. Manag.*, 21(4):3775–3796, 2024.
- [RLF⁺20] Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing plausible infrastructure configurations for evaluating edge computing systems. In *Proc. 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 2020)*, 2020.
- [RV15] Björn Rabenstein and Julius Volz. Prometheus: A Next-Generation monitoring system (talk). Dublin, May 2015. USENIX Association.
- [Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

- [SWDTS24] José Santos, Tim Wauters, Filip De Turck, and Peter Steenkiste. Towards optimal load balancing in multi-zone kubernetes clusters via reinforcement learning. In *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2024.
- [TGD25] Imane Taleb, Jean-Loup Guillaume, and Benjamin Duthil. A survey on services placement algorithms in integrated cloud-fog / edge computing. *ACM Comput. Surv.*, 57(11):271:1–271:36, 2025.
- [TW20] Ming Tang and Vincent WS Wong. Deep reinforcement learning for task offloading in mobile edge computing systems. *IEEE Transactions on Mobile Computing*, 21(6):1985–1997, 2020.
- [WRY⁺22] Qingkun Wang, Yi Ren, Saqing Yang, Jianbo Guan, Bao Li, Jianfeng Zhang, and Yusong Tan. Proxydwr: a dynamic load balancing approach for heterogeneous-cpu kubernetes clusters. In *2022 IEEE International Conference on Joint Cloud Computing (JCC)*, pages 65–72. IEEE, 2022.
- [ZSX⁺23] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proc. 2023 ACM Symposium on Cloud Computing (SoCC)*, pages 142–157, 2023.