Tail-Latency-Aware Scheduling and Graceful Degradation in Edge-Cloud Continuum

Yinan Cao

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

September 2026

ICTEAM
Louvain School of Engineering
Université Catholique de Louvain
Louvain-la-Neuve, Belgium

Thesis Committee:

Pr. Etienne Rivière **Supervisor**	UCLouvain, Belgium
Pr. Ramin Sadre **Supervisor**	UCLouvain, Belgium
Pr. Charles Pecheur **President of Jury**	UCLouvain, Belgium
Pr. Cristel Pelsser **Secretary of Jury**	UCLouvain, Belgium
Dr. Christian Perez **Member of Jury**	Inria, France
Pr. Anne Remke **Member of Jury**	University of Münster, Germany

Tail-Latency-Aware Scheduling and Graceful Degradation in Edge-Cloud Continuum

by Yinan Cao

© Yinan Cao 2026

ICTEAM
Université Catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was supported by Project RAINDROP, an ARC project funded by the Fédération Wallonie-Bruxelles, Belgium, and by Grid'5000, France.

"All models are wrong,
but some are useful."
GEORGE E. P. BOX

Abstract

With the proliferation of latency-critical services and Edge AI, the edge-cloud continuum has emerged as a vital infrastructure for processing data closer to users. Managing resources in these environments is challenging, however, due to the heterogeneous network latencies between users and sites, the limited capacity of edge infrastructures, and the nonlinear queueing dynamics of containerized applications under load. Existing orchestrators rely on hardware utilization metrics or mean performance statistics and consequently fail to provide strict guarantees on tail latency, leading to Service Level Objective (SLO) violations during periods of contention or resource scarcity.

This thesis proposes a unified framework for latency-aware scheduling and quality assurance, developed through four progressive contributions. We first establish an analytical foundation, introducing a queueing-theoretic model that predicts the end-to-end tail latency of containerized applications by faithfully capturing the per-instance request dispatching performed by Kubernetes. We then turn this prediction into a decision procedure, formulating the placement and scaling of a single application as a mixed-integer linear program whose tail-latency constraints are linearized without loss of accuracy. Extending the formulation to multi-tenant environments, we jointly schedule co-located applications with heterogeneous resource footprints under a shared capacity constraint, and introduce heuristics that keep the resulting problem tractable as the number of applications grows. To address the hard capacity limits of the edge, we then introduce a graceful degradation mechanism based on mixed-criticality principles, allowing the system to protect high-criticality workloads while preserving bounded service for low-priority ones through controlled and predictable trade-offs. Finally, we specialize this approach for Edge AI, jointly optimizing model variant selection, placement, and scaling so that resource shortfalls are absorbed by reducing inference quality rather than responsiveness. Validated on real distributed testbeds, the proposed methodology consistently satisfies SLO compliance, improves resource efficiency, and maximizes service quality under varying levels of resource pressure.

Résumé

Avec la prolifération des services critiques en latence et de l'IA en périphérie, le continuum edge-cloud s'est imposé comme une infrastructure essentielle pour traiter les données au plus près des utilisateurs. La gestion des ressources y demeure toutefois difficile, en raison de l'hétérogénéité des latences réseau entre utilisateurs et sites, de la capacité limitée des infrastructures de périphérie, et de la dynamique non linéaire des files d'attente des applications conteneurisées en charge. S'appuyant sur des métriques d'utilisation matérielle ou des statistiques de performance moyennes, les orchestrateurs existants échouent à garantir strictement la latence de queue, ce qui conduit à des violations des objectifs de niveau de service (SLO) en période de contention ou de pénurie de ressources.

Cette thèse propose un cadre unifié pour l'ordonnancement conscient de la latence, développé à travers quatre contributions progressives. Nous établissons d'abord un fondement analytique fondé sur la théorie des files d'attente, capable de prédire la latence de bout en bout en queue de distribution des applications conteneurisées, en reproduisant fidèlement la répartition des requêtes par instance effectuée par Kubernetes. Nous transformons ensuite cette prédiction en procédure de décision, en formulant le placement et le dimensionnement d'une application comme un programme linéaire en nombres entiers mixtes, dont les contraintes de latence sont linéarisées sans perte de précision. Étendant la formulation aux environnements multi-locataires, nous ordonnançons conjointement des applications co-localisées aux empreintes de ressources hétérogènes sous une contrainte de capacité partagée, et introduisons des heuristiques qui maintiennent le problème résoluble à mesure que le nombre d'applications augmente. Pour répondre aux limites de capacité de la périphérie, nous introduisons un mécanisme de dégradation gracieuse fondé sur les principes de criticité mixte, qui protège les charges de travail les plus critiques tout en préservant un service borné pour celles de moindre priorité. Enfin, nous spécialisons cette approche à l'IA en périphérie, en optimisant conjointement la sélection des variantes de modèles, leur placement et leur mise à l'échelle, de sorte que les déficits de ressources soient absorbés par une réduction de la qualité d'inférence plutôt que de la réactivité. Validée sur des bancs d'essai distribués réels, la méthodologie proposée assure le respect des SLO, améliore l'efficacité des ressources et maximise la qualité de service sous différents niveaux de pression.

Acknowledgments

I personally consider this section to be the most challenging part of the thesis, even more so than the theories and equations. There are simply too many people to whom I owe my deepest gratitude, and the brief words on these pages seem far from enough.

First and foremost, my most sincere gratitude goes to my two supervisors, Prof. Ramin Sadre and Prof. Etienne Rivière, for their unwavering guidance and support throughout my doctoral journey. They opened the gateway to science and guided me through the misty, uncharted forest of research, arming me with the swords of methodology and the arrows of theory. When storms raged and the path grew treacherous, their patience and passion served as my shield and fortress. At the end of this journey, I come to realize that the title itself fades in comparison to the wisdom they imparted (or at least tried their best to), and above all, the inspiring example they set as mentors, which will remain with me for a lifetime.

I would like to extend my gratitude to the members of my jury. My heartfelt thanks go to Dr. Christian Perez from Inria, France, and Prof. Anne Remke from the University of Münster, Germany, for taking the time to travel to my defense and for their patient, insightful, and inspiring questions. Special thanks also go to Prof. Charles Pecheur and Prof. Cristel Pelsser, who served as the president and secretary of the jury, respectively, for their warm encouragement and steadfast support. My gratitude also goes to Prof. Pierre Schaus, who unreservedly inspired me in optimization, a critical component of my research with which I was initially unfamiliar. My sincere thanks also go to Prof. Anthony Cleve and Maxime André from UNamur for the insightful discussions during our meetings.

From the very first day I embarked on my doctoral journey at UCLouvain, the wonderful people of the INGI department have made my life more colorful and joyous than I ever dared to imagine. My thanks go first to Fabien and Igor, the postdocs who kicked off my research with their unreserved encouragement and support. Next, to my dearest François "Framinem" DK and Gorby "Les Gorbes", Lionel, Rémi, and Pierre "Rapunzel", for all the support, brainstorming, and laughter we shared in A.253. My appreciation also extends to those who were once in or around the office (cafetaria included): Mathias the B-to-be, Bastien the coolest "flame-on", Patient, John, Benoît le BG, CHBVO, Rosana, Edouard, Benoît R, Hored, Momo, Ioannis, Carole, Louis

and Emma. To Matthieu and Christophe, I deeply enjoyed each of our smoke breaks and lovely jokes. To the lovely vibrant crew upstairs: Nikita, Clément, Alessandro, and Soyong, thank you for bringing so much energy. To Donatien, Pierre M and Samy, I thoroughly enjoyed all our shared TPs and am grateful for all the help. Last but not least, my heartfelt thanks to Vanessa, the GSOAT, who has always been there with her assistance full of patience, no matter how busy she was.

Outside of work, I am deeply grateful to those who brought inspiration, joy, and unwavering support to my personal life. Special thanks to Xinxin, my BFF (*yuk*), whom I am truly blessed to know, and who brought me to a group of wonderful basketball buddies including Tom, Tonton, M. Fâcher and many others. To Amand, Andrea and Yod, I deeply cherished every moment spent in your company. My thanks also go to all the gym folks who witnessed and encouraged my physical progress over these years.

Finally, my family. There is no way I could have come this far without the unconditional love and support of my beloved parents. Nothing compares to the depth of their love, and I feel endlessly fortunate to be their child. Before I finish, I dedicate this special space to Dr. Yanan Huo: thank you for everything you have done for me, and for every single day we share together alongside Higgins and Chopin. You are the gentle anchor of my restless soul, and the sweetest home I could ever wish for.

As the honorably appointed Official Chartreuse Guy (OCG), I say: *"We'll tak a cup o' kindness yet"* and, **cul sec!**

致谢

此致谢始于文成后二月有余，本以为成竹于胸，是时必能文如泉涌；然启笔时却如鲠在喉，如芒在背，心绪万千，以致辞不达意笔不从心者，十之八九。虽师夷长技，余欲于满篇西文之中，揽我母语一席之地；然飘零海外，去国近十载，久疏汉墨，固知文采浅陋，深思再三，终执此意。此非泥古守旧、守祖宗之法不化，盖因华夏文脉，博大精深，历久而弥新。遂不揣冒昧，舞文弄墨，以昭华夏文化之深韵，彰炎黄薪火之绵长。

余幼时即嗜玩，胸无大志，不谙世事。兜转浮沉，机缘巧合，始萌攻读博士之念，遂履科研之途，时光荏苒，转眼已至而立之年。幸得二位恩师倾囊相授，谆谆教诲，有教无类，受益良多。求学十余载，唯此四年最为澄明，始得以摒除纷扰，静心治学，终能独立思考，尽己所能，方得略窥科学之一隅，粗通计算之精妙。世人常叹未遇伯乐以致才不加施，然际遇之日，又恐己之学浅才疏不足以当其识拔与厚恩也。

本谓身寄异邦，举目无亲，当孤灯独行；幸得诸君相伴，佳人相依，四载光阴由是绚烂，寸心皆驻胜景。此间常有崎岖坎坷、不遂人意之时，幸得朋辈同道并肩，或研机析理于困顿之间，或共饮倾谈于仲夏之夜，同舟共济，使万里异乡亦复有平生欢愉；更幸得至爱佳偶相濡，朝夕相处间温我平生，乃此身羁旅中最温柔之锚泊。世人常叹未遇知音佳偶以致身孤寂而心难安，然际遇之日，又恐己之所付不足以当其所授之欢愉与依恃也。

自弱冠离家，挂帆远游，双亲始终如岳立于后，默默垂护。幼蒙庭训，常谓严教乃系望子成龙之寄、托成才报国之期；然及至而立之年，方悟双亲所祈，不过许余以己之意，度余之一生，纵使此途须以久缺晨昏定省为代价，纵隔万里仍蒙倾力庇荫，亦未尝萌丝毫怨言。爱子之深，计之深远，感念于心，结草难酬。世人常叹当奉亲尽孝以致恪守于心，然当报之日，又恐己之绵薄何以酬其生鞠劬劳之深恩也。

凡百过往，皆为序章。三十载身经际遇，尝谓独一无二、波澜独具；然若通观史册，方知红尘跌宕、荣枯浮沉，皆已被古人阅尽千帆，吾辈之喜怒交集、心境起伏，亦早被先贤剖析至尽，所谓：“世殊事异，所以兴怀，其致一也。”而于科学之长河，余所拼凑之绵薄微功，得博士之微名，亦无非凭仁前人肩膀，归纳阐发，以奉用于今世而已。世人常叹身如蜉蝣以致功名渺茫，然归去来兮，方觉以梦为马，不负岁华；心之所向，无愧自我。余愿怀此敬畏与热忱，不困于过往，不畏于未来，自度度人，坦荡前行。

Introduction 1

The rapid digitization of society has ushered in an era of ubiquitous, real-time services that demand instantaneous interaction between the physical and digital worlds. Modern applications have evolved from simple static content delivery to complex, data-driven ecosystems, such as immersive Augmented and Virtual Reality (AR/VR) [1], autonomous vehicular networks, and real-time industrial monitoring [2]. These services increasingly rely on high-frequency data processing and Artificial Intelligence (AI) [3] to ensure user safety and Quality of Experience [4]. To support this evolution, the industry has shifted toward containerization and microservice architectures, which allow applications to be decoupled from underlying hardware and flexibly deployed across diverse infrastructures [5–7]. The underlying systems must provide not only high throughput but also deterministic, ultra-low latency performance. This shift has challenged the traditional centralized cloud model, pushing the boundaries of distributed systems toward the network edge and creating a multi-tiered Distributed Edge-Cloud Hierarchy. This thesis explores the intersection of resource orchestration and performance modeling within this evolving landscape, aiming to provide robust quality guarantees for the next generation of latency-sensitive applications.

1.1 Context

The digital landscape is undergoing a rapid architectural evolution, transitioning from centralized Cloud Computing to a multi-tiered Distributed Edge-Cloud Hierarchy, as illustrated in Figure 1.1. While the cloud provides vast, elastic resources, its physical distance from the majority of end-users introduces significant network propagation delays that are unacceptable for modern interactive services [8–10]. To address this, Edge Computing has emerged as a critical extension, placing computational resources at the network periphery to process data closer to users [11]. This hierarchy allows applications to be dynamically orchestrated across both high-capacity cloud centers and low-latency edge sites. However, this environment is characterized by extreme hardware heterogeneity and a fundamental tension: the cloud offers near-infinite capacity at the cost of high geographical latency, while edge sites offer low-latency access but are inherently limited in capacity and geographically constrained [8].

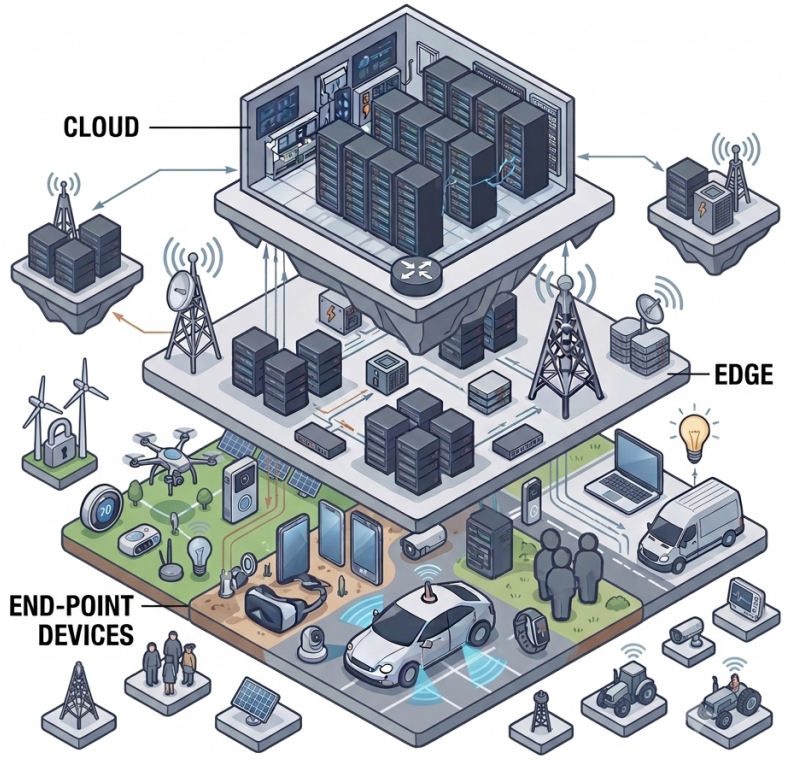


Figure 1.1: A typical edge-cloud continuum infrastructure.

The primary gap in current orchestration systems [9, 10, 12–19] lies in the inability of widely used container orchestration systems to provide strict tail-latency guarantees across these distributed tiers. Existing schedulers and autoscalers typically rely on hardware consumption metrics, such as CPU or RAM utilization [13,17,19,20]. Some proactive approaches managed to schedule the workload but mainly based on limited performance statistics, such as mean round-trip time of users' requests [10, 12, 16]. These metrics are often latency-blind: they fail to account for the non-linear queueing delays that form on resource-constrained edge nodes and for the network propagation delays caused by the geographical distribution of edge sites relative to their users. Consequently, even when hardware utilization appears within safe limits, requests may still suffer long tail latencies that violate Service Level Objectives (SLOs). Bridging this gap requires a unified scheduling methodology that evolves from precise analytical provisioning to graceful degradation and mixed-criticality awareness, ensuring service continuity even when resources are no longer abundant.

A second, equally consequential gap emerges once the edge infrastructure approaches its capacity limits. Unlike the cloud, where additional resources

can be provisioned on demand, edge clusters operate under a hard physical ceiling, and during peak demand the aggregate resource requirement of all co-located applications may simply exceed what the infrastructure can offer. Existing orchestration systems provide no principled response to this situation: they either declare the scheduling problem infeasible, evict applications abruptly to reclaim their resources [12, 16, 17], or allow SLOs to be violated in an uncontrolled and unpredictable manner [21–23]. Bridging both gaps requires a unified scheduling methodology that evolves from precise analytical provisioning to graceful degradation and mixed-criticality awareness, ensuring service continuity within acceptable levels even when resources are no longer abundant.

1.2 Summary of Contributions

Building on the context established above, this thesis identifies and addresses a series of progressively more complex challenges that current orchestration systems fail to handle. The foundational challenge is one of precision: given an application, a population of geographically distributed users, and a tail-latency SLO, how many instances should be deployed and at which sites? The answer is far from obvious, because tail latency is not a linear function of the load. Placing an application at the closest edge site minimizes network latency but exposes it to saturation as demand grows, at which point queueing delays escalate sharply and the tail of the latency distribution deteriorates far faster than its mean. Without the ability to anticipate this behavior before it occurs, operators are left to choose between massive over-provisioning, which wastes expensive and limited edge resources, and frequent SLO violations during peak traffic.

Beyond provisioning a single service, the multi-tenant nature of edge-cloud continuum infrastructure introduces the challenge of mutual interference. Concurrent applications compete for the same shared resources [8], and standard load-balancing mechanisms do not account for this contention [12, 20], leading to unpredictable *Noisy Neighbor* effects [24] that degrade tail-latency performance across the entire edge-cloud continuum clusters.

A further challenge arises from the computing resources ceiling of the edge clusters. In scenarios of extreme demand, the system may reach a state of resource scarcity where SLOs can no longer be met through scaling alone. Existing platforms lack a safety valve for such situations and may fail unpredictably, with no mechanism to protect mission-critical tasks from the performance degradation caused by resource contention between applications, or even unavailability due to abrupt evictions of services [21, 23].

This last challenge takes on a particular acuity in the case of Edge AI applications, whose rapid proliferation is reshaping the workloads that edge clus-

ters must accommodate. Deep learning inference is computationally heavy: a handful of such services can exhaust a cluster that would comfortably host dozens of conventional applications, so scarcity ceases to be an exceptional condition and becomes the ordinary operating regime [25]. Compounding this, the criticality of co-located AI services can differ enormously, from a vision pipeline detecting hazards in an industrial workspace to a recommendation engine serving nearby visitors, yet the coarse instruments available to existing orchestrators, namely scaling, throttling, and eviction, are blind to this distinction. They can decide only whether an application runs or does not, and offer no means of expressing which services must be shielded from resource pressure and which can absorb it, nor by how much [23, 26].

To address these challenges, this thesis presents a unified methodology for latency-aware orchestration, developed through the following contributions:

- We establish a formal analytical framework that models both the infrastructure and the requests it serves. On the system side, we define the architectural elements of the edge-cloud hierarchy: the cloud and edge sites forming the continuum, the uniform resource slots through which their capacity is expressed, and the Local User Groups (LUGs) abstracting the geographical distribution of users and their network distances to each site. On the request side, we decompose the end-to-end Round-Trip Time (RTT) into its three constituent parts, the network latency, the queueing delay, and the request processing time. Leveraging queueing theory, we then derive a model that predicts the mean as well as any specified tail-latency percentile, such as P95 or P99, providing a mathematical foundation for performance estimation in stochastic edge environments.

- Building on this foundation, we introduce Lassy, a latency-aware scheduler that operationalizes the analytical model. A key element is that it faithfully captures the internal load-balancing dynamics of Kubernetes, modeling how kube-proxy distributes traffic independently across replicas. Lassy transforms scheduling into an optimization problem with customizable objectives, such as cost minimization or resource efficiency, and determines the number and placement of replicas required to satisfy an application's P99 tail-latency SLO.

- We extend the methodology to multi-application concurrency with the Scylla scheduler. Recognizing that edge nodes are rarely dedicated to a single task, Scylla generalizes the single-application formulation to jointly schedule co-located applications competing for shared resources, ensuring that all meet their respective SLOs without mutual interference. As joint optimization causes the problem scale and computa-

tion time to grow rapidly, we introduce the *Layered Permutation Search* heuristic, which decomposes the global problem into a sequence of single-application problems explored at controllable depth, achieving near-optimal placements within practical time limits.

- To address scenarios where edge resources are insufficient to meet all demands, we introduce Cerberus and its graceful degradation mechanism. Building on mixed-criticality principles [27], it keeps high-priority services fully compliant with their SLOs by systematically and transparently degrading lower-priority ones within user-defined bounds, replacing the unpredictable failures of standard orchestrators with controlled, predictable trade-offs. Its degradation procedure invokes the scheduler repeatedly, motivating the *Monte Carlo Sampling* heuristic, which explores application orderings by random sampling rather than the depth-bounded search, offering a different point in the quality-runtime trade-off.

- We specialize the framework for Edge AI with the Occamy scheduler, which leverages a property unique to Deep Neural Network workloads: the same inference task can be served by multiple model variants trading accuracy for inference time and resource footprint. By jointly optimizing variant selection, placement, and replica scaling, Occamy absorbs resource shortfalls by reducing inference accuracy rather than responsiveness, preserving latency compliance for all applications even under extreme resource pressure.

- Finally, we design a complete experimental framework that validates every contribution on real infrastructure rather than in simulation. All experiments run on the Grid'5000 [28] testbed, where we deploy a Kubernetes cluster over bare-metal nodes, emulate the WAN latencies of the edge-cloud continuum, and generate Poisson request workloads. The full pipeline, from node reservation and cluster deployment to network emulation, workload injection, and measurement collection, is automated with EnOSlib [29], ensuring that all results are reproducible and that scenarios can be reconfigured at scale.

Together, these contributions form a coherent methodology for managing latency-sensitive applications, progressing from performance modeling to quality-aware resilience under resource pressure.

Naming of the Systems. The four scheduling systems presented in this thesis are named with a recurring canine and mythological theme, reflecting the guardian role each plays over the applications it schedules. *Lassy* is an acronym for *Latency-Aware SLO-Sufficing* scheduler, while also evoking

Lassie, the faithful watchdog. *Scylla* takes its name from the sea monster that guarded a narrow strait, much as the scheduler routes the traffic of many applications through the limited capacity of the edge. *Cerberus*, the many-headed watchdog of the underworld, oversees multiple co-located applications at once, protecting the most critical under resource scarcity. *Occamy*, a serpentine creature that grows or shrinks to fit the space available to it, mirrors the accuracy-aware scheduler that expands or contracts its model variants to fit the resources at hand.

1.3 Structure of the Thesis

The remainder of this thesis is organized as follows:

- **Chapter 2** establishes the background and motivation for the research presented in this thesis. It introduces the edge-cloud continuum infrastructure, describes how modern applications are deployed and orchestrated using containerization and Kubernetes, and characterizes the workloads and quality-of-service requirements of their users. It concludes by identifying the key open challenges that motivate the contributions of the following chapters.
- **Chapter 3** develops the queueing-theoretic latency model that serves as the analytical foundation for all scheduling systems presented in this thesis. It formalizes the system model, presents the $M|D|1$ queueing model as the appropriate tool for characterizing per-instance queueing behavior in Kubernetes deployments, and derives closed-form expressions for both mean and tail latency prediction.
- **Chapter 4** presents the experimental methodology shared by all contributions of this thesis, including the Grid'5000 testbed, the Kubernetes deployment, the network emulation of the edge-cloud continuum, the workload generation, the slot-based resource allocation model, the target applications, and the baseline systems. Within this framework, it empirically validates the latency model of Chapter 3, demonstrating that it accurately predicts tail latencies of real containerized deployments and correctly determines the number of instances required to meet a tail-latency SLO at a single site.
- **Chapter 5** builds on the validated latency model to formulate the single-application scheduling problem in the edge-cloud continuum as a combinatorial optimization problem. It presents Lassy, a Latency-Aware SLO-Sufficing Scheduler that computes deployment plans minimizing operational costs while provably satisfying tail-latency SLOs, and evaluates its effectiveness in realistic multi-site scenarios.

- **Chapter 6** addresses the complexities of resource contention by building upon the core model. It explores how the internal load-balancing mechanisms of Kubernetes, such as kube-proxy [30], impact request distribution and tail latency. We propose a global optimization strategy that coordinates the placement and scaling of multiple concurrent applications in shared edge environments.

- **Chapter 7** investigates the challenge of resource scarcity. We introduce the concept of mixed-criticality into our scheduling framework, allowing the system to distinguish between mission-critical and best-effort services. We present a novel mechanism for graceful degradation, where the performance of low-priority tasks is strategically reduced to preserve the integrity of high-priority SLOs during periods of extreme load or hardware failure.

- **Chapter 8** specializes the evolved framework for Edge AI workloads. We move beyond black-box application models to leverage the inherent flexibility of Deep Neural Network variants. We describe an integrated approach that jointly optimizes model selection, node placement, and replica scaling to maximize global service quality while maintaining strict adherence to P99 latency targets.

- **Chapter 9** concludes the thesis with a summary of the major findings and the scientific impact of our work. We reflect on the evolution of our methodology from simple modeling to complex quality-of-service guarantees and suggest promising avenues for future research, such as mobility-aware scheduling and energy-efficient edge orchestration.

1.4 Related Publications

Doctoral Symposium:

- **Better Orchestration for SLO-Oriented Cross-site Microservices in Multi-tenant Cloud/Edge Continuum.** Yinan Cao. In Proceedings of the 24th ACM/IFIP International Middleware Conference: Demos, Posters and Doctoral Symposium, pp. 9-10. 2023. Bologna, Italy.

Full Papers:

- **LASSY: a Latency-Aware SLOs-Sufficing Scheduling System for the Cloud/Edge Continuum**. Yinan Cao, Etienne Rivière, and Ramin Sadre. In Proceedings of the 25th IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 276-285, 2025. Tromsø, Norway.

- **Scylla: Scheduling Multiple Latency-Sensitive Applications in the Edge-Cloud Continuum**. Yinan Cao, Etienne Rivière, and Ramin Sadre. In Proceedings of the IEEE/ACM 34th International Symposium on Quality of Service (IWQoS), pp. 901-910, 2026. Istanbul, Turkey.

- **Cerberus: Latency-Aware Scheduling with Graceful Degradation for Constrained Edge Infrastructures**. Yinan Cao, Etienne Rivière, and Ramin Sadre. In Proceedings of the 45th IEEE International Symposium on Reliable Distributed Systems (SRDS), 2026. Roma, Italy.

  This paper received the **Best Paper Award** at SRDS 2026.

- **Occamy: Latency-Aware Scheduling with Controlled Degradation for Mixed-Criticality AI at the Edge**. Yinan Cao, Patient Ntumba Wa Ntumba, Etienne Rivière, Ramin Sadre and Stefano Secci. In Proceedings of the 11th ACM/IEEE Symposium on Edge Computing (SEC), 2026. Santa Clara, CA, United States.

1.5 Generative AI Usage Statement

In the interest of transparency, this section discloses the use of generative AI tools in the preparation of this thesis. Such tools were used in two limited capacities: to assist with language, by checking grammar and refining the phrasing of the written text, and to generate the conceptual Figure 1.1, which illustrates the edge-cloud continuum at a schematic level. They were not used for any part of the research itself: the problem formulations, analytical models, algorithms, experimental designs, and all code developed for the schedulers and their evaluation are the author's own work, and all experiments were conducted, measured, and analyzed on real infrastructure without AI assistance. The scientific content, results, and conclusions of this thesis, together with responsibility for the final text, rest entirely with the author.

Background and Motivation

2

This chapter establishes the foundational background and motivation for the research presented in this thesis. We begin by describing the edge-cloud hierarchy infrastructure, introducing the physical and logical organization of cloud and edge sites, and the concept of Local User Groups that abstracts the geographical distribution of users. We then examine how modern applications are deployed and orchestrated across this infrastructure using containerization and Kubernetes, the latter being the dominant orchestration platform in practice and the one adopted throughout this thesis, though not the only one available, and characterize the workloads and quality-of-service expectations of their users. Building on this foundation, we identify the key open challenges that arise when scheduling latency-sensitive applications in resource-constrained edge environments, namely the performance inversion problem, resource contention among co-located applications, and resource scarcity under peak demand. These challenges collectively motivate the research contributions of this thesis, which are presented in the following chapters.

2.1 The Edge-Cloud Hierarchy Infrastructure

The infrastructure discussed in this thesis is structured as a multi-tiered hierarchy designed to balance the trade-off between massive computational capacity and geographical proximity to end-users. This architecture is formed of two physical tiers, the Cloud and the Edge Sites, which together form an integrated Edge-Cloud Continuum.

2.1.1 Cloud and Edge Sites

Cloud Site. At the top of the hierarchy lies the centralized cloud, composed of large data center with vast computational and storage resources. The cloud serves as the primary engine for heavy processing tasks, long-term data archival, and global management. However, because these data centers are often located far from the majority of end-users, they introduce significant network propagation delay [8]. While the cloud is highly cost-efficient and elas-

tic, its inherently high latency often makes it unsuitable for the millisecond-level response times required by real-time and latency-sensitive applications.

Edge Sites. At the opposite end of the hierarchy, closest to end-users, are the Edge Sites. These are localized compute clusters—often referred to as cloudlets or micro-data centers—deployed on-premise or within the last-mile network infrastructure. By processing data at the network periphery, edge sites drastically reduce the round-trip network latency, enabling low-latency service delivery. However, unlike the cloud, edge sites are inherently resource-constrained, possessing limited CPU and memory, and their fixed geographical locations mean they cannot easily absorb local spikes in demand without incurring a latency penalty due to queueing.

Scope of the Edge. It is important to delineate the infrastructure layer considered in this thesis from the endpoint device layer. We do not consider individual IoT sensors, smartphones, or embedded devices, commonly referred to as the *far edge*, as part of our edge compute pool. While these devices generate data and consume services, they typically lack the standardized virtualization and orchestration capabilities required for a managed, multi-tenant environment. Instead, our focus remains on infrastructure-grade edge sites, which are localized clusters of servers with sufficient resources to run containerized workloads under a common orchestration platform.

2.1.2 The Edge-Cloud Continuum

The synergy between these two layers forms the *Edge-Cloud Continuum*. Rather than treating cloud and edge as isolated tiers, this integrated model allows applications to be deployed across multiple sites at different levels of the hierarchy, with each site contributing its distinct characteristics in terms of computational capacity and network proximity to users. A given application may have instances running simultaneously at an edge site close to its users and at a cloud site with greater resource availability, with requests routed to the most appropriate site depending on the current conditions.

Figure 2.1 illustrates a representative edge-cloud continuum infrastructure, showing multiple geographically distributed edge sites interconnected with cloud site, aligns with the infrastructure model used by related work [8, 12, 14, 31]. Users at different locations experience different network latencies to each site, and application instances deployed at each site share the available computational resources. This infrastructure forms the physical substrate on which all scheduling problems addressed in this thesis are defined. Since placement decisions in this continuum are inherently tied to the geographical distribution of users, we next introduce the concept of Local User Groups as a structured abstraction for reasoning about user locations and their network distances to the available sites.

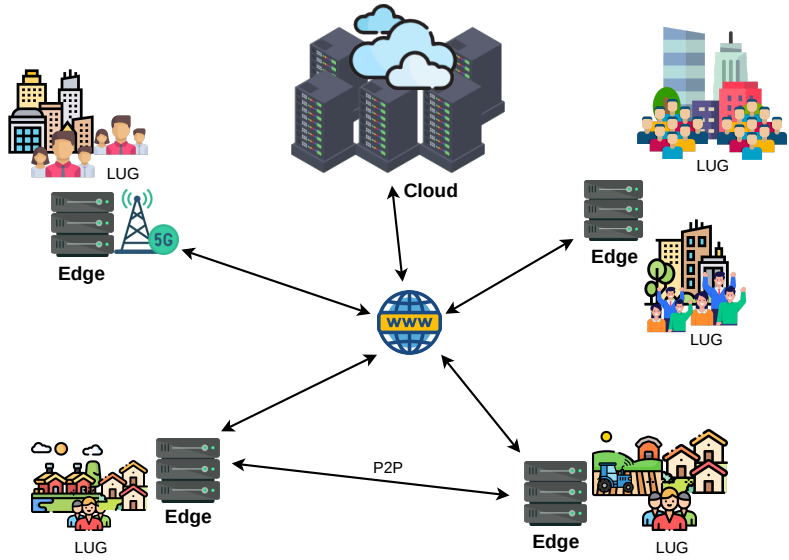


Figure 2.1: A typical edge-cloud continuum infrastructure

2.1.3 Local User Groups

Users of applications deployed in the edge-cloud continuum are not uniformly distributed. They are concentrated in geographically distinct areas, such as cities, campuses, or industrial zones, each with different network distances to the available cloud and edge sites. Users located in the same city will experience similar network latency to each site, while users in different cities may have vastly different latency profiles. Treating each individual user as a separate entity in a scheduling model would be computationally intractable and unnecessary, given this spatial regularity. Thus, groups of users with similar network latencies to a given edge site can be modeled as a *Local User Group* (LUG) [32]. Following established research [12, 33, 34] and industrial practice [35, 36], we treat each LUG as a single entity, who issues requests to applications deployed on different cloud or edge sites.

Definition. We adopt the concept of a *Local User Group* (LUG), which aggregates users that are geographically co-located and issue requests to the *same application*, and therefore share both a common workload and similar network latencies to the available sites. Each LUG is characterized by two properties: its aggregate request arrival rate, representing the combined demand generated by all users in the group for that application, and its average round-trip network latency to each site in the continuum, representing the network latency experienced by users in that group when communicating

with a given site. A LUG is thus tied to a single application: users at the same physical location who consume several applications are represented by several distinct LUGs, one per application, which may share identical network latencies to each site but carry independent workloads. The LUG thus serves as the basic unit of demand in the scheduling models developed throughout this thesis.

Practical Identification. In practice, LUGs can be identified through a variety of techniques, including passive network measurements, active probing such as ping-based RTT estimation, or indirect methods based on network coordinate [37]. In cellular networks, the serving base station provides a natural proxy for user location. Since the concept and justification of LUGs are well-established in the literature [12, 38–40], the concrete mechanism for identifying them is considered outside the scope of this thesis. We assume that LUG memberships and their associated average network latencies to each site are known inputs to our scheduling systems. If the network latency experienced by a group of users changes significantly, the LUG can either be reassigned to a more representative existing LUG, or subdivided into smaller, more refined LUGs that better capture the updated latency distribution of its members. Either case triggers a rescheduling of the affected groups.

2.2 Distributed Application Deployment

Deploying and managing applications across the heterogeneous nodes of the edge-cloud continuum requires a robust abstraction layer between the application and the underlying hardware. Without such an abstraction, differences in operating systems, hardware architectures, and software dependencies across cloud and edge sites would make consistent deployment extremely difficult. Two complementary technologies address this challenge and form the backbone of modern distributed application deployment: *containerization* and *container orchestration*.

2.2.1 Containerization

A container packages an application together with all of its dependencies, such as libraries, runtime, and configuration, into a single, portable unit. Unlike traditional virtual machines, which emulate an entire operating system stack, containers share the host kernel and isolate only the application's user-space environment. This lightweight design results in fast startup times, low memory overhead, and high deployment density, making containers particularly well-suited for resource-constrained edge sites where every unit of compute capacity is valuable [41, 42].

Container Images and Runtimes. A container is instantiated from a *container image*, which is a static, layered snapshot of the application and its dependencies. Images are built once and can be distributed and run on any host with a compatible container runtime, ensuring deployment consistency across the heterogeneous nodes of the edge-cloud continuum. The *Docker* runtime [43] is the de facto standard for building and distributing container images, while the *Open Container Initiative (OCI)* [44] provides the interoperability specifications that allow these images to run on any compliant runtime, decoupling the image format from the underlying execution environment.

Resource Isolation. Containers leverage two core Linux kernel primitives to provide isolation: *namespaces* [45], which restrict a container's view of the system to its own processes, network interfaces, and filesystem, and *control groups (cgroups)* [46], which enforce hard limits on the CPU, memory, and I/O resources a container may consume. This resource isolation mechanism is of particular relevance to this thesis: by assigning each application instance a fixed and identical share of the host node's CPU and memory resources, we can reason about the capacity of a site in terms of the number of instances it can concurrently host, independently of the specifics of the underlying hardware.

2.2.2 Orchestration with Kubernetes

While containerization solves the problem of consistent application packaging, managing a large fleet of containers across a distributed, multi-site infrastructure requires additional automation for scheduling, scaling, health monitoring, and network routing. *Kubernetes* [47] has emerged as the dominant open-source platform for this purpose and is the orchestration system used throughout this thesis.

Cluster Architecture. A Kubernetes cluster consists of a *control plane* and a set of *worker nodes*. The control plane maintains the desired state of the system and is responsible for making scheduling decisions, while worker nodes execute the containerized workloads. The atomic unit of deployment in Kubernetes is a *pod* [48], which typically encapsulates a single container instance along with its associated storage and network configuration.

Kubernetes Scheduling Framework. As shown in Figure 2.2, the default Kubernetes scheduler assigns pods to nodes based on a two-step process: *filtering*, which eliminates nodes that do not meet the pod's resource requirements or other constraints, and *scoring*, which ranks the remaining nodes based on various heuristics such as resource utilization and affinity rules. While this default scheduler is designed for general-purpose workloads, it

does not account for application-specific performance metrics such as tail latency, which are critical in the edge-cloud continuum. This gap motivates the development of custom scheduling solutions that can optimize for these specific objectives, as mainly explored in Chapters 5 and 6. Kubernetes readily accommodates such solutions: through mechanisms such as the scheduling framework's extension points, custom schedulers, and the node affinity and node selector APIs, an externally computed placement plan can be enforced directly by the platform, allowing the schedulers developed in this thesis to be realized without modifying Kubernetes itself. We validated the feasibility of this integration through a proof-of-concept custom scheduler embedded in the Kubernetes scheduling framework; the four schedulers presented in this thesis, however, compute their placement plans externally and apply them to the cluster rather than running as in-cluster schedulers, and their native integration into the scheduling framework remains a direction for future work.

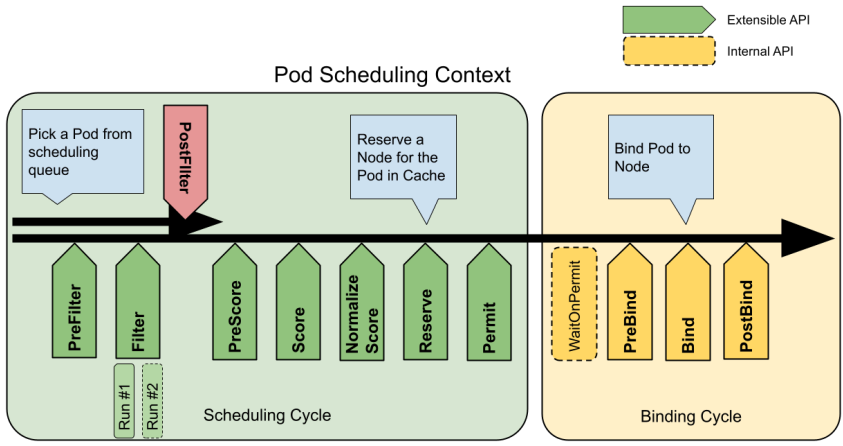


Figure 2.2: Kubernetes scheduling framework. The default scheduler performs filtering and scoring based on general-purpose heuristics, while custom schedulers can be implemented as extensions to optimize for specific application-level objectives.

Deployments and Scaling. Applications in Kubernetes are managed through *Deployments*, which declaratively specify the desired number of pod replicas of a given container image. The control plane continuously reconciles the actual state of the cluster with this desired state, automatically restarting failed pods and rescheduling them onto available nodes when necessary. The number of replicas can be adjusted manually by an operator or automatically by the *Horizontal Pod Autoscaler* (HPA) [20], which scales the replica count in response to observed resource utilization metrics such as average CPU usage. While HPA provides a reactive scaling mechanism suitable for

general-purpose workloads, it reacts to resource pressure only after it has materialized, rather than provisioning proactively.

Request Dispatching. When multiple pods serve the same application, incoming requests must be distributed across them. Kubernetes handles this through *kube-proxy* [30], the cluster's built-in service proxy, which by default distributes requests in a round-robin or random fashion, independently and uniformly across all available pods, as shown in Figure 2.3. Each pod therefore receives an approximately equal share of the incoming traffic and maintains its own independent local request queue. As we will discuss in Chapter 5, this distinction is foundational to the accuracy of the latency models underpinning the scheduling systems presented in this thesis.

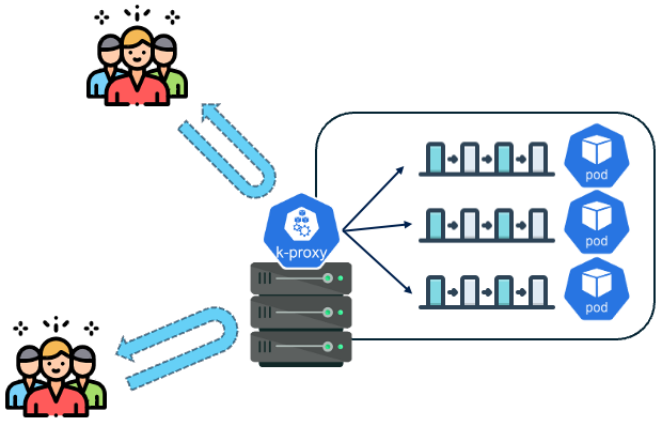


Figure 2.3: Traffic dispatching of Kubernetes' kube-proxy.

2.3 Workload Characterization

Understanding how users interact with applications deployed in the edge-cloud continuum, and what performance guarantees they expect, is essential for designing effective scheduling systems. This section characterizes the workload generated by users and the quality-of-service requirements that applications must satisfy.

2.3.1 Application Model

Modern networked applications take many forms, ranging from tightly coupled monolithic services to loosely coupled distributed architectures. It is important for situating the scope of this thesis and justifying the modeling assumptions that underpin the scheduling systems presented in subsequent chapters.

Scope. The scheduling systems presented in this thesis focus on a specific but widely applicable class of applications: *stateless, monolithic services* that can be deployed as independent, interchangeable instances across the edge-cloud continuum. Each instance is capable of handling any incoming request without relying on state from previous interactions, and instances do not communicate with each other. This model is well-suited to a broad and growing category of applications, including image processing pipelines, natural language processing services, and in particular *Edge AI inference services*, where a deep neural network model processes an input and returns a result in a single, self-contained transaction.

The stateless monolithic model offers two important properties that make it tractable for analytical scheduling. First, because any instance can serve any request, incoming traffic can be freely distributed across all available instances at a site by the Kubernetes service proxy, enabling the independent per-instance queueing model described in Section 2.2.2. Second, because instances do not share state or communicate, the performance of a site can be characterized purely in terms of the number of deployed instances and the arrival rate of incoming requests, without reasoning about inter-instance coordination overhead.

While microservice and stateful application scheduling are important research directions, they introduce additional complexity that builds upon, rather than replaces, the problem addressed here. A microservice application is ultimately a composition of individual services, each of which must itself be placed and scaled to meet its own latency and resource constraints; solving this problem rigorously for a single stateless service is therefore a prerequisite for reasoning about their composition. We consequently scope this thesis to stateless monolithic services, which already present significant and largely unsolved challenges in the edge-cloud continuum, as detailed in Section 2.5, and which provide the foundation on which the scheduling of more complex modern application architectures can subsequently be constructed.

2.3.2 Request-Response Dynamics

Applications considered in this thesis follow a *request-response* interaction model. A user sends a request to an application instance deployed at a site, the instance processes the request and returns a response. The time elapsed between the moment the request leaves the user and the moment the response is received is referred to as the *End-to-End Latency (EEL)*. This metric is the primary performance indicator considered throughout this thesis, as it directly reflects the quality of experience perceived by the user.

Components of End-to-End Latency. The end-to-end latency of a request is composed of three distinct contributions, as illustrated in Figure 2.4. The

first is the *network latency* (T_{NL}), which is the time required for the request and response to travel between the user and the site over the network. This component is determined by the physical distance between the user and the site, the routing path, and the characteristics of the network infrastructure. The second component is the *queueing delay* (T_Q), which is the time a request spends waiting at the application instance before being processed. This delay arises when the instance is busy serving a previous request and the incoming request must wait in the instance's local queue. The third component is the *service time* (T_S), which is the time the instance takes to process the request and produce a response.

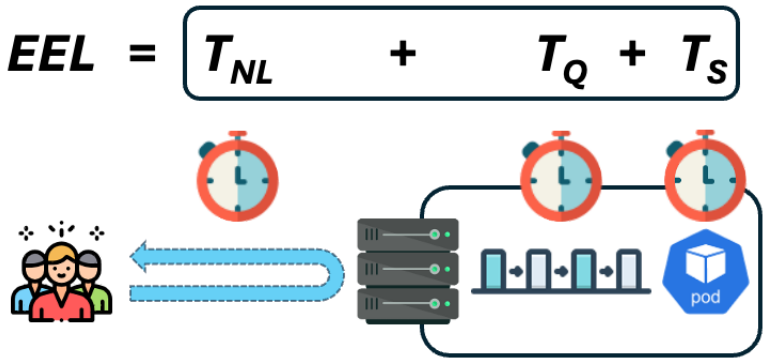


Figure 2.4: Composition of End-to-End Latency.

2.3.3 Quality of Service Requirements

The *Quality of Service* (QoS) of an application refers to the measurable performance characteristics that determine the level of service experienced by its users. For the latency-sensitive applications considered in this thesis, QoS is expressed in terms of the end-to-end latency of user requests.

Service Level Objectives. A *Service Level Objective* (SLO) specifies a measurable performance target that an application is expected to meet, most commonly, for latency-sensitive applications, an upper bound on the end-to-end latency that users may experience. Rather than bounding the average latency, which can mask the poor experience of a significant fraction of users, such SLOs are typically expressed as a constraint on a high percentile of the latency distribution, for example requiring that the 99th percentile of end-to-end latency not exceed a specified threshold. This ensures that the performance guarantee holds not just on average, but for the vast majority of individual requests. We define an SLO as *satisfied* if the empirically observed latency at the target percentile remains at or below the specified threshold under the given workload conditions.

Tail Latency. The high-percentile latency targeted by SLOs is commonly referred to as *tail latency*. While average latency characterizes the typical experience, tail latency captures the worst-case experience of users at the extremes of the latency distribution. In large-scale systems, even a small fraction of slow requests can have a disproportionate impact on user experience, particularly when a user interaction triggers multiple sequential service calls [49]. Furthermore, under high load conditions, queueing delays grow superlinearly with utilization, causing the tail of the latency distribution to grow far more rapidly than the mean, making tail latency a more sensitive and meaningful indicator of service quality.

The P99 Latency Target. Throughout this thesis, we focus on the 99th percentile of the end-to-end latency distribution, commonly denoted *P99*, as the primary SLO metric. An SLO expressed in terms of P99 guarantees that at most one percent of requests experience an end-to-end latency exceeding the defined threshold. This choice reflects industry practice for latency-sensitive services [50] and provides a stringent yet tractable target for the scheduling and optimization problems addressed in this thesis.

2.4 Open Challenges and Thesis Objectives

The background established in the preceding sections reveals that deploying latency-sensitive applications in the edge-cloud continuum is far from straightforward. While edge sites offer the proximity necessary to reduce network latency, and Kubernetes provides the orchestration machinery to manage application instances, the interaction between limited edge resources, geographically distributed users, and strict tail-latency SLOs gives rise to a set of fundamental challenges that existing systems do not fully address, as we summarized in Section 2.5. This section describes these challenges in detail, as they collectively motivate the research contributions of this thesis.

2.4.1 Performance Inversion Problem

The most immediate challenge in edge-cloud scheduling is the *performance inversion problem* [8]. The intuitive strategy for serving latency-sensitive users is to deploy application instances at the edge site closest to them, thereby minimizing network latency. However, this strategy can backfire when the demand directed at an edge site exceeds its available processing capacity.

The Inversion Mechanism. As the load on an edge site increases, application instances become increasingly utilized and incoming requests must wait longer in the local queue before being processed. As established in Section 2.3.3, queueing delays grow superlinearly with utilization and can rapidly dominate the end-to-end latency. When queueing delays at an overloaded

edge site grow sufficiently large, the total end-to-end latency experienced by nearby users can exceed the latency they would have experienced had their requests been routed to a more distant cloud site with greater available capacity. The presumed advantage of edge proximity is thus inverted: the site that should offer the best performance instead delivers the worst.

Limits of Reactive Scaling. The straightforward remedy for an overloaded edge site is to scale out the application by deploying additional instances. However, as discussed in Section 2.1.3, edge sites are inherently resource-constrained, and the number of additional instances that can be deployed is strictly bounded by the available capacity. When this capacity is exhausted, scaling out is no longer possible and the operator must either redirect users to a more distant site, which is trading network latency for reduced queueing delay, or accept the SLO violation. Determining the right balance between these options, across multiple sites and user groups simultaneously, requires a principled scheduling approach that jointly models network latency, queueing behavior, and resource availability.

2.4.2 Noisy Neighbors and Resource Contention

In a multi-tenant edge environment, multiple applications share the limited resources of the same edge site. While each application may individually fit within the available capacity, their combined resource consumption can lead to contention, a phenomenon commonly referred to as the *noisy neighbor problem* [51–53].

Contention Under Co-location. When multiple applications are co-located on the same edge site, they compete for the same pool of CPU and memory resources. Even if each application is allocated a fixed share of resources, contention for shared hardware components—such as CPU caches, memory bandwidth, and network interfaces—can cause each application's effective service rate to degrade below its nominal value. This degradation increases service times and, consequently, queueing delays, pushing the end-to-end latency of affected applications toward and beyond their SLO thresholds [13, 18].

Cascading SLO Violations. The noisy neighbor problem is particularly dangerous in the context of elastic, replica-based deployments. When an application detects that its SLO is being violated due to resource contention, a natural response is to spawn additional instances at the same or neighboring edge sites. However, adding instances increases the total resource consumption at those sites, which in turn exacerbates contention for other co-located applications, potentially triggering their own SLO violations and further scaling responses. This feedback loop can lead to *cascading SLO violations* [54, 55],

where the independent scaling decisions of individual applications collectively degrade the performance of all applications on the site. Avoiding this cascade requires a global scheduling perspective that considers the resource demands of all co-located applications simultaneously, rather than scheduling each application in isolation.

2.4.3 Resource Scarcity and Service Inconsistency

The challenges described above assume that a feasible scheduling plan exists, that there is some allocation of instances across sites that allows all applications to meet their SLOs simultaneously. In practice, however, the aggregate demand from all applications may exceed the total available capacity of the edge infrastructure, making it impossible to satisfy all SLOs at once.

Resource Scarcity. Resource scarcity arises when the edge infrastructure is collectively saturated: even with an optimal allocation of instances across all sites, the available capacity is insufficient to absorb the incoming workload while respecting every application's tail-latency SLO. This situation is not exceptional but rather an expected operating condition during peak demand periods, when user activity surges beyond the provisioned capacity. While the cloud can in principle provide additional resources on demand, it is not always a viable fallback: many edge deployments operate under data-sovereignty, privacy, or connectivity constraints that prohibit offloading to the cloud, and even when offloading is permitted, the network latency to a distant cloud site may itself exceed the tail-latency SLO of the affected applications. In such edge-only or edge-constrained settings, the infrastructure operates under a hard capacity ceiling that cannot be dynamically relaxed, and the scheduler must resolve the resource deficit locally.

Eviction and Its Drawbacks. A common response to resource scarcity is to *evict* low-criticality applications from the edge site, freeing their allocated resources for higher-priority workloads. While eviction is effective at restoring capacity, it completely interrupts the service for evicted applications and their users, resulting in a severely degraded experience. Furthermore, eviction decisions are often irreversible in the short term, and the repeated cycle of eviction and redeployment as demand fluctuates can cause system instability and unpredictable service availability.

Service Inconsistency. Beyond full eviction, resource scarcity can also manifest as *service inconsistency*: applications remain nominally deployed but operate under conditions where their SLOs are chronically violated, without any controlled or predictable degradation policy. Users of these applications experience erratic response times that vary unpredictably with the overall load on the site, making it impossible for application owners to provide meaningful performance guarantees. This lack of predictability is arguably as dam-

aging to user experience as outright eviction, and it highlights the need for scheduling mechanisms that can manage resource scarcity in a controlled, transparent, and priority-aware manner.

2.4.4 Mixed-Criticality Edge AI Allocation

The rapid proliferation of AI-powered applications at the edge is fundamentally changing the nature of the workloads that edge infrastructures must support. Applications such as real-time video analytics for traffic and public safety monitoring [56], autonomous driving perception pipelines [57], augmented reality [1], and industrial visual inspection [2] increasingly rely on deep neural network inference executed close to the data source to meet their stringent latency requirements. Unlike traditional lightweight services, modern deep neural network based inference applications are computationally demanding: processing a single request may require significant CPU or GPU resources and tens to hundreds of milliseconds of inference time. As the adoption of edge AI accelerates across domains such as autonomous vehicles, industrial automation, and public safety surveillance, the aggregate resource demand placed on edge infrastructures is growing rapidly, pushing these already resource-constrained environments closer to their capacity limits.

Heterogeneous Criticality in AI Workloads. A defining characteristic of edge AI deployments is the coexistence of applications with vastly different levels of criticality. A real-time video analytics system detecting accidents or abnormal events on a public road [56] operates under fundamentally different stakes than a recommendation engine suggesting augmented reality content to museum visitors [1]. Both applications may compete for the same limited edge resources, yet the consequences of a performance degradation or service interruption are incomparable. For safety-critical applications, a violation of tail-latency SLOs or an unexpected service interruption during peak demand is not merely a quality-of-experience issue, but can directly compromise human safety.

Inadequacy of Existing Eviction Policies. As discussed in Section 2.4.3, existing approaches to resource scarcity typically resort to evicting applications in low criticality to free resources for higher-priority ones. In the context of mixed-criticality AI workloads, however, the boundary between critical and non-critical is not always clear-cut, and abrupt eviction policies fail to account for the continuous and heterogeneous nature of application criticality. Furthermore, the computational demands of AI inference workloads make resource contention during peak periods not an exceptional event but a foreseeable and recurring operating condition. Without a principled, priority-aware scheduling framework that can reason about the criticality of co-located AI applications and manage resource pressure in a controlled and

predictable manner, edge infrastructures risk becoming unreliable platforms for the safety-critical applications they are increasingly called upon to support.

2.4.5 Thesis Objectives

The four challenges described above are not independent: they form a progression of increasing complexity, from provisioning a single application accurately, to coordinating multiple applications, to managing scarcity, and finally to exploiting the internal adaptability of edge AI workloads. This progression defines the central goal of this thesis: to develop a unified, latency-aware scheduling methodology for the edge-cloud continuum or edge clusters, that provides strict tail-latency guarantees and degrades gracefully to guarantee the highly prioritized applications when resources become insufficient. We decompose this goal into four research questions, each addressing one of the challenges and each answered by one of the contributions presented in the following chapters.

RQ1: Accurate tail-latency provisioning. How can the tail latency of a containerized application be accurately predicted as a function of its resource allocation, workload, and the network distances to its users, and how can this prediction be used to compute deployment plans that satisfy strict tail-latency SLOs at minimal cost? This question addresses the performance inversion problem of Section 2.4.1, and is answered in Chapters 3 and 5, which develop the queueing-theoretic latency model and the Lassy single-application scheduler.

RQ2: Global multi-application scheduling. How can multiple co-located applications competing for the same shared edge resources be scheduled jointly, so that the tail-latency SLOs of all applications are satisfied simultaneously while avoiding the cascading contention that arises from per-application scheduling? This question addresses the noisy neighbor problem of Section 2.4.2, and is answered in Chapter 6, which presents the Scylla global scheduling framework.

RQ3: Graceful degradation under scarcity. When the available resources are insufficient to satisfy all SLOs simultaneously, how can the scheduler degrade the performance of less critical applications in a controlled, bounded, and priority-aware manner, rather than resorting to abrupt eviction or uncontrolled SLO violations? This question addresses the resource scarcity problem of Section 2.4.3, and is answered in Chapter 7, which introduces the Cerberus graceful degradation mechanism.

RQ4: Accuracy-aware scheduling for edge AI. For AI inference applications that can be served by multiple model variants of different sizes, how can

the scheduler exploit this internal flexibility to maintain latency compliance under resource pressure by trading inference accuracy for reduced resource consumption, rather than relaxing latency bounds alone? This question addresses the mixed-criticality edge AI challenge of Section 2.4.4, and is answered in Chapter 8, which presents the Occamy accuracy-aware scheduler.

2.5 State-of-the-Art Scheduling Approaches

A substantial body of work has addressed the problem of scheduling latency-sensitive applications across cloud and edge infrastructures. These approaches can be broadly categorized according to their target infrastructure, their scheduling methodology, and the performance metrics they optimize. This section surveys the most relevant existing systems, identifying their contributions and the limitations that motivate the research presented in this thesis. Table 2.1 summarizes the key characteristics of the surveyed approaches.

Name	Cloud	Distributed Edge	Tail Latency	Real-world Testbed
INVAR [12]	✓	✓	✗	✗
FIRM [9]	✓	✗	✗	✓
Cilantro [19]	✓	✗	✗	✓
KaiS [18]	✓	✓	✗	✓
SORA [10]	✓	✗	✓	✓
Polaris [16]	✓	✓	✗	✗
Atlas [13]	✓	✗	✗	✓
iPlace [58]	✓	✗	✗	✗
Phare [17]	✓	✓	✗	✗
Mu [14]	✓	✓	✗	✓
Oakestra [59]	✓	✓	✗	✓
Komet [60]	✗	✓	✓	✗
Ets-ddpg [61]	✓	✓	✗	✗
FGSLO [62]	✗	✓	✗	✗

Table 2.1: General related work for resource scheduling.

Reactive Autoscaling. The most widely deployed scheduling mechanism in practice is reactive autoscaling, exemplified by the Kubernetes Horizontal Pod Autoscaler (HPA) [20]. HPA monitors aggregate resource utilization metrics such as average CPU usage across all pods of an application, and triggers the addition or removal of replicas when this metric crosses a predefined threshold. This approach is simple, stateless, and requires no prior knowledge of the application's performance characteristics. However, it is fundamentally reactive: it responds to resource pressure after it has already materialized, and is entirely agnostic to the violations of SLO. As a result, HPA cannot anticipate the queueing delays that drive tail latency violations,

and it has no mechanism for reasoning about the network distances between users and sites when making scaling decisions.

Cloud-Focused Analytical Schedulers. A more principled class of approaches employs analytical or machine-learning-based models to predict application performance and inform scheduling decisions proactively. FIRM [9] ensures high utilization and reduced SLO violations by leveraging telemetry data and machine learning techniques to detect and mitigate resource contention among microservices. Cilantro [19] uses online learning mechanisms to map resource allocation to performance outcomes in real-time, optimizing resource use for varying user-defined objectives. SORA [10] is a management framework that dynamically adjusts soft resource allocations, such as server threads and database connections, for latency-sensitive microservices. While these systems represent significant advances over reactive autoscaling, they are designed for cloud-only infrastructures with abundant and homogeneous resources, and do not account for the distributed nature of the edge-cloud continuum or the varying network latencies between geographically distributed users and sites.

Multi-Component and Microservice Schedulers. Several works address the scheduling of multi-component applications, where end-to-end latency depends on the chained latencies of multiple internal service calls. Polaris [16] is an SLO- and topology-aware scheduler for microservices at the edge, while Atlas [13] is a hybrid cloud migration advisor for interactive microservices that optimizes inter-component communication latency. Phare [17] focuses on scheduling multi-component applications across federated edge sites, optimizing inter-application API call latencies. KaiS [18] proposes a learning-based scheduling framework for Kubernetes-oriented edge-cloud systems that uses a multi-agent actor-critic algorithm and graph neural networks for resource and request dispatch management. These works address an important class of applications but target a fundamentally different problem from the one considered in this thesis, which focuses on stateless monolithic applications where the dominant latency factor is the network distance between users and the serving site, rather than inter-component communication.

Edge-Cloud Continuum Schedulers. The most directly related class of work targets the scheduling of applications across the edge-cloud continuum, explicitly accounting for the heterogeneity of available resources and the varying network latencies between users and sites. Mu [14] proposes an autoscaling framework for serverless applications on resource-constrained edge sites, focusing on latency within the site without considering the network latency from users to the edge. iPlace [15] reduces the deployment delay for cloud-based applications across the continuum. INVAR [12] is the most closely related prior work: it uses queueing theory to predict the mean

end-to-end latency for a single application and determines the number of required instances at edge sites along with a mapping of users to sites, using the deployment budget as the optimization objective. However, INVAR targets mean latency rather than tail latency, relies on an $M|M|c$ queueing model that does not reflect the per-instance request dispatching behavior of Kubernetes, and has been validated only through simulations with a proprietary parameter set. These limitations are discussed in detail in Chapter 3.

Edge-Oriented Schedulers and Orchestration Frameworks. A broad line of work develops schedulers and orchestration frameworks tailored to the specific constraints of edge infrastructures, such as heterogeneity, geographical distribution, energy budgets, and mobility. Oakestra [59] introduces a lightweight, hierarchical orchestration framework for heterogeneous edge infrastructures that leverages delegated service scheduling and semantic overlay networking to optimize both service-to-service and service-to-user latencies across federated, multi-operator clusters. Komet [60] targets serverless low-Earth orbit edge computing, using orbit-aware scheduling and zero-downtime migration to preserve service availability against satellite movement. ETS-DDPG [61] combines an $M|G|1|PR$ priority queuing model with deep reinforcement learning to jointly minimize task response time and energy consumption in geo-distributed data centers. FGSLO [62] applies Lyapunov optimization to jointly manage service placement, service lifetime, and task scheduling under strict energy budgets in cloud-free edge-edge collaboration. While these systems address important dimensions of edge orchestration, none of them jointly predicts tail latency, optimizes the placement of multiple co-located applications globally, and provides a controlled degradation mechanism under resource scarcity, which is the combination that distinguishes the work presented in Chapter 7 and Chapter 8.

Summary and Positioning. No existing system simultaneously provides accurate tail-latency prediction, SLO-aware global optimization, and validation on real edge-cloud infrastructure. The research presented in this thesis addresses this gap progressively across four contributions, each extending the scope and capability of the scheduling framework to cover a broader and more realistic set of operating conditions.

2.6 Motivation and Summary

The challenges described in the preceding sections reveal a fundamental tension at the heart of edge computing. Edge sites are attractive precisely because of their proximity to users, yet this same proximity comes with resource constraints that make them difficult to manage under realistic workload conditions. The performance inversion problem, resource contention among co-

located applications, resource scarcity under peak demand, and the mixed-criticality nature of edge AI workloads are not corner cases, but expected operating conditions in any densely utilized edge infrastructure. Addressing them is made increasingly urgent by the emergence of next-generation latency-sensitive applications, from augmented reality to autonomous systems, for which tail latency is not a secondary concern but the primary measure of service quality, and for which violating a P99 SLO is unacceptable.

As surveyed in Section 2.5, current orchestration tools are ill-equipped to meet this bar. Reactive, utilization-based autoscalers such as Kubernetes HPA are latency-blind: they respond to average resource consumption rather than to the queueing dynamics that drive tail latency, and they have no awareness of the network distances between users and sites. Existing analytical schedulers take a more principled approach but target mean latency rather than tail latency, schedule a single application at a time, and rely on queueing models that do not faithfully reflect the per-instance request dispatching behavior of Kubernetes. None of these systems can reliably guarantee tail-latency SLOs across the range of conditions encountered in practice, and none provides a controlled response when resources become insufficient.

This thesis addresses this gap. Our goal is to develop scheduling systems that are simultaneously accurate in their tail-latency predictions, global in their optimization scope, and robust under resource scarcity. We pursue this goal progressively across four contributions, each extending the scheduling framework to a broader and more realistic set of operating conditions: from single-application tail-latency scheduling, to multi-application global optimization, to graceful degradation under scarcity, and finally to accuracy-aware quality management for edge AI workloads. The infrastructure model, workload characterization, and SLO definitions established in this chapter are shared across all four contributions and referenced throughout the remainder of the thesis.

Latency Modeling and Tail Latency Prediction

3

Accurately predicting the latency experienced by users of applications deployed in the edge-cloud continuum is a prerequisite for any principled latency-aware scheduling system. Without a reliable model of how end-to-end latency responds to changes in resource allocation, workload intensity, and network conditions, scheduling decisions must rely on reactive heuristics that are blind to the queueing dynamics driving tail latency violations. This chapter develops the analytical foundation that all four scheduling systems presented in this thesis build upon: a queueing-theoretic model that predicts the end-to-end latency distribution of user requests to containerized application instances deployed in a Kubernetes environment, and in particular its tail.

We begin by defining the system model, which formalizes the infrastructure, the request arrival process, and the decomposition of end-to-end latency into its constituent components. We then present the $M|D|1$ queueing model as the appropriate analytical tool for capturing the per-instance queueing behavior that arises from Kubernetes' independent request dispatching mechanism, and derive closed-form expressions for both mean and tail latency prediction. Finally, we discuss the key modeling choices and assumptions that underpin the model, and contrast our approach with the $M|M|c$ model used by prior work [12].

3.1 System Model

Before deriving a latency prediction model, we must precisely define the system we are modeling. This section formalizes the infrastructure, the workload, and the end-to-end latency structure that together form the basis of the queueing-theoretic analysis developed in Section 3.2. The notation used throughout this chapter and the remainder of this thesis is summarized in Table 3.1.

3.1.1 Infrastructure and Request Dispatching

We consider a edge-cloud continuum infrastructure consisting of a set of sites $ST = \{st_1, \ldots, st_{|ST|}\}$, each hosting a cluster of worker nodes managed by a

Name	Description
$st \in ST$	Cloud and edge sites
$nslots_{st}$	Number of slots of computing resource available at site st
$lug \in LUG$	Local user groups
$l_{lug,st}$	Round trip network latency between users lug and site st
λ_{lug}	Request rate of user group lug
μ	Service rate of an instance
slo	Maximum tolerated θth percentile of the end-to-end latency defined in SLO
$\Lambda_{st,lug}$	Maximum arrival rate for the instances at site st can accept without breaking the SLO for user group lug
$ninsts_{st}$	Number of instances deployed on each site st
$lugsite_{lug}$	Site that user group lug is redirected to
LUG_{st}	Set of user groups sending requests to site st
w_{st}	Queueing delay at site st
ρ_{st}	Utilization of an instance at site st

Table 3.1: Model parameters (top) and variables (bottom)

common Kubernetes control plane. Sites differ in their computational capacity and their network distance to users. Each site st provides a number of uniform computation slots $nslots_{st}$, where each slot represents a fixed and identical unit of CPU and memory resources. In this chapter, we consider a single application and assume for simplicity that each of its instances occupies exactly one slot. This assumption keeps the latency model uncluttered while we establish its foundations; it is relaxed in Chapter 6, where instances of different applications may consume different numbers of slots, yielding a more complete and realistic resource model.

An application instance deployed at a site is realized as a Kubernetes pod, and each pod occupies exactly one slot. In this chapter, we consider a single application and characterize its latency behavior, assuming for simplicity that each instance consumes exactly one resource slot. This uniform resource assumption simplifies capacity reasoning and is well aligned with the practice of assigning fixed, identical resource quotas to application replicas in Kubernetes. The number of pods deployed for the application at site st is denoted $ninsts_{st}$, with the constraint $ninsts_{st} \leq nslots_{st}$. The more general case, where multiple applications compete for shared resources across sites, is studied in following chapters.

When multiple pods serve the same application at a site, incoming requests are distributed across them by the Kubernetes service proxy, kube-proxy, in a round-robin or random fashion as described in Section 2.2.2. Each pod therefore receives an approximately equal share of the total incoming

traffic and maintains its own independent local queue. This dispatching behavior is central to the modeling approach adopted in this chapter: rather than treating all pods at a site as a single shared multi-server queue, we model each pod as an independent single-server queue, a choice we motivate analytically in Section 3.2 and validate empirically in Chapter 4.

3.1.2 Workload and Arrival Process

The workload directed at a site originates from one or more Local User Groups (LUGs), as defined in Section 2.1.3. Each LUG $lug \in LUG$ generates requests at an aggregate rate λ_{lug}, and all requests from a given LUG are routed to exactly one site $lugsite_{lug} \in ST$. The set of LUGs assigned to site st is denoted $LUG_{st} = \{lug \in LUG \mid lugsite_{lug} = st\}$, and the total arrival rate at site st is therefore:

$$\Lambda_{st} = \sum_{lug \in LUG_{st}} \lambda_{lug} \tag{3.1}$$

Since each LUG aggregates a large number of independently acting users, each issuing requests at irregular intervals, the aggregate arrival process at a site is well approximated by a Poisson process [63]. Under this model, inter-arrival times between consecutive requests are exponentially distributed, and the number of requests arriving in any fixed time interval follows a Poisson distribution with a mean determined by the aggregate arrival rate. When the total arrival rate Λ_{st} is evenly distributed across the $ninsts_{st}$ pods deployed at the site, each pod receives requests at a rate of:

$$\lambda_{st}^{inst} = \frac{\Lambda_{st}}{ninsts_{st}} = \frac{\sum_{lug \in LUG_{st}} \lambda_{lug}}{ninsts_{st}} \tag{3.2}$$

3.1.3 End-to-End Latency Decomposition

The end-to-end latency of a request issued by a user in LUG lug and served by a pod at site $lugsite_{lug}$ is composed of three distinct contributions, as introduced in Section 2.3.2.

Network Latency. The first contribution is the *network latency* $l_{lug,lugsite_{lug}}$, defined as the average round-trip network latency between the users of LUG lug and site $lugsite_{lug}$. As discussed in Section 2.1.3, we assume that network latencies are relatively stable and can be estimated from network measurements, allowing us to approximate the instantaneous round-trip network latency by its average value $\bar{l}_{lug,lugsite_{lug}}$.

Queueing Delay. The second contribution is the *queueing delay* w_{st}, which is the time a request spends waiting in the local queue of the pod before being processed. This component is stochastic and depends on the arrival rate, the

number of deployed pods, and the service time distribution. It is the primary source of variability in the end-to-end latency and the component that the queueing model developed in Section 3.2 is designed to characterize.

Service Time. The third contribution is the *service time* S, defined as the time required by a pod to process a request and produce a response. The service time is dominated by a near-constant model inference time that is largely independent of the content of the request. We therefore model the service time as deterministic with constant rate μ, where $S = 1/\mu$, an assumption whose implications for the choice of queueing model are discussed in Section 3.2.

The end-to-end latency of a request from a *lug* served at site $lugsite_{lug}$ is thus:

$$L_{lug,lugsite_{lug}} = \bar{l}_{lug,lugsite_{lug}} + w_{lugsite_{lug}} + \frac{1}{\mu} \tag{3.3}$$

Since the network latency and service time are treated as deterministic constants for a given LUG-site pair, the distribution of the end-to-end latency is entirely determined by the distribution of the queueing delay w_{st}, which we derive in the following section.

3.2 Queueing-Theoretic Latency Model

This section first introduces the fundamentals of queueing theory and then presents the $M|D|1$ model as the appropriate tool for characterizing the per-instance queueing behavior of a Kubernetes deployment, and derives closed-form expressions for both mean and tail latency prediction.

3.2.1 Queueing Theory

Queueing theory is a branch of applied mathematics concerned with the analysis of waiting lines, or queues, that form when demand for a shared resource exceeds its instantaneous processing capacity. It provides analytical tools for characterizing the statistical behavior of systems in which jobs arrive, wait for service, and depart after being processed. In the context of computing systems, queueing theory has been widely applied to model the performance of servers, networks, and distributed systems under load [64–67].

Kendall's Notation. Queueing models are commonly described using Kendall's notation [68], which characterizes a queue by three elements written as $A|S|c$. The first element A denotes the inter-arrival process of incoming jobs, the second element S denotes the service time distribution, and the third element c denotes the number of servers. Common distributions used in this notation include M for Markovian, meaning exponentially distributed, D for deterministic, meaning constant, and G for general, meaning an arbitrary distribution. For example, an $M|M|1$ queue has Poisson arrivals, exponentially

distributed service times, and a single server, while an $M|D|1$ queue has Poisson arrivals, deterministic service times, and a single server.

Key Performance Metrics. The primary performance metrics of interest in a queueing system are the *waiting time* w, defined as the time a job spends in the queue before being served, and the *sojourn time*, defined as the total time a job spends in the system including both waiting and service. The distribution of these metrics depends on the arrival process, the service time distribution, and the number of servers. A queueing system is said to be *stable* if the queue does not grow unboundedly over time, which requires that the utilization $\rho = \lambda/(c \cdot \mu)$ satisfies $\rho < 1$, where λ is the arrival rate, μ is the service rate of a single server, and c is the number of servers. As the utilization approaches 1, waiting times grow superlinearly and the system approaches saturation.

3.2.2 The M|D|1 Queueing Model

The choice of queueing model is determined by the characteristics of the arrival process, the service time distribution, and the number of servers. As established in Section 3.1, the arrival process at each pod follows a Poisson process, the service time is deterministic with rate μ, and each pod operates as an independent single-server queue. These three properties correspond precisely to the assumptions of the $M|D|1$ queue: Markovian (Poisson) arrivals, Deterministic service times, and a single server. This model is therefore a natural and principled choice for characterizing the queueing behavior of a single application pod in a Kubernetes deployment.

Utilization. A fundamental parameter of the $M|D|1$ queue is the *utilization* ρ_{st}, defined as the fraction of time the server is busy processing requests. For a pod at site st receiving requests at rate λ_{st}^{inst} with service rate μ, the utilization is:

$$\rho_{st} = \frac{\lambda_{st}^{inst}}{\mu} \tag{3.4}$$

The queue is stable, meaning it does not grow unboundedly, if and only if $\rho_{st} < 1$. As ρ_{st} approaches 1, the queueing delay grows superlinearly, corresponding to the performance inversion phenomenon described in Section 2.4.1.

3.2.3 Mean Latency Prediction

For an $M|D|1$ queue, the mean queueing delay is given by the Pollaczek-Khinchine (P-K) mean value formula [63]:

$$\mathbb{E}[w_{st}] = \frac{\rho_{st}}{2\mu(1 - \rho_{st})} \tag{3.5}$$

This expression reveals the superlinear dependence of the mean queueing delay on utilization: as ρ_{st} increases toward 1, the denominator $(1 - \rho_{st})$ approaches zero and the mean queueing delay diverges. The mean end-to-end latency for a request from LUG *lug* served at site $lugsite_{lug}$ is then obtained by substituting Equation 3.5 into Equation 3.3:

$$\mathbb{E}[L_{lug,lugsite_{lug}}] = \bar{l}_{lug,lugsite_{lug}} + \frac{\rho_{lugsite_{lug}}}{2\mu(1 - \rho_{lugsite_{lug}})} + \frac{1}{\mu} \tag{3.6}$$

The ability to predict mean end-to-end latency is the level of granularity targeted by existing schedulers such as INVAR [12], which use it as the primary performance metric for scheduling decisions. However, as argued in Section 2.3.3, mean latency is an insufficient indicator of service quality for latency-sensitive applications: it can remain well within acceptable bounds while a significant fraction of users experience response times that far exceed the SLO threshold. Industry practice and the academic literature on Quality of Service [49] have converged on tail latency—specifically high-percentile metrics such as P99—as the appropriate standard for defining and enforcing performance guarantees. We therefore move beyond mean latency prediction and derive in the following subsection the full queueing delay distribution, which enables direct prediction and control of tail latency.

3.2.4 Tail Latency Prediction

While the mean latency provides a useful summary statistic of the typical experience of users, it is insufficient for reasoning about SLO compliance. The mean latency aggregates the full distribution of response times into a single value, masking the variability that arises from queueing under load. In particular, under high utilization, the latency distribution becomes heavily right-skewed: the majority of requests are served quickly, while a small but non-negligible fraction experience disproportionately long waiting times due to queueing. These outliers are precisely the requests that determine whether an SLO is satisfied or violated, yet they have little influence on the mean. As established in Section 2.3.3, it is the tail of the latency distribution — not its center — that matters for latency-sensitive applications. We therefore require the full distribution of the queueing delay w_{st} in order to predict the θth percentile of the end-to-end latency and verify whether the SLO *slo* is satisfied.

Queueing Delay Distribution. For an $M|D|1$ queue, the cumulative distribution function of the queueing delay is given by [69]:

$$P(w_{st} \leq t) = (1 - \rho_{st}) \sum_{i=0}^{\lfloor t \cdot \mu \rfloor} e^{-\lambda_{st}^{inst}(i/\mu - t)} \frac{(\lambda_{st}^{inst})^i (i/\mu - t)^i}{i!} \tag{3.7}$$

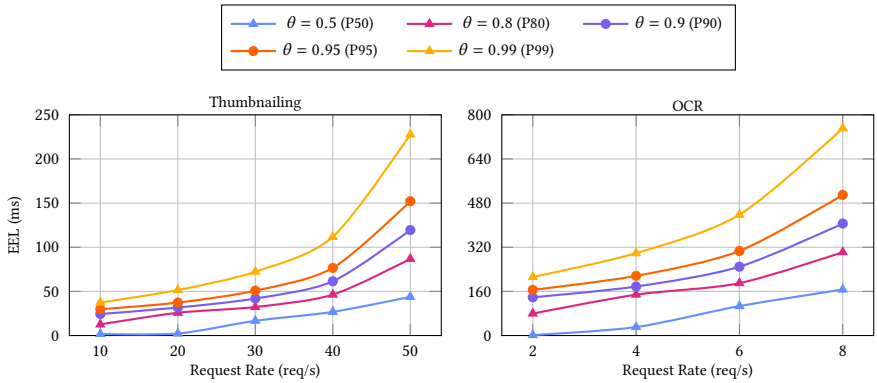


Figure 3.1: Predicted end-to-end latency (EEL) at the 50th, 80th, 90th, 95th, and 99th percentiles for single instance of two target applications, as a function of the request rate.

where $\rho_{st} = \lambda_{st}^{inst}/\mu$ is the utilization of each pod at site st, and $\lfloor t \cdot \mu \rfloor$ denotes the floor of $t \cdot \mu$. Here, t is the waiting-time threshold at which the distribution is evaluated, and gives the probability that a request waits no longer than t time units in the queue before being served. The summation index i ranges over the number of requests already ahead of an arriving request in the queue. Since each request receives deterministic service of duration $1/\mu$, clearing i requests ahead takes time i/μ; an arriving request therefore begins service within t only if at most $\lfloor t \cdot \mu \rfloor$ requests are ahead of it, which fixes the upper limit of the sum. The expression is derived from the analysis of the $M|D|1$ queue by Erlang [69]. Note that the distribution depends only on the utilization ρ_{st} and the per-instance arrival rate λ_{st}^{inst}, both of which are determined by the number of deployed instances and the total incoming request rate at the site, as established in Section 3.1.

Percentile Computation. Given the cumulative distribution function in Equation 3.7, the θth percentile of the queueing delay at site st, denoted w_{st}^{θ}, is defined as the smallest value of t such that $P(w_{st} \leq t) \geq \theta\%$:

$$w_{st}^{\theta} = \inf \{t \geq 0 \mid P(w_{st} \leq t) \geq \theta\%\} \tag{3.8}$$

Since $P(w_{st} \leq t)$ is a non-decreasing, right-continuous function of t, this percentile is well-defined and can be computed numerically from Equation 3.7 for any target θ.

As illustrated in Figure 3.1, we apply Equation 3.8 to compute the end-to-end latency at several percentiles, from the median ($\theta = 0.5$) to the tail ($\theta = 0.99$), for two of the target applications that will be introduced in next chapter, as the request rate increases toward saturation. The gap between the percentiles widens sharply as the load approaches saturation: the median

remains modest while the tail grows several times faster. This is precisely why bounding the mean is insufficient for latency-sensitive services, and why the model predicts any target percentile θ directly. In the remainder of this thesis we take $\theta = 0.99$ as the SLO target, though the formulation applies unchanged to any percentile an application may require.

End-to-End Tail Latency. Recalling the end-to-end latency decomposition from Equation 3.3, the θth percentile of the end-to-end latency for a request from user group *lug* served at site $lugsite_{lug}$ is:

$$L^{\theta}_{lug,lugsite_{lug}} = \bar{l}_{lug,lugsite_{lug}} + w^{\theta}_{lugsite_{lug}} + \frac{1}{\mu} \tag{3.9}$$

Since $\bar{l}_{lug,lugsite_{lug}}$ and $1/\mu$ are deterministic constants for a given user group and site, the distribution of the end-to-end latency is entirely determined by the distribution of the queueing delay in Equation 3.7. The θth percentile of the end-to-end latency therefore depends only on the θth percentile of the queueing delay, shifted by the sum of the network latency and the service time. This additive structure is a key property of the model: it allows the tail latency to be predicted analytically for any combination of resource allocation, workload, and network conditions, without requiring simulation or measurement.

3.3 Discussion

The queueing-theoretic model derived in Section 3.2 rests on a set of analytical assumptions about the arrival process, service time distribution, and request dispatching behavior of Kubernetes. This section discusses the key modeling choices that underpin this model, contrasts our approach with the $M|M|c$ model used by prior work, and identifies the assumptions and limitations that bound its applicability.

Network Latency. The model assumes that the network latency between a LUG and a site is stable and can be approximated by its average value $\bar{l}_{lug,st}$. In practice, network latency can fluctuate due to congestion, routing changes, or variability in the underlying physical infrastructure. Short-term fluctuations are unlikely to significantly affect scheduling decisions, as the scheduler operates on timescales much longer than individual request round-trip times. However, sustained changes in network conditions — such as a degradation in the link quality between a user population and its assigned site — may cause the model to underestimate or overestimate the network latency component of the end-to-end latency. Such changes can be detected through continuous network monitoring, and a rescheduling can be triggered when the observed latency deviates significantly from the model's assumption.

Service Time. The accuracy of the $M|D|1$ model is contingent on the validity of its underlying assumptions. The deterministic service time assumption holds well for applications where the processing time per request is near-constant, regardless of the request content. This is the case for a broad class of workloads, including image processing pipelines, data transformation services, and AI inference workloads where a fixed neural network model processes every request with a predictable computation time. The assumption may be less accurate for applications with highly variable processing times, such as database queries or dynamic content generation, where the service time depends heavily on the complexity or size of individual requests. In such cases, the $M|D|1$ model would underestimate queueing delays, as the additional service time variability would increase the tail latency beyond what the deterministic model predicts. A first mitigation is to use an upper-bound estimate of the service time derived from empirical observations, allowing the model to conservatively account for variability without requiring knowledge of the full service time distribution. Alternatively, the deterministic queue can be replaced by a model that explicitly captures service time variability, such as an $M|G|1$ queue parameterized by the empirically measured variance of the service time, or an $M|M|1$ queue in the limiting case of exponentially distributed service times. Since the remainder of the scheduling framework interacts with the queueing model only through the resulting waiting time distribution, such a substitution can be made without altering the scheduling formulation or algorithms, an extension we discuss further as future work in Chapter 9.

Request Payload. The model assumes that the size of requests and responses is small relative to the available network bandwidth, such that transmission delays are negligible and the network latency is independent of the individual request content. This assumption holds for a broad class of applications where inputs and outputs are compact, such as image processing services, data transformation pipelines, and AI inference workloads where inputs are typically images or short text snippets and responses are compact structured outputs. For applications that transfer large payloads — such as video streams or large binary files — transmission delays may constitute a significant and variable component of the end-to-end latency that the current model does not capture.

Arrival Process. The Poisson arrival assumption is well justified for large user populations composed of many independently acting users, each generating requests at irregular intervals. However, it may be less accurate for small or highly correlated user groups, or for workloads exhibiting strong temporal patterns such as periodic batch jobs or synchronized request bursts. In such cases, the inter-arrival time distribution may deviate significantly

from the exponential distribution assumed by the Poisson model, leading to prediction errors in both mean and tail latency.

Instance Resource Assignment. As mentioned in Section 3.1, we assume for simplicity that each application instance occupies exactly one uniform resource slot, allowing us to reason about site capacity purely in terms of the number of deployable instances. This holds well for applications deployed with fixed, identical resource quotas, a common practice in Kubernetes for predictable performance, but does not capture cases where different applications or model variants consume different amounts of resources. The generalization to heterogeneous resource footprints competing for shared capacity is addressed in Chapter 6 and the chapters that follow.

3.4 Conclusion

This chapter has developed the queueing-theoretic latency model that serves as the analytical foundation for all scheduling systems presented in this thesis. We began by defining the system model for a single application, formalizing the infrastructure, the workload arrival process, and the decomposition of end-to-end latency into its three constituent components: network latency, queueing delay, and service time. We then introduced the fundamentals of queueing theory and presented the $M|D|1$ model as the appropriate analytical tool for characterizing the per-instance queueing behavior that arises from Kubernetes' independent request dispatching mechanism, and derived closed-form expressions for both mean and tail latency prediction. Finally, we discussed the key modeling choices underpinning our approach, contrasting the $M|D|1$ model with the $M|M|c$ model used by prior work and identifying the assumptions and limitations that bound its applicability.

The model developed in this chapter plays a dual role in the remainder of this thesis. First, it enables accurate prediction of the end-to-end latency distribution for a given resource allocation and workload, which is the prerequisite for any principled SLO-aware scheduling system. Second, the closed-form expression for the queueing delay distribution provides an analytical handle on the tail latency constraint, which as we will show in Chapter 5 can be incorporated directly into a combinatorial optimization problem to compute deployment plans that provably satisfy P99 SLOs. The empirical validation of the model, along with a direct experimental comparison with the $M|M|c$ model, is presented in Chapter 4 in the context of the scheduling system that builds upon it. The limitations of the model identified in Section 3.3, in particular its assumption of deterministic service times, are revisited in subsequent chapters as the scheduling framework is extended to more complex deployment scenarios.

Experimental Methodology and Latency Model Validation

4

The four contributions presented in this thesis are all evaluated experimentally on a real Kubernetes testbed, without relying on simulations. These evaluations share a common infrastructure, workload generation methodology, and resource allocation model, and draw on a common pool of target applications, of which each contribution uses the subset relevant to its scope. This chapter presents this common experimental framework once, so that the following chapters can focus on the evaluation aspects that are specific to their respective contributions. In addition, this chapter provides the empirical validation of the $M|D|1$ latency prediction model introduced in Chapter 3, through a microbenchmark of prediction accuracy and a single-site scale-out evaluation. These two experiments form the empirical foundation on which the schedulers of Chapters 5 to 8 are built.

4.1 Introduction

Evaluating latency-aware schedulers for the edge-cloud continuum requires an experimental environment that reproduces the defining characteristics of this setting: geographically distributed sites with heterogeneous capacities, WAN latencies between users and sites, and latency-sensitive containerized applications under realistic request workloads. Commercial edge-cloud offerings exist, for instance on Microsoft Azure, but they are costly to operate at scale, offer limited control over placement and network conditions, and exhibit fluctuating latencies that complicate the measurement of tail latencies. We therefore construct such an environment on a dedicated testbed, combining physical machines, Kubernetes orchestration, and network emulation. This approach preserves the fidelity of real container execution and request dispatching, while providing the control and reproducibility necessary for a rigorous evaluation.

The remainder of this chapter is organized as follows. Section 4.2 describes the testbed infrastructure, including the Grid'5000 platform, the Kubernetes deployment, the network emulation layer, and the automation of the

experimental pipeline. Section 4.3 presents the workload generation methodology used to emulate local user groups, the slot-based resource allocation model, the target applications, and the baseline approaches against which our model is compared. Section 4.4 empirically validates the $M|D|1$ model of Chapter 3, first through microbenchmarks of prediction accuracy and then through single-site evaluations of its ability to drive scale-out decisions. Section 4.5 discusses the modeling choices underlying these results, and Section 4.6 concludes the chapter.

4.2 Testbed Infrastructure

This section describes the physical and software infrastructure on which all experiments of this thesis are conducted: the Grid'5000 platform providing the bare-metal machines, the Kubernetes deployment orchestrating the applications, the network emulation reproducing the WAN latencies of the edge-cloud continuum, and the tooling automating the experimental pipeline.

4.2.1 The Grid'5000 Testbed

All experiments presented in this thesis are conducted on Grid'5000 [28], a large-scale distributed platform for experimental research in computer science, with sites in France, Belgium, and Luxembourg, hosted by a scientific interest group including Inria, CNRS, RENATER, and several universities. Grid'5000 provides bare-metal access to clusters of homogeneous physical machines, allowing experiments to run on dedicated, non-virtualized hardware. This eliminates the performance variability introduced by co-located tenants in public clouds, a property that is essential when measuring tail latencies, which are by definition sensitive to rare interference events.

Two hardware configurations are used across the experiments of this thesis. The experiments of Chapter 5 use the *paradoxe* cluster located in Rennes, in which each node is equipped with two 8-core Intel Xeon E5-2630 v3 CPUs and 128 GB of RAM, and each actor of the system, i.e., a cloud or edge site or a local user group, is deployed on a dedicated physical node with exclusive access to its resources. The experiments of Chapter 4, 6, 7, 8 use the *gros* cluster located in Nancy, whose larger node count accommodates multi-application scenarios; each node is equipped with an 18-core Intel Xeon Gold 5220 CPU and 96 GB of RAM, and nodes are logically partitioned into sites and local user groups as described next. In both configurations, the hardware is homogeneous within the cluster, ensuring that observed performance differences are attributable to scheduling decisions rather than hardware heterogeneity.

4.2.2 Kubernetes Deployment

We deploy a Kubernetes [47] cluster over the reserved nodes. A dedicated node hosts the control plane and does not run any application workloads, isolating control-plane activity from the worker nodes so that it does not interfere with the resources available to the measured applications. In the configuration of Chapter 5, each site consists of one or more dedicated worker nodes. In the configuration of Chapters 6 to 8, we logically isolate groups of nodes using Kubernetes labels to form cloud and edge sites as well as local user groups. Label-based isolation guarantees that pods belonging to a site are only scheduled on the nodes of that site, while allowing a single physical cluster to host the complete emulated edge-cloud topology.

Application instances are deployed as Kubernetes pods by the control plane. Incoming requests are dispatched to the pods of a service by the service proxy (kube-proxy), following the default dispatching policy described in Chapter 2. In the experiments of Chapter 8, Kubernetes is additionally configured with the static CPU management policy [70], which guarantees exclusive core access per pod. This configuration was not required in the earlier chapters, where the service times of the deployed applications were already stable under the default policy. The AI inference workloads of Chapter 8, however, exhibited noticeable service time variability without it, which the static policy removed by eliminating the interference caused by CPU sharing. Since the deterministic service time assumption of the $M|D|1$ model rests on this stability, the static policy is used throughout that chapter.

4.2.3 Network Emulation

Grid'5000 clusters are interconnected by low-latency local networks that do not reflect the WAN latencies characteristic of the edge-cloud continuum. We therefore emulate network latencies between local user groups and sites using the Linux `traffic control` (`tc`) facility [71]. In the experiments of Chapter 5, latencies are configured directly, with the cloud site set to a significantly higher network latency to users than the edge sites. For the experiments of Chapters 6 to 8, we implement a network topology generator based on network coordinates [72]: system actors are placed in an Euclidean plane and the latency between any two actors is derived from their distance in that plane. This approach produces internally consistent latency matrices, in which the triangle inequality between actors holds by construction. Real Internet paths, however, do not always satisfy the triangle inequality, since routing policies and peering agreements can make an indirect path shorter than the direct one, and this is a well-documented limitation of Euclidean network coordinate systems [73]. Our emulated topology is therefore an idealization in this respect, which does not affect the validity of our results, as the scheduler treats the la-

tency matrix as an opaque input and never assumes any metric property of it. The concrete latency values used in the multi-site scenarios are additionally grounded in real-world edge-cloud latency measurements [8] and are given in the respective chapters.

The multi-application scenarios of Chapters 6 to 8 involve up to 40 LUGs across 10 user locations, which would be impractical to emulate using one dedicated physical node per LUG. Instead, we co-locate multiple LUGs on the same physical node by leveraging Linux network namespaces, with each LUG isolated in its own namespace and assigned a unique virtual IP address via a `veth` [74] interface pair connected to a shared bridge. Per-destination network latencies are configured independently for each namespace using `tc netem` [75], with traffic marked using `iptables` [76] and routed to per-destination queuing disciplines via HTB classes, allowing each LUG to experience a different round-trip time to each site regardless of the physical node it shares with other LUGs.

4.2.4 Experiment Automation

The experimental scenarios of this thesis involve multiple sites, multiple applications, tens of LUGs, and several workload configurations. To manage this complexity, we rely on EnOSlib [29], a Python library designed for reproducible and automated experimentation on testbed infrastructures such as Grid'5000. EnOSlib provides abstractions for node reservation, software deployment, network emulation, and experiment orchestration, allowing the full experimental pipeline to be defined and executed programmatically. This includes the automated deployment of the Kubernetes cluster, the configuration of network latencies between LUGs and sites, the deployment of application instances, the execution of load generation, and the collection of measurements. The use of EnOSlib ensures that all experiments are fully reproducible and that the experimental setup can be easily reconfigured for different scenario sizes and workload configurations.

4.3 Workload and Experimental Components

Having described the physical testbed in the previous section, we now turn to the elements that populate it: the workload that drives the applications, the resource abstraction through which site capacity is allocated, and the applications themselves.

4.3.1 Workload Generation and Local User Groups

We emulate the workload of LUGs using `httperf` [77], a load generator able to send requests to a remote service at a high rate with inter-request times

following an exponential distribution. This is key to emulating the behavior of a group of independently acting users, and matches the arrival process assumed by the queueing model of Chapter 3. Each LUG is materialized by one `httperf` process, and these processes are pinned on dedicated cores using `taskset` [78] to avoid scheduling hazards that would distort the generated arrival process or the latency measurements.

Measurement protocol. Unless stated otherwise, every configuration is executed once for five minutes. The first 10% of measurements are discarded to eliminate warm-up effects, such as container cold starts and cache population, and the remaining requests form the latency distribution reported in the figures.

4.3.2 Resource Allocation Strategy

We allocate computing resources at cloud and edge sites at the granularity of a *slot*, comprising 1 CPU core and 1 GB of RAM. Each application instance is deployed as a Kubernetes pod and occupies a fixed, integer number of slots, specified per application. Slot capacity is homogeneous across all sites, so that a slot represents the same amount of computing resources regardless of where it is allocated.

Slot-based allocation is a well-established practice in production cluster management. Google's Borg [79] and Apache Hadoop YARN [80] allocate resources to tasks as fixed-size bundles of CPU and memory, and Kubernetes itself expresses pod resource requests and limits as explicit CPU and memory quantities [47]. Fixing the per-instance resource footprint reflects the common operational practice of sizing a container once, through profiling, and adapting to workload variations by changing the number of instances. Consistently with this practice and with the application model of Chapter 3, applications in this thesis are scaled horizontally, by adding or removing instances (scale-out and scale-in), and never vertically: the resources assigned to a running instance are not modified. This choice also keeps the service rate μ of an instance constant, which is a prerequisite for the latency predictions of the $M|D|1$ model.

4.3.3 Applications

We select four representative containerized applications with diverse resource needs and service time profiles. All four are stateless request-response services, matching the application model of Chapter 3. Table 4.1 summarizes their characteristics. Because the service rate of an application is hardware-dependent, it must be re-profiled whenever the underlying platform changes. The multi-site experiments of Chapter 5 were conducted on the Rennes clus-

ter and use only the Thumbnailing and OCR applications, whose service rates were calibrated on that platform. All subsequent chapters, from Chapter 6 onward, were conducted on the Nancy cluster, equipped with 18-core Intel Xeon Gold 5220 CPUs as described in Section 4.2, and use all four applications with their service rates re-calibrated on that platform. Table 4.1 reports the service rates measured on both clusters, with each instance allocated the number of slots indicated; the dashes denote applications not used in the Rennes experiments.

Applications	Slots per Instance	Service rate (req/s) Rennes	Service rate (req/s) Nancy
Thumbnailing	1	46	58
OCR	2	7.8	10
MatrixProduct	2	-	20
Translator	8	-	4.5

Table 4.1: The four target applications used throughout this thesis (except for Chapter 8), their resource footprint in slots, and their measured service rates on Rennes and Nancy cluster.

Thumbnailing. This application accepts a base64 string representation of an image and returns a resized image in the same format.

Optical Character Recognition (OCR). This application implements text recognition from an image provided as input. We implement it using Tesseract [81], a reference OCR engine based on an LSTM neural network.

MatrixProduct. This application multiplies two random matrices and returns the result. The matrix size is configurable and we use 42×42 matrices in our experiments for a reasonable service time.

Translator. This application implements a translation service. We implement it with EasyNMT [82] and the neural machine translation model Opus-MT [83].

These four applications cover the spectrum of latency-sensitive edge-cloud workloads: network-dominated (Thumbnailing), computation-dominated (MatrixProduct), and resource-heavy inference services powering scenarios such as augmented reality and mobile applications (OCR and Translator). The experiments of Chapter 5 use Thumbnailing and OCR, while the experiments of Chapters 6 and 7 use all four applications. Chapter 8 targets edge AI applications available in multiple model variants; the corresponding applications and their variants are specific to that contribution and are introduced in Chapter 8 directly.

4.3.4 Baselines

We use two established baseline systems to validate the latency model of this chapter, representing, respectively, the state of the art in model-based edge-cloud scheduling and the most common industrial practice for elastic resource allocation. Both serve as points of comparison for the validation experiments of this chapter, and INVAR reappears in the microbenchmark of Chapter 8, where the model is validated for the AI application variants. Neither is used in the multi-site evaluations of the following chapters: as our results show, neither can handle tail-latency SLOs, and neither supports the multi-application scheduling that the subsequent contributions addr ess. The multi-site scenarios of the following chapters are instead compared against baselines specific to each contribution, namely a *local-first* placement heuristic in Chapter 6, a *priority-aware eviction* scheduling policy in Chapter 7, a *local-first* placement heuristic and the model-variant scheduler in Chapter 8, each will be introduced in its respective chapter.

INVAR. INVAR [12] is a model-based scheduler that generates placement plans for edge-cloud scenarios. It focuses on a single application and on SLOs expressed over the *mean* end-to-end latency. INVAR models each deployment as an $M|M|c$ queueing system, i.e., it assumes exponentially distributed service times and a single queue served by multiple instances. As discussed in Chapter 3, both assumptions conflict with the behavior of containerized deployments: service times of containerized applications show low variability, and Kubernetes' service proxy dispatches requests to instances upon arrival, resulting in one queue per instance rather than a single shared queue. INVAR does not predict tail latencies and was evaluated by its authors in simulation only. It is the closest related approach to the prediction model of this thesis and therefore serves as our primary model-based baseline.

Kubernetes Horizontal Pod Autoscaling. Kubernetes' built-in Horizontal Pod Autoscaling (HPA) [20] is one of the most commonly deployed resource allocation mechanisms in industry. HPA reactively triggers the scale-out of an application, i.e., adds instances, when its average CPU utilization over a specified observation period exceeds a threshold. In contrast to INVAR and to the contributions of this thesis, HPA does not use a model of application performance and only reacts to system-level metrics. To account for our focus on latency-sensitive applications, we set its threshold to 60% average CPU utilization rather than the default 80% [20], a configuration that favors the baseline by provisioning resources more aggressively.

4.4 Latency Model Validation

Before the latency model of Chapter 3 can serve as the foundation for the schedulers of the following chapters, its predictions must be shown to hold on a real deployment. This section validates the model through two complementary single-site experiments conducted within the framework described above. The first, a microbenchmark, assesses the raw accuracy of the model by comparing its predicted latencies against measurements for a fixed number of instances. The second closes the loop by evaluating whether these predictions are accurate enough to drive the elementary scheduling decision of provisioning the right number of instances to meet a tail-latency SLO.

4.4.1 Microbenchmark: Prediction Accuracy Evaluation

The first experiment evaluates whether the $M|D|1$ queueing model of Chapter 3 accurately predicts the tail latency of requests sent to an application instance running in a real Kubernetes deployment. We compare its predictions to those of the $M|M|c$ model used by INVAR.

Setup. We deploy a single instance of each of the four target applications, each allocated the number of computing resource slots specified in Table 4.1, and use `httperf` to send requests at a controlled mean rate with Poisson inter-arrival times. We progressively increase the request rate from a lightly loaded regime up to the saturation point of each application, as determined by its service rate. The network latency between the load generator and the instance is set to zero, so that the measured end-to-end latency isolates the queueing and service delays that the model must capture. We set $\theta = 0.99$ and thus evaluate the prediction of the P99 latency, a percentile that guarantees a high quality of service for the vast majority of requests.

Results Analysis. Figure 4.1 shows the measured mean and P99 end-to-end latencies (EEL) for the four applications under increasing request rates, together with the predictions of the $M|D|1$ model and of INVAR's $M|M|c$ model. Note that INVAR does not provide a prediction of the tail latency but only of its mean.

The $M|D|1$ model accurately predicts the tail latency for low to medium request rates, and only slightly overestimates it for high request rates close to the saturation point of three of the applications. This overestimation is conservative, a property we revisit in Section 4.5. Its prediction of the mean latency achieves an average relative error of 10%, against 49% for INVAR's $M|M|c$ model. These results confirm empirically the modeling decisions made in Chapter 3.

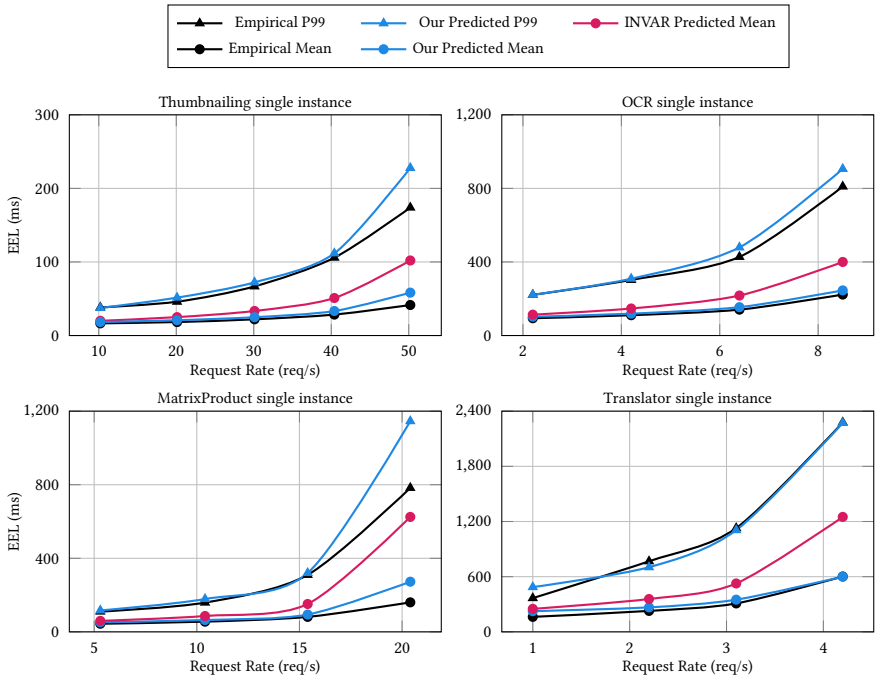


Figure 4.1: Evaluation of performance and prediction accuracy for the four applications' single instances used in isolation and in a local setting. Black curves are measured results for mean (dot) and P99 (triangle) end-to-end (EEL) latency. Blue curves indicate mean (dots) and P99 (triangles) predictions by our model, while red curves indicate mean (dots) latency prediction by INVAR.

4.4.2 Single-Site: Instance Count Evaluation

The second experiment evaluates whether the latency predictions of the $M|D|1$ model can drive scale-out decisions, i.e., determine the number of instances necessary to meet a tail-latency SLO in a single-site setting. This sizing capability is the elementary building block of all the schedulers presented in the following chapters. We focus on the Thumbnailing and OCR applications as representative cases.

Setup. We configure a single site with a single LUG using the Thumbnailing and OCR applications. The P99 SLO is set to 100 ms for the Thumbnailing application and to 250 ms for the OCR application. The network latency between the LUG and the site is set to zero. For increasing request rates, we deploy the minimal number of instances such that the P99 latency predicted by the $M|D|1$ model, with the arrival rate divided evenly among instances, remains below the SLO. We compare against INVAR and HPA, letting each baseline control the number of deployed instances for the same workload.

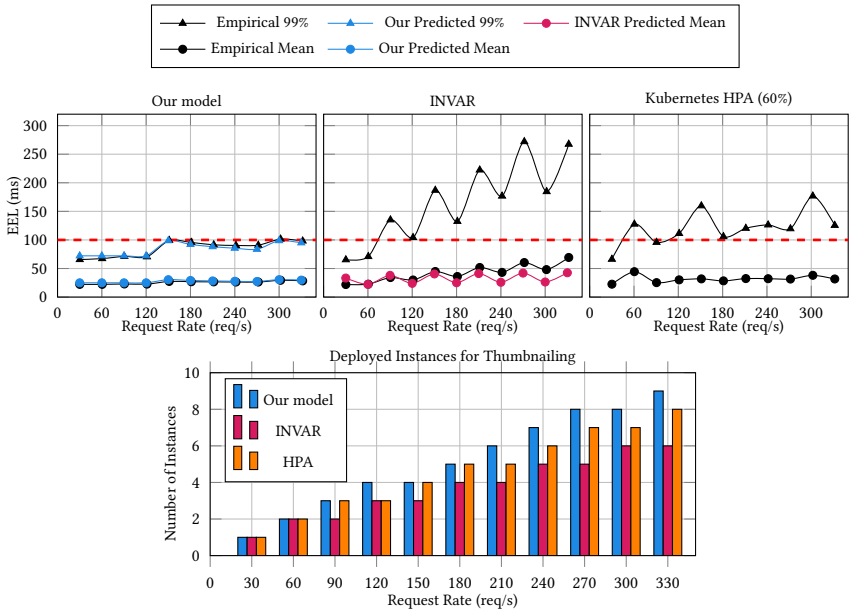


Figure 4.2: Empirical and predicted end-to-end latencies (EEL) for a single site scenario using our model, INVAR, and Kubernetes HPA to control the number of instances deployed for the *Thumbnailing* application. Black curves indicate the empirical results for the P99 (triangles) and mean (dots) latency for each deployment, while blue and red curves indicate predictions made by our model and INVAR, respectively. The dashed red line indicates the application's SLO for P99 latency.

Since INVAR only supports mean latency targets, we set its objective to half the target tail latency, i.e., $slo/2$, giving it a deliberately generous provisioning budget; configured directly against slo as a mean target, it would provision so few instances that the resulting tail latencies would be too severe to constitute a meaningful comparison.

Results Analysis. Figure 4.2 and Figure 4.3 show the empirical mean and P99 end-to-end latency and the number of instances deployed by each approach for the Thumbnailing and OCR applications, respectively. The model-driven sizing accurately determines the number of instances necessary to keep the empirical P99 latency below the SLO threshold across all tested request rates, with predictions closely matching the empirical measurements. INVAR and HPA, by contrast, consistently deploy too few instances under high load, resulting in P99 latency SLO violations of up to 171% and 76%, respectively, even when their mean latency stays below target.

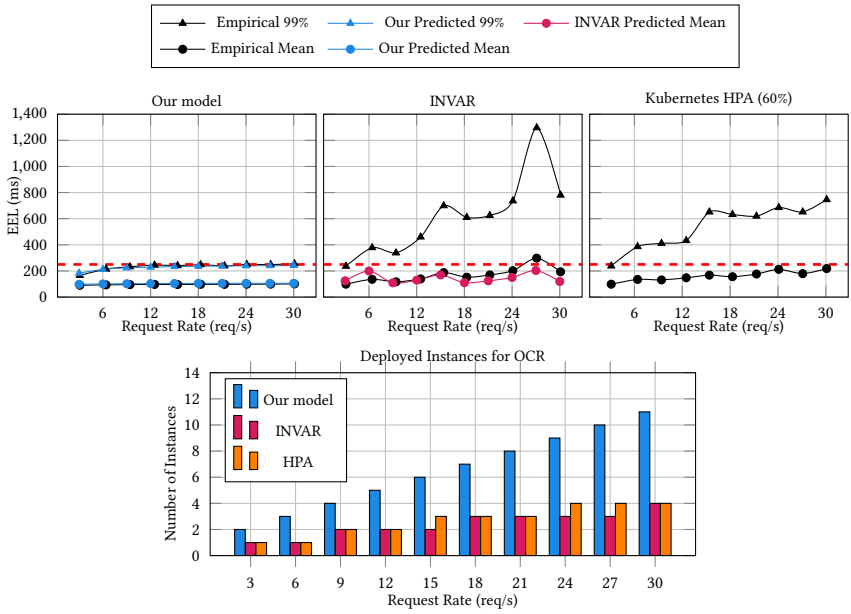


Figure 4.3: Empirical and predicted end-to-end latencies (EEL) for a single site scenario using our model, INVAR, and Kubernetes HPA to control the number of instances deployed for the *OCR* application.

4.5 Discussion

4.5.1 Queueing Model Comparison

As the closest existing approach to ours, INVAR [12] warrants a detailed comparison. The prediction error reported in Section 4.4.1 originates from two modeling assumptions that distinguish the $M|M|c$ model from ours.

The first concerns the service time distribution. The $M|M|c$ model assumes exponentially distributed service times, which introduces artificial variability and consequently overestimates queueing delays for workloads where service times are near-constant. The $M|D|1$ model assumes deterministic service times, a significantly better fit for applications where the processing time per request is largely independent of the request content.

The second difference concerns the queueing topology. The $M|M|c$ model treats all instances at a site as a single pool of servers sharing a common queue, so an arriving request is always dispatched to the next available instance. Our model instead treats each instance as an independent $M|D|1$ queue, reflecting the actual behavior of kube-proxy, which distributes requests in a round-robin or random fashion without knowledge of the current queue length at each instance. A request may therefore arrive at a busy instance even if another is idle, producing queueing delays that the $M|M|c$

model structurally cannot represent. This topological mismatch, invisible in INVAR's simulation-only evaluation, is what the single-instance and scaled-out measurements of Section 4.4.1 expose on a real deployment.

4.5.2 Tail Latency Prediction Accuracy

A more fundamental limitation of the $M|M|c$ approach is that it predicts only the mean latency. The $M|D|1$ model instead provides a closed-form expression for the full queueing delay distribution, enabling direct prediction of tail latency at any target percentile θ. This is what makes SLO enforcement on the tail possible at all: the schedulers of the following chapters take a P99 target as input and provision against it, whereas a mean-only model can at best target the mean and hope the tail follows. The conservative bias observed near saturation in Section 4.4.1 interacts favorably with this: provisioning against a slight tail overestimate errs toward SLO compliance rather than violation, a safety margin that a mean-based scheduler cannot provide regardless of how it is tuned.

4.6 Conclusion

This chapter has presented the experimental framework shared by all the contributions of this thesis: a Kubernetes testbed deployed on the bare-metal Grid'5000 platform, network emulation reproducing realistic WAN latencies, Poisson workload generation emulating local user groups, a slot-based resource allocation model, and four representative latency-sensitive applications. It has also introduced INVAR and Kubernetes HPA, the two baseline systems used to validate our latency model.

Within this framework, we have empirically validated the $M|D|1$ latency prediction model of Chapter 3. The microbenchmark shows that the model accurately predicts the P99 end-to-end latency of real containerized applications, with an average relative error of 10% on the mean latency, where INVAR's $M|M|c$ model exhibits an error of 49%. The single-site evaluation shows that these predictions suffice to determine the minimal number of instances meeting a tail-latency SLO, a capability that neither INVAR nor HPA provides, as both consistently under-provision and violate the SLO by up to 171% and 76%, respectively.

The model is thus able to answer the question of *how many* instances an application needs at a single site. It does not, however, answer the questions that arise in the distributed edge-cloud continuum: at *which* sites should these instances be placed, and to which site should each user group direct its requests, given heterogeneous network latencies, limited site capacities, and operational costs. The next chapter presents Lassy, a scheduler that embeds the

validated $M|D|1$ model into an optimization problem to answer these questions jointly, computing placement plans that guarantee tail-latency SLOs for a single application across the sites of the edge-cloud continuum.

Single-Application Scheduling with Lassy

5

Chapter 3 established how to predict the tail latency of an application from its resource allocation, its workload, and the network distances to its users. Prediction, however, is only a means to an end. Deploying an application across the edge-cloud continuum requires answering two coupled questions: how many instances to place at each site, and which users each site should serve. The latency model makes it possible to evaluate whether a given pairing of these decisions satisfies the tail-latency SLO, but it does not indicate which pairing to choose, and the set of feasible plans is typically large, with members differing substantially in quality. Selecting among them therefore requires an explicit optimization objective, whose definition we deliberately keep flexible: an application owner may wish to minimize the total number of deployed instances, the total monetary cost weighted by site-specific pricing, or the number of active sites, and the same framework accommodates all three. This chapter formulates the resulting scheduling problem as a combinatorial optimization problem that incorporates the tail-latency modeling of Chapter 3 directly into its feasible region, and presents Lassy, a Latency-Aware SLO-Sufficing Scheduler that solves it efficiently. The central technical difficulty is that these constraints are nonlinear in the decision variables, and we show how they can be linearized without loss of accuracy, yielding a Mixed-Integer Linear Program that off-the-shelf solvers handle at realistic problem scales.

The chapter is organized as follows. Section 5.1 formulates the scheduling problem and derives its constraints. Section 5.2 presents the Lassy scheduler and its algorithm. Section 5.3 present the experimental evaluation of Lassy and results analysis. Section 5.4 concludes the chapter.

5.1 Scheduling as an Optimization Problem

The latency model developed in Chapter 3 provides a closed-form expression for the tail latency experienced by users of an application deployed at a given site, as a function of the number of deployed instances and the incoming request rate. We now leverage this model to formulate the scheduling problem as a combinatorial optimization problem. The goal is to determine a *placement*

plan that specifies how many application instances to deploy at each site and which site each user group should be assigned to, such that the tail-latency SLO is satisfied for all user groups while minimizing operational costs. The notation used throughout this chapter is summarized in Table 5.1.

Name	Description
$st \in ST$	Cloud and edge sites
$nslots_{st}$	Number of slots of computing resource available at site st
$lug \in LUG$	Local user groups
$l_{lug,st}$	Round trip network latency between users lug and site st
λ_{lug}	Request rate of user group lug
μ	Service rate of an instance
slo	Maximum tolerated θth percentile of the end-to-end latency defined in SLO
$\Lambda_{st,lug}$	Maximum arrival rate for the instances at site st can accept without breaking the SLO for user group lug
c_{st}	Cost per instance at site st
$ninsts_{st}$	Number of instances deployed on each site st
$lugsite_{lug}$	Site that user group lug is redirected to
LUG_{st}	Set of user groups sending requests to site st
w_{st}	Queueing delay at site st
ρ_{st}	Utilization of an instance at site st
$x_{st,lug}$	Whether user group lug is assigned to site st
y_{st}	Whether site st is active, i.e. hosts at least one instance

Table 5.1: Model parameters (top) and variables (bottom). Rows with a grey background are introduced in this chapter.

5.1.1 Decision Variables and Capacity Constraints

The scheduling problem has two sets of decision variables. The first is the number of application instances $ninsts_{st} \in \mathbb{N} \cup \{0\}$ to deploy at each site $st \in ST$. The second is the site assignment for each user group $lug \in LUG$, represented by a binary variable $x_{st,lug} \in \{0, 1\}$ that takes the value 1 if user group lug is assigned to site st and 0 otherwise. Together, these two variables fully define a placement plan.

The number of instances deployed at a site is bounded by the available resource capacity. As defined in Chapter 3, we let each instance occupy exactly one resource slot for instance, and the total number of slots available at site st is $nslots_{st}$. The capacity constraint is therefore:

$$ninsts_{st} \leq nslots_{st} \quad \forall st \in ST \tag{5.1}$$

And each user group must be assigned to exactly one site, enforced by the assignment constraint:

$$\sum_{st \in ST} x_{st,lug} = 1 \quad \forall lug \in LUG \tag{5.2}$$

5.1.2 Arrival Rate and SLO Constraint

Given the site assignment of all user groups, the set of user groups assigned to site st is $LUG_{st} = \{lug \in LUG \mid x_{st,lug} = 1\}$, and the total arrival rate at site st is $\sum_{lug \in LUG_{st}} \lambda_{lug}$. As established in Chapter 3, this total rate is evenly distributed across all $ninsts_{st}$ instances at the site by kube-proxy, so each instance receives requests at rate:

$$\lambda_{st}^{inst} = \frac{\sum_{lug \in LUG_{st}} \lambda_{lug}}{ninsts_{st}} \tag{5.3}$$

The SLO requires that the θth percentile of the end-to-end latency experienced by every user group does not exceed the threshold slo. As derived in Chapter 3, since network latency and service time are deterministic, this constraint reduces to a constraint on the queueing delay distribution:

$$P\left(w_{lugsite_{lug}} \leq slo - \bar{l}_{lug,lugsite_{lug}} - \frac{1}{\mu}\right) \geq \theta\% \quad \forall lug \in LUG \tag{5.4}$$

This constraint is nonlinear in the decision variables, since the queueing delay distribution depends nonlinearly on the instance arrival rate λ_{st}^{inst}, which itself depends on both $ninsts_{st}$ and $x_{st,lug}$. Directly incorporating this constraint into a mixed-integer program is therefore not tractable and requires a linearization step.

5.1.3 Constraint Linearization

To linearize the SLO constraint, Lassy pre-computes for each pair of site st and user group lug the *maximum admissible arrival rate* $\Lambda_{st,lug}$, defined as the maximum per-instance arrival rate at site st such that the SLO is satisfied for user group lug, assuming that group is assigned to that site. Formally, $\Lambda_{st,lug}$ is the unique solution to:

$$P\left(w_{st} \leq slo - \bar{l}_{lug,st} - \frac{1}{\mu}\right)\Bigg|_{\lambda_{st}^{inst} = \Lambda_{st,lug}} = \theta\% \tag{5.5}$$

This equation has a unique solution because the queueing delay increases monotonically with the arrival rate. If the latency budget $slo - \bar{l}_{lug,st} - 1/\mu$ is negative or too small to be feasible, no solution exists and the pair (st, lug) is

marked as infeasible. Lassy solves Equation 5.5 for all feasible site-user pairs using the Brent root-finding method [84].

With $\Lambda_{st,lug}$ pre-computed, the nonlinear SLO constraint in Equation 5.4 is replaced by the following linear constraint:

$$x_{st,lug} \cdot \lambda_{st}^{inst} \leq \Lambda_{st,lug} \quad \forall st \in ST, lug \in LUG \tag{5.6}$$

This constraint ensures that if user group *lug* is assigned to site *st*, the per-instance arrival rate at that site does not exceed the maximum admissible rate for that group. The constraints in Equations 5.1, 5.2, and 5.6 together define the feasible region of the scheduling problem.

5.1.4 Optimization Objectives

The scheduling problem as formulated above may admit multiple feasible solutions. An optimization objective is therefore required to select the best among them. We consider three different objectives that reflect different practical concerns of application owners, as described below. Since cloud and edge providers typically charge fees based on the number of allocated resource slots, the primary optimization objective considered in this thesis is the minimization of total operational cost:

$$\begin{aligned} &\min \sum_{st \in ST} c_{st} \cdot ninsts_{st} \\ &\text{s.t.} \quad (5.1),\ (5.2),\ (5.6), \end{aligned} \tag{5.7}$$

where c_{st} is the cost per instance slot at site *st*. This objective reflects the practical concern of application owners who wish to meet their SLOs at the lowest possible resource cost.

An alternative objective that is also considered in our evaluation is the minimization of the total number of deployed instances, regardless of their cost:

$$\begin{aligned} &\min \sum_{st \in ST} ninsts_{st} \\ &\text{s.t.} \quad (5.1),\ (5.2),\ (5.6), \end{aligned} \tag{5.8}$$

This objective is appropriate when the application owner wishes to minimize resource consumption independently of site-specific pricing, for example to preserve edge capacity for other applications, or the general costs due to fragmentation.

A third objective is the minimization of the number of active sites:

$$\begin{aligned} &\min \sum_{st \in ST} y_{st} \\ &\text{s.t.} \quad (5.1),\ (5.2),\ (5.6), \end{aligned} \tag{5.9}$$

This objective is useful when deploying an application on a new site entails significant overhead, such as bootstrapping costs, reduced cache efficiency, or administrative burden. By minimizing the number of active sites, the scheduler consolidates the deployment onto as few sites as possible while still satisfying the SLO constraints for all user groups.

The constraints in Equations 5.1, 5.2, and 5.6, together with either of the optimization objectives above, form a Mixed-Integer Linear Program (MILP) that could be solved, where the decision variables are the optimized placement plan defined by $ninsts_{st}$ and $x_{st,lug}$ for all sites and user groups.

5.2 Lassy Scheduler

Having formulated the scheduling problem as a Mixed-Integer Program in Section 5.1, we now describe how Lassy solves it in practice. This section first establishes the computational complexity of the problem, then presents the optimization solver used to tackle it, and finally describes the end-to-end scheduling algorithm.

5.2.1 Computational Complexity

The scheduling problem formulated above is NP-hard in general. To see this, consider a simplified instance where all sites have identical network latencies and service rates, and the SLO constraint reduces to a simple capacity constraint on the number of instances per site. In this case, the problem of assigning user groups to sites while respecting site capacity reduces to a variant of the *bin packing problem*, in which user groups correspond to items with weights determined by their request rates, and sites correspond to bins with capacities determined by the maximum admissible arrival rate $\Lambda_{st,lug}$. Since bin packing is NP-hard [85], the scheduling problem is NP-hard as well. In practice, the additional constraints introduced by heterogeneous network latencies and site-specific SLO feasibility further increase the difficulty of the problem.

5.2.2 Gurobi Optimization Solver

Given the NP-hard nature of the scheduling problem, Lassy relies on the *Gurobi Optimizer* [86], a state-of-the-art commercial mixed-integer programming solver, to obtain optimal placement plans. Gurobi implements a branch-and-bound algorithm augmented with cutting planes, heuristics, and presolve techniques that allow it to efficiently solve MILP problems of practical size. It accepts the problem formulation directly in terms of decision variables, linear constraints, and an optimization objective, and returns the optimal values of all decision variables upon convergence.

The MILP formulated in Section 5.1 is well-suited for Gurobi. The decision variables $ninsts_{st}$, y_{st}, and $x_{st,lug}$ are integer and binary respectively, the constraints are all linear after the pre-computation of $\Lambda_{st,lug}$, and the optimization objectives are linear functions of the decision variables. The pre-computation of $\Lambda_{st,lug}$ using the Brent root-finding method [84] is performed outside Gurobi as a preprocessing step, and its results are passed to the solver as fixed parameters. This separation between the nonlinear latency model and the MILP is what makes the problem tractable in practice.

5.2.3 Algorithm Description

The Lassy scheduling algorithm consists of three phases: preprocessing, constraint construction, and optimization. In the preprocessing phase, the maximum admissible arrival rates $\Lambda_{st,lug}$ are computed for all site-user group pairs, linearizing the SLO constraint and reducing the problem to a MILP. In the constraint construction phase, the MILP constraints are assembled and passed to the Gurobi solver. In the optimization phase, Gurobi solves the MILP and returns the optimal placement plan. We describe each phase below and illustrate the key implementation steps with code excerpts from the Lassy prototype.

Preprocessing. In the preprocessing phase, Lassy computes the maximum admissible arrival rate $\Lambda_{st,lug}$ for each feasible site-user group pair by solving Equation 5.5 using the Brent root-finding method. The waiting time distribution $P(w_{st} \leq t)$ is implemented directly from the closed-form expression derived in Equation 3.7 as the `waitingTimeDistr` function. For each site-user group pair, the available latency budget $slo - \bar{l}_{lug,st} - 1/\mu$ is first evaluated, and pairs with a non-positive budget are marked as infeasible. For feasible pairs, the Brent solver finds the unique arrival rate at which the θth percentile of the queueing delay exactly equals the latency budget, yielding $\Lambda_{st,lug}$. Listing 5.1 shows the implementation of both the waiting time distribution and the pre-computation of $\Lambda_{st,lug}$.

Assignment Constraints. These constraints collectively ensure that each user group is assigned to exactly one active site, that any active site has at least one instance deployed, and that the number of user groups assigned to each site is tracked. While not all of these constraints are formalized as explicit equations in Section 5.1, they are necessary in the implementation to ensure the correctness of the placement plan. Listing 5.2 shows their implementation.

Resource Constraints. The capacity constraint ensures that the number of deployed instances does not exceed the available slots at each site, and that instances are only deployed on active sites. Listing 5.3 shows its implementation.

```
def waitingTimeDistr(t, lm):
    mu = service_rate
    s = 0.0
    for i in range(int(t*mu)+1):
        s = s + math.exp(-lm*(i/mu-t)) * pow((i/mu-t)*lm,i) / math.
            factorial(i)
    return (1-lm/mu)*s

lambdaLimits = {
    i: {
        j: spo.fsolve(
            lambda lm: waitingTimeDistr(
                slo - latency_node_user[i,j] - service_time, lm
            ) - theta,
            0.5
        )[0]
        for j in users
    }
    for i in sites
}
```

Listing 5.1: Implementation of the $M|D|1$ waiting time distribution $P(w_{st} \leq t)$ from Equation 3.7 and pre-computation of maximum admissible arrival rates $\Lambda_{st,lug}$ for each site-user group pair.

SLO Constraints. The linearized SLO constraint ensures that the per-instance arrival rate at each site does not exceed the maximum admissible rate $\Lambda_{st,lug}$ pre-computed in Listing 5.1 for each assigned user group, directly enforcing Equation 5.6. A constraint is added for each site-user group pair, as shown in Listing 5.4.

Optimization Objective. Lassy sets the optimization objective and invokes Gurobi to solve the MILP. The objective is configurable and can be set to minimize the total operational cost, the total number of instances, or the number of active sites, as described in Section 5.1. Only one objective is active at a time, selected by the application owner prior to scheduling. Listing 5.5, Listing 5.6 and Listing 5.7 shows the three supported objectives respectively, of which exactly one is passed to Gurobi before the solver is invoked.

Upon convergence, Gurobi returns the optimal values of $ninsts_{st}$ and $x_{st,lug}$ for all sites and user groups, which together define the placement plan. This plan is then enacted by the Kubernetes orchestration platform, which deploys or removes pods accordingly and updates the routing configuration to direct each user group to its assigned site. Lassy can be re-invoked whenever the input parameters change significantly, for example when the request rate of a user group increases beyond a threshold or when network latencies shift, triggering a rescheduling that produces an updated placement plan adapted to the new conditions.

```
model.addConstrs(
    (x.sum('*', j) == 1 for j in users), name="assignToOne"
    )

model.addConstrs(
    (x[i, j] <= y[i] for i in sites for j in users), name="linkToOpen")

model.addConstrs(
    (instance_per_site[i] >= y[i] for i in sites),
        name="atLeastOneInstanceIfOpen")

model.addConstrs(
    (u[i] == x.sum(i, '*') for i in sites), name="numUsers"
)
```

Listing 5.2: Assignment constraints added to the Gurobi model.

```
model.addConstrs(
    (instance_per_site[i] * y[i] <= capacities[i] for i in sites),
    name="capacity"
)
```

Listing 5.3: Resource capacity constraint added to the Gurobi model.

5.3 Experimental Evaluation

This section evaluates Lassy on a real Kubernetes testbed. The experimental setup, applications, workload generation, and baselines are those described in Chapter 4. The accuracy of the underlying $M|D|1$ latency model and its ability to correctly determine the number of instances necessary to meet tail-latency SLOs in a single-site setting were validated in Sections 4.4.1 and 4.4.2, respectively, and are not repeated here. We focus instead on the aspects specific to Lassy: its ability to compute effective placement plans in a realistic multi-site scenario featuring multiple edge sites and user groups, under varying workload conditions and optimization objectives, and the computation time of the scheduling algorithm. Lassy is evaluated with the Thumbnailing and OCR applications introduced in Section 4.3. As Lassy schedules a single application at a time, these two applications, representing the network-dominated and computation-heavy ends of the workload spectrum, are sufficient to cover the relevant service time profiles; the full four-application set of Chapter 4 is used from Chapter 6 onward, where the simultaneous scheduling of multiple applications is the object of study.

```
model.addConstrs(
    (x[i, j] * instance_arrival_rate[i] <= lambdaLimits[i][j]
    for i in sites for j in users),
    name="lambdaMax"
)
```

Listing 5.4: Linearized SLO constraints added to the Gurobi model.

```
# Objective min_cost: minimize total operational cost
model.setObjective(price.sum(), GRB.MINIMIZE)

model.optimize()
```

Listing 5.5: Optimization objectives of minimizing the total operational cost.

5.3.1 Multi-site: Realistic Placement Evaluation

This experiment evaluates Lassy in a realistic multi-site edge-cloud continuum scenario, assessing its ability to compute effective placement plans across multiple sites and user groups, under different workload configurations and optimization objectives.

Setup. The multi-site scenario features one cloud site, three edge sites, and five LUGs. The cloud site has ample computing resources, while edge sites are resource-constrained. The network latency setup is based on real-world measurements from Ali-Eldin et al. [8], where the cloud site generally has significantly higher network latency to users than edge sites, while some edge sites have higher latency than the cloud to certain LUGs. The resource capacities, slot costs, and network latencies are summarized in Table 5.2. Edge resources are more valuable and therefore more expensive than cloud resources. The SLOs are identical to those used in Section 4.4.2: $\theta = 0.99$ with $slo = 100$ ms for thumbnailing and $slo = 400$ ms for OCR.

Setups	Slots	Pricing	Network latency (ms)				
			LUG1	LUG2	LUG3	LUG4	LUG5
Cloud	15	5	40	50	60	60	50
Edge 1	5	10	5	5	65	65	5
Edge 2	8	8	45	10	5	20	60
Edge 3	8	8	50	60	20	5	10

Table 5.2: Resource slots and latency setup for multi-site experiments

We evaluate seven workload configurations, summarized in Table 5.3. In configurations #1 to #4, the request rate is progressively increased uniformly

```
# Objective min_slots: minimize total number of instances
model.setObjective(instance_per_site.sum(), GRB.MINIMIZE)

model.optimize()
```

Listing 5.6: Optimization objectives of minimizing the total number of instances.

```
# Objective min_sites: minimize number of opening sites
model.setObjective(y.sum(), GRB.MINIMIZE)

model.optimize()
```

Listing 5.7: Optimization objectives of minimizing the number of opening sites

across all five LUGs. In configurations #5 to #7, the request rate of LUG1 is drastically increased while other LUGs maintain the rate of configuration #4, emulating a localized demand burst caused for example by a local event. We evaluate Lassy under two optimization objectives: minimizing the total number of instances and minimizing the total operational cost.

Setups	Configurations	Request Rate (req/s)				
		LUG1	LUG2	LUG3	LUG4	LUG5
Thumbnailing	#1	5	5	5	5	5
	#2	15	15	15	15	15
	#3	30	30	30	30	30
	#4	45	45	45	45	45
	#5	60	45	45	45	45
	#6	80	45	45	45	45
	#7	100	45	45	45	45
OCR	#1	1	1	1	1	1
	#2	2	2	2	2	2
	#3	3	3	3	3	3
	#4	4	4	4	4	4
	#5	6	4	4	4	4
	#6	8	4	4	4	4
	#7	10	4	4	4	4

Table 5.3: Request rate configurations from #1 to #7 for multi-site experiments

SLO Compliance Analysis. Figure 5.1 shows the empirical mean and P99 end-to-end latency for all LUGs across all configurations and both optimization objectives. The placement plans computed by Lassy successfully keep the P99 latency below the SLO threshold in the vast majority of cases for both applications and both optimization objectives. In the few cases where

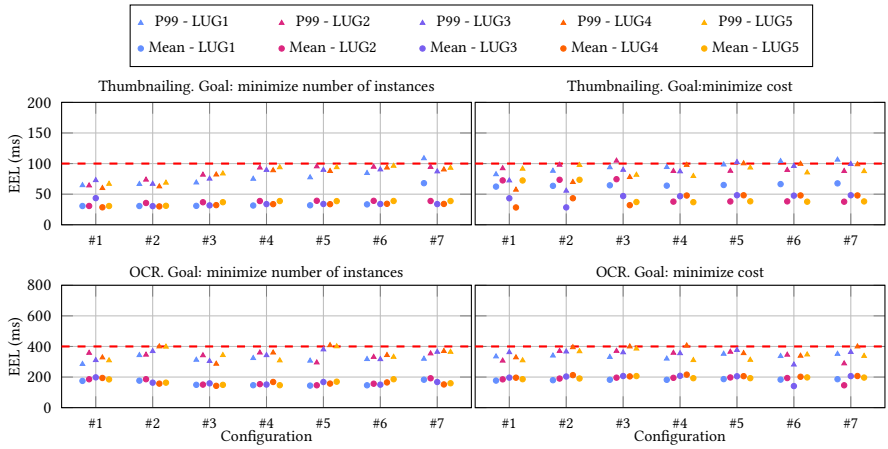


Figure 5.1: Empirical mean and P99 end-to-end latency (EEL) for different LUGs, different multi-site configurations, different optimization goals, and the two applications. The dashed line indicates the P99 tail latency SLO.

the SLO is marginally exceeded, the violation is small and attributable to the conservative underestimation behavior of the $M|D|1$ model at high utilization, as identified in Section 4.4.1. Overall, these results confirm that Lassy can effectively enforce tail latency SLOs in a complex multi-site scenario with varying workload conditions and optimization goals.

Figure 5.2 shows the empirical cumulative distribution of the end-to-end latency for configurations #3, #5, and #7, confirming that the latency distribution remains well-controlled even under the localized demand burst of configurations #5 to #7.

Placement Plans and Optimization Objectives Analysis. Figure 5.3 visualizes the placement plans computed by Lassy under the two optimization objectives across all configurations. The two objectives produce qualitatively different placement strategies. When minimizing the number of instances, Lassy favors edge sites over the cloud as long as sufficient edge capacity is available, since edge sites offer lower network latency to nearby users. Only at very high request rates, in configuration #7, does Lassy resort to deploying a significant number of instances at the cloud site to absorb the excess demand that edge sites cannot handle. When minimizing cost, the behavior changes: since the cloud site is cheaper than the edge sites in our setup, Lassy shifts more instances to the cloud, exploiting the fact that the cloud's greater resource availability compensates for its higher network latency. These results demonstrate the flexibility of Lassy in adapting its placement strategy to different optimization goals while consistently satisfying the SLO constraints.

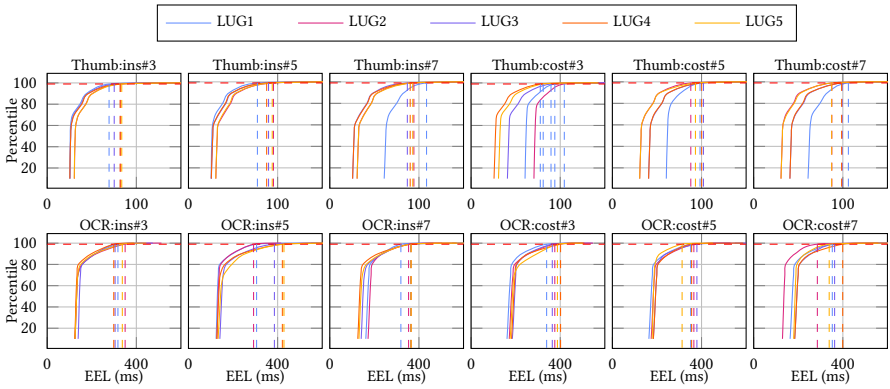


Figure 5.2: End-to-end latency distribution under different multi-site configurations for the two applications and the two optimization goals based on the placement plan computed by Lassy. The label "Thumb:ins#3" means "Thumbnailing application, minimizing the number of instances, configuration #3". The dashed lines indicate the P99 end-to-end latency for the different LUGs.

5.3.2 Computation Time

Figure 5.4 shows the time required by Lassy to compute a placement plan as a function of the number of LUGs, for the four-site setup used in these experiments. The computation time grows with the number of LUGs but remains within practical limits for all realistic scenario sizes, confirming that Lassy is suitable for deployment in real edge-cloud environments where rescheduling must be completed within a reasonable time window.

5.4 Conclusion

This chapter has presented Lassy, a Latency-Aware SLO-Sufficing Scheduler for single applications in the edge-cloud continuum. Building on the queueing-theoretic latency model developed in Chapter 3, we formulated the scheduling problem as a Mixed-Integer Linear Program that directly enforces tail-latency SLO constraints while minimizing predefined operational costs. The key technical contribution of the formulation is the linearization of the non-linear SLO constraint through the pre-computation of maximum admissible arrival rates, which makes the problem tractable for off-the-shelf MIP solvers while preserving the accuracy of the underlying latency model. The modeling and design choices underlying Lassy, including the queueing model, the request distribution strategy, and the choice of optimization solver, are shared by all the schedulers presented in this thesis. We therefore defer their discussion to the general discussion in Chapter 9.

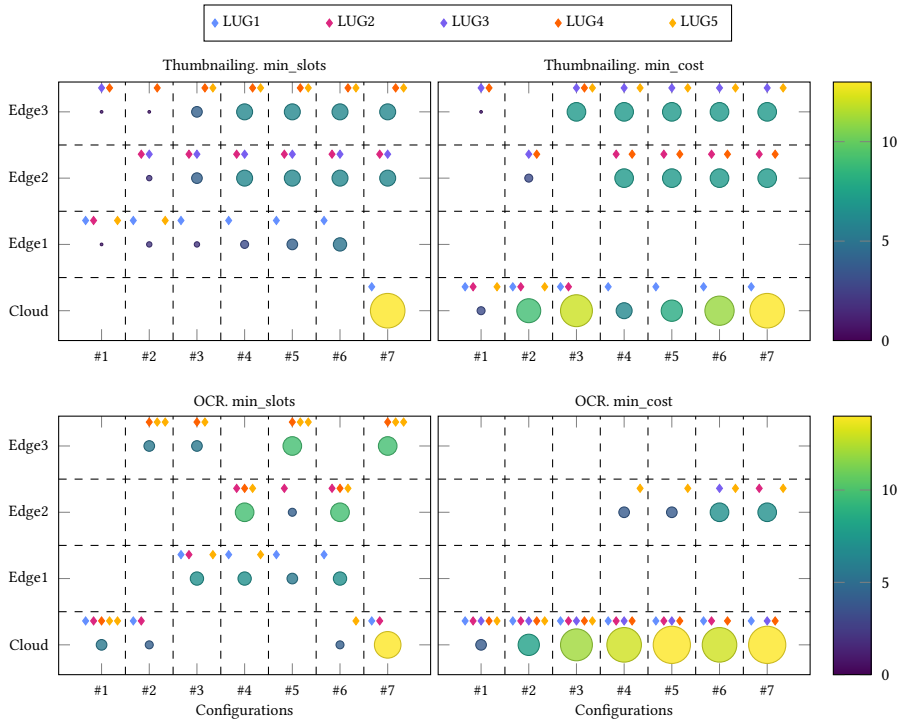


Figure 5.3: Placement plans of the instances and user-to-site mapping under different configurations and optimization goals. The discs represent the number of instances allocated per site. The diamond markers represent the LUGs assigned to the sites. Lassy could effectively gives different placement plans based on different optimization goals.

The experimental evaluation on a real Kubernetes testbed demonstrated three main results. First, the $M|D|1$ model achieves a mean prediction error of 10% compared to 49% for the $M|M|c$ model used by INVAR, confirming that the deterministic service time and per-instance dispatching assumptions are a significantly better fit for the class of workloads considered in this thesis. Second, Lassy correctly determines the number of instances necessary to meet P99 SLOs across a wide range of request rates, reducing the tail latency by up to 30% for the thumbnailing service and up to 50% for the OCR service during high concurrency periods compared to INVAR and Kubernetes HPA. Third, in a realistic multi-site scenario with multiple edge sites and user groups, Lassy consistently satisfies SLO constraints under varying workload conditions and demand bursts, while adapting its placement strategy flexibly to different optimization objectives.

Despite these contributions, Lassy addresses only the scheduling of a single application in isolation. In a multi-tenant edge environment where multi-

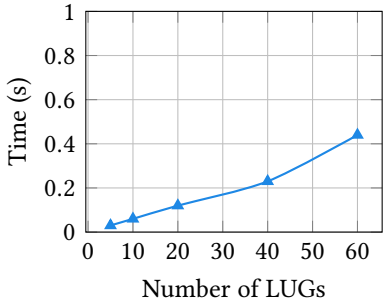


Figure 5.4: Computation time of Lassy for four sites.

ple applications share the same limited resources, scheduling each application independently can lead to resource contention and cascading SLO violations, as discussed in Section 2.4.2. The following chapter addresses this limitation by extending the scheduling framework to multiple co-located applications, introducing a global optimization approach that considers the resource demands and SLO requirements of all applications simultaneously.

Multi-Application Scheduling with Scylla

6

The Lassy scheduler developed in Chapter 5 treats an application as the sole occupant of the infrastructure, implicitly assuming that the resources of every site are exclusively available to it. Real edge deployments are multi-tenant: a single site hosts several applications at once, each with its own workload, resource footprint, and tail-latency SLO, all drawing from the same limited pool of slots. Scheduling each of them in isolation is not merely suboptimal but actively harmful, since the placement decisions taken for one application silently constrain those available to the others. This chapter lifts the single-application assumption and formulates the scheduling of all co-located applications as a single, globally optimized problem. Doing so raises a computational challenge absent from the previous chapter, as the joint decision space grows exponentially with the number of applications, and a substantial part of this chapter is devoted to the heuristic algorithms that make the problem tractable at realistic scales.

6.1 Introduction

Scheduling a single application in the edge-cloud continuum, as addressed in Chapter 5, already presents significant challenges in balancing network latency, queueing delays, and resource constraints. In practice, however, edge sites host multiple applications simultaneously, each with its own workload characteristics, resource requirements, and tail-latency SLOs. Scheduling each application independently, as in Lassy, ignores the interactions between co-located applications competing for the same limited edge resources. As discussed in Section 2.4.2, this can lead to resource contention, cascading SLO violations, and poor global resource utilization, outcomes that a globally aware scheduler can avoid.

This chapter presents Scylla, a multi-application latency-aware scheduler for the edge-cloud continuum. Scylla extends the scheduling framework of Chapter 5 to jointly consider multiple co-located applications, enabling global optimization objectives that go beyond the individual view of a single application. It builds on the same queueing-theoretic latency model developed in Chapter 3, extending it to account for resource contention among applications

sharing the same edge site. Scylla computes globally optimized placement plans that ensure the tail-latency SLOs of all applications are simultaneously satisfied, while minimizing system-wide objectives such as the total number of deployed instances, the total operational cost, or the number of active sites.

This chapter is organized as follows. Section 6.2 reviews related work on multi-application scheduling in cloud and edge environments. Section 6.3 extends the problem formulation of Chapter 5 to the multi-application setting. Section 6.4 presents the Scylla scheduler and its algorithm. Section 6.5 evaluates Scylla on a real Kubernetes testbed with multiple applications. Section 6.6 discusses the results and the limitations of the approach. Section 6.7 concludes the chapter.

6.2 Related Work

The general landscape of scheduling approaches for latency-sensitive applications across cloud and edge environments is covered in Section 2.5, and the works most directly related to single-application scheduling are discussed in Chapter 5. We focus here specifically on works that address the scheduling of multiple applications in shared infrastructure, which is the core challenge addressed by Scylla.

As established in Section 2.5, existing schedulers targeting the edge-cloud continuum — including INVAR [12], Mu [14], Phare [17], and Polaris [16] — all schedule a single application at a time. This per-application view forfeits the ability to reason about the interactions between co-located applications competing for shared resources, and cannot prevent the cascading SLO violations that arise from independent scaling decisions in a multi-tenant edge environment.

In the cloud computing literature, several works address resource allocation for multiple co-located applications. ATOM [87] proposes a model-driven autoscaling framework for microservices that uses queuing models to inform scaling decisions. However, like most cloud-focused schedulers, it assumes abundant and homogeneous resources and does not account for the distributed nature of the edge-cloud continuum or the varying network latencies between geographically distributed users and sites. Workflow scheduling approaches [88, 89] address the scheduling of multiple interdependent tasks across cloud resources, but target batch workloads whose performance is measured by the completion time of the workflow as a whole, rather than by the tail of the per-request latency distribution of a continuously served request stream.

To the best of our knowledge, no existing system simultaneously accounts for an edge-cloud distributed setting, accurate tail-latency prediction validated against real Kubernetes deployments, and a global optimization ap-

proach that jointly schedules multiple applications while minimizing system-wide resource objectives. Scylla addresses this gap, extending the single-application framework of Chapter 5 to a globally optimized multi-application scheduler for the edge-cloud continuum.

6.3 Multi-application Scheduling Modeling

This section extends the system model and optimization problem formulation of Chapters 3 and 5 to the multi-application setting, introducing the elements required to reason about shared resource consumption and to jointly schedule applications with heterogeneous resource requirements and individual tail-latency SLOs.

6.3.1 Multi-Tenancy in Kubernetes

Deploying several applications on the same site introduces two considerations absent from the single-application setting of Chapter 5, where a site's resources were exclusively available to one application and kube-proxy distributed its traffic uniformly across all of its pods, as described in Section 2.2.2.

Namespace Isolation. Kubernetes uses *namespaces* to logically isolate applications from each other within the same cluster. Each application is deployed in its own namespace, with its own set of pods and its own Kubernetes Service endpoint. The kube-proxy instance at each node routes incoming traffic to the correct set of pods based on the destination service, ensuring that requests for one application are never dispatched to pods of another. This isolation guarantees that the per-instance queueing model developed in Chapter 3 remains valid in the multi-tenant setting: each pod of a given application receives an equal share of that application's incoming traffic and maintains its own independent queue, independently of the pods of other applications running on the same node. The edge-cloud continuum infrastructure with multi-application scheduling is illustrated in Figure 6.1.

Shared Resource Pool. While namespace isolation ensures traffic separation, all applications deployed on the same site draw from the same finite pool of CPU and memory available on its nodes. Kubernetes assigns each pod a fixed share of these resources through its resource requests and limits, enforced by the cgroups mechanism described in Section 2.2. Applications therefore compete for how much of the node's capacity each of them may claim, rather than for the throughput of an allocation already granted. Pods of different applications do not necessarily claim the same amount: a lightweight image processing service may require far less CPU and memory per pod than a heavy AI inference application. This heterogeneity must be

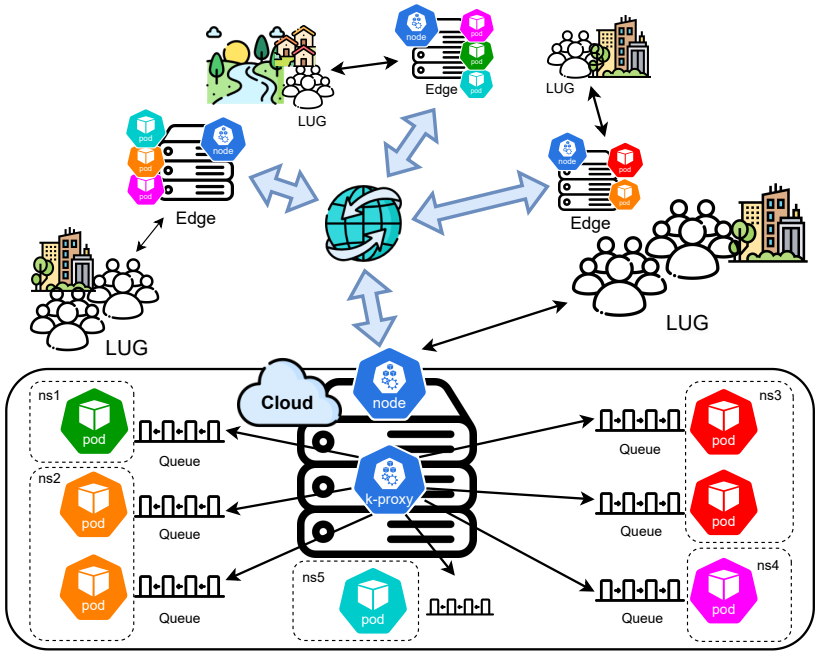


Figure 6.1: A typical edge-cloud continuum infrastructure with multiple applications co-located at each site and namespace isolation provided by Kubernetes.

explicitly accounted for in the scheduling model, so that the combined demand of all co-located applications does not exceed the capacity available at any site.

6.3.2 Tail Latency Prediction for Multiple Applications

Namespace isolation in Kubernetes, as established in Section 6.3.1, ensures that each application's traffic is dispatched independently across its own pods, each maintaining its own queue. The $M|D|1$ per-instance model of Chapter 3 therefore applies to each application independently, without modification. The notation used throughout this chapter is summarized in Table 6.1.

Per-Application Latency Model. We extend the notation of Chapter 5 to the multi-application setting by introducing an application identifier $app \in APP$. Each application app has its own service rate μ_{app}, its own set of LUGs, and its own tail-latency SLO defined as a maximum acceptable P99 end-to-end latency slo_{app}. Furthermore, each application app requires σ_{app} computation slots, reflecting the heterogeneous resource footprints of different applications.

Name	Description
$st \in ST$	Cloud and edge sites
$app \in APP$	Applications
$nslots_{st}$	Slots of computing resource available at site st
$lug \in LUG$	Local user group(s)
$l_{lug,st}$	Round trip netw. latency between user group lug and site st
λ_{lug}	Request rate of user group lug
$\Lambda_{st,lug}$	Maximum arrival rate for the instances at site st can accept without breaking the SLO for user group lug
$lugapp_{lug}$	Application used by LUG lug
σ_{app}	Slots required by each instance of application app
μ_{app}	Service rate of an application instance
c_{st}	Cost per slot at site st
slo_{app}	SLO: Maximum tail latency (P99) for application app
$ninsts_{st,app}$	Instances of app deployed on site st
$lugsite_{lug}$	Site that user group lug is directed to
$x_{st,lug}$	Whether user group lug is assigned to site st
y_{st}	Whether site st is active, i.e. hosts at least one instance
$L_{st,app}$	Set of user groups sending requests for app to site st
$w_{st,app}$	Queueing delay at site st for app

Table 6.1: Model parameters (top) and variables (bottom). Rows with a grey background are introduced in this chapter.

The number of instances of application app deployed at site st is denoted $ninsts_{st,app}$, and the per-instance arrival rate at site st for application app is:

$$\lambda^{inst}_{st,app} = \frac{\sum_{lug \in LUG_{st,app}} \lambda_{lug}}{ninsts_{st,app}} \tag{6.1}$$

where $LUG_{st,app}$ denotes the set of LUGs assigned to site st for application app. The utilization of each instance of application app at site st is:

$$\rho_{st,app} = \frac{\lambda^{inst}_{st,app}}{\mu_{app}} \tag{6.2}$$

The queueing delay distribution for each instance of application app at site st follows the same $M|D|1$ cumulative distribution function derived in Equation 3.7, parameterized by $\lambda^{inst}_{st,app}$ and μ_{app}:

$$P(w_{st,app} \leq t) = (1 - \rho_{st,app}) \sum_{i=0}^{\lfloor t \cdot \mu_{app} \rfloor} e^{-\lambda^{inst}_{st,app}(i/\mu_{app} - t)} \frac{(\lambda^{inst}_{st,app})^i (i/\mu_{app} - t)^i}{i!} \tag{6.3}$$

The end-to-end latency for a request from LUG *lug* for application *app*, served at site $lugsite_{lug}$, is then:

$$L_{lug,lugsite_{lug},app} = l_{lug,lugsite_{lug}} + w_{lugsite_{lug},app} + \frac{1}{\mu_{app}} \tag{6.4}$$

and the SLO constraint for application *app* requires that:

$$P\left(w_{lugsite_{lug},app} \leq slo_{app} - l_{lug,lugsite_{lug}} - \frac{1}{\mu_{app}}\right) \geq \theta\% \quad \forall lug \in LUG_{app} \tag{6.5}$$

Independence Across Applications. A key observation is that under the namespace isolation guarantee of Kubernetes, the queueing delay of one application's pods is independent of the load generated by other applications, as each application's pods maintain separate queues and receive only their own traffic. The tail latency of each application can therefore be predicted independently using the same $M|D|1$ model, with the only coupling between applications occurring at the level of slot availability, which is captured by the capacity constraint introduced in the following subsection. This holds as long as each instance receives a genuinely exclusive share of the node's resources; where that isolation is imperfect, the resulting interference would degrade the effective service rate, an effect the model does not capture.

6.4 Scylla Scheduler

Having formulated the multi-application scheduling problem in Section 6.3, we now describe how Scylla solves it in practice. For small-scale scenarios, Scylla employs a globally optimal solver. For larger scenarios where the computational cost of global optimization becomes prohibitive, Scylla provides a practical heuristic that trades a small degree of optimality for a significant reduction in computation time.

6.4.1 Globally Optimal Scheduling

The single-application MILP formulated in Chapter 5 determines the number of instances to deploy at each site and the assignment of user groups to sites for a single application. We now extend this formulation to the multi-application setting, introducing the additional constraints required to reason about shared resource consumption across multiple co-located applications.

Extended Resource Capacity Constraint. The key extension to the optimization problem concerns the resource capacity constraint. In the single-application case, each instance occupied exactly one uniform resource slot, and the capacity constraint simply bounded the number of instances at each

site by the available slots. Instances of different applications do not necessarily have the same resource footprint. We therefore associate each application app with a slot demand σ_{app}, the number of resource slots consumed by one of its instances, reflecting the heterogeneous requirements of the applications used in our model. The shared capacity constraint must therefore account for the aggregate resource consumption of all co-located applications at each site:

$$\sum_{app \in APP} ninsts_{st,app} \cdot \sigma_{app} \leq nslots_{st} \quad \forall st \in ST \tag{6.6}$$

This constraint couples the scheduling decisions of all applications at a site: deploying more instances of one application reduces the resources available for all others, and the scheduler must jointly optimize the instance counts of all applications to find a globally feasible and cost-efficient placement plan. Each user group must be assigned to exactly one site, enforced by the assignment constraint:

$$\sum_{st \in ST} x_{st,lug} = 1 \quad \forall lug \in LUG \tag{6.7}$$

Per-Application SLO Constraints and Linearization. The SLO constraint of Equation 6.5 is defined independently for each application app and each LUG $lug \in LUG_{app}$. As in Chapter 5, each nonlinear SLO constraint is linearized by pre-computing the maximum admissible per-instance arrival rate $\Lambda_{st,lug,app}$ for each feasible site-user group-application triple, defined as the unique solution to:

$$P\left(w_{st,app} \leq slo_{app} - l_{lug,st} - \frac{1}{\mu_{app}}\right)\Bigg|_{\lambda^{inst}_{st,app}=\Lambda_{st,lug,app}} = \theta\% \tag{6.8}$$

The linearized SLO constraint then becomes:

$$x_{st,lug} \cdot \lambda^{inst}_{st,app} \leq \Lambda_{st,lug,app} \quad \forall st \in ST, lug \in LUG_{app}, app \in APP \tag{6.9}$$

Optimization Objectives. The optimization objectives supported by Scylla generalize those of Lassy to the multi-application setting. The cost minimization objective accounts for the heterogeneous resource consumption of each application:

$$\begin{aligned} &\min \sum_{st \in ST, app \in APP} c_{st} \cdot ninsts_{st,app} \cdot \sigma_{app} \\ &\text{s.t.} \quad (6.6),\ (6.7),\ (6.9), \end{aligned} \tag{6.10}$$

Alternatively, the total number of deployed instances across all applications can be minimized:

$$\begin{aligned} &\min \sum_{st \in ST, app \in APP} ninsts_{st,app} \\ &\text{s.t.} \quad (6.6),\ (6.7),\ (6.9), \end{aligned} \tag{6.11}$$

Or the number of active sites can be minimized, allowing unused sites to be shut down:

$$\begin{aligned} &\min \sum_{st \in ST} y_{st} \\ &\text{s.t.} \quad (6.6),\ (6.7),\ (6.9), \end{aligned} \tag{6.12}$$

The constraints in Equations 6.6, 6.9, and 5.2, together with any of the optimization objectives above, form the multi-application MILP that Scylla solves using the Gurobi Optimizer [86], referred to as G_{opt}.

6.4.2 Computational Complexity

The multi-application MILP formulated in Section 6.4.1 is a generalization of the single-application problem presented in Chapter 5, which was already shown to be NP-hard by reduction to a multi-dimensional multiple-choice knapsack problem (MMKP) [90]. The multi-application extension is potentially harder: the shared resource capacity constraint of Equation 6.6 couples the scheduling decisions of all applications, and the problem can be seen as a generalization of the multiple knapsack problem extended with latency constraints.

We denote the solver algorithm that computes *globally* the optimal solution for our scheduling problem as G_{opt}. In practice, the computation time of G_{opt} grows exponentially with the number of applications, sites, and user groups. While G_{opt} remains tractable for small scenarios involving a handful of applications, its computation time becomes prohibitive as the problem scale increases, due to the explosion of the joint search space over all application placement decisions. As we demonstrate in Section 6.5.2, G_{opt} becomes impractical beyond four to five applications for the scenario sizes considered in this thesis. This motivates the need for a heuristic algorithm that can produce high-quality placement plans within practical time limits for larger-scale scenarios, which we present in the following subsection.

6.4.3 Heuristic Algorithms

The key observation shared by all heuristics presented in this section is that the joint multi-application MILP can be decomposed into a sequence of single-application MILPs solved greedily, one application at a time, on the remaining available resources. The order in which applications are scheduled matters: scheduling a resource-hungry application first may leave insufficient resources for others, while a different ordering may yield a globally better solution. We first present the Full Permutation Search heuristic H_{Full} as the naive baseline that motivates the need for more scalable approaches, then

present the Layered Permutation Search H_{LPS} used in Scylla, addressing the scalability limitation of H_{Full}.

Naive Baseline: Full Permutation Search. The simplest heuristic approach is to enumerate all $|APP|!$ possible orderings of the applications, schedule them sequentially using G_{opt} for each ordering, and return the solution found with the best *objective value*. For each permutation π, applications are scheduled one by one: G_{opt} is run for the first application in π on the full available resources, then for the second application on the remaining resources after the first is placed, and so on until all applications are placed or one is found to be infeasible. The best solution across all permutations is retained as the output. As we demonstrate in Section 6.5.2, H_{Full} returns solutions of the same quality as G_{opt} for all scenario sizes we evaluate, while computing them considerably faster, since each single-application subproblem is far smaller than the full multi-application MILP. This equivalence is an empirical observation on the scenarios considered here rather than a guarantee: the sequential decomposition offers no optimality bound, and nothing precludes a gap from appearing at larger problem scales, where the coupling between application placements is stronger. H_{Full} is moreover limited by the factorial growth in the number of permutations with the number of applications, which makes it impractical beyond five to six applications. This motivates the two scalable heuristics presented below.

Layered Permutation Search Heuristic. Due to the factorial growth of the computational time in H_{Full}, we propose the Layered Permutation Search (LPS) Heuristic. The LPS strategy is inspired and can be viewed as a generalized `Beam Search` [91] where the beam width is determined by the prefix depth d. At depth $d = 1$, H_{LPS} explores $|APP|$ permutations, each starting with a different application. At depth $d = |APP|$, it exhaustively explores all $|APP|!$ permutations, equivalent to a full search. This progressive structure exposes a controllable depth-runtime trade-off: a shallow depth yields fast computation at the cost of potential optimality loss, while increasing the depth improves solution quality at the cost of exponentially growing computation time.

H_{LPS} builds directly on the full permutation search principle of H_{Full}, Instead of enumerating all $|APP|!$ permutations, H_{LPS} takes a depth parameter d and evaluates all ordered prefixes of d applications, while appending the remaining $|APP| - d$ applications in a random order. For each permutation, applications are scheduled sequentially via G_{opt} (lines 6–11): for each application in turn, G_{opt} determines the optimal instance count and *LUG* to *ST* assignment subject to the tail latency SLO constraint and the remaining available resource budget; if no feasible placement exists, the permutation is discarded and the next one is tried. If the resulting solution $\mathcal{S}'$ yields a lower objective value $f(\mathcal{S}')$ than the best found so far, the global solution is

Algorithm 1 H_{LPS}: Layered Permutation Search Heuristic

```
1: Input: d                                              ▷ Desired depth parameter
2: S ← ∅, V ← ∞                                          ▷ Best solution and objective value
3: P ← GeneratePrefixPermutations(APP, d)
4: for π ∈ P do
5:     avail_resources ← nslot_S, S' ← ∅
6:     for app in π do
7:         G_opt(app, avail_resources)
8:         if scheduling feasible then
9:             Update S' and avail_resources
10:        else
11:            S' ← ∅; break
12:    if S' ≠ ∅ and f(S') < V then
13:        S ← S', V ← f(S')
14: return S
```

updated (line 13). By increasing d from 1 to $|APP|$, the search space is explored progressively: at $d = 1$, only $|APP|$ permutations are evaluated, while at $d = |APP|$, all $|APP|!$ permutations are covered. The depth d thus acts as a tunable trade-off between solution quality and computational cost.

6.5 Experimental Evaluation

This section evaluates Scylla on a real Kubernetes testbed. The experimental setup, applications, workload generation, and baselines are those described in Chapter 4. The accuracy of the $M|D|1$ latency model for the four applications used in this chapter, as well as its ability to determine the number of instances necessary to meet tail-latency SLOs at a single site, were validated in Sections 4.4.1 and 4.4.2, respectively, and are not repeated here. We focus instead on the aspects specific to Scylla: its ability to compute placement plans that satisfy the tail-latency SLOs of all applications simultaneously in a realistic multi-site, multi-application scenario, under different workload configurations and optimization objectives, and the scalability and solution quality of the H_{LPS} heuristic compared to G_{opt} and H_{Full}.

6.5.1 Multi-Site: Realistic Placement Evaluation

This experiment evaluates Scylla in a realistic multi-site, multi-application scenario, assessing its ability to compute globally optimized placement plans that satisfy the tail-latency SLOs of all four applications simultaneously, under varying workload conditions and different optimization objectives.

Setup. The infrastructure consists of one cloud site and four edge sites, with capacities and pricing as summarized in Table 6.2. The network latencies between LUGs and sites are based on real-world measurements [8] and are

	Slots	Price	Network latency from LUG to site (ms)									
			LUGs 1-4	LUGs 5-8	LUGs 9-12	LUGs 13-16	LUGs 17-20	LUGs 21-24	LUGs 25-28	LUGs 29-32	LUGs 33-36	LUGs 37-40
C	72	$1	38	58	34	31	34	60	57	42	60	60
E1	36	$3	56	108	16	49	84	10	61	42	60	110
E2	36	$4	12	70	52	81	40	60	107	20	110	60
E3	36	$2	88	58	48	19	60	60	11	92	16	60
E4	36	$2	44	20	84	51	16	110	57	70	66	10

Table 6.2: Resource and pricing for the cloud (C) and edge (E) sites, and relative latencies (ms).

	#1	#2	#3	#4	#5	#6	#7	#Imp
Thumbnailing	5	10	107	205	304	402	500	**290**
OCR	5	10	19	24	33	41	50	24
MatrixProduct	5	10	29	46	66	82	100	46
Translator	3	5	6	7	8	9	10	7

Table 6.3: Volume of requests for each of the four applications in the seven different configurations (req/s). Configuration "#Imp" represents an impulse in the popularity of the *Thumbnailing* application from configuration #4.

presented in Table 6.2. We place ten user locations distributed across the network coordinate space, with four LUGs per location, one for each application, yielding 40 LUGs in total. The P99 tail-latency SLOs are set to slo = 100 ms for Thumbnailing, slo = 400 ms for OCR, slo = 200 ms for MatrixProduct, and slo = 800 ms for Translator.

Baseline. We compare Scylla to a *local-first heuristic* baseline, which assigns each LUG to its closest edge site and only uses cloud resources when the closest edge site is insufficient. This heuristic mimics a scenario where application developers independently decide where to place their own application without global coordination, and serves as a baseline for the global optimization objectives of Scylla.

We evaluate seven workload configurations with progressively increasing request rates across all four applications, as summarized in Table 6.3. In addition, we evaluate an impulsive load configuration #Imp, derived from configuration #4, where the request rate of the Thumbnailing application increases by an additional 85 req/s from a single LUG, representing a sudden localized demand surge. We test Scylla under three optimization objectives: (1) minimizing the total number of slots, marked as *min_slots*; (2) minimizing the total deployment cost, marked as *min_cost*; (3) minimizing the number of ac-

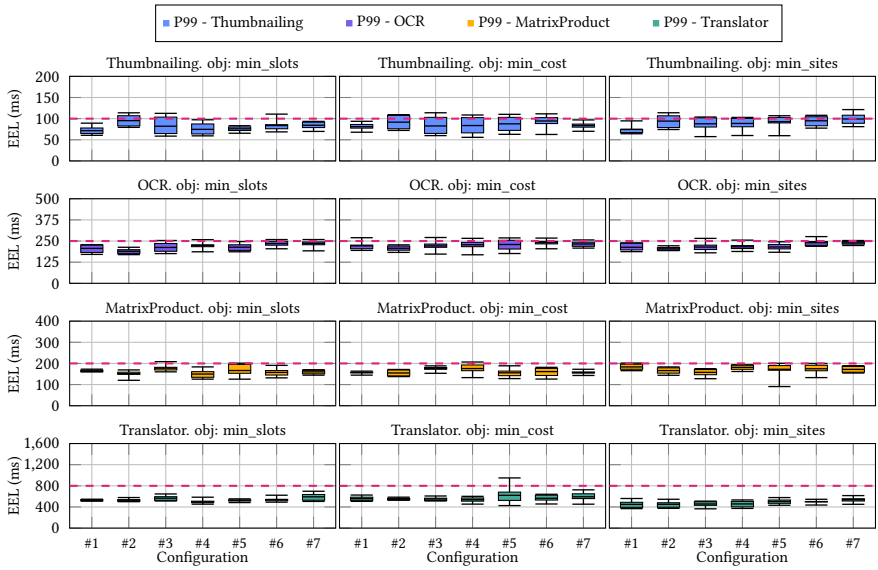


Figure 6.2: P99 latency distributions for the LUGs across the seen configurations and the impulsive configuration, for the four target applications and three global optimization objectives. The red dashed line in each plot indicates the application SLO set for its P99 latency. Box plots read as follows: the upper edge, lower edge, and the bar in the middle represent the maximum, minimum, and mean value for P99 latency of all LUGs.

tive sites, marked as *min_sites*.

Results Analysis. Figure 6.2 shows the empirical mean and P99 end-to-end latency for all applications and all LUGs across all seven configurations under all three optimization objectives. The placement plans computed by Scylla successfully keep the P99 latency below the SLO threshold for all applications in the vast majority of configurations, with a single exception for the Translator application under one configuration for the minimize cost objective, where the tail latency marginally exceeds the threshold.

Figure 6.3 shows the empirical mean and P99 end-to-end latency for all applications and all LUGs under the base configuration #4 and the impulsive configuration #Imp, across all three optimization objectives. As in the other configurations, the placement plans computed by Scylla successfully keep the P99 latency below the SLO threshold for all applications under the abrupt workload change introduced by #Imp, where the request rate of the Thumbnailing application increases sharply from a single LUG. This result confirms that Scylla can jointly satisfy the SLOs of multiple co-located applications across a complex multi-site infrastructure, including under localized demand bursts that are not uniformly distributed across user groups.

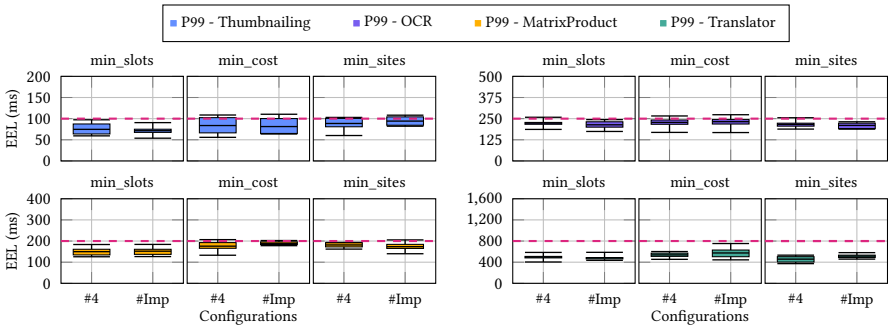


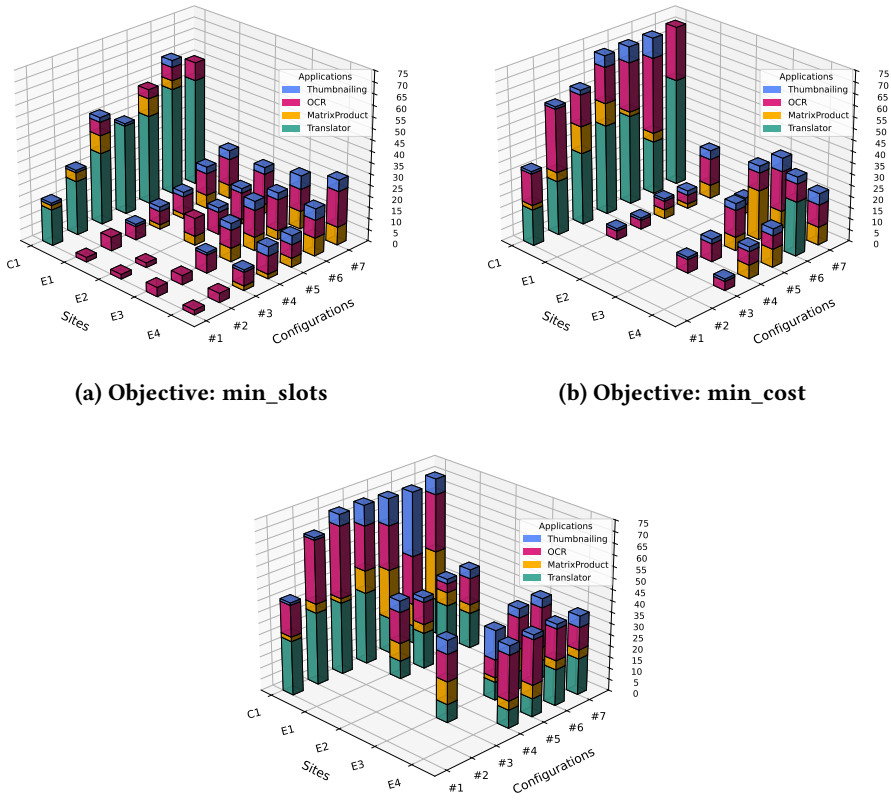
Figure 6.3: P99 latency distributions for the LUGs across the #4 and the impulsive configurations, for the four target applications and three global optimization objectives.

Figures 6.4a, 6.4b and 6.4c shows the distribution of deployed slots per site under the three optimization objectives, respectively. The three objectives produce qualitatively different placement strategies. When minimizing the total number of slots, Scylla deploys instances preferentially at edge sites, placing the Translator application at the cloud site to benefit from its greater resource availability while deploying lighter applications at the edge. When minimizing cost, Scylla shifts instances toward the cheaper cloud site, using edge sites increasingly as the load grows and cloud resources become insufficient to satisfy latency constraints alone. When minimizing the number of active sites, Scylla consolidates deployments onto fewer sites, reducing the number of active edge sites at the cost of higher total resource consumption and operational cost.

Table 6.4 reports the numerical results under the three optimization objectives and the local-first heuristic for two representative configurations #3 and #7. Each objective from Scylla achieves the best performance on its targeted metric, while the local-first heuristic consistently incurs higher resource consumption and cost across all configurations. These results demonstrate that Scylla's global optimization approach produces significantly better outcomes than independent per-application scheduling, both in terms of resource efficiency and SLO compliance.

6.5.2 Heuristic Algorithms Evaluation

This experiment evaluates the scalability and solution quality of H_{LPS} compared to G_{opt} and H_{Full}, assessing whether H_{LPS} provides a practical alternative to global optimization for larger-scale multi-application scenarios. Note that the evaluation of H_{MCS}, the Monte Carlo Sampling heuristic used in Cerberus, is deferred to Chapter 7, where it is evaluated in the context of the graceful



(a) Objective: min_slots

(b) Objective: min_cost

(c) Objective: min_sites

Figure 6.4: The placement plans generated by Scylla for all configuration in multi-site multi-app scenario with three optimization objectives.

degradation scheduling framework that builds upon it.

Setup. We use the same infrastructure parameters as in Section 6.5.1, with five sites and ten LUGs per application, and vary the number of applications from 1 to 10 under the minimize slots objective. Computation times are measured on an Intel CPU with 32 cores. G_{opt} uses Gurobi's multi-threading capability across all available cores, while H_{Full} and H_{LPS} explore permutations in 8 parallel processes, each with 4 Gurobi cores.

Results Analysis. Figure 6.5 shows the computation times on a log scale and the objective values for G_{opt}, H_{Full}, and H_{LPS} with depths $d = 1$ and $d = 2$, as a function of the number of pseudo applications. For all multi-application experiments presented in Section 6.5.1, H_{LPS} and G_{opt} return identical scheduling solutions, confirming that the heuristic achieves optimal solution quality for the scenario sizes used in the main evaluation.

Config	Objective	Used slots	Total cost	Used sites
#3	min_slots	**73**	105	4
	min_cost	77	**99**	4
	min_sites	107	143	**2**
	local-first	92	243	4
#7	min_slots	**152**	312	5
	min_cost	156	**276**	5
	min_sites	178	321	**4**
	local-first	160	368	5

Table 6.4: Global results obtained with the three optimization objectives and local-first heuristic as baseline in configuration #3 and #7 as examples.

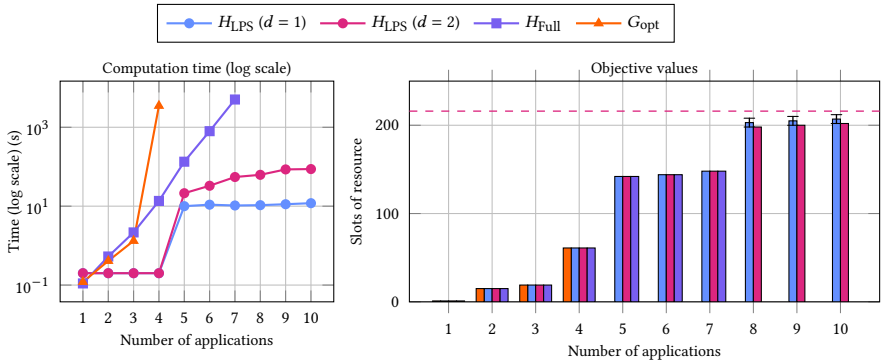


Figure 6.5: Computing time (log scale) for H_{LPS} with $d = 1$, $d = 2$, H_{Full} and G_{opt} under the objective min_slot (left), along with the objective values (right). Error bars in the right plot indicate the variation in objective values caused by the random ordering of the remaining applications beyond d in LPS at different depths.

The computation time results reveal three distinct scalability regimes. G_{opt} becomes computationally intractable beyond four applications due to the explosion of the joint search space, and its results are therefore omitted for larger numbers of applications. H_{Full} remains tractable up to six applications but suffers from the factorial growth in the number of permutations, becoming impractical beyond that point. H_{LPS} with $d = 1$ achieves computation times up to 500 times lower than H_{Full}, while producing indistinguishable objective values for up to seven applications. Beyond seven applications, when edge resources become scarce and scheduling decisions become tightly coupled, H_{LPS} with $d = 1$ exhibits an optimality gap of around 5% compared to $d = 2$, illustrating the inherent trade-off between runtime and solution quality as the depth parameter increases. These results confirm that H_{LPS} is a practical and scalable algorithm for periodic rescheduling in complex multi-

application edge-cloud scenarios.

6.6 Discussion

The experimental results presented in Section 6.5 demonstrate that Scylla accurately predicts tail latency for multiple co-located applications and computes globally optimized placement plans that satisfy P99 SLOs while minimizing system-wide objectives. This section discusses the key design choices and trade-offs underlying these results.

6.6.1 Heterogeneous Resource Requirements

A key extension of Scylla over Lassy is the explicit modeling of heterogeneous resource consumption across applications through the slot parameter σ_{app}. In Lassy, every instance consumed exactly one uniform slot, which reduced the capacity of a site to a simple bound on the number of instances it could host. That uniformity was not merely a simplification of notation: with a single application whose instances are interchangeable, there is nothing to pack, and site capacity enters the problem as a scalar limit rather than as a resource to be apportioned. Introducing σ_{app} is what turns placement into a packing problem, and it is the parameter through which the applications of a multi-tenant site genuinely compete with one another.

The consequences are visible throughout the experimental evaluation. The Translator application, requiring eight slots per instance against one for Thumbnailing, dominates the capacity constraint at most sites and acts as the binding resource consumer in high-load configurations. Its footprint also produces an asymmetry that a uniform model cannot express: residual capacity is not fungible across applications, since a site left with three free slots can still accommodate three Thumbnailing instances but not a single Translator instance. Placement decisions for resource-heavy applications must therefore be coordinated with those for lighter ones, both to avoid starving the latter and to avoid fragmenting the infrastructure into pockets too small for the former. This same asymmetry underlies the ordering sensitivity of the sequential heuristics of Section 6.4.3: scheduling a heavy application early secures the contiguous capacity it requires but may exhaust the sites best positioned for the lighter ones, while scheduling it late risks finding no site able to host it at all.

6.6.2 Limits Under Resource Scarcity

The scheduling framework presented in this chapter assumes that a feasible placement plan always exists, that is, that the available resources across all

sites are sufficient to simultaneously satisfy the tail-latency SLOs of all applications under the given workload. This assumption holds well under the workload conditions evaluated in this chapter, where the aggregate resource demand of all applications remains within the available capacity of the infrastructure. However, it does not hold in general, and resource scarcity represents a fundamental limitation of the current approach.

As the heuristic evaluation in Section 6.5.2 reveals, when the number of applications grows and edge resources become increasingly constrained, the feasible space of the scheduling problem shrinks and scheduling decisions become more tightly coupled. In such conditions, even H_{LPS} with increasing depth may fail to find a feasible solution, not because the heuristic is suboptimal, but because no feasible solution exists under the given resource budget. This situation is not exceptional in practice: edge sites are inherently resource-constrained and are expected to operate near capacity during peak demand periods, precisely when the demand for latency-sensitive services is highest.

When no feasible solution exists, the current framework provides no mechanism for graceful recovery. The only options available are to reject the scheduling request entirely, to evict one or more applications to free resources for others, or to accept SLO violations without any control over their magnitude or distribution across applications. As discussed in Section 2.4.3 of Chapter 2, all of these outcomes are unsatisfactory: outright rejection and eviction disrupt service continuity, while uncontrolled SLO violations leave application owners without any predictability or guarantee on the quality of service they can expect. This limitation motivates the extension of the scheduling framework with a principled mechanism for handling resource scarcity in a controlled and priority-aware manner, which is the central contribution of the following chapter.

6.7 Conclusion

This chapter has presented Scylla, which lifts the single-application assumption of Chapter 5 and schedules all co-located applications jointly. Two extensions distinguish it from Lassy: a per-application slot demand σ_{app}, which turns placement into a packing problem in which applications genuinely compete for capacity, and a shared capacity constraint that couples their scheduling decisions at every site. The premise is that scheduling applications one at a time forfeits any control over how they interact, and that only a joint formulation can prevent the contention and resource waste this produces in a multi-tenant edge environment. Because that formulation grows exponentially with the number of applications, we further introduced heuristics that decompose it into a sequence of single-application problems explored over

application orderings.

The evaluation on a real Kubernetes testbed with four applications confirmed both points. The placement plans computed by Scylla satisfy the P99 SLOs of all applications simultaneously across a wide range of workload conditions, optimization objectives, and demand patterns including localized bursts, while a local-first baseline that places each application independently consistently incurs higher resource consumption and cost. The H_{LPS} heuristic reduced computation times by up to a factor of 500 relative to exhaustive permutation search while returning the same solutions at the scenario sizes evaluated.

Scylla nonetheless assumes throughout that a feasible placement plan exists. When aggregate demand exceeds the available edge capacity, it offers no controlled recovery, returning either infeasibility or plans that violate SLOs arbitrarily. The following chapter addresses this by extending the framework with a graceful degradation mechanism that resolves scarcity in a predictable and priority-aware manner.

Scheduling with Graceful Degradation with Cerberus

7

The global scheduling framework developed in Chapter 6 targets the edge-cloud continuum, where cloud resources act as a virtually unbounded capacity buffer that can absorb excess demand when edge sites are insufficient. However, in infrastructure formed exclusively by edge sites, without access to cloud resources, this safety net disappears. Edge-only deployments are increasingly common in scenarios where data sovereignty, privacy constraints, or connectivity limitations prevent offloading to the cloud, such as industrial automation, healthcare monitoring, and tactical networks. In such settings, the fixed and limited resources of the edge sites must absorb the full demand from all co-located applications, and during peak demand periods the aggregate resource consumption may exceed the available capacity of the infrastructure. When this occurs, no feasible placement plan exists that simultaneously satisfies the tail-latency SLOs of all applications, and the scheduler must make a difficult decision about how to handle the resulting resource deficit. Existing approaches either evict low-criticality applications entirely or accept uncontrolled SLO violations, both of which are unsatisfactory outcomes that damage user experience and system stability. This chapter addresses this limitation by extending the scheduling framework of the previous Chapter with a principled graceful degradation mechanism that handles resource scarcity in a controlled, predictable, and priority-aware manner.

7.1 Introduction

Resource scarcity is an expected and recurring operating condition in edge-only environments, not an exceptional event. Unlike the edge-cloud continuum considered in Chapter 6, where cloud resources provide a capacity buffer for absorbing excess demand, edge-only infrastructures operate under hard resource constraints that cannot be relaxed by offloading to a more resourceful tier. Edge sites are deployed with fixed computational budgets, and the demand from latency-sensitive applications fluctuates unpredictably with user activity. During peak periods, the aggregate demand from all co-located appli-

cations may exceed the available capacity, making it impossible to satisfy all SLOs simultaneously. In such situations, a scheduler must decide which applications to prioritize and how to manage the performance of lower-priority ones in a way that is predictable and bounded, rather than arbitrary or disruptive.

Existing solutions are inadequate for this challenge. Eviction-based approaches free resources for high-priority applications by completely interrupting the service of low-priority ones, causing severe disruptions to user experience and triggering repeated reallocation cycles as demand fluctuates. Resource-agnostic approaches such as Kubernetes HPA simply scale applications reactively without any awareness of their relative criticality, leading to unpredictable performance degradation distributed arbitrarily across all applications. Classical mixed-criticality scheduling approaches from real-time systems prioritize high-criticality tasks but provide no guarantees or minimum performance thresholds for low-criticality ones, which are throttled or dropped entirely without bounds [92].

This chapter presents Cerberus, a scheduler that integrates the globally optimized scheduling framework of Chapter 6 with a novel graceful degradation mechanism tailored for resource-constrained edge environments. Rather than evicting applications or accepting uncontrolled violations, Cerberus degrades the SLOs of low-criticality applications in a controlled and predictable manner, minimizing the magnitude of degradation while preserving strict SLO compliance for high-criticality applications. Each application is assigned a priority level and a Service Level Threshold (SLT) that defines the maximum acceptable degradation of its SLO, bounding the extent to which its performance may be relaxed under resource pressure.

This chapter is organized as follows. Section 7.2 reviews related work on mixed-criticality scheduling and graceful degradation. Section 7.3 presents the extended system model and problem formulation with graceful degradation, and describes the Cerberus scheduler and its algorithms. Section 7.4 evaluates Cerberus on a real Kubernetes testbed. Section 7.5 discusses the results and the implications of the approach. Section 7.6 concludes the chapter.

7.2 Related Work

The general landscape of scheduling approaches for latency-sensitive applications and the works most directly related to single and multi-application scheduling in the edge-cloud continuum are covered in Sections 2.5, and 6.2 respectively. We focus here specifically on works addressing mixed-criticality scheduling, graceful degradation, and resource scarcity management, which are the core themes of this chapter.

Mixed-Criticality Scheduling. The notion of mixed-criticality scheduling

has been studied extensively in the context of real-time and embedded systems, where workloads with varying safety or priority levels must coexist on shared resources [27]. Classical mixed-criticality approaches ensure that high-criticality applications meet their SLOs even under overloaded conditions, typically by throttling or discarding low-criticality tasks [93]. More recent work extends mixed-criticality concepts to cloud and edge environments. Tango [21] proposes a joint optimization framework for mixed services in Kubernetes-based edge clouds, balancing resource allocation between latency-sensitive and best-effort applications. Amer et al. [22] propose a collaborative scheduling algorithm for prioritized tasks with shared resources in mobile edge and cloud computing systems. While these works introduce priority awareness into edge scheduling, they provide no minimum performance guarantees or bounded degradation thresholds for low-criticality applications, which are either throttled arbitrarily or dropped entirely when resources become scarce.

Graceful Degradation in Real-Time Systems. Graceful degradation strategies have been explored in real-time and safety-critical systems, where applications must maintain a minimum level of functionality even under resource pressure. Sun et al. [94] propose elastic scheduling for graceful degradation of mixed-criticality systems, allowing applications to switch between high and low criticality modes. Sundar and Easwaran [95] introduce a practical degradation model for mixed-criticality systems with bounded performance tradeoffs. Zou et al. [96] propose context-aware graceful degradation for mixed-criticality scheduling in autonomous systems. Li et al. [93] address mixed-criticality federated scheduling for parallel real-time tasks. While these works provide valuable insights into degradation mechanisms, they typically assume centralized schedulers with full system visibility, tightly bounded task execution times, and predictable resource availability. These assumptions are difficult to maintain in geographically distributed edge deployments with heterogeneous resources and dynamic workloads. Furthermore, none of these works are integrated with modern container orchestration platforms such as Kubernetes.

Graceful Degradation in Edge-Cloud Scenarios. Agrawal and Jyothi [23] propose cooperative graceful degradation in containerized clouds, where applications collaborate to absorb resource deficits without complete service interruption. While this work shares the motivation of avoiding abrupt eviction, it targets cloud environments with abundant resources and does not consider the geographical distribution of edge sites and users, and does not address the problem of keeping the tail latency of degraded applications within an acceptable predefined bound, which is central to Cerberus. To the best of our knowledge, Cerberus is the first system to integrate accurate tail-latency

prediction, globally optimized multi-application scheduling, and a principled graceful degradation mechanism in a unified framework validated on real edge infrastructure.

7.3 Scheduling With Graceful Degradation

The scheduling framework presented in Chapter 6 computes globally optimized placement plans that simultaneously satisfy the tail-latency SLOs of all co-located applications. As established in Section 7.1, this framework assumes that a feasible solution always exists, an assumption that breaks down under resource scarcity in edge-only deployments. This section extends the framework with a graceful degradation mechanism that handles resource scarcity in a controlled and priority-aware manner. We first introduce the concepts of application priority and Service Level Threshold that define the degradation space for each application, then describe the degradation mechanism itself, and finally present the extended problem formulation and the Cerberus scheduling algorithm that integrates degradation into the global optimization.

7.3.1 Application Priority and Service Level Threshold

When resources are insufficient to simultaneously satisfy the SLOs of all applications, the scheduler must decide which applications to protect and which to degrade. Cerberus addresses this by extending the application model with two new concepts: a priority level that captures the relative criticality of each application, and a Service Level Threshold that bounds the extent to which its performance may be relaxed. Together, these two parameters define the degradation space of each application and govern the order and magnitude of degradation applied under resource scarcity.

Table 7.1 extends the notation of Chapter 6 with the new parameters introduced by the graceful degradation mechanism. The degraded SLO slo_{app}^{degr}, the SLT slt_{app}, and the priority level $prio_{app}$ are all application-level parameters provided as inputs to the scheduler, while $\Lambda_{st,lug,app}^{degr}$ is a derived quantity computed during the preprocessing phase.

Priority Level. In a multi-tenant edge environment, not all applications are equally critical. A real-time safety monitoring system and a content recommendation service may share the same edge site, but the consequences of a performance degradation are fundamentally different for each. To capture this heterogeneity, Cerberus assigns each application a discrete *priority level* $prio_{app} \in \mathbb{N}$, where a lower value indicates higher criticality. Priority level 1 denotes the most critical applications, whose SLOs must be preserved at all costs, while higher priority values indicate progressively less critical applications whose performance may be relaxed under resource pressure. The

Name	Description
$st \in ST$	Edge sites
$app \in APP$	Applications
$nslots_{st}$	Slots of computing resource available at site st
$lug \in LUG$	Local user group(s)
$l_{lug,st}$	Round trip network latency between user group lug and site st
λ_{lug}	Request rate of user group lug
$lugapp_{lug}$	Application used by LUG lug
σ_{app}	Slots required by each instance of app
μ_{app}	Service rate of an application instance
c_{st}	Cost per slot at site st
slo_{app}	SLO: Maximum P99 latency for app
$\Lambda_{st,lug}$	Maximum arrival rate for the instances at site st can accept without breaking the SLO for user group lug
slo^{degr}_{app}	Degraded SLO: Degraded maximum P99 latency for app
slt_{app}	SLT: Maximum acceptable degraded P99 latency for app
$prio_{app}$	Priority level of app (Priority 1 = highest priority)
$ninsts_{st,app}$	Instances of app deployed on site st
$lugsite_{lug}$	Site that user group lug is directed to
$x_{st,lug}$	Whether user group lug is assigned to site st
$L_{st,app}$	Set of user groups sending requests for app to site st
$w_{st,app}$	Queueing delay at site st for app
r	Degradation steps

Table 7.1: Model parameters (top) and variables (bottom). Rows with a grey background are introduced in this chapter.

priority level is provided by the application owner as part of the application's deployment policy and reflects the operational importance of the service in its deployment context. How priorities are determined in practice depends on the operator: they may follow the service tier the application owner has paid for, the intrinsic criticality of the service, as for life-saving or safety monitoring applications. Our scheduler treats the priority as a given input and is agnostic to the policy that produced it.

Service Level Threshold. In addition to its SLO slo_{app}, which defines the target tail-latency bound under normal operating conditions, each application specifies a *Service Level Threshold* (SLT) $slt_{app} \geq slo_{app}$. The SLT represents the maximum acceptable degraded tail-latency bound that the application owner is willing to tolerate under resource scarcity. It is provided by the application owner as part of the application's deployment policy and serves as a hard upper bound on the extent of degradation: Cerberus guarantees that the tail-latency bound of any degraded application will never exceed its

SLT, ensuring that degradation remains predictable and bounded rather than arbitrary. The gap between slo_{app} and slt_{app} defines the *degradation space* of the application, within which Cerberus may relax the SLO constraint when resources are insufficient. Importantly, the SLT is specified independently for each application and need not correspond to a uniform fraction of its SLO, as the acceptable extent of degradation depends on the nature of the application itself. This per-application specification gives the scheduler the flexibility to accommodate applications with widely differing latency tolerances within the same deployment scenario.

Degradation Granularity. The degradation space of each application is divided into n discrete steps, where n is a predefined parameter of the deployment policy. At degradation step $r \in \{0, 1, \ldots, n\}$, the degraded SLO bound of application *app* is set to:

$$slo_{app}^{degr} = slo_{app} + (slt_{app} - slo_{app}) \cdot \frac{r}{n} \tag{7.1}$$

At step $r = 0$, the degraded SLO equals the original SLO, meaning no degradation is applied. At step $r = n$, the degraded SLO reaches the SLT, meaning the application is fully degraded to its maximum acceptable bound. This discrete stepping mechanism ensures that degradation is applied progressively and predictably, allowing the scheduler to find the minimum necessary degradation at each step before proceeding further.

7.3.2 Graceful Degradation Mechanism

When the scheduling problem as formulated in Chapter 6 has no feasible solution under the current resource constraints, Cerberus activates its graceful degradation mechanism. Rather than evicting applications or accepting uncontrolled SLO violations, the mechanism progressively relaxes the SLO constraints of low-priority applications in a controlled and priority-aware manner, until a feasible placement plan is found or all degradable applications have been fully degraded to their SLT bounds.

Priority-Ordered Degradation. The degradation mechanism operates in priority order, always targeting the least critical applications first. Applications at the lowest priority level are degraded before those at higher priority levels, and within a given priority level all applications are degraded simultaneously and symmetrically to ensure fairness. High-criticality applications at priority level 1 are never degraded: their original SLO constraints remain enforced throughout the process, and the mechanism guarantees their strict compliance regardless of the resource pressure experienced by the system.

Iterative Relaxation. The degradation process proceeds iteratively across discrete degradation steps. At each step r, the degraded SLO bounds of all

currently degradable applications are updated according to Equation 7.1, and the scheduler attempts to find a feasible placement plan under the updated constraints. If a feasible plan is found, the degradation process terminates and the plan is returned. If no feasible plan is found after all applications at the current priority level have been fully degraded to their SLT bounds, the degradation process is extended to the next higher priority level, which is then included in the set of degradable applications for subsequent steps. Already fully degraded applications remain fixed at their SLT values throughout the remainder of the process.

Termination and Guarantees. The degradation process terminates either when a feasible placement plan is found, or when all degradable applications have been fully degraded to their SLT bounds and the scheduling problem remains infeasible. In the latter case, Cerberus returns an empty solution, signaling that no placement plan exists even under maximum degradation. This situation indicates that the available edge resources are fundamentally insufficient to serve all applications even at their minimum acceptable quality level, and that a more drastic intervention such as load shedding or infrastructure expansion is required. Throughout the process, Cerberus provides two key guarantees: high-criticality applications always operate at their original SLO, and the degradation applied to low-criticality applications never exceeds their specified SLT, ensuring that all performance trade-offs remain within the bounds agreed upon by application owners.

7.3.3 Problem Formulation with Degradation

The scheduling problem with graceful degradation extends the multi-application MILP formulated in Chapter 6 by replacing the fixed SLO constraints with degradable ones. All decision variables, capacity constraints, and optimization objectives defined in Section 6.4.1 remain unchanged. The only modification concerns the SLO constraint of Equation 6.5, which is parameterized by the degraded SLO bound slo_{app}^{degr} rather than the original slo_{app}.

Degraded SLO Constraint. At degradation step r, the SLO constraint for application app is relaxed to:

$$P\left(w_{lugsite_{lug},app} \leq slo_{app}^{degr} - l_{lug,lugsite_{lug}} - \frac{1}{\mu_{app}}\right) \geq \theta\% \quad \forall lug \in LUG_{app} \tag{7.2}$$

where slo_{app}^{degr} is defined by Equation 7.1. For high-criticality applications that are not subject to degradation, $slo_{app}^{degr} = slo_{app}$ at all steps, preserving their original SLO constraint. For degradable applications, slo_{app}^{degr} increases progressively with r, relaxing the latency budget available to the queueing delay

and allowing the scheduler to provision fewer instances while still satisfying the relaxed constraint.

Linearization. As in Chapter 6, the nonlinear SLO constraint is linearized by pre-computing the maximum admissible arrival rate $\Lambda^{degr}_{st,lug,app}$ for each feasible site-user group-application triplet under the degraded SLO bound:

$$P\left(w_{st,app} \leq slo^{degr}_{app} - l_{lug,st} - \frac{1}{\mu_{app}}\right)\Bigg|_{\lambda^{inst}_{st,app}=\Lambda^{degr}_{st,lug,app}} = \theta\% \tag{7.3}$$

Since $slo^{degr}_{app} \geq slo_{app}$, the degraded latency budget is larger than the original, meaning $\Lambda^{degr}_{st,lug,app} \geq \Lambda_{st,lug,app}$. A relaxed SLO therefore allows each instance to absorb a higher arrival rate before violating the constraint, directly reducing the number of instances required to serve a given workload. This is the fundamental mechanism through which graceful degradation frees resources for high-criticality applications.

Notation Extension. Table 7.1 extends the notation of Chapter 6 with the new parameters introduced by the graceful degradation mechanism. The degraded SLO slo^{degr}_{app}, the SLT slt_{app}, and the priority level $prio_{app}$ are all application-level parameters as inputs to the scheduler, while $\Lambda^{degr}_{st,lug,app}$ is a derived quantity computed during the preprocessing phase.

7.3.4 Cerberus Scheduling Algorithms

The Cerberus scheduling algorithm combines a new heuristic algorithm, which addresses the scalability limitation from a different angle, with the graceful degradation mechanism described in Section 7.3.2.

Monte Carlo Sampling Heuristic. Apart from Layered Permutation Search heuristic H_{LPS} presented in Section 6.4, we propose the Monte Carlo Sampling heuristic H_{MCS}. Inspired by Monte Carlo Sampling methods [97], H_{MCS} introduces a sampling parameter $p \in (0, 1]$ that directly controls the fraction of all $|APP|!$ possible orderings to explore, randomly sampling a subset of permutations rather than enumerating them in a structured manner. This approach provides direct control over the trade-off between solution quality and computation time without the overhead of prefix enumeration.

As shown in Algorithm 2, H_{MCS} first generates all permutations of APP (line 4) and samples $\lfloor p \times |\Pi| \rfloor$ of them uniformly at random (line 5). For each sampled permutation π, applications are scheduled sequentially using G_{opt} on the remaining available resources (lines 9–15): if scheduling an application is feasible, the solution and available resources are updated; otherwise the permutation is abandoned. If the cost of the resulting solution $\mathcal{S}'$ is lower than the best found so far (line 16), it is retained as the new best (lines 17–18).

Algorithm 2 H_{MCS}: Monte Carlo Sampling Heuristic

```
1: p ∈ (0, 1]                                          ▷ percentage of permutations to explore
2: S ← ∅                                               ▷ optimal solution among all permutations
3: C ← ∞                                               ▷ cost of optimal solution
4: Π ← permutations of order(APP)
5: Π_sampled ← select ⌊p × |Π|⌋ rand. permutations from Π
6: for π ∈ Π_sample do
7:     available_resources ← nslots_ST
8:     S' ← ∅                                          ▷ solution for permutation π
9:     for app in π do
10:        Run G_opt(app, available_resources)
11:        if scheduling is feasible then
12:            Update S' with solution found by G_opt for app
13:            Update available_resources
14:        else
15:            Break and try next permutation
16:    if cost of S' < C then
17:        S ← S'
18:        C ← cost of S'
19: return S
```

At $p = 1$, H_{MCS} is equivalent to H_{Full}, while smaller values of p trade solution quality for reduced computation time by exploring only a random subset of orderings.

Graceful Degradation Algorithm. Under normal operating conditions, when a feasible placement plan exists without any degradation, Cerberus simply invokes H_{MCS} with the original SLO constraints and returns the resulting plan. When no feasible plan exists, Cerberus activates the graceful degradation algorithm D_{opt}, shown in Algorithm 3, which iteratively relaxes the SLO constraints of low-priority applications until a feasible plan is found or all degradable applications have been fully relaxed to their SLT bounds.

D_{opt} takes the number of degradation steps n as a parameter and maintains three state variables: the best solution found $\mathcal{S}_{\text{deg}}$, the set of fully degraded applications $\mathcal{F}$, and the index i_{pri} tracking the current highest priority level subject to degradation. The priority levels are sorted in ascending order from lowest to highest criticality, and the algorithm begins by initializing the set of degradable applications $\mathcal{A}$ to those at the lowest priority level (line 6).

The outer loop iterates over all priority-levels, starting from the lowest $i_{pri} = 1$ to the second highest, assuming that the applications in the highest priority level are not degradable (line 6). The middle loop iterates over degradation steps $r = 0, 1, \ldots, n$ (line 7). At each step, the inner loop processes the current set of degradable applications $\mathcal{A}$ (line 9). For each application $app \in \mathcal{A}$ that has not yet been fully degraded, the degraded SLO bound is updated according to Equation 7.1 (lines 10–11). Applications that have al-

Algorithm 3 Graceful Degradation D_{opt}

1: Parameter: n ▷ Number of degradation steps
2: $\mathcal{S}_{\text{deg}} \leftarrow \emptyset$ ▷ Optimal solution with degradation
3: $\mathcal{F} \leftarrow \emptyset$ ▷ Fully degraded applications
4: $\mathcal{P} \leftarrow$ priority levels sorted in ascending order (lowest → highest)
5: $\mathcal{A} \leftarrow \{app \in APP \mid prio_{app} = \mathcal{P}[1]\}$ ▷ Lowest-priority applications
6: **for** $i_{pri} = 1, \ldots, |\mathcal{P}| - 1$ **do**
7: **for** $r = 0, 1, \ldots, n$ **do**
8: **while** $\mathcal{A} \neq APP$ **do**
9: **for** $app \in \mathcal{A}$ **do**
10: **if** $app \notin \mathcal{F}$ **then**
11: $slodegr_{app} \leftarrow slo_{app} + (slt_{app} - slo_{app}) \cdot \frac{r}{n}$
12: **else**
13: $slodegr_{app} \leftarrow slt_{app}$
14: **if** $r = n$ **then**
15: $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{A}$ ▷ Application fully degraded
16: $\mathcal{S}_{\text{deg}} \leftarrow H_{\text{MCS}}(APP \text{ with } slodegr)$ ▷ Search for feasible solution with current degradations
17: **if** Feasible solution exists **then**
18: **return** $\mathcal{S}_{\text{deg}}$
19: **if** $(\forall app \in \mathcal{A}, app \in \mathcal{F})$ **then**
20: $\mathcal{A} \leftarrow \{app \in APP \mid prio_{app} = \mathcal{P}[1..i_{pri}]\}$ ▷ Include applications in next priority level for degradation
21: **return** $\emptyset$ ▷ No feasible solution found after full degradation of all degradable applications

ready reached their SLT are fixed at that value (lines 12–13). When $r = n$, all applications in $\mathcal{A}$ are marked as fully degraded by adding them to $\mathcal{F}$ (lines 14–15).

After updating the degraded SLO bounds, D_{opt} invokes H_{MCS} with the updated constraints to attempt to find a feasible placement plan (line 16). If a feasible plan is found, it is immediately returned (lines 17–18). Otherwise, if all applications at the current priority level have been fully degraded and higher priority levels remain, the degradation is extended to the next priority level by incrementing i_{pri} and expanding $\mathcal{A}$ accordingly (lines 19–20). If no feasible plan is found after all degradable applications have been fully relaxed to their SLT bounds, D_{opt} returns an empty solution (line 21), signaling that no placement plan exists even under maximum degradation.

7.4 Experimental Evaluation

The experimental setup, applications, workload generation, and baselines are those described in Chapter 4, and the validation of the $M|D|1$ latency model as well as the single-site instance count evaluation of Sections 4.4.1 and 4.4.2 apply unchanged to this chapter. We focus on the aspects that are specific

	Priority	#1	#2	#3	#4	#5	#6	#7
Thumbnailing	High	60	80	100	120	140	160	180
OCR	Medium	4	4	4	4	4	4	4
MatrixProduct	Low	20	20	20	20	20	20	20

Table 7.2: Requests per second for each of the four applications in the seven different configurations.

to Cerberus: the effectiveness of the graceful degradation mechanism under resource scarcity, first in a single-site and then in a realistic multi-site edge-only setting, and the scalability and solution quality of the H_{MCS} heuristic.

7.4.1 Single-Site: Instance Count Evaluation with Degradation

This experiment builds on the single-site instance count evaluation of Section 4.4.2, extending it to a resource-constrained setting with multiple applications of different criticality levels. It evaluates the effectiveness of Cerberus's graceful degradation mechanism under resource scarcity, demonstrating that Cerberus correctly applies degradation in priority order while preserving strict SLO compliance for high-criticality applications.

Setup. We deploy three applications at a single site with a capacity of 10 slots, each served by a single LUG with zero network latency. The three applications are assigned distinct priority levels: Thumbnailing at the highest priority, OCR at medium priority, and MatrixProduct at the lowest priority. The SLO and SLT parameters are set as follows: $slo = slt = 100$ ms for Thumbnailing, meaning no degradation is tolerated; $slo = 250$ ms and $slt = 500$ ms for OCR; and $slo = 200$ ms and $slt = 400$ ms for MatrixProduct. The number of degradation steps is set to $n = 5$. We start from a non-degraded configuration and progressively increase the request rate to Thumbnailing to force the total resource demand to exceed the site capacity, triggering the graceful degradation mechanism, as defined in Table 7.2.

Results Analysis. Figure 7.1 shows the number of slots allocated to each application by Cerberus and the empirical and predicted P99 end-to-end latency for each application across all configurations. In the initial configuration, all slots are in use and all applications operate within their original SLO bounds. As the request rate to Thumbnailing increases and the total resource demand exceeds the site capacity, Cerberus activates its degradation mechanism. In accordance with the priority ordering, Cerberus first degrades MatrixProduct, the lowest-priority application, freeing slots for Thumbnailing. As the demand continues to grow, Cerberus subsequently degrades OCR to release additional resources. Throughout the process, Thumbnailing, the highest-priority application, continues to operate strictly within its original

SLO of 100 ms without any degradation. The degraded P99 latencies of MatrixProduct and OCR remain within their respective SLT bounds at all times, confirming that the degradation applied by Cerberus is controlled, bounded, and predictable. These results validate the graceful degradation mechanism and confirm that Cerberus correctly enforces the priority ordering and SLT constraints under resource scarcity.

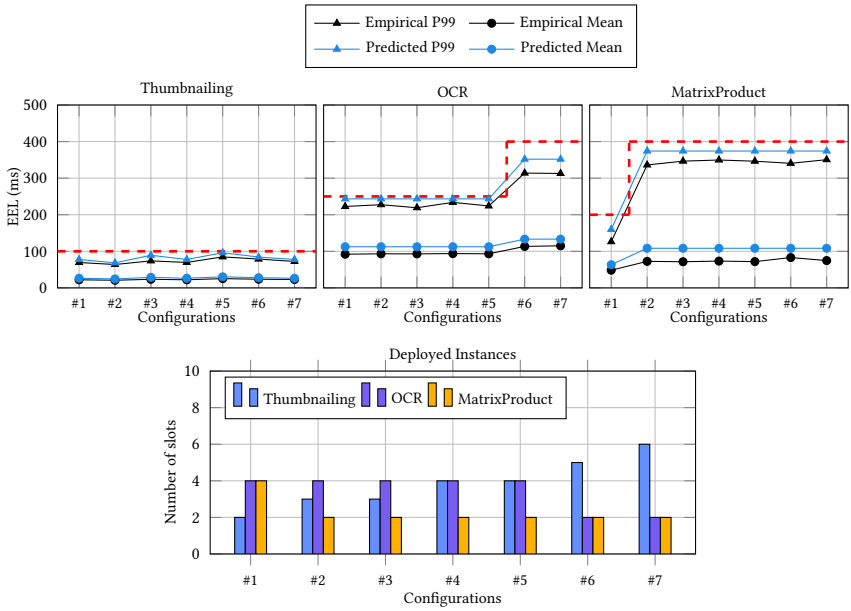


Figure 7.1: Empirical and predicted P99 end-to-end latencies (EEL) for a single site scenario with fully loaded sites and graceful degradation using Cerberus to control the number of instances deployed for the *Thumbnailing*, *OCR* and *MatrixProduct* applications. Black curves indicate the empirical results for the P99 latency for each application while blue curves indicate predictions made by Cerberus. The dashed red line indicates the application's SLO or SLT for P99 latency.

7.4.2 Multi-Site Realistic Placement Evaluation

This experiment evaluates Cerberus in a realistic multi-site edge-only scenario with multiple applications and LUGs, assessing its ability to compute globally optimized placement plans that satisfy the tail-latency SLOs of all applications simultaneously under varying workload conditions, and to apply graceful degradation in a controlled and priority-aware manner when resources become insufficient.

Setup. The infrastructure consists of five edge sites (E1–E5), each with a capacity of 30 slots, with no cloud site, reflecting the edge-only deployment

Site	Slots	Price	Network latency from LUG to site (ms)									
			LUGs 1-4	LUGs 5-8	LUGs 9-12	LUGs 13-16	LUGs 17-20	LUGs 21-24	LUGs 25-28	LUGs 29-32	LUGs 33-36	LUGs 37-40
E1	30	3$	45	46	13	59	53	37	54	80	61	64
E2	30	4$	79	10	63	59	53	67	24	30	111	114
E3	30	3$	95	46	63	109	103	17	74	80	61	64
E4	30	2$	29	60	37	9	17	87	54	80	79	64
E5	30	2$	45	96	37	59	67	37	104	130	29	14

Table 7.3: Resource, pricing, and latencies for sites E1-E5.

setting targeted by Cerberus. Resource, pricing, and network latency configurations are summarized in Table 7.3. We place ten user locations with diverse latency profiles, each hosting four LUGs for one LUG per application, yielding 40 LUGs in total. The SLOs and SLTs are set as follows: $slo = slt = 100$ ms for Thumbnailing, meaning no degradation is tolerated; $slo = 250$ ms and $slt = 350$ ms for OCR; $slo = 200$ ms and $slt = 400$ ms for MatrixProduct; and $slo = 800$ ms and $slt = 1600$ ms for Translator. Compared to the single-site experiment, the *slt* of OCR is tightened from 500 ms to 350 ms, narrowing its degradation space so that it becomes fully degraded within the range of load configurations considered here, making the progression of degradation across priority levels observable in a single experiment. The priority ordering assigns Thumbnailing the highest priority, followed by OCR and MatrixProduct at medium priority, and Translator at the lowest priority. The number of degradation steps is set to $n = 5$.

Baseline. We introduced a *priority-aware eviction* policy, which serves as the direct foil for the graceful degradation mechanism of Cerberus. Under resource scarcity, this policy evicts applications in order of increasing criticality, one at a time, until a feasible placement is found for the surviving applications. Evicted applications become fully unavailable for the duration of the scheduling period.

We evaluate seven workload configurations with progressively increasing request rates, as summarized in Table 7.4. Configuration #7 includes a sharp demand burst where the request rate to Thumbnailing increases by an additional 100 req/s from a single LUG, emulating a localized surge in user activity. We test Cerberus under two optimization objectives: minimizing the total number of slots and minimizing the total operational cost.

Results Analysis. Figure 7.2 shows the distribution of observed P99 latencies for all applications across all configurations and both optimization objectives. In configurations #1 and #2, no degradation is necessary as the resource demand from all applications remains within the available capacity, and Cer-

	#1	#2	#3	#4	#5	#6	#7
Thumbnailing	205	304	402	452	500	655	**1055**
OCR	13	24	33	41	46	50	76
MatrixProduct	46	66	82	90	100	130	146
Translator	7	8	9	9	10	10	10

Table 7.4: Total request rates for each of the four applications in the seven different configurations. From #6 to #7 there is a burst in the popularity of the *Thumbnailing* application.

berus computes placement plans that satisfy the original SLO bounds for all applications. As the load increases and edge sites become fully utilized, Cerberus begins to introduce graceful degradation from configuration #3, targeting Translator as the lowest-priority application first. As Translator reaches its SLT and demand continues to grow, Cerberus progressively extends degradation to MatrixProduct and OCR in configurations #5 to #7, freeing resources to maintain strict SLO compliance for Thumbnailing throughout all configurations.

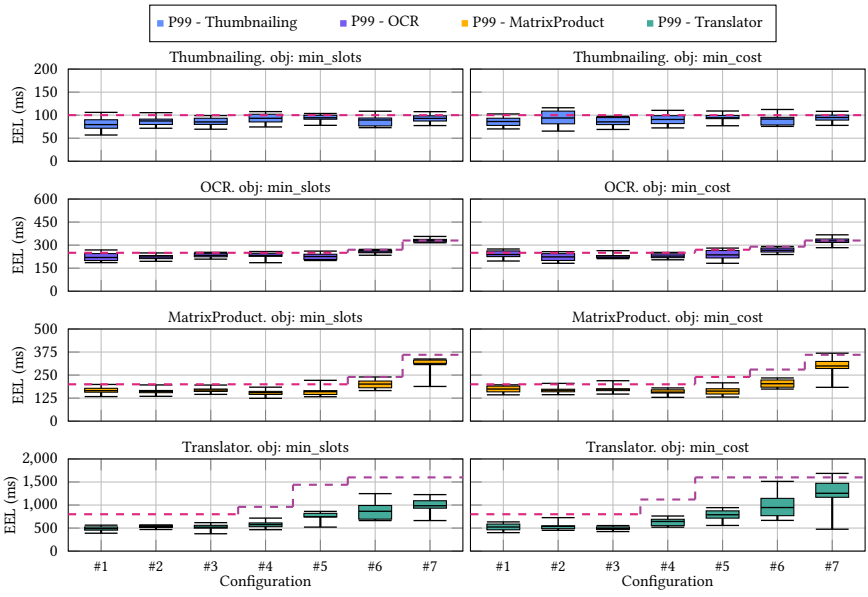
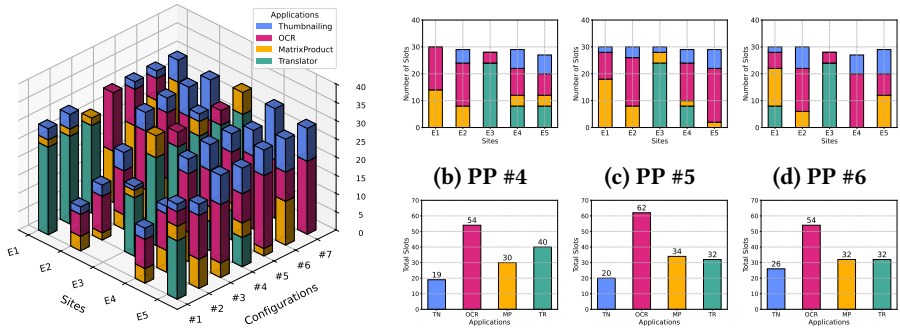


Figure 7.2: P99 latency distributions for the LUGs across the seen configurations and the impulsive configuration, for the four target applications and two global optimization objectives. The red dashed line in each plot indicates the application SLO or SLT set for its P99 latency. Box plots read as follows: the upper edge, lower edge, and the bar in the middle represent the maximum, minimum, and mean value for P99 latency of all LUGs.

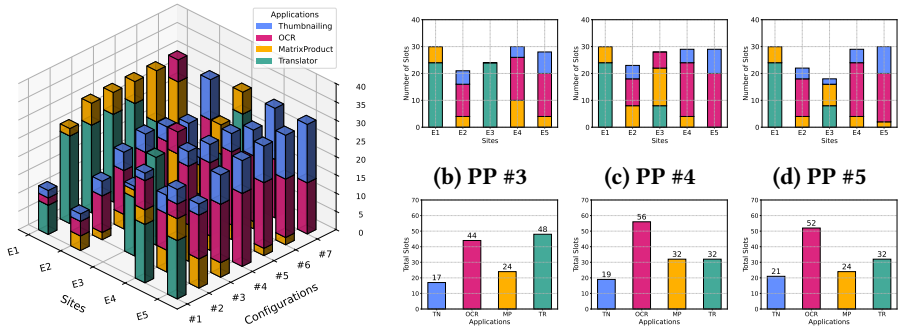
The placement plans computed by Cerberus successfully keep the mean of the observed P99 latency below the SLO or SLT threshold for all applications in almost all configurations, with only minor violations in the maximum of the P99 distribution in certain configurations. Figures 7.3 and 7.4 show the placement plans under the two optimization objectives for representative configurations, illustrating how Cerberus redistributes slots across sites and applications as degradation progresses. Under the minimize cost objective, Cerberus avoids deploying instances at the most expensive site E2 as much as possible, preferring cheaper sites when their network latency to the relevant LUGs remains within the SLO or SLT budget. These results confirm that Cerberus correctly enforces the priority ordering and SLT bounds across a complex multi-site, multi-application scenario, including under the localized demand burst of configuration #7, and that the graceful degradation mechanism produces predictable and bounded performance trade-offs across all applications.



(a) Obj: min_slots (b) PP #4 (c) PP #5 (d) PP #6 (e) Slots #4 (f) Slots #5 (g) Slots #6

Figure 7.3: Placement plans (PP) generated by Cerberus for multi-site multi-application scenarios under objective min_slots. Plot (a) shows the overall placement plan for all configurations. Plots (b)–(d) detail the placement plans for configuration #4 to #6, while plots (e)–(g) show the corresponding resource slot allocation per application.

Table 7.5 compares Cerberus against the eviction baseline under resource scarcity in configurations #5 and #7, respectively representing moderate and severe scarcity. In configuration #5, the eviction policy drops the lowest-priority application Translator to free sufficient capacity, leaving its users without service, whereas Cerberus retains all four applications by degrading Translator within its SLT bound. In configuration #7, where scarcity is more severe, eviction is forced to drop both Translator and MatrixProduct entirely, while Cerberus keeps all applications running, degrading OCR, MatrixProduct, and Translator within their respective SLT bounds while preserving full SLO compliance for Thumbnailing. These results confirm that



(a) Obj: min_cost (b) PP #3 (c) PP #4 (d) PP #5 (e) Slots #3 (f) Slots #4 (g) Slots #5

Figure 7.4: Placement plans (PP) generated by Cerberus for multi-site multi-application scenarios under objective min_cost. Same layout as Figure 7.3, but with configurations #3 to #5 to better illustrate the degradation among applications.

Config	Strategy	Thumbnailing	OCR	MatrixProduct	Translator
#5	Eviction	✓	✓	✓	**Evicted**
	Cerberus	✓	✓	✓	✓ (SLT)
#7	Eviction	✓	✓	**Evicted**	**Evicted**
	Cerberus	✓	✓ (SLT)	✓ (SLT)	✓ (SLT)

Table 7.5: Comparison of Cerberus and eviction-based scheduling under resource scarcity. ✓ indicates SLO compliance, ✓ (SLT) graceful degradation within the SLT bound, and Evicted service unavailability.

graceful degradation provides a strictly superior availability guarantee under resource scarcity compared to eviction-based strategies. This evaluation demonstrates that different global optimization objectives can guide Cerberus to select different SLO-compliant solutions, thereby answering RQ4.

7.4.3 Heuristic Algorithm Evaluation

This experiment evaluates the scalability and solution quality of the H_{MCS} heuristic compared to G_{opt}, assessing the impact of the sampling parameter p on both computation time and solution quality under ample and scarce resource conditions.

Setup. We use the same infrastructure parameters as in Section 7.4.2, with five sites and 40 LUGs, and vary the number of applications from 1 to 6 under both the minimize slots and minimize cost objectives. Computation times are measured on an Intel CPU with 32 cores. G_{opt} uses Gurobi's multi-threading capability across all available cores with a timeout of 1000 seconds, while

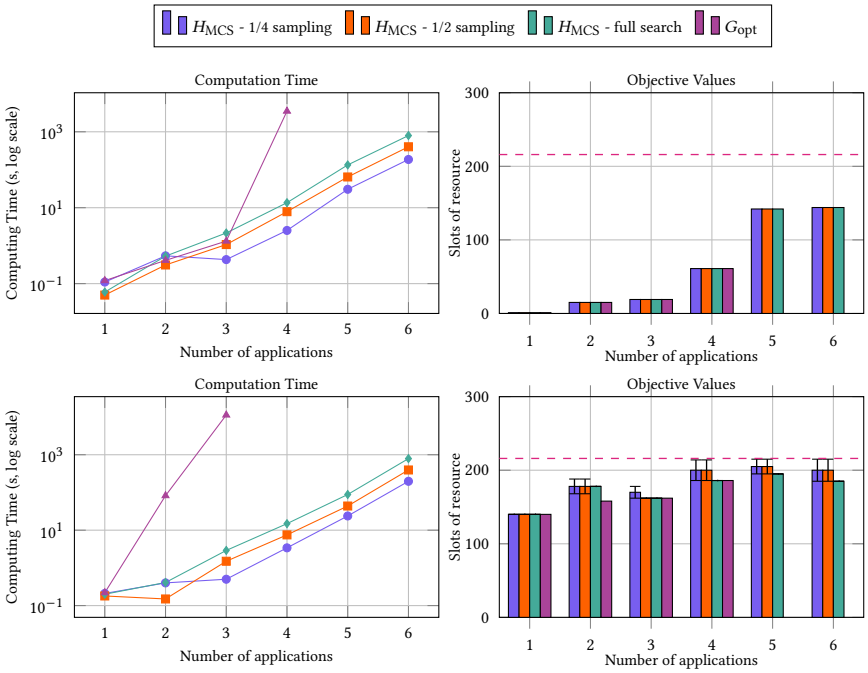
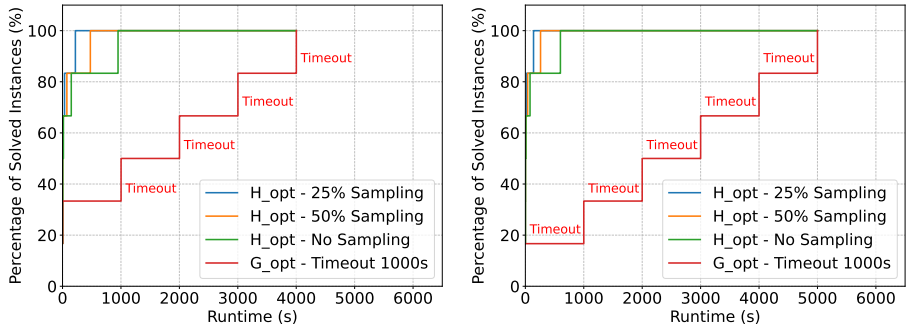


Figure 7.5: Computing time (log scale) for H_{MCS} and G_{opt} (left) under objective min_slots and results (right) with ample (upper) and scarced resource (bottom).

H_{MCS} explores permutations in 8 parallel processes with 4 Gurobi cores each. We evaluate H_{MCS} under three sampling ratios: $p = 1/4$, $p = 1/2$, and $p = 1$ (full search).

Results Analysis. Figure 7.5 shows the computation times and objective values for G_{opt} and H_{MCS} with different sampling ratios, under both ample and scarce resource conditions. The results reveal that G_{opt} becomes computationally intractable very quickly in the presence of graceful degradation: its computation time becomes prohibitive beyond two applications under ample resources and beyond a single application under scarce resources, where the tighter coupling of scheduling decisions significantly expands the search space. In contrast, H_{MCS} achieves orders-of-magnitude reductions in computation time across all scenario sizes, with computation time growing linearly with the number of sampled permutations rather than exponentially.

Figure 7.6 shows Cactus plots of the computation times for both optimization objectives when solving all six scheduling problems sequentially. G_{opt} hits the 1000-second timeout after two and one applications under the minimize slots and minimize cost objectives respectively, while H_{MCS} completes all six problems within practical time limits regardless of the sampling ratio.



(a) Objective min_slots **(b) Objective min_cost**

Figure 7.6: Computation runtime and objective value of H_{MCS} with different sampling ratios compared to G_{opt} with 1000 second timeout under two optimization objectives.

The sampling ratio has a negligible impact on solution quality under ample resources, where all three sampling ratios produce solutions indistinguishable from G_{opt}. The bottom-right plot of Figure 7.5 shows that under scarce resources, lower sampling ratios introduce an optimality gap of up to 16% with $p = 1/4$ compared to $p = 1$, reflecting the increased importance of exploring a broader set of permutations when the feasible space is tighter and application ordering becomes more critical. These results confirm that H_{MCS} with full sampling is the recommended configuration for scenarios involving graceful degradation, providing practical computation times while preserving near-optimal solution quality even under resource scarcity.

7.5 Discussion

The experimental results presented in Section 7.4 demonstrate that Cerberus accurately predicts tail latency, computes globally optimized placement plans that satisfy P99 SLOs under normal operating conditions, and applies graceful degradation in a controlled and priority-aware manner under resource scarcity. This section discusses the key design choices and trade-offs underlying these results, and identifies the limitations that motivate the research presented in the following chapter.

7.5.1 Degradation Ordering Rationale

The priority-ordered degradation strategy of Cerberus, which targets the applications in the lowest priority first and progresses toward higher-priority ones only when necessary, is a deliberate design choice that reflects a fundamental principle: degradation should be minimized in both scope and magni-

tude. An alternative approach would be to degrade all applications simultaneously and proportionally, distributing the resource shortfall evenly across all co-located services. While such an approach may appear fairer in a symmetric sense, it has two significant drawbacks. First, it exposes high-criticality applications to performance degradation even when sufficient resources could be freed by degrading only low-criticality ones, unnecessarily compromising the quality of the most important services. Second, it provides application owners with weaker predictability guarantees: under proportional degradation, the performance of any application may be affected by the behavior of any other, making it difficult to reason about worst-case performance bounds.

The priority-ordered approach avoids both of these issues. By degrading the least critical applications first, Cerberus minimizes the number of applications affected at any given resource pressure level and ensures that high-criticality applications operate at their original SLO for as long as possible. Within a given priority level, all applications are degraded simultaneously and symmetrically, ensuring fairness among applications of equal criticality. This design gives application owners a clear and predictable guarantee: their application will only be degraded when all lower-priority applications have been fully degraded to their SLT bounds, and the degradation applied will never exceed the SLT they specified.

7.5.2 The SLT as a Performance Contract

The Service Level Threshold is a central concept in Cerberus that distinguishes it from prior mixed-criticality scheduling approaches. Rather than providing binary guarantees — either the SLO is met or the application is evicted — the SLT establishes a continuous performance contract between the application owner and the infrastructure operator, defining the range of acceptable performance under resource pressure. This contract-based model has several practical implications.

From the application owner's perspective, the SLT provides a meaningful bound on the worst-case performance they can expect under peak demand conditions. Unlike eviction-based approaches where service interruption can occur without warning or bound, Cerberus guarantees that the tail-latency bound of any degraded application will remain within the SLT at all times. This predictability is particularly valuable for applications that must maintain a minimum level of service quality even under adverse conditions, such as augmented reality services that can tolerate reduced frame rates but must never completely stop responding.

From the infrastructure operator's perspective, the SLT provides a quantitative handle for capacity planning and resource allocation: knowing the SLT of each application allows the operator to reason about the minimum re-

sources required to keep all applications operational and to make informed decisions about infrastructure sizing. In practice, SLT values would be determined through negotiation between application owners and infrastructure operators, potentially informed by empirical measurements of application behavior under degraded conditions or by service-level agreements with end users.

7.5.3 Heuristic Behavior Under Scarcity

The evaluation of Section 7.4.3 shows that H_{MCS} matches G_{opt} when resources are ample, but that reduced sampling ratios incur an optimality gap of up to 16% under scarcity, for the reasons discussed in Chapter 9. The practical implication for Cerberus is that the sampling ratio should be set to $p = 1$ wherever resource scarcity is anticipated, trading computation time for solution quality. Full sampling remained practical for every scenario evaluated in this thesis, completing all scheduling problems well within the time limits required for periodic rescheduling. For larger scenarios where full sampling becomes prohibitive, a bound on the gap can still be obtained without solving the problem to optimality: running G_{opt} under a short time limit yields a valid dual bound on the optimum, against which the heuristic's objective value can be compared. Increasing the sampling ratio until this bound is met to within an acceptable margin provides a principled stopping criterion.

7.6 Conclusion

This chapter has presented Cerberus, a scheduler that extends the global multi-application scheduling framework of Chapter 6 with a principled graceful degradation mechanism for resource-constrained edge-only environments. While Scylla assumes that cloud resources are always available to absorb excess demand, Cerberus targets deployments where the edge infrastructure must operate self-sufficiently under hard capacity constraints, and where resource scarcity during peak demand periods is an expected and recurring condition rather than an exceptional event.

The key contribution of Cerberus is the introduction of application priority levels and Service Level Thresholds as first-class scheduling parameters, enabling the scheduler to make principled and predictable trade-offs between the performance of co-located applications under resource pressure. Rather than evicting applications or accepting uncontrolled SLO violations, Cerberus degrades the SLO constraints of low-priority applications in a controlled and progressive manner, minimizing the magnitude of degradation while preserving strict SLO compliance for high-criticality applications. The degradation algorithm D_{opt} integrates seamlessly with the H_{MCS} heuristic from Sec-

tion 6.4.3, extending it to handle resource scarcity without requiring modifications to the underlying scheduling model or optimization problem.

The experimental evaluation on a real Kubernetes testbed with five edge sites and four representative applications demonstrated three main results. First, Cerberus performs as well as Scylla under normal operating conditions, confirming that the graceful degradation mechanism does not introduce overhead or compromise scheduling quality when resources are sufficient. Second, the graceful degradation mechanism correctly enforces the priority ordering and SLT bounds across a wide range of workload conditions, including localized demand bursts, applying degradation only when necessary and only to the least critical applications. Third, the H_{MCS} heuristic with full sampling provides practical computation times for all scenario sizes evaluated, though its optimality gap under scarce resources motivates the use of full sampling in resource-constrained deployments.

Despite these contributions, Cerberus degrades application performance exclusively at the level of SLO constraints, without modifying the underlying execution of the application. For AI inference workloads that offer multiple model variants with different resource and accuracy trade-offs, a more expressive and resource-efficient form of degradation is available, one that reduces the computational footprint of the application itself rather than merely relaxing its latency bound. This observation motivates the final contribution of this thesis, presented in the following chapter.

Scheduling with Application-Specific Degradation with Occamy

8

The graceful degradation mechanism introduced in Chapter 7 relaxes the tail-latency SLOs of low-priority applications when resources become scarce, allowing their instances to operate at higher utilization and freeing capacity for higher-priority ones. As discussed in Section 7.5, this form of degradation is effective for general latency-sensitive applications, but it does not change the computation performed by the application: a degraded instance still executes the same code and consumes the same resources per request, with the savings coming entirely from tolerating higher queueing delays. For AI inference applications, a more expressive and resource-efficient form of degradation is available. This chapter exploits the internal adaptability of edge AI workloads, where the same inference task can be served by multiple deep neural network model variants with different accuracy, inference time, and resource trade-offs, to degrade the resource footprint of the application itself rather than merely relaxing its latency bound.

Collaboration Statement. The contributions of this chapter result from a collaboration with Patient Ntumba wa Ntumba (CNAM, Paris), who prepared the Edge AI applications and their model variants, surveyed the related work, and implemented the MVSP baseline for comparison. The queueing-theoretic latency prediction for model variants, the controlled degradation model and algorithm, and the full experimental evaluation constitute the contribution of this thesis.

8.1 Introduction

An increasing portion of the workloads deployed at the edge consists of AI inference tasks built on deep neural networks. These workloads are computationally demanding: processing a single inference request may require significant CPU or memory resources and tens to hundreds of milliseconds of computation. Such workloads are frequently accelerated using GPUs or dedicated inference accelerators, which substantially reduce inference times

but are scarce, costly, and unevenly available across edge sites. In this thesis, we consider CPU-based inference, which remains widespread in edge deployments where accelerators are absent or shared among tenants, and we express resource consumption through the uniform slot abstraction introduced in Chapter 3. The scheduling framework is not tied to this choice: extending the resource model to accelerator-equipped sites is discussed as future work in Chapter 9. In resource-constrained edge environments, the computational weight of inference workloads makes AI applications particularly prone to saturation during peak demand, and particularly difficult to schedule under strict tail-latency SLOs.

A defining property of many AI inference applications is that the same task can be performed by multiple model variants offering different trade-offs. A large, high-accuracy model may consume several resource slots and incur a long inference time, while a smaller, compressed, or quantized variant of the same model performs the task faster and with a lighter resource footprint, at the cost of reduced inference accuracy. This internal flexibility, which general-purpose schedulers ignore by treating applications as black boxes, opens a fundamentally different avenue for handling resource scarcity: rather than relaxing the latency bound of a low-priority application, the scheduler can switch it to a lighter model variant that consumes fewer resources while still meeting its original latency SLO, trading inference accuracy for resource savings. We refer to this as *application-specific degradation*, since the scheduler acts on a configuration knob exposed by the application itself rather than on its own performance targets.

This chapter presents Occamy, a priority-aware scheduler for mixed-criticality AI applications across geographically distributed edge sites. Occamy jointly optimizes three decisions for each application: the selection of a model variant, the placement of instances across edge sites, and the number of instances to deploy. Building on the queueing-theoretic latency model of Chapter 3 and the priority-based degradation framework of Chapter 7, Occamy replaces the SLO relaxation of Cerberus with application-specific degradation through model-variant switching. When resources are insufficient to run all applications at their highest accuracy, Occamy progressively switches low-priority applications to lighter variants in a controlled, priority-ordered manner, preserving strict P99 latency compliance for all deployed applications while degrading inference accuracy only as much as necessary and only for the least critical applications.

This chapter is organized as follows. Section 8.2 reviews related work on AI inference scheduling and model-variant selection. Section 8.3 introduces the model-variant abstraction, extends the problem formulation with variant selection, and presents the priority-based controlled degradation algorithm. Section 8.4 evaluates Occamy on a real distributed edge testbed.

Work	Model Variants	Controlled Degradation	Distributed Edge	P99 Latency Prediction	Mixed-Criticality
CLOVER [98]	✗	✓	✗	✓	✗
EcoEdgeInfer [99]	✗	✗	✗	✓	✗
HyperDrive [100]	✗	✗	✓	✗	✗
INFaaS [101]	✗	✓	✗	✗	✗
InfAdapter [102]	✗	✓	✗	✓	✗
InferRouter [103]	✓	✓	✗	✓	✗
LayerCake [104]	✓	✓	✗	✓	✗
MVSP [25]	✓	✗	✓	✗	✗
Occamy	✓	✓	✓	✓	✓

Table 8.1: Comparison of Occamy with related work.

Section 8.5 discusses the results and the broader implications of application-specific degradation. Section 8.6 concludes the chapter and the technical contributions of this thesis.

8.2 Related Work

The general landscape of scheduling approaches for latency-sensitive applications is covered in Section 2.5, single and multi-application scheduling in the edge-cloud continuum in Chapters 5 and 6, and mixed-criticality scheduling with graceful degradation in Chapter 7. We focus here specifically on works addressing AI inference scheduling, model-variant selection, and controlled degradation of AI applications, which are the core themes of this chapter. Table 8.1 summarizes the key characteristics of the most relevant systems and positions Occamy with respect to them.

Degradation Strategies for AI Applications. Application degradation allows services to continue operating under reduced performance or limited features rather than being interrupted entirely. For DNN inference applications specifically, three main degradation approaches exist [105]: *inference quality degradation*, which switches to lower-quality model variants to reduce resource consumption; *network latency degradation*, which offloads requests to more distant nodes to access additional resources at the cost of higher network latency; and *traffic rate reduction*, which lowers request waiting times by shedding load but requires replication to preserve service continuity. Each approach trades off resource usage, latency, and inference quality differently. The graceful degradation mechanism of Cerberus in Chapter 7 can be seen as a form of latency-budget relaxation, while Occamy adopts inference quality degradation through model-variant switching, the application-specific degradation mechanism introduced in Section 8.1, which directly reduces the resource footprint of the application.

Model-Variant Selection Systems. Several systems exploit the availability of multiple model variants for AI inference serving. INFaaS [101] automatically generates and selects model variants based on inference quality and latency requirements, using ILP-based autoscaling for large-scale inference. CLOVER [98] dynamically adjusts mixtures of model variants to meet strict P99 latency SLOs, while InfAdapter [102] formulates an ILP to maximize inference quality under the same constraints. However, these systems assume centralized cloud resources and the concurrent execution of multiple model variants, both of which are often infeasible in resource-constrained edge environments. InferRouter [103] achieves local graceful degradation on a single edge node by jointly using multiple model variants when predicted queueing delays threaten P99 latency SLOs, while LayerCake [104] routes inference requests across heterogeneous on-device and cloud resources, selecting the model variant and execution location that maximize inference quality under latency constraints. EcoEdgeInfer [99] dynamically tunes hardware and software parameters to jointly optimize P99 inference latency and energy consumption on a single edge device. While these works demonstrate the value of model-variant adaptation, they operate at the granularity of a single node or device and do not address multi-site placement, global optimization, or mixed-criticality scheduling across applications.

Model-Variant Placement in Distributed Edge. Closest to our work is MVSP [25], a heuristic for model-variant selection and placement in distributed edge environments subject to end-to-end latency constraints. MVSP incrementally places and replicates applications to meet resource and workload demands, evaluating solutions on resource cost and average latency. Unlike Occamy, however, it does not address controlled degradation, mixed-criticality workloads, or P99 tail-latency prediction: its lack of queueing-based latency modeling means it cannot anticipate the queueing delays that dominate tail latency under load, a limitation that manifests as extreme SLO violations in our experimental comparison in Section 8.4.

Positioning of Occamy. As summarized in Table 8.1, no existing system simultaneously combines model-variant awareness, controlled degradation, distributed edge placement, P99 tail-latency prediction, and mixed-criticality scheduling. Systems that exploit model variants operate on single nodes or assume cloud resources, while distributed edge schedulers lack variant awareness or tail-latency modeling. Occamy fills this gap by integrating all five capabilities into a unified scheduling framework, building on the latency model of Chapter 3 and the priority-based degradation principle of Chapter 7, and extending them with model-variant switching as a first-class scheduling decision.

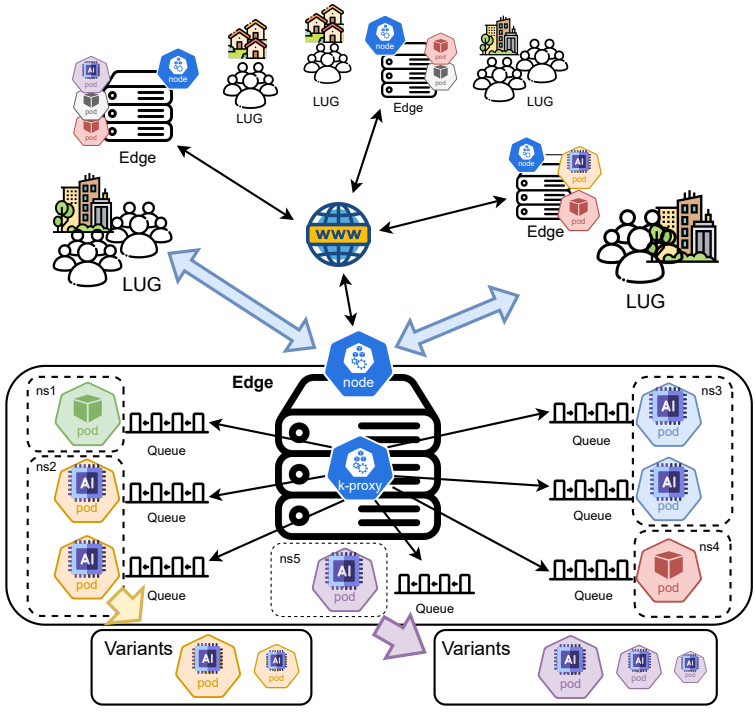


Figure 8.1: A typical edge infrastructure with AI workloads and their variants.

8.3 Scheduling with Model-Variant Degradation

The scheduling framework developed in the previous chapters provides the foundation for this chapter: the $M|D|1$ per-instance latency model of Chapter 3, the multi-application formulation with shared capacity constraints of Chapter 6, and the priority-based degradation principle of Chapter 7 all carry over unchanged.

What is new is the edge AI applications with variants and the degradation mechanism itself. Instead of relaxing the tail-latency SLO of a low-priority application, Occamy applies *application-specific degradation*: it switches the application to a lighter model variant that consumes fewer resources and serves requests faster, while keeping its original SLO intact. This section introduces the model variant abstraction, extends the problem formulation with variant selection as a new scheduling decision, and establishes the complexity of the resulting problem. The edge-cloud continuum infrastructure with edge AI applications is illustrated in Figure 8.1.

8.3.1 Model Variants and Degradation Levels

The notation used throughout this chapter is summarized in Table 8.2. Each AI application $app \in APP$ may be associated with $D_{app} \geq 1$ model variants, capturing different trade-offs between inference quality, inference time, and resource consumption. Applications with $D_{app} = 1$ do not support degradation and operate with a single, fixed model, like the applications considered in the previous chapters. An application operates with one model variant at a time, indexed by $d \in \{0, \ldots, D_{app}\}$ and referred to as the *degradation level*, where higher degradation levels correspond to lighter, faster, but less accurate variants and 0 refers to the original model. The variant at degradation level d determines two properties of every instance of the application: its resource footprint σ^d_{app}, the number of slots required by one instance, and its service rate μ^d_{app}, the rate at which one instance processes inference requests. Switching to a higher degradation level therefore simultaneously reduces the resource consumption and increases the service rate of the application, at the cost of reduced inference quality.

Inference Quality. Inference quality refers to the task-specific performance of a model variant, measured using standard AI metrics such as classification accuracy or mean average precision. In practice, model variants can be obtained through established DNN optimization techniques such as compression, pruning, or quantization, or by uniformly scaling the dimensions of a baseline architecture, such as its number of layers and channels per layer. The generation and profiling of model variants is an application-level concern: Occamy takes as input the set of variants of each application along with their profiled resource footprints and service rates, and does not itself generate variants. The deterministic service time assumption of the $M|D|1$ model from Chapter 3 is particularly well justified in this setting, as the architecture of a given variant is fixed, leading to nearly constant inference times with very small variance.

Priority as Joint Criticality. The priority levels and priority classes APP_p, $p \in \{1, \ldots, K\}$, are inherited from Chapter 7. What changes in this chapter is their interpretation: priority now reflects the joint criticality of an application with respect to both inference quality and latency tolerance. A high-priority application is one for which neither inference quality degradation nor latency violations are acceptable, while a low-priority application can tolerate switching to a lower-quality model variant under resource pressure while still meeting its latency SLO. This joint treatment is natural for DNN inference workloads, where model variants simultaneously affect inference quality, inference time, and resource footprint: an application that is sensitive to inference quality loss is typically also the one whose service is most critical to preserve end-to-end.

Name	Description
$st \in ST$	Edge sites
$app \in APP$	AI applications
$nslots_{st}$	Available resource slots at site st
$lug \in LUG$	Set of user group(s)
$l_{lug,st}$	Round trip network latency between lug and site st
λ_{lug}	Request rate of user group lug
$lugapp_{lug}$	Application used by LUG lug
σ^d_{app}	Slots demand of app at degradation level d
c_{st}	Cost per slot at site st
μ^d_{app}	Service rate of app instance at degradation level d
slo_{app}	Max. P99 end-to-end tail latency SLO per app
$prio_{app}$	Priority level of application app
$\Lambda_{st,lug}$	Maximum arrival rate for the instances at site st can accept without breaking the SLO for user group lug
D_{app}	Number of model variants per app
$d \in [0, D_{app}]$	Degradation level / model variant index
$APP_p \subseteq APP$	Set of applications belonging to priority class p
K	Total number of priority classes
$ninsts_{st,app}$	Number of instances of app deployed at site st
$lugsite_{lug}$	Edge site assigned to serve user group lug
$x_{st,lug}$	Whether user group lug is assigned to site st
$w_{st,app}$	Queueing delay for application app at site st
$L_{st,app}$	Set of user groups sending requests for app to site st
z^d_{app}	1 if app runs at degradation d, 0 otherwise

Table 8.2: Model parameters (top) and variables (bottom). Rows with a grey background are introduced in this chapter.

Contrast with SLO Relaxation. The degradation mechanism of this chapter differs fundamentally from that of Cerberus. Under SLO relaxation, the scheduler acts on its own performance target: a degraded application executes the same computation and consumes the same resources per instance, the savings arise from tolerating higher per-instance utilization, and users experience higher latencies. Under application-specific degradation, the scheduler acts on a configuration exposed by the application itself: a degraded application executes a lighter computation, so each instance consumes fewer slots and serves requests faster, and fewer resources are needed to serve the same workload within the *original* latency SLO. Users of a degraded application continue to experience compliant response times; what degrades is the quality of the inference results. This makes application-specific degradation both more resource-efficient and less disruptive to the user experience, at the price of being applicable only to applications that expose multiple variants.

8.3.2 Problem Formulation with Model Switching

The problem formulation extends the multi-application MILP of Chapter 6 with model-variant selection as a new scheduling decision. All elements of the inherited formulation carry over with the variant-dependent parameters substituted in place of the fixed ones: the shared resource capacity constraint of Chapter 6 applies with the per-variant slot demand σ^d_{app} in place of σ_{app}, as in Equation 8.1:

$$\sum_{app \in APP} ninsts_{st,app} \cdot \sigma^d_{app} \leq nslots_{st}, \quad \forall st \in ST \tag{8.1}$$

and the end-to-end latency decomposition and $M|D|1$ queueing delay distribution of Chapter 3 apply with the per-variant service rate μ^d_{app} in place of μ_{app}, and the SLO constraint is linearized as in the previous chapters, with the maximum admissible arrival rates re-computed for the service rate of the currently selected variant, as in Equation 8.2:

$$x_{st,lug} \cdot \lambda_{inst,st} \leq \Lambda_{st,lug} \text{ for all } st \in ST, lug \in LUG. \tag{8.2}$$

Each user group lug is assigned to exactly one site:

$$\sum_{st \in ST} x_{st,lug} = 1 \quad \forall lug \in LUG \tag{8.3}$$

Variant Selection. We introduce a binary decision variable z^d_{app} such that $z^d_{app} = 1$ if application app is deployed at degradation level $d \leq D_{app}$, and $z^d_{app} = 0$ otherwise. Each application must be deployed with exactly one model variant:

$$\sum_{d=1}^{D_{app}} z^d_{app} = 1, \quad \forall app \in APP. \tag{8.4}$$

A defining property of this formulation is that the SLO threshold slo_{app} of every application remains unchanged at every degradation level. Unlike Cerberus, where degradation relaxes the latency bound itself, degradation here acts on μ^d_{app} and σ^d_{app}, enlarging the feasible region by making the application cheaper and faster to serve while its latency guarantee stays intact.

Lexicographic Optimization Objective. The optimization objective must capture two goals in strict order of importance: preserve inference quality according to the priority hierarchy, and minimize deployment cost. Let $APP_p \subseteq APP$ denote the set of applications belonging to priority class $p \in \{1, \ldots, K\}$, where $p = 1$ is the highest priority and $p = K$ is the lowest. We define the aggregate degradation of priority class p as:

$$\Phi_p = \sum_{app \in APP_p} \sum_{d=0}^{D_{app}} d \cdot z^d_{app}. \tag{8.5}$$

The overall optimization problem is then formulated as the following lexicographic program:

$$\begin{aligned} \min_{\text{lex}} \quad & \left(\Phi_1, \Phi_2, \ldots, \Phi_K, C\right) \\ \text{s.t.} \quad & C = \sum_{st \in ST} \sum_{app \in APP} c_{st} \cdot ninsts_{st,app} \cdot \sigma_{app}^{d}. \\ & (8.1),\ (8.2),\ (8.3),\ (8.4). \end{aligned} \tag{8.6}$$

subject to the variant selection constraint of Equation 8.4 and the inherited capacity and SLO constraints. The lexicographic formulation enforces a strict priority hierarchy in the degradation process: degradation is minimized first for the highest-priority class APP_1, and only among solutions achieving that minimum is degradation minimized for APP_2, and so forth. Consequently, applications in priority class p are degraded only if the problem remains infeasible even after all applications in lower-priority classes have been fully degraded. The final objective C minimizes the total resource cost among all solutions achieving minimal degradation; as in the previous chapters, an alternative objective minimizing the total number of occupied slots is also supported.

Problem Complexity. The NP-hardness of the scheduling problem was established in Chapter 5, where the multi-application extension was anticipated to reduce to a multi-dimensional multiple-choice knapsack problem (MMKP). It is the variant selection introduced in this chapter that completes this reduction: the model variants of each application constitute the multiple mutually exclusive configurations from which exactly one must be chosen, the defining feature of the MMKP. Compared to the formulation of Chapter 6, variant selection multiplies the decision space by $\prod_{app \in APP} D_{app}$, further widening the gap between problem scale and what optimal solvers can handle, and motivating the use of the H_{MCS} heuristic of Chapter 6 within the degradation procedure described next.

8.3.3 Priority-Based Controlled Degradation Algorithm

The scheduling algorithm of Occamy follows the same priority-based iterative structure as the D_{opt} algorithm of Chapter 7, with one fundamental difference: each degradation step switches applications to the next lighter model variant rather than relaxing their SLO bounds. At each step, the H_{MCS} heuristic presented in Chapter 6 is invoked with the updated variant configurations to determine whether a feasible deployment exists; the heuristic itself is used unchanged.

Algorithm 4 shows the procedure. All applications initially operate with their highest inference quality variant, corresponding to degradation level

Algorithm 4 Priority-Based Application-Specific Degradation

1: $\mathcal{S}_{\text{deg}} \leftarrow \emptyset$ ▷ Optimal solution with degradation
2: $\mathcal{F} \leftarrow \emptyset$ ▷ Set of fully degraded applications
3: $\mathcal{P} \leftarrow$ priority levels sorted in ascending order (lowest → highest)
4: $p \leftarrow 1$ ▷ Index of current priority level
5: $\mathcal{A} \leftarrow \{app \in APP \mid prio_{app} = \mathcal{P}[p]\}$ ▷ Applications at current priority level
6: $d \leftarrow 1 \quad \forall app \in APP$ ▷ Current degradation stage of each application
7: **while** $\mathcal{A} \neq APP$ **do**
8: **for** $app \in \mathcal{A}$ **do**
9: $D_{app} \leftarrow$ maximum degradation level of app
10: $D_{\max} \leftarrow \max_{app \in \mathcal{A}} D_{app}$
11: **for** $deg \leftarrow 1$ to $D_{\max}$ **do**
12: **for** $app \in \mathcal{A}$ **do**
13: **if** $d < D_{app}$ **then**
14: $\sigma_{app} \leftarrow \sigma_{app^{(deg)}},\ \mu_{app} \leftarrow \mu_{app^{(deg)}},\ d \leftarrow deg$
15: **else**
16: $\sigma_{app} \leftarrow \sigma_{app^{(D_{app})}},\ \mu_{app} \leftarrow \mu_{app^{(D_{app})}},\ \mathcal{F} \leftarrow \mathcal{F} \cup \{app\}$
17: $\mathcal{S}_{\text{deg}} \leftarrow H_{\text{MCS}}(APP)$ ▷ Find solution with current degradations
18: **if** feasible solution exists **then**
19: **return** $\mathcal{S}_{\text{deg}}$
20: **if** $(\forall app \in \mathcal{A}, app \in \mathcal{F})$ **AND** $(p < |\mathcal{P}|)$ **then**
21: $p \leftarrow p + 1$
22: $\mathcal{A} \leftarrow \{app \in APP \mid prio_{app} = \mathcal{P}[p]\}$

$d = 1$ (line 6). Applications are grouped by priority level and processed in ascending order, from lowest to highest priority (lines 3–5). At each degradation step (line 11), the algorithm switches every not-yet-fully-degraded application at the current priority level to its next lighter variant, updating its resource footprint σ^d_{app} and service rate μ^d_{app} to those of the selected variant (lines 13–14). Applications that have reached their maximum degradation level D_{app} remain fixed at their lightest variant and are added to the set $\mathcal{F}$ of fully degraded applications (lines 15–16). After each adjustment, H_{MCS} is invoked with the current variant configuration (line 17); if a feasible solution exists, it is returned immediately (lines 18–19). If all applications at the current priority level are fully degraded and no feasible solution has been found, the algorithm proceeds to the next priority level while keeping lower-priority applications at their degraded variants (lines 20–22).

By construction, a higher-priority application is switched to a lighter variant only after every application at each lower-priority level has been fully degraded and no feasible deployment exists, ensuring that resource shortfalls are absorbed by the least critical applications first. This realizes the lexicographic objective of Equation 8.6 procedurally: rather than solving the lexicographic program directly, which would be intractable, the algorithm explores degradation configurations in exactly the order that the lexicographic hierar-

chy prescribes, terminating at the first feasible one and thereby guaranteeing minimal degradation for the highest-priority classes.

8.4 Experimental Evaluation

This section evaluates Occamy on a real distributed edge testbed. The experimental design follows the pattern established in the previous chapters, with the applications replaced by AI inference services exposing multiple model variants. We first validate the accuracy of the $M|D|1$ latency model for each application and each of its variants, since the variants introduce service rates and resource footprints not covered by the validation of the earlier chapters. We then evaluate the controlled degradation mechanism in a single-site setting under progressively increasing resource scarcity, and finally assess Occamy in a realistic multi-site, multi-application scenario driven by network latencies and request traces derived from a real-world 5G dataset, comparing it against MVSP, the closest state-of-the-art model-variant scheduler.

8.4.1 Experimental Setup

Infrastructure. The experiments of this chapter follow the methodology of Chapter 4. We reserve 50 nodes for the larger scenarios of this chapter. As the number of slots per instance now depends on the model variant, we configure Kubernetes with the static CPU management policy [70], guaranteeing exclusive core access per pod and eliminating the CPU-sharing interference that would otherwise confound the per-variant service rates.

Applications and Model Variants. We select three representative AI applications with heterogeneous resource requirements, two of which are degradable through multiple model variants: (**a**) OCR who based on the LSTM-based Tesseract engine [81]; (**b**) ResNet [106] for image classification and (**c**) YOLO [107] for object detection. Model variants are generated using a data-driven profiling approach inspired by EfficientNet [108]: for each degradable application, we first identify a baseline DNN architecture via grid search, then uniformly scale its dimensions, such as the number of layers and the number of channels per layer, to produce multiple variants. Each variant is trained and evaluated on the COCO dataset [109], and only the top-performing variants in terms of inference quality are retained. Table 8.3 summarizes the resulting variants with their resource footprints and service rates. OCR is not degradable and operates with a single model.

Application priorities are assigned as follows. OCR represents a lightweight but mission-critical service, such as an AR pipeline in a medical or industrial context, where degradation poses safety risks, and is therefore assigned

Applications	Priority	Variants	Slots per instance	Service rate (req/s)
OCR	High	-	2	10
ResNet	Medium	ResNet-large	10	13
		ResNet-medium	8	21
YOLO	Low	YOLO-large	16	6
		YOLO-medium	14	10.5
		YOLO-small	12	21.5
		YOLO-mini	10	36

Table 8.3: AI applications and their resource requirements in slots.

the highest priority. ResNet and YOLO are more computationally demanding but less critical, and are assigned medium and low priority respectively. This assignment reflects the joint criticality interpretation introduced in Section 8.3.1: OCR tolerates neither quality degradation nor latency violations, while YOLO can absorb resource shortfalls by switching to lighter variants.

Baselines and Configuration. We compare Occamy against four baselines, used in different experiments. (**a**) INVAR [12] and (**b**) Kubernetes HPA [20], configured as in the previous chapters, are used in the single-site experiments. (**c**) MVSP [25], introduced in Section 8.2, places model variants randomly and assigns users via a sequential strategy that considers latency and load, scaling or deploying new variants until a feasible solution is found. (**d**) Similarly to Section 6.5, a *local-first heuristic* which abides by the SLO requirements of each application, but uses only resources on the closest edge site to each LUG and only uses further edges' resource if resources at the closest edge site are insufficient, mimicking the scheduling without global decision making. The latter two are used for the multi-site placement comparison, as neither INVAR nor HPA supports multi-application scheduling with model variants. All results use H_{MCS} with a sampling ratio of $p = 1$. Each experiment is repeated three times for 10 minutes, with the first 15% of samples discarded as warm-up and averages reported across runs.

8.4.2 Microbench: Model Variant Accuracy Evaluation

This experiment follows the same methodology as the microbenchmark of Section 4.4.1, applied here to the AI applications of this chapter. It validates the accuracy of the $M|D|1$ latency model for each AI application and each of its model variants, extending the validation of Chapter 4 to the new applications and to the variant-dependent service rates introduced in this chapter.

Setup. We deploy a single instance of each application and of each of its variants, and generate requests at progressively increasing rates following a Pois-

son arrival process. The network latency is set to zero to isolate the queueing delay as the sole contributor to end-to-end latency beyond the inference time.

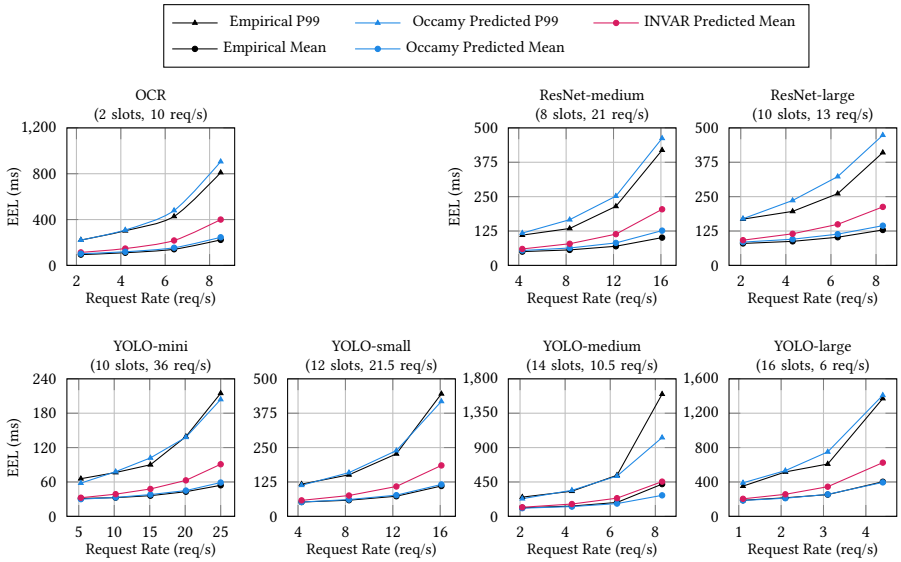


Figure 8.2: Evaluation of performance and prediction accuracy for the three applications and their variants (Table 8.3) used in isolation and in a local setting.

Results Analysis. Figure 8.2 shows the empirical and predicted mean and P99 end-to-end latencies for the three applications and all their variants under increasing request rates. Occamy accurately predicts the tail latency at low to moderate request rates for every variant, and only slightly overestimates it as the load approaches saturation, consistent with the conservative behavior observed in Chapter 4. INVAR predicts only mean latency, with an average relative error of 49%, for the reasons analyzed in Section 4.5. The key result here is that this accuracy holds across all model variants, whose fixed architectures yield nearly constant inference times: the deterministic service time assumption of the $M|D|1$ model is therefore satisfied per variant, validating the variant-parameterized latency model of Section 8.3.2.

8.4.3 Single-Site: Controlled Degradation under Scarcity

This experiment assesses the effectiveness of the controlled degradation mechanism of Occamy under resource scarcity. It follows the same design as the single-site degradation experiment of Section 7.4.1, progressively increasing the load until the total resource demand exceeds the site capacity and observing how the scheduler resolves the shortfall. The difference lies in what

Series	App	Priority	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
§1	OCR	High	1	2	3	4	5	6	7	8	9	10
	YOLOs	Low	2	2	2	2	2	2	2	2	2	2
§2	OCR	High	2	2	2	2	2	2	2	2	2	2
	YOLOs	Low	2	4	6	8	10	12	14	16	18	20

Table 8.4: Two series of request rates (req/s) for each of the two applications in ten different configurations.

degradation does: rather than relaxing the SLO of the low-priority application, Occamy switches it to a lighter model variant, freeing resources for the high-priority application while preserving the original SLOs of all deployed applications.

Setup. We deploy two applications at two priority levels on a single site with a capacity of 20 slots: OCR at high priority and YOLO at low priority, each requested by a single LUG with negligible network latency. We set $slo = 250$ ms for OCR and $slo = 600$ ms for YOLO, and let Occamy determine the number of instances for each application. We evaluate two configuration series, summarized in Table 8.4. Starting from a non-degraded situation, series §1 progressively increases the request rate to OCR while keeping YOLO's rate fixed, and series §2 progressively increases the request rate to YOLO while keeping OCR's rate fixed. In both series, the total resource demand eventually exceeds the site capacity, triggering degradation.

Results Analysis. Figure 8.3 and 8.4 shows the number of slots allocated to each application under each configuration series, along with the empirical and predicted P99 latencies. In series §1, all slots are in use from configuration #3. As the request rate to OCR increases, Occamy first degrades YOLO from YOLO-large to YOLO-medium at #5, then to YOLO-small at #7, and eventually to YOLO-mini at #9, progressively freeing resources for OCR. In series §2, the resources remaining after serving OCR are already insufficient to host even a single replica of YOLO-large from #1, so Occamy directly deploys YOLO-medium at #2, and subsequently degrades to YOLO-small at #4 and YOLO-mini at #6 as YOLO's own demand grows.

In both series, Occamy guarantees the P99 latency SLO for the high-priority OCR application throughout, while degrading YOLO in a controlled manner by switching to progressively lighter variants only when resources become insufficient. Crucially, YOLO continues to satisfy its own P99 SLO of 600 ms at every degradation level: what degrades is the inference quality of the deployed variant, not the responsiveness of the service. This is the defining behavioral difference from Cerberus: application-specific degradation preserves responsiveness and sacrifices output quality, whereas SLO

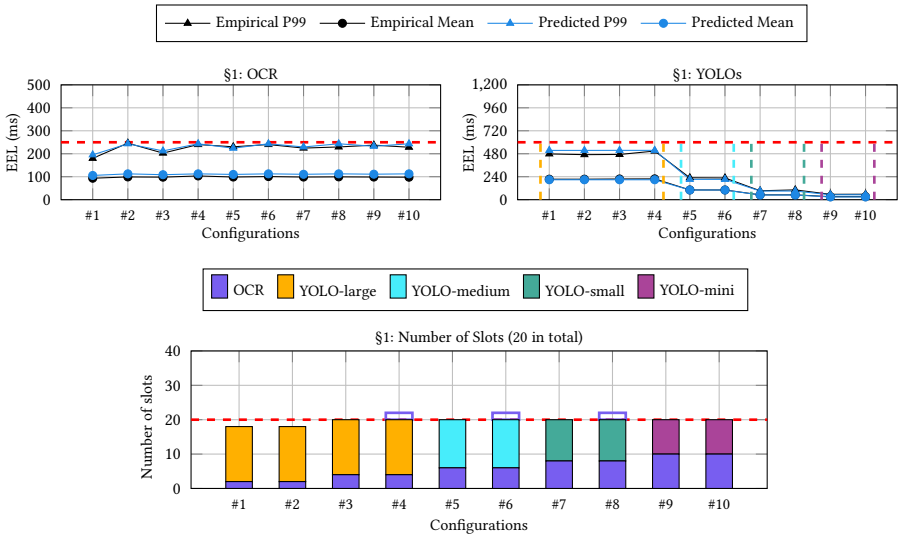


Figure 8.3: The number of instances deployed for *OCR* and *YOLO* variants (left) for the single site scenario with graceful degradation using Occamy, along with empirical and predicted P99 and mean end-to-end latencies (EEL) (middle & right) correspond to the configuration series §1 in Tables 8.4. Shadow bar in the plot on the bottom indicates the lack of computing resource slot(s) to deploy such application's instance.

relaxation preserves output quality and sacrifices responsiveness.

8.4.4 Multi-Site: Realistic Placement Evaluation

This experiment evaluates Occamy in a complex multi-site, multi-application scenario driven by a real-world network trace, and compares its placement plans against those of MVSP.

Setup. We consider five edge sites E1-E5, each with a capacity of 72 slots and per-slot pricing varying across sites, with E4 and E5 being the most expensive, as summarized in Table 8.5. We place ten user locations with diverse latency profiles, ensuring each location has one nearby edge site and distinct network latencies to the others. The network latencies and request loads are derived from TCP interactions in a large-scale 5G network dataset [110], where flows are aggregated into tracking area codes representing LUGs and remote autonomous systems representing sites. At each location we place three LUGs, one per application, for a total of 30 LUGs. The P99 SLOs are set to slo = 250 ms for OCR, 500 ms for ResNet and its variants, and 600 ms for YOLO and its variants.

We increase the load through seven configurations, with the total request rates shown in Table 8.6, distributed among LUGs according to the session

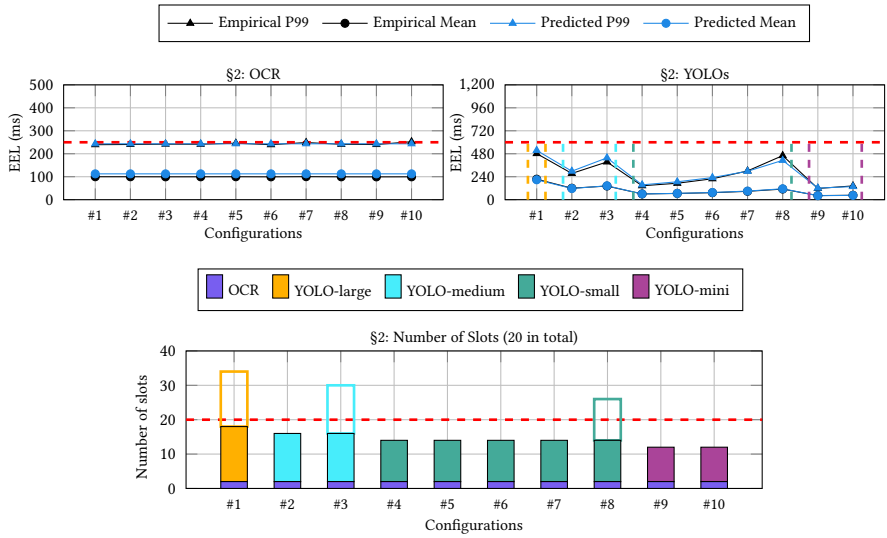


Figure 8.4: The number of instances deployed for *OCR* and *YOLO* variants (left) for the single site scenario with graceful degradation using Occamy, along with empirical and predicted P99 and mean end-to-end latencies (EEL) (middle & right) correspond to the configuration series §2 in Tables 8.4.

statistics observed in the trace. Configuration #7 includes a sharp increase of 30 req/s from a single LUG for OCR, emulating a sudden localized surge in user activity. We evaluate both the min slots and min cost optimization objectives.

Results Analysis. Figure 8.5 shows the distribution of empirical P99 latencies from the placement plans generated by Occamy and MVSP across all configurations and both objectives. For Occamy, no degradation is required in configurations #1 and #2, where the aggregate resource demand of all applications is met by the available resources. As demand increases and all sites become fully loaded, Occamy introduces controlled degradation starting from the low-priority level, degrading YOLO to free resources for higher-priority applications in configurations #3 to #6. Once YOLO is fully degraded to its YOLO-mini variant and demand continues to grow, Occamy begins degrading the medium-priority ResNet from configuration #6 to #7, freeing additional resources for the highest-priority OCR application. Throughout all configurations, including the localized demand burst of #7, the P99 SLOs of all three applications are satisfied, confirming that the priority ordering enforced by the lexicographic objective of Equation 8.6 holds in a realistic multi-site setting.

Table 8.7 contrasts the two objectives with the *local-first heuristic*, which is consistently dominated by Occamy: it uses more resource slots and incurs

Site	Slots	Price	Network latency from LUG to site (ms)									
			LUGs 1-3	LUGs 4-6	LUGs 7-9	LUGs 10-12	LUGs 13-15	LUGs 16-18	LUGs 19-21	LUGs 22-24	LUGs 25-27	LUGs 28-30
E1	72	$2	67	61	40	65	30	40	58	67	80	47
E2	72	$3	117	11	80	115	44	90	58	69	80	27
E3	72	$3	67	61	90	77	80	60	8	19	30	47
E4	72	$4	67	61	30	65	20	40	108	117	130	77
E5	72	$4	17	111	40	27	56	10	58	67	80	97

Table 8.5: Resource in slots, pricing, and latencies for sites E1-E5.

	#1	#2	#3	#4	#5	#6	#7
OCR	13	24	33	40	71	104	**169**
ResNet	13	27	33	42	72	104	127
YOLO	13	26	32	43	72	105	129

Table 8.6: Total request rates for each of the three applications in the seven different configurations. From #6 to #7 there is a burst in the popularity of the *OCR* application.

a higher total cost under configuration #5 and #6, and in configuration #7 the local-first heuristic fails to find any feasible placement within the same degradation level, while both Occamy objectives still return SLO-compliant solutions. Notably, in configuration #5 both objectives use the same number of slots (252), yet min_cost achieves it at a lower total cost (732 versus 772), an artifact of this particular configuration where the cost-minimal placement happens to require no additional slots. This confirms that latency-greedy placement, though intuitive, wastes resources and can even become infeasible under resource scarcity, motivating the need for global scheduling as in Occamy.

MVSP, by contrast, exhibits extreme SLO violations exceeding 500 times the target thresholds. This is a direct consequence of its lack of a queueing-theoretic latency model: MVSP selects and places variants based on average latency and resource cost without any ability to predict the queueing delays that dominate tail latency under load, and consequently under-provisions instances to a degree that drives the system deep into saturation. Figure 8.7 and Figure 8.8 show the placement plans computed by Occamy across sites and configurations, illustrating how variant selection and instance placement co-evolve as resource pressure increases, with the scheduler consolidating high-priority OCR instances on the cheaper sites under the min cost objective while progressively lightening the variants of the lower-priority applications.

Config	Objective	Used slots	Total cost
#5	min_slots	**252**	772
	min_cost	**252**	**732**
	local-first	264	930
#6	min_slots	**314**	1008
	min_cost	316	**984**
	local-first	324	1060
#7	min_slots	**324**	1044
	min_cost	326	**1028**
	local-first	Infeasible	Infeasible

Table 8.7: Global results obtained from Occamy with two optimization objectives and *local-first heuristic* in configuration #5 to #7 as examples. Infeasible indicates that no feasible solution found under the same degradation level.

8.5 Discussion

The experimental results presented in Section 8.4 demonstrate that Occamy accurately predicts tail latency across model variants, and that model-variant switching provides an effective degradation mechanism that preserves latency compliance for all applications while reducing inference quality only where necessary. This section discusses the design choices underlying these results and the limitations that bound their applicability.

8.5.1 Degradation Level as a Proxy for Quality Loss

The lexicographic objective of Equation 8.6 minimizes the aggregate degradation Φ_p of each priority class, defined as the sum of the degradation levels d of its applications. This formulation treats the degradation level as a proxy for inference quality loss, implicitly assuming that the quality penalty incurred by moving from level d to $d+1$ is comparable across variants and across applications. In practice, this assumption is an approximation: the accuracy drop between two adjacent variants of the same application is rarely uniform, and the quality metrics of different applications, such as classification accuracy for ResNet and mean average precision for YOLO, are not directly commensurable.

A more faithful formulation would replace the degradation level with the measured quality loss of each variant, expressed as a normalized penalty relative to the highest-quality variant of the same application. This would allow the scheduler to prefer a two-level degradation of an application whose variants degrade gracefully over a one-level degradation of an application with

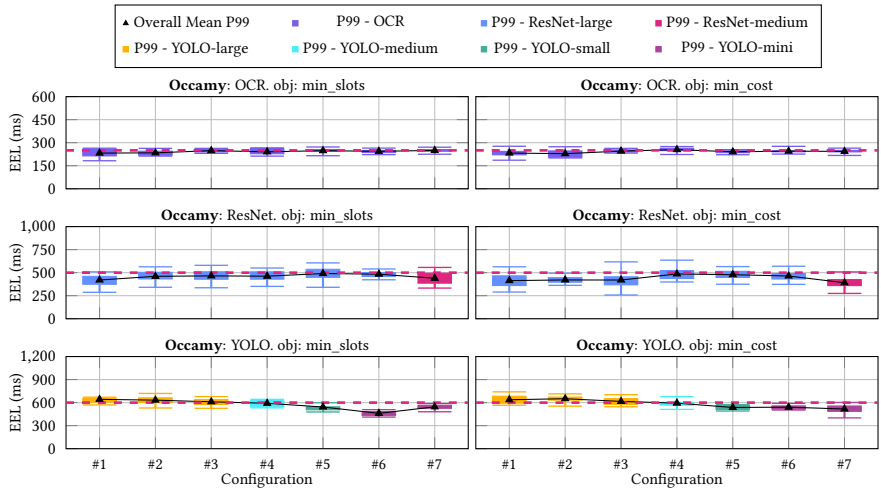


Figure 8.5: P99 latency distributions for all LUGs from Occamy across all seven configurations, for the three target applications and two global optimization objectives.

a steep quality cliff. The formulation supports this substitution directly: the coefficient d in Equation 8.6 can be replaced by any application-specific and variant-specific penalty weight without altering the structure of the problem or the degradation algorithm. We adopt the degradation level in this work because it requires no cross-application normalization of quality metrics, which is itself a non-trivial modeling question, and because it makes the priority semantics transparent to the infrastructure provider assigning them.

8.5.2 Complementarity of the Two Degradation Mechanisms

The SLO relaxation of Chapter 7 and the application-specific degradation of this chapter act on different levers, and are therefore complementary rather than competing. SLO relaxation is scheduler-side: it applies to any latency-sensitive application, including those with a single fixed implementation, and frees resources by tolerating higher per-instance utilization. Application-specific degradation is application-side: it applies only to applications exposing multiple variants, and frees resources by reducing the computational footprint of each instance while leaving the latency guarantee intact.

Each mechanism has a domain where it is clearly preferable. For an application whose users tolerate slower responses but require exact results, such as a batch analytics query, SLO relaxation is the appropriate lever. For an interactive AI service whose users require responsiveness above all, such as an augmented reality overlay, model-variant switching is preferable, since a slightly less accurate result delivered on time is more useful than an exact

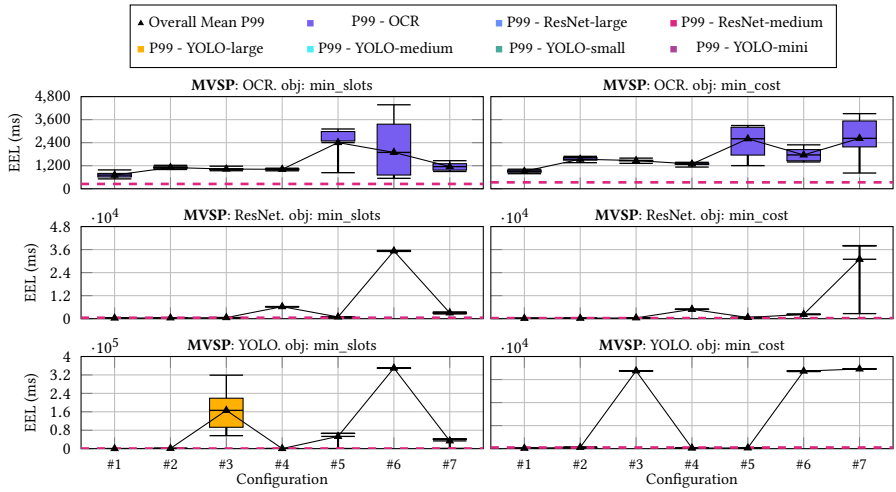


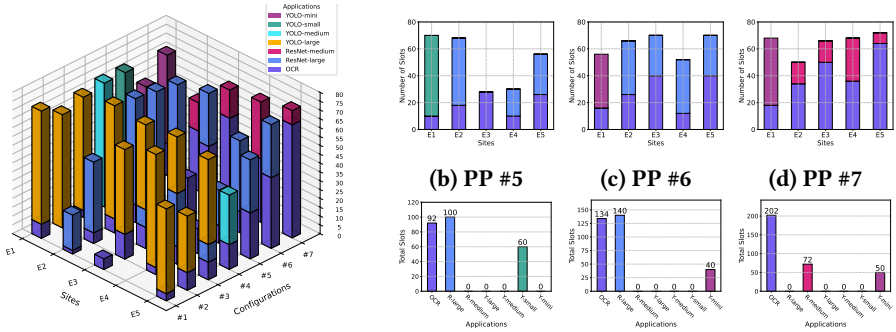
Figure 8.6: P99 latency distributions for all LUGs from MVSP across all seven configurations, for the three target applications and two global optimization objectives.

result delivered late. A unified scheduler could expose both levers simultaneously, treating the choice between relaxing the SLO and switching the variant as an additional decision in the optimization, and selecting per application whichever trade-off its owner declares acceptable. Realizing this unified degradation space would require a common currency in which latency degradation and quality degradation can be compared, which is a natural extension of the penalty-weight formulation discussed above and an interesting direction for future work.

8.5.3 Variant Generation and Profiling

Occamy takes the set of model variants of each application as an input, together with their profiled resource footprints and service rates. The generation of these variants, whether by compression, pruning, quantization, or architectural scaling, and their offline profiling on the target hardware, are treated as application-level concerns outside the scope of the scheduler. This separation is deliberate: it keeps the scheduling framework agnostic to the specific DNN optimization technique used, and allows application owners to expose whichever variants they consider acceptable for their service.

The separation does, however, place two requirements on the deployment pipeline. First, profiling must be performed on hardware representative of the target edge sites, since the service rate of a variant is hardware-dependent and an inaccurate μ^d_{app} propagates directly into inaccurate tail-latency predictions. Second, the resource footprint of each variant must be express-



(a) Obj: min_slots (b) PP #5 (c) PP #6 (d) PP #7

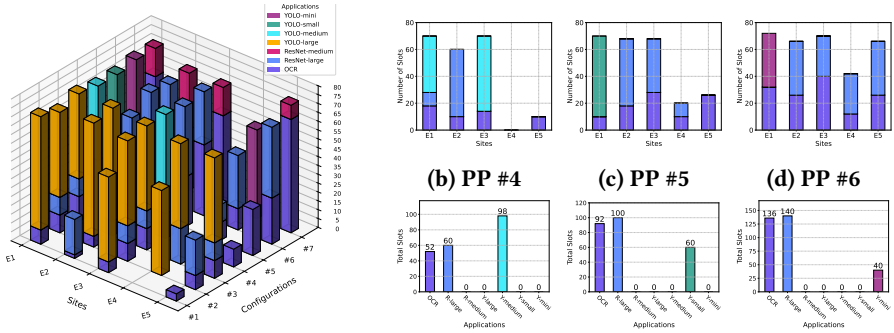
(e) Slots #5 (f) Slots #6 (g) Slots #7

Figure 8.7: Placement plans (PP) generated by Occamy for multi-site multi-application scenarios under objective min_slots. Plot (a) shows the overall placement plan for all configurations. Plots (b)–(d) detail the placement plans for configuration #5 to #7 (sliced from the 3D bar chart on the left), while plots (e)–(g) show the corresponding resource slot allocation per application.

ible in the uniform slot abstraction of the infrastructure, which as discussed in Chapter 6 becomes limiting when variants differ along resource dimensions the abstraction does not capture, most notably when some variants are accelerator-bound and others are CPU-bound. Extending the slot model to a multi-dimensional resource vector would allow the scheduler to reason about such heterogeneous variants natively.

8.5.4 Applicability Boundary

Application-specific degradation is available only to applications that expose multiple model variants. Applications with $D_{app} = 1$, including all non-AI services and AI services deployed with a single fixed model, cannot be degraded by Occamy and are always served at their only configuration. In a deployment mixing degradable and non-degradable applications, the resource shortfall must therefore be absorbed entirely by the degradable ones, regardless of their priority relative to the non-degradable applications. This is an acceptable outcome when the degradable applications are also the least critical, as in our evaluation, but it constrains the priority assignments that the infrastructure provider can meaningfully express: assigning the lowest priority to a non-degradable application yields no benefit under scarcity, since that application cannot contribute resources to the shortfall. Combining the two degradation mechanisms as discussed above would remove this constraint, allowing non-degradable applications to contribute through SLO relaxation while degradable ones contribute through variant switching.



(a) Obj: min_cost (b) PP #4 (c) PP #5 (d) PP #6 (e) Slots #4 (f) Slots #5 (g) Slots #6

Figure 8.8: Placement plans (PP) generated by Occamy for multi-site multi-application scenarios under objective min_cost. Same layout as Figure 8.7, but with configurations #4 to #6 to better illustrate the degradation among different steps.

8.6 Conclusion

This chapter has presented Occamy, an accuracy-aware scheduler for mixed-criticality AI applications deployed across distributed edge sites. Occamy exploits a property of AI inference workloads that general-purpose schedulers ignore, namely that the same inference task can be served by multiple model variants offering different trade-offs between inference quality, inference time, and resource footprint. Building on the queueing-theoretic latency model of Chapter 3, the multi-application formulation of Chapter 6, and the priority-based degradation principle of Chapter 7, Occamy introduces model-variant selection as a first-class scheduling decision, jointly optimizing the variant, the placement, and the number of instances of every application under strict P99 tail-latency constraints.

The central conceptual contribution of the chapter is the shift in what degradation means. Where Cerberus relaxes the latency bound of a low-priority application, leaving its computation unchanged and its users waiting longer, Occamy switches the application to a lighter variant that consumes fewer resources and serves requests faster, so that fewer resources suffice to serve the same workload within the original latency guarantee. Users of a degraded application continue to experience compliant response times; what degrades is the quality of the inference results. This makes application-specific degradation both more resource-efficient, since it reduces the computational footprint rather than merely tolerating higher utilization, and less disruptive, since responsiveness is preserved. The lexicographic optimization objective ensures that this degradation is applied in strict priority order, and the resulting problem, which completes the multi-dimensional multiple-

choice knapsack reduction anticipated in Chapter 5, is solved with the H_{MCS} heuristic of Chapter 6 driven by a priority-based controlled degradation procedure.

The experimental evaluation on a real distributed edge testbed with three AI applications and their variants demonstrated three results. First, the $M|D|1$ latency model predicts tail latency accurately across all model variants, whose fixed architectures yield nearly constant inference times, while INVAR, which predicts only mean latency, exhibits an average relative error of 49%. Second, the controlled degradation mechanism correctly enforces the priority hierarchy under resource scarcity, degrading low-priority applications through progressively lighter variants while preserving the SLOs of all deployed applications, including those being degraded. Third, in a multi-site scenario driven by real 5G network traces, Occamy satisfies the P99 SLOs of all applications across all load configurations, including a localized demand burst, whereas MVSP, which lacks a queueing-theoretic latency model, exhibits SLO violations exceeding 500 times the target thresholds.

Occamy completes the progression of scheduling systems developed in this thesis. Beginning with accurate tail-latency prediction and single-application provisioning, extending to globally optimized multi-application scheduling, then to controlled degradation under resource scarcity, and finally to accuracy-aware degradation for edge AI, the four contributions together form a unified latency-aware scheduling methodology for the edge-cloud continuum. The following chapter concludes the thesis, summarizing these contributions and outlining directions for future research.

Conclusion and Discussion

9

This thesis has addressed the problem of scheduling latency-sensitive containerized applications across the distributed edge-cloud continuum under strict tail-latency Service Level Objectives. This final chapter first summarizes the contributions developed throughout the preceding chapters, then revisits the design choices that recur across the four scheduling systems and discusses them in a consolidated manner, and finally outlines the directions for future research that emerge from their limitations.

9.1 Conclusion

The starting point of this thesis was the observation that mainstream orchestration platforms are latency-blind: they manage resources based on hardware utilization metrics and mean performance statistics, and consequently cannot anticipate the queueing delays and network propagation latencies that drive tail-latency violations in the edge-cloud continuum. Chapter 2 established the technical background of this problem, characterized the workloads and quality-of-service requirements of latency-sensitive applications, and identified four open challenges: the performance inversion problem, resource contention among co-located applications, resource scarcity under peak demand, and the allocation of mixed-criticality edge AI workloads. These challenges defined the four research questions that structured the contributions of this thesis.

Chapter 3 developed the analytical foundation shared by all subsequent contributions. Modeling each application instance as an independent $M|D|1$ queue, a choice that reflects both the near-constant service times of the targeted workloads and the per-instance request dispatching performed by kube-proxy, we derived a closed-form expression for the full end-to-end latency distribution, decomposed into network latency, queueing delay, and service time. From this distribution the model predicts not only the mean but any percentile of the end-to-end latency, and in particular the P99 tail latency that defines the SLOs pursued throughout the thesis. This turned the informal notion of a latency guarantee into a quantity that can be computed in advance from an application's workload, its resource allocation, and the network dis-

tances to its users.

Chapter 4 established the experimental framework on which all contributions of this thesis are evaluated, and validated the model of Chapter 3 against reality. Rather than relying on simulation, we deploy a Kubernetes cluster on the bare-metal Grid'5000 testbed, emulate the WAN latencies of the edge-cloud continuum, and generate Poisson workloads representative of independently acting user groups over four latency-sensitive applications spanning network-dominated, computation-dominated, and inference-heavy service profiles. Within this framework, the $M|D|1$ model was shown to predict the P99 end-to-end latency of real containerized deployments accurately, reducing the mean prediction error from the 49% of the $M|M|c$ model used by prior work to 10%, and to slightly overestimate the tail only near saturation, a conservative behavior that is desirable for SLO enforcement. The model further proved sufficient to determine the minimal number of instances meeting a tail-latency SLO at a single site, where INVAR and Kubernetes HPA both under-provision and violate the SLO by up to 171% and 76%, respectively. This validated foundation answers how many instances an application requires at one site, but leaves open where those instances should be placed across a distributed infrastructure.

Chapter 5 answered that question with Lassy, a scheduler for a single application in the edge-cloud continuum. By pre-computing the maximum admissible arrival rate for each site and user group pair, the nonlinear tail-latency constraint is linearized, transforming the scheduling problem into a Mixed-Integer Linear Program with customizable objectives that provably satisfies P99 SLOs while minimizing operational costs. Experiments on a real Kubernetes testbed showed that Lassy computes placement plans that keep the P99 latency of all user groups within their SLOs across a wide range of workload configurations, including localized demand bursts, and that different optimization objectives lead it to qualitatively different placement strategies while consistently preserving SLO compliance.

Chapter 6 generalized the framework to multiple co-located applications with Scylla, introducing heterogeneous per-application resource footprints and a shared capacity constraint that couples the scheduling decisions of all applications. To confront the exponential growth of the joint optimization problem, the chapter introduced heuristic algorithms that decompose the global problem into sequences of single-application problems explored over application orderings, with the Layered Permutation Search achieving computation times up to 500 times lower than exhaustive search while returning placement plans indistinguishable from the global optimum in the evaluated scenarios.

Chapter 7 confronted resource scarcity in edge-only infrastructures with Cerberus, which extends the scheduling framework with a graceful degrada-

tion mechanism. By introducing application priority levels and Service Level Thresholds as first-class scheduling parameters, Cerberus relaxes the SLOs of low-criticality applications in a controlled, bounded, and priority-ordered manner when no feasible placement plan exists, preserving strict compliance for high-criticality applications instead of resorting to abrupt eviction or uncontrolled violations.

Chapter 8 completed the progression with Occamy, which exploits the internal adaptability of edge AI workloads. By treating the selection of a deep neural network model variant as a first-class scheduling decision, Occamy degrades the computational footprint of low-priority applications rather than their latency bounds: a degraded application switches to a lighter, faster variant that continues to satisfy its original SLO, trading inference quality for resource savings. The resulting problem completes the multi-dimensional multiple-choice knapsack reduction anticipated in Chapter 5, and is solved by a priority-based controlled degradation procedure built on the heuristics of Chapter 6. Experiments on a real distributed edge testbed driven by 5G network traces showed that Occamy satisfies the P99 SLOs of all applications across all load conditions, while the closest state-of-the-art model-variant scheduler exhibits violations exceeding 500 times the target thresholds.

Together, these contributions form a unified, progressively constructed methodology for latency-aware orchestration in the edge-cloud continuum: from accurate tail-latency prediction, to SLO-sufficing scheduling for single application, to globally optimized multi-application placement, to controlled degradation under scarcity, and finally to accuracy-aware scheduling of mixed-criticality edge AI. All contributions were validated on real testbed infrastructure with production-grade orchestration, demonstrating that principled, model-driven scheduling with strict tail-latency guarantees is achievable in practice.

9.2 General Discussion

Across the four scheduling systems presented in this thesis, several design choices recur and underpin the results of every chapter. Because they are shared rather than specific to any one contribution, we discuss them here in a consolidated manner: the queueing model at the core of the latency prediction, the accuracy of the resulting tail-latency estimates, the strategy for mapping user groups to sites, and the choice of optimization solver.

9.2.1 Queueing Model

The $M|D|1$ model introduced in Chapter 3 is the analytical foundation shared by all four schedulers, and its accuracy relative to the $M|M|c$ model used by

prior work such as INVAR [12] rests on two modeling differences. The first concerns the service time distribution. The $M|M|c$ model assumes exponentially distributed service times, which introduces artificial variability and consequently overestimates queueing delays for workloads whose service times are near-constant. The $M|D|1$ model assumes deterministic service times, a significantly better fit for applications whose processing time per request is largely independent of the request content.

The second difference concerns the queueing topology. The $M|M|c$ model treats all instances at a site as a single pool of servers sharing a common queue, so that an arriving request is always dispatched to the next available instance. Our model instead treats each instance as an independent $M|D|1$ queue, reflecting the actual behavior of kube-proxy, which distributes requests in a round-robin or random fashion without knowledge of the current queue length at each instance. Consequently, a request may arrive at a busy instance even when another is idle, producing queueing delays that the $M|M|c$ model cannot capture. The microbenchmark evaluation of Section 4.4.1 confirmed the practical consequence of these two differences: the $M|D|1$ model achieves a mean prediction error of 10% against 49% for the $M|M|c$ model, and the same accuracy held across the four applications of Chapter 7 and the model variants of Chapter 8, whose fixed architectures likewise yield near-constant service times.

The accuracy of the $M|D|1$ model is nonetheless contingent on this near-constant service time assumption. For workloads with highly variable processing times, the deterministic model would underestimate queueing delays. In such cases a model that explicitly captures service time variability, such as an $M|M|1$ or, more generally, an $M|G|1$ queue parameterized by the empirically measured variance, would be more appropriate. Since the entire scheduling framework interacts with the queueing model only through the resulting waiting time distribution, substituting the model requires no change to the optimization formulation or the algorithms, an extension we return to in Section 9.3.

9.2.2 Tail Latency Prediction Accuracy

Beyond the mean, the $M|D|1$ model provides a closed-form expression for the full queueing delay distribution, enabling direct prediction of tail latency at any target percentile. As observed consistently across the experimental chapters, the model predicts the P99 latency accurately at low to moderate utilization and conservatively overestimates it as the load approaches saturation. This conservative behavior is a desirable property for SLO enforcement: a scheduler relying on an overestimate of the tail latency provisions slightly more instances than strictly necessary, but never underestimates the latency

and thereby never risks an SLO violation through optimism. The modest over-provisioning this induces at high utilization is a deliberate and acceptable price for the guarantee that satisfied predictions translate into satisfied SLOs in practice.

9.2.3 Request Distribution

A design choice shared by all four schedulers is the strict one-to-one mapping between each Local User Group and a single site: all requests from a given LUG are directed to exactly one site per placement plan. This contrasts with INVAR, which splits a LUG's traffic proportionally across several sites simultaneously, and the difference works consistently in our favour. First, it simplifies the routing configuration decisively: the mapping is realized as a single static routing rule in the network infrastructure or the Kubernetes ingress, with no per-request load-balancing logic that must be kept aware of the current scheduling plan, whereas proportional splitting requires exactly such plan-aware routing at every ingress point. Second, it keeps the optimization problem more tractable, since the assignment of each LUG is a binary choice rather than a continuous proportion, reducing both the number and the type of decision variables. Third, because all requests from a LUG always reach the same site, the mapping is directly compatible with stateful applications that require session affinity, offering a straightforward path toward supporting them; INVAR's proportional splitting cannot provide this guarantee without additional affinity mechanisms layered on top.

The mapping is nonetheless less flexible than proportional splitting in distributing load. When a LUG generates a request rate that a single site cannot absorb within its SLO budget, our schedulers must either redirect the entire LUG to a more resourceful but potentially more distant site, or deploy sufficient instances at the current site to handle the full load, whereas proportional splitting could in principle spread the excess across several sites. As the experimental results across all chapters demonstrate, however, the one-to-one mapping proves sufficient to meet SLOs across a wide range of realistic workload conditions when combined with accurate tail-latency prediction and globally optimized placement.

9.2.4 Granularity of Local User Groups

The one-to-one mapping discussed above raises a natural question about the granularity at which Local User Groups are defined. Because our schedulers assign each LUG to a single site, the LUG is also the smallest unit of demand that can be redirected: a LUG whose aggregate request rate exceeds what any single reachable site can absorb within its SLO budget cannot be served at all under the mapping, even when the combined capacity of several nearby

sites would suffice. Rather than relaxing the mapping to permit proportional splitting, with the routing complexity and reduced tractability this entails, the same effect can be achieved upstream, by defining LUGs at a finer granularity.

Splitting a LUG partitions the users it aggregates into several smaller groups, each becoming an independent unit of demand that the scheduler may assign to a different site. A LUG whose demand overwhelms its closest site can thus be divided so that part of its traffic is served locally while the remainder is redirected to sites with spare capacity, recovering much of the load-distribution flexibility of proportional splitting without altering the scheduling model. The granularity of LUGs is therefore a tuning parameter: coarse groups keep the problem small and the routing simple, while finer groups grant the scheduler more freedom to balance load across sites, at the cost of additional decision variables.

The same mechanism provides the means to adapt to changes in the underlying conditions. As noted in Section 2.1.3, the network latencies and request rates that characterize a LUG are treated as known inputs, valid for the duration of a scheduling round. When these conditions shift, for example when the users within a LUG cease to share a common latency profile to the available sites, or when the group's aggregate demand grows beyond what a single site can serve, the LUG can be subdivided into smaller, more homogeneous groups that better capture the updated distribution, and the affected groups rescheduled. In this sense, LUG granularity is the lever through which the schedulers of this thesis accommodate spatial and temporal variation in the workload, complementing the periodic rescheduling on which they already rely.

9.2.5 Computing Resource Abstraction

Resource capacity is expressed throughout this thesis through a uniform slot abstraction, refined progressively across the contributions: Chapters 3 and 5 assume that each instance occupies exactly one slot, Chapter 6 introduces a per-application slot demand σ_{app} to capture heterogeneous footprints, and Chapter 8 makes this demand variant-dependent. In all cases, a slot denotes a fixed and identical bundle of CPU and memory, which is appropriate for the CPU-bound and memory-bound workloads of our evaluation but collapses what are in reality several distinct resource dimensions into a single scalar.

This collapse is limiting wherever applications differ along a dimension the bundle does not separate. GPUs and dedicated inference accelerators are the clearest case: they are essential for accelerating deep neural network inference, are typically scarce and unevenly distributed across edge sites, and cannot be expressed as a multiple of CPU slots. An application whose variants differ in whether they are accelerator-bound or CPU-bound, as discussed in

Chapter 8, is therefore not representable in the current model. Power is a second such dimension, as edge sites increasingly operate under explicit power budgets [111], and network bandwidth a third, becoming binding for applications that transfer large payloads.

Generalizing the slot to a vector of resource types, with each instance consuming a specified quantity of each, would turn the capacity constraint into a multi-dimensional knapsack constraint. The scheduling framework accommodates this change structurally, since the constraint remains linear and the heuristics are indifferent to its dimensionality, but the search space grows accordingly and the practical scalability of the resulting formulation would need to be re-established. This extension concerns what the scheduler can represent; the related question of what it should optimize over these additional dimensions is taken up in Section 9.3.

9.2.6 Optimization Solvers

All four schedulers rely on the Gurobi Optimizer [86] to solve their Mixed-Integer Linear Programs. Gurobi is widely regarded as one of the most performant MILP solvers available, combining branch-and-bound with cutting planes, presolve routines, and parallel search, and it integrates naturally with the preprocessing pipeline through which the pre-computed maximum admissible arrival rates are supplied. Several alternatives could be substituted without changing the formulation. CPLEX [112] offers comparable performance and is equally widely used in operations research. SCIP [113] is a high-performance open-source alternative suitable when a commercial license is unavailable. Knitro [114] is designed primarily for nonlinear and mixed-integer nonlinear programs, making it less suited to our purely linear formulation, while Google OR-Tools [115], though a versatile open-source framework, does not match dedicated MILP solvers on problems of the scale and structure considered here. Gurobi was selected for its performance, robustness, and ease of integration.

9.2.7 Heuristic Algorithms

The evaluations of Sections 6.5.2 and 7.4.3 reveal a clear scalability hierarchy among the scheduling approaches used in this thesis. G_{opt} produces provably optimal solutions but becomes intractable beyond a handful of applications, making it unsuitable for deployments where the number of co-located applications is larger. H_{Full} reduces computation time by decomposing the joint MILP into a sequence of single-application problems, but the factorial growth in the number of permutations limits it to scenarios with up to six applications in our setups. The two scalable heuristics address this limitation from complementary angles, and both expose an explicit parameter controlling the

trade-off between solution quality and computation time, which neither G_{opt} nor H_{Full} offers.

The depth parameter d of H_{LPS} governs this trade-off through structured exploration. Within the scope of our experiments in Chapter 6, at $d = 1$, only $|APP|$ permutations are explored, yielding computation times up to 500 times lower than H_{Full} while producing optimal or near-optimal solutions when edge resources are relatively abundant. As resource pressure increases and placement decisions become more tightly coupled, the ordering of applications in the sequential decomposition grows more consequential, and $d = 2$ consistently closes the optimality gap observed at $d = 1$ at a moderate increase in computation time. In Chapter 7, the sampling ratio p of H_{MCS} governs the same trade-off through random exploration, and exhibits the same qualitative behavior: sampling ratios have negligible impact on solution quality when resources are ample, whereas under scarcity $p = 1/4$ incurs an optimality gap of up to 16% relative to full sampling.

That both heuristics degrade in quality precisely when the feasible space tightens is not coincidental. When resources are abundant, many application orderings lead to equally good solutions and any exploration strategy suffices; when they are scarce, few orderings remain viable and a partial exploration is correspondingly more likely to miss them. This suggests a common deployment strategy for both: operate at a low depth or sampling ratio for routine periodic rescheduling under normal conditions, and increase it during resource scarcity or when solution quality is at a premium.

9.3 Future Work

The contributions of this thesis open several directions for future research, which we organize along four axes.

Alternative Queueing Models for Variable Workloads. The $M|D|1$ model at the core of this thesis owes its accuracy to the deterministic service time assumption, which holds well for the targeted class of applications where processing time is largely independent of request content. As discussed in Chapter 3, this assumption weakens for workloads with highly variable processing times, such as database queries or dynamic content generation, for which the model underestimates queueing delays. A natural extension is to make the queueing model itself a configurable component of the scheduler: applications with near-constant service times would retain the $M|D|1$ model, while applications with higher service time variability would be assigned an $M|M|1$ or, more generally, an $M|G|1$ model parameterized by the empirically measured variance of their service times. Since the entire scheduling pipeline interacts with the queueing model only through the waiting time distribution used in the pre-computation of maximum admissible arrival rates, sub-

stituting the model requires no change to the optimization formulation or the heuristics. The challenge lies in automatically selecting the appropriate model per application, for instance by monitoring the observed service time distribution at runtime, and in quantifying the loss of prediction tightness that more conservative models introduce, since looser tail bounds translate directly into over-provisioning.

Richer Optimization Objectives and Multi-Objective Trade-offs. The optimization objectives considered in this thesis, minimizing slots, monetary cost, or active sites, are representative but far from exhaustive. As discussed in Chapter 6, emerging operational concerns suggest several additional objectives: GPU utilization for accelerator-equipped edge sites, energy consumption under site-level power budgets, and the carbon footprint of the infrastructure, which varies both spatially across sites and temporally with the energy mix. Incorporating such objectives raises two modeling questions. The first is how to express them in the optimization problem: some, like energy, require extending the resource model to a multi-dimensional vector as discussed in Chapter 6, while others, like carbon intensity, introduce time-varying coefficients that couple the scheduling decision to external signals. The second is how to balance multiple objectives that are in tension, for example minimizing carbon footprint may favor distant sites powered by renewable energy at the expense of network latency headroom. Two classical approaches deserve investigation in this context: scalarization through a weighted sum, which preserves the single-objective MILP structure and the heuristics of this thesis but requires careful weight elicitation, and true multi-objective optimization computing Pareto-optimal frontiers, which offers operators an explicit view of the trade-off space at a higher computational cost. The lexicographic structure introduced in Chapter 8 suggests a middle path, where objectives are ranked by operator preference and optimized hierarchically.

Supporting Stateful Applications. The application model of this thesis assumes stateless services, in which any instance can serve any request and instances can be added, removed, or relocated without data movement. Many latency-sensitive applications, however, maintain per-user sessions, caches, or persistent state, which breaks the assumptions underlying both the latency model and the schedulers. Statefulness affects the model first: when requests must be routed to the instance holding the relevant state, the even traffic splitting performed by kube-proxy no longer applies, and the arrival process at each instance becomes workload-dependent and potentially skewed, invalidating the per-instance $M|D|1$ abstraction with uniform arrival rates. It affects scheduling second: relocating an instance is no longer free, as state must be migrated or replicated, introducing transition costs between succes-

sive placement plans that the current formulations, which compute each plan independently, do not capture. A natural extension is to distinguish state-bearing from stateless tiers, model sticky routing through per-instance arrival rates derived from the state partitioning, and augment the optimization objective with migration penalties that make successive placement plans state-aware. Consistency requirements for replicated state would additionally constrain which sites can host instances of the same application, coupling placement decisions across sites in ways that resemble, and could build upon, the affinity relations discussed next.

Scheduling Interconnected Microservices. This thesis deliberately scoped its contributions to stateless, monolithic applications, where any instance can serve any request and instances do not communicate. Extending the framework to interconnected microservices is a substantial but valuable direction, as microservice architectures dominate modern cloud-native development. The extension raises fundamental modeling challenges. The end-to-end latency of a user-facing request becomes the composition of latencies along a call graph spanning multiple services, so the tail latency of the composite depends on the joint distribution of the per-service latencies rather than on any single queue. The placement problem correspondingly changes character: instances of communicating services should be co-located or placed on nearby sites to minimize inter-service network latency, introducing affinity relations between placement decisions that the current per-application formulation does not capture. A promising starting point is to model each microservice with the per-instance queueing approach of this thesis, compose the per-service tail bounds along the call graph using network calculus or moment-based approximations, and extend the MILP with inter-service traffic terms. The heuristic decomposition of Chapter 6 would also need revisiting, since the ordering-based decomposition assumes applications are independent, which no longer holds when services share a call graph.

Learning-Based Acceleration of Placement Computation. The heuristics developed in this thesis reduce scheduling times from hours to minutes or seconds, but they remain iterative search procedures whose cost grows with the problem scale. An appealing complementary direction is to train a machine learning model, for instance a graph neural network operating on the bipartite structure of user groups and sites, on a large corpus of optimal placement plans pre-computed offline by G_{opt} across representative problem instances. At runtime, the trained model would infer a near-optimal placement plan in milliseconds, enabling reaction to workload changes at a timescale that no exact or search-based method can match. This approach inverts the cost structure of the schedulers in this thesis: the expensive optimization is paid once, offline and per problem scope, while the online cost becomes a

single forward pass. Its principal limitations mirror those of learned systems generally. The model is reliable only within the scope of the training distribution, so infrastructure changes such as adding sites or applications require retraining or fine-tuning, whereas the analytical schedulers of this thesis adapt immediately. Moreover, a learned model provides no feasibility guarantee, so its output must be verified against the constraints, using the latency model of Chapter 3, and repaired or recomputed when verification fails. A pragmatic architecture therefore combines the two: the learned model serves as a fast first-line scheduler for in-distribution conditions, the verification step catches its failures, and the heuristics of this thesis serve as the authoritative fallback and as the generator of new training data, closing a virtuous loop between exact optimization, heuristic search, and learned inference. Comparing this hybrid against the pure heuristics of Chapters 6 and 7, in solution quality, reaction time, and robustness to distribution shift, constitutes a complete research agenda in itself.

9.4 Closing Remarks

The edge-cloud continuum will continue to grow in importance as latency-sensitive and AI-driven services proliferate, and with it the tension between the proximity that edge sites offer and the scarcity of the resources they hold. This thesis has shown that this tension can be managed in a principled way: that tail latency can be predicted analytically rather than observed reactively, that placement can be optimized globally rather than negotiated application by application, and that scarcity can be absorbed gracefully rather than suffered unpredictably. We hope that the methodology developed here, and the systems that embody it, contribute a step toward edge infrastructures that honor the performance promises made to their users, even, and especially, when resources run short.

Bibliography

[1] M. Erol-Kantarci and S. Sukhmani, "Caching and computing at the edge for mobile augmented reality and virtual reality (ar/vr) in 5g," in *Ad Hoc Networks: 9th International Conference, AdHocNets 2017, Niagara Falls, ON, Canada, September 28–29, 2017, Proceedings*. Springer, 2018, pp. 169–177.

[2] K. Huang, Z. Tao, C. Wang, T. Guo, C. Yang, and W. Gui, "Cloud-edge collaborative method for industrial process monitoring based on error-triggered dictionary learning," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 12, pp. 8957–8966, 2022.

[3] R. Singh and S. S. Gill, "Edge AI: a survey," *Internet of Things and Cyber-Physical Systems*, vol. 3, 2023.

[4] J. Xia, G. Cheng, D. Guo, and X. Zhou, "A qoe-aware service-enhancement strategy for edge artificial intelligence applications," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9494–9506, 2020.

[5] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. M. Elfadel, "Containerized microservices: A survey of resource management frameworks," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3775–3796, 2024.

[6] A. Bhardwaj and C. R. Krishna, "Virtualization in cloud computing: Moving from hypervisor to containerization—a survey," *Arabian Journal for Science and Engineering*, vol. 46, no. 9, pp. 8585–8601, 2021.

[7] Hivenet, "Hivenet: Distributed cloud storage & computing," https://www.hivenet.com/.

[8] A. Ali-Eldin, B. Wang, and P. Shenoy, "The hidden cost of the edge: a performance comparison of edge and cloud latencies," in *Intl. Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC, 2021.

[9] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *14th USENIX symposium on operating systems design and implementation*, ser. OSDI, 2020.

[10] J. Liu, Q. Wang, S. Zhang, L. Hu, and D. Da Silva, "Sora: A latency sensitive approach for microservice soft resource adaptation," in *24th International Middleware Conference*, 2023.

[11] A. Ali, A. A. Laghari, I. A. Kandhro, K. Kumar, and S. Younus, "Systematic analysis of on-premise and cloud services," *International Journal of Cloud Computing*, vol. 13, no. 3, pp. 214–242, 2024.

[12] B. Wang, D. Irwin, P. Shenoy, and D. Towsley, "INVAR: Inversion aware resource provisioning and workload scheduling for edge computing," in *IEEE International Conference on Computer Communications*, ser. INFOCOM, 2024.

[13] K.-H. Chow, U. Deshpande, V. Deenadayalan, S. Seshadri, and L. Liu, "Atlas: Hybrid cloud migration advisor for interactive microservices," in *19th ACM European Conference on Computer Systems*, ser. EuroSys, 2024.

[14] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. Ramakrishnan, and T. Wood, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *ACM Symposium on Cloud Computing*, ser. SoCC, 2021.

[15] F. Caglar, S. Shekhar, and A. Gokhale, "iplace: An intelligent and tunable power-and performance-aware virtual machine placement technique for cloud-based real-time applications," in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 2014, pp. 48–55.

[16] T. Pusztai, S. Nastic, A. Morichetta, V. C. Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polaris scheduler: Slo- and topology-aware microservices scheduling at the edge," in *IEEE/ACM 15th International Conference on Utility and Cloud Computing*, ser. UCC, 2022.

[17] G. Castellano, S. Galantino, F. Risso, and A. Manzalini, "Scheduling multi-component applications across federated edge clusters with Phare," *Open Journal of the Communications Society*, 2024.

[18] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, "Tailored learning-based scheduling for Kubernetes-oriented Edge-Cloud system," in *IEEE conference on computer communications*, ser. INFOCOM, 2021.

[19] R. Bhardwaj, K. Kandasamy, A. Biswal, W. Guo, B. Hindman, J. Gonzalez, M. Jordan, and I. Stoica, "Cilantro: Performance-Aware resource allocation for general objectives via online feedback," in *17th USENIX*

Symposium on Operating Systems Design and Implementation, ser. OSDI, 2023.

[20] Kubernetes horizontal pod autoscaling. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[21] S. Shen, Y. Feng, M. Xu, Y. Ren, X. Wang, V. C. Leung, and W. Wang, "Tango: Harmonious optimization for mixed services in kubernetes-based edge clouds," *IEEE Transactions on Services Computing*, 2024.

[22] A. A. Amer, I. E. Talkhan, R. Ahmed, and T. Ismail, "An optimized collaborative scheduling algorithm for prioritized tasks with shared resources in mobile-edge and cloud computing systems," *Mobile Networks and Applications*, vol. 27, no. 4, pp. 1444–1460, 2022.

[23] K. Agrawal and S. Abdu Jyothi, "Cooperative graceful degradation in containerized clouds," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 214–232.

[24] L. Tomás, C. Vázquez, J. Tordsson, and G. Moreno, "Reducing noisy-neighbor impact with a fuzzy affinity-aware scheduler," in *2015 International Conference on Cloud and Autonomic Computing*. IEEE, 2015, pp. 33–44.

[25] M. Bensalem, J. Dizdarević, and A. Jukan, "DNN placement and inference in edge computing," in *43rd International Convention on Information, Communication and Electronic Technology*, ser. MIPRO, 2020, pp. 479–484.

[26] J. J. Meza, T. Gowda, A. Eid, T. Ijaware *et al.*, "Defcon: Preventing overload with graceful feature degradation," in *17th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2023, pp. 607–622.

[27] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–37, 2017.

[28] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367.

[29] R.-A. Cherrueau, M. Delavergne, A. Van Kempen, A. Lebre, D. Pertin, J. R. Balderrama, A. Simonet, and M. Simonin, "Enoslib: A library for experiment-driven research in distributed computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1464–1477, 2021.

[30] Kube-proxy documentation. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/

[31] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *IEEE/IFIP Network Operations and Management Symposium*, ser. NOMS. IEEE, 2018.

[32] Microsoft Learn. (2023) Azure network round-trip latency statistics. [Online]. Available: https://learn.microsoft.com/en-us/azure/networking/azure-network-latency?tabs=Americas%2CWestUS

[33] B. Jamil, H. Ijaz, M. Shojafar, K. Munir, and R. Buyya, "Resource allocation and task scheduling in fog computing and internet of everything environments: A taxonomy, review, and future directions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, 2022.

[34] M. Bouet and V. Conan, "Mobile edge computing resources optimization: A geo-clustering approach," *IEEE Transactions on Network and Service Management*, vol. 15, no. 2, pp. 787–796, 2018.

[35] S. Lee, Z. Guo, O. Sunercan, J. Ying, T. Kooburat, S. Biswal, J. Chen, K. Huang, Y. Cheung, Y. Zhou *et al.*, "Shard manager: A generic shard management framework for geo-distributed applications," in *ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP, 2021.

[36] D. Chou, T. Xu, K. Veeraraghavan, A. Newell, S. Margulis, L. Xiao, P. M. Ruiz, J. Meza, K. Ha, S. Padmanabha *et al.*, "Taiji: managing global user traffic for large-scale internet services at the edge," in *ACM SIGOPS 27th ACM symposium on operating systems principles*, ser. SOSP, 2019.

[37] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 15–26, 2004.

[38] Q. M. Tran, P. H. Nguyen, T. Tsuchiya, and M. Toulouse, "Designed features for improving openness, scalability and programmability in the fog computing-based iot systems," *SN Computer Science*, vol. 1, no. 4, 2020.

[39] B. Charyyev, E. Arslan, and M. H. Gunes, "Latency comparison of cloud datacenters and edge servers," in *IEEE GLOBECOM Global Communications Conference*, 2020.

[40] H. Zhang, S. Huang, M. Xu, D. Guo, X. Wang, V. C. Leung, and W. Wang, "How far have edge clouds gone? a spatial-temporal analysis of edge network latency in the wild," in *IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*, 2023.

[41] I. Miell and A. Sayers, *Docker in practice*. Simon and Schuster, 2019.

[42] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[43] Docker: The leading container runtime. [Online]. Available: https://www.docker.com/products/container-runtime/

[44] Open container initiative. [Online]. Available: https://opencontainers.org/

[45] Michael Kerrisk, "Linux/unix system programming training - namespaces." [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html

[46] ——, "Linux/unix system programming training - cgroups." [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html

[47] Kubernetes: Production-grade container orchestration. [Online]. Available: https://kubernetes.io/

[48] Kubernetes Documentation, "Kubernetes pods." [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/

[49] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, 2013.

[50] Z. Fan, E. Song, B. Jiang, Y. Song, Y. Hong, B. Yang, Y. Lv, Y. Zhou, J. Cai, C. Wang *et al.*, "Understanding the long tail latency of tcp in large-scale cloud networks," in *Proceedings of the 9th Asia-Pacific Workshop on Networking*, 2025, pp. 106–113.

[51] P. Veitch, E. Curley, and T. Kantecki, "Performance evaluation of cache allocation technology for nfv noisy neighbor mitigation," in *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017.

[52] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," in *ACM conference on Computer and communications security*, ser. CCS, 2012.

[53] S. Akundi, S. Prabhu, N. U. BK, and S. C. Mondal, "Suppressing noisy neighbours in 5g networks: An end-to-end nfv-based framework to detect and suppress noisy neighbours," in *Proceedings of the 21st International Conference on Distributed Computing and Networking*, 2020.

[54] Z. Zhao, Y. Hu, G. Yang, Z. Gong, C. Shen, L. Zhao, W. Li, X. Liu, and W. Qu, "Slopt: Serving real-time inference pipeline with strict latency constraint," *IEEE Transactions on Computers*, vol. 74, no. 4, pp. 1431–1445, 2025.

[55] X. Zhang and D. Kim, "Enabling slo-aware 5g multi-access edge computing with smec," *arXiv preprint arXiv:2601.19162*, 2026.

[56] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[57] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of field robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[58] M. Adeppady, P. Giaccone, H. Karl, and C. F. Chiasserini, "Reducing microservices interference and deployment time in resource-constrained cloud systems," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, 2023.

[59] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott, "Oakestra: A lightweight hierarchical orchestration framework for edge computing," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 215–231.

[60] T. Pfandzelter and D. Bermbach, "Komet: A serverless platform for low-earth orbit edge services," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024, pp. 866–882.

[61] J. Zhao, Y. Xia, X. Sun, T. Long, Q. Peng, S. Guo, F. Meng, Y. Dong, and Q. Xia, "Ets-ddpg: an energy-efficient and qos-guaranteed edge task scheduling approach based on deep reinforcement learning," *Wireless Networks*, vol. 31, no. 2, pp. 1241–1254, 2025.

[62] H. Zou, J. Guo, J. Zeng, Y. Li, J. Cao, and T. Wang, "Fine-grained service lifetime optimization for energy-constrained edge-edge collaboration," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2024, pp. 565–576.

[63] L. Kleinrock, *Queueing systems: theory*. wiley, 1974, vol. 2.

[64] M. Harchol-Balter, "Open problems in queueing theory inspired by datacenter computing," *Queueing Systems*, vol. 97, no. 1, pp. 3–37, 2021.

[65] D. Terekhov, T. T. Tran, D. G. Down, and J. C. Beck, "Integrating queueing theory and scheduling for dynamic scheduling problems," *Journal of Artificial Intelligence Research*, vol. 50, pp. 535–572, 2014.

[66] J. Prados-Garzon, P. Ameigeiras, J. J. Ramos-Munoz, J. Navarro-Ortiz, P. Andres-Maldonado, and J. M. Lopez-Soler, "Performance modeling of softwarized network services based on queuing theory with experimental validation," *IEEE Transactions on Mobile Computing*, vol. 20, no. 4, pp. 1558–1573, 2019.

[67] R. Nelson, *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling*. Springer Science & Business Media, 2013.

[68] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain," *The Annals of Mathematical Statistics*, pp. 338–354, 1953.

[69] A. K. Erlang, "Sandsynlighedsregning og telefonsamtaler," *Nyt tidsskrift for Matematik*, vol. 20, 1909.

[70] Kubernetes Documentation, "Kubernetes cpu management policies." [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/

[71] Michael Kerrisk, "Linux/unix system programming training - tc." [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[72] B. Donnet, B. Gueye, and M. A. Kaafar, "A survey on network coordinates systems, design, and security," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 4, 2010.

[73] C. Lumezanu, R. Baden, N. Spring, and B. Bhattacharjee, "Triangle inequality variations in the internet," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, 2009, pp. 177–183.

[74] Michael Kerrisk, "Linux/unix system programming training - veth." [Online]. Available: https://man7.org/linux/man-pages/man4/veth.4.html

[75] ——, "Linux/unix system programming training - tc-netem." [Online]. Available: https://man7.org/linux/man-pages/man8/tc-netem.8.html

[76] ——, "Linux/unix system programming training - iptables." [Online]. Available: https://man7.org/linux/man-pages/man8/iptables.8.html

[77] D. Mosberger and T. Jin, "httperf—a tool for measuring web server performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 3, dec 1998.

[78] Michael Kerrisk, "Linux/unix system programming training - taskset." [Online]. Available: https://man7.org/linux/man-pages/man1/taskset.1.html

[79] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the tenth european conference on computer systems*, 2015, pp. 1–17.

[80] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[81] A. Kay, "Tesseract: an open-source optical character recognition engine," *Linux J.*, vol. 2007, no. 159, jul 2007.

[82] EasyNMT. [Online]. Available: https://github.com/UKPLab/EasyNMT

[83] J. Tiedemann and S. Thottingal, "Opus-mt–building open translation services for the world," in *Proceedings of the 22nd annual conference of the European Association for Machine Translation*, 2020, pp. 479–480.

[84] Brent. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brent.html

[85] H. Kellerer, U. Pferschy, and D. Pisinger, "Introduction to np-completeness of knapsack problems," in *Knapsack problems*. Springer, 2004, pp. 483–493.

[86] Gurobi Optimizer. [Online]. Available: https://www.gurobi.com/

[87] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *39th International Conference on Distributed Computing Systems*, ser. ICDCS. IEEE, 2019.

[88] Q. Wu, M. Zhou, and J. Wen, "Endpoint communication contention-aware cloud workflow scheduling," *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 2, 2021.

[89] Q. Zhang, Q. Wu, M. Zhou, J. Wen, and S. Yao, "A communication contention-cognizant scheduling approach for workflow execution across public and private clouds," *IEEE Transactions on Automation Science and Engineering*, vol. 21, no. 4, 2023.

[90] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin, Heidelberg: Springer, 2004.

[91] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Upper Saddle River, NJ: Pearson, 2020.

[92] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE International Real-Time Systems Symposium (RTSS)*, 2007.

[93] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu, "Mixed-criticality federated scheduling for parallel real-time tasks," *Real-time systems*, vol. 53, no. 5, pp. 760–811, 2017.

[94] Z. Sun, M. Sudvarg, and C. Gill, "Elastic scheduling for graceful degradation of mixed-criticality systems," in *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, ser. RTNS, 2024, pp. 218–228.

[95] V. K. Sundar and A. Easwaran, "A practical degradation model for mixed-criticality systems," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing*, ser. ISORC. IEEE, 2019, pp. 171–180.

[96] J. Zou, X. Dai, and J. A. McDermid, "Context-aware graceful degradation for mixed-criticality scheduling in autonomous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 3, pp. 788–801, 2023.

[97] A. Shapiro, "Monte carlo sampling methods," *Handbooks in operations research and management science*, vol. 10, pp. 353–425, 2003.

[98] B. Li, S. Samsi, V. Gadepally, and D. Tiwari, "Clover: Toward sustainable AI with carbon-aware machine learning inference service," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC. New York, NY, USA: Association for Computing Machinery, 2023.

[99] S. P. Rachuri, N. Shaik, M. Choksi, and A. Gandhi, "Ecoedgeinfer: Dynamically optimizing latency and sustainability for inference on edge devices," in *2024 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2024, pp. 191–205.

[100] T. Pusztai, C. Marcelino, and S. Nastic, "Hyperdrive: Scheduling serverless functions in the edge-cloud-space 3d continuum," in *2024 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2024, pp. 265–278.

[101] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *USENIX Annual Technical Conference*, ser. ATC, 2021, pp. 397–411.

[102] M. Salmani, S. Ghafouri, A. Sanaee, K. Razavi, M. Mühlhäuser *et al.*, "Reconciling high accuracy, cost-efficiency, and low latency of inference serving systems," in *3rd Workshop on Machine Learning and Systems*, ser. EuroMLSys, 2023, pp. 78–86.

[103] M. Yuan, L. Zhang, D. Duan, L. Zeng, M.-H. Song *et al.*, "Mitigating tail latency for on-device inference with load-balanced heterogeneous models," *IEEE Transactions on Mobile Computing*, 2025.

[104] S. S. Ogden and T. Guo, "Layercake: Efficient inference serving with cloud and mobile resources," in *IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing*, ser. CCGrid. IEEE, 2023, pp. 191–202.

[105] L. Wu, W. Hanafy, T. Abdelzaher, D. Irwin *et al.*, "Faillite: Failure-resilient model serving for resource-constrained edge environments," in *Proceedings of the 2025 ACM Symposium on Cloud Computing*, 2025, pp. 416–429.

[106] PyTorch. (2026) ResNet Models. [Online]. Available: https://docs.pytorch.org/vision/stable/models/resnet.html

[107] Ultralytics. (2026) YOLOv8. [Online]. Available: https://docs.ultralytics.com/models/yolov8/

[108] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, ser. ICML. PMLR, 2019, pp. 6105–6114.

[109] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[110] P. Ntumba, N.-E.-H. Yellas, S. Bin-Ruba, S. Secci *et al.*, "Data pipeline system designs for in-network learning," in *20th International Conference on Network and Service Management*. IEEE, 2024, pp. 1–9.

[111] L. Wu, W. A. Hanafy, A. Souza, K. Nguyen, J. Harkes, D. Irwin, M. Satyanarayanan, and P. Shenoy, "Carbonedge: Leveraging mesoscale spatial carbon-intensity variations for low carbon edge computing," in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, 2025, pp. 1–13.

[112] CPLEX Optimizer. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio

[113] SCIP Optimizer. [Online]. Available: https://www.scipopt.org/

[114] Knitro Optimizer. [Online]. Available: https://www.artelys.com/solvers/knitro/

[115] Google OR-Tools. [Online]. Available: https://developers.google.com/optimization