Lancer: A Multi-Objective Framework for Container Provisioning in the Edge-to-Cloud Continuum

Nicolas Flores Feijó, Paulo Silas Severo de Souza (advisor)

[1]Federal University of Pampa — Alegrete Campus
Tiaraju Avenue, 810 — Ibirapuitã, Alegrete - RS, 97546-550

{nicolasfeijo.aluno, paulosilas}@unipampa.edu.br

Abstract. *Containerized workloads have become very popular by delivering rapid scalability and alleviating network load during deployments through a layered image file system that enables multiple containers to share common dependencies. The benefits of containerization are particularly significant in distributed infrastructures, such as the Edge-to-Cloud Continuum, where the network can be unstable and computational resources are heterogeneous and geographically dispersed. Given the promising match between containers and Edge-to-Cloud infrastructures, several studies have explored the management of containerized workloads in such environments. However, these works often treat container provisioning as a single decision, overlooking that resource management involves multiple interrelated decisions. For instance, when performance issues arise, where should containerized applications be allocated to improve performance? Once an allocation decision is made, through which network paths should the necessary data be transferred? After deciding the paths, how should bandwidth be allocated, especially when multiple competing flows are present? To address these challenges, we propose Lancer, a framework that uses multi-objective optimization to jointly make these interrelated decisions. Simulation results show that Lancer's decision mechanisms independently enhance the performance of established policies such as the Kubernetes Scheduler. When combined, they yield even larger gains, optimizing application performance and the infrastructure's energy consumption by 60,17% and 6,8%, respectively, over the baseline approaches.*

Resumo. *As cargas de trabalho conteinerizadas tornaram-se amplamente populares por oferecerem rápida escalabilidade e por reduzirem a carga de rede durante os processos de implantação, por meio de um sistema de arquivos de imagens em camadas que permite que múltiplos contêineres compartilhem dependências comuns. Os benefícios da conteinerização são particularmente relevantes em infraestruturas distribuídas, como o Edge-to-Cloud Continuum, nas quais a rede pode ser instável e os recursos computacionais são heterogêneos e geograficamente dispersos. Diante da afinidade promissora entre contêineres e infraestruturas Edge-to-Cloud, diversos estudos têm investigado o gerenciamento de cargas de trabalho conteinerizadas nesses ambientes. Contudo, tais trabalhos frequentemente tratam o provisionamento de contêineres como uma decisão única, negligenciando o fato de que o gerenciamento de recursos envolve múltiplas decisões inter-relacionadas. Por exemplo, quando surgem problemas de desempenho, onde as aplicações conteinerizadas devem ser alocadas para melhorar o desempenho? Uma vez tomada a decisão de alocação, por quais caminhos de rede os dados necessários devem ser transferidos? Após a definição dos caminhos, como a largura de banda deve ser alocada, especialmente na presença de múltiplos fluxos concorrentes? Para enfrentar esses desafios, propomos o Lancer, um* framework *que utiliza otimização multiobjetivo para tomar, de forma conjunta, essas decisões inter-relacionadas. Resultados de simulação demonstram que os mecanismos de decisão do Lancer melhoram de forma independente o desempenho de políticas consolidadas, como o* Kubernetes Scheduler. *Quando combinados, esses mecanismos proporcionam ganhos ainda mais expressivos, otimizando o desempenho das aplicações e o consumo energético da infraestrutura em 60,17% e 6,8%, respectivamente, em relação às abordagens de referência.*

1. Introduction

In the modern digital landscape, software applications are becoming increasingly complex and distributed, requiring advanced infrastructure solutions to meet performance and reliability demands. Over the past decade, Cloud Computing has emerged as the dominant paradigm for hosting applications, driven by its ability to provide on-demand resource provisioning and high scalability. This popularity also stems from key underlying technologies, most notably virtualization and, more recently, containerization. Virtualization abstracts physical hardware, allowing multiple virtual machines to run on a single physical server, thereby improving resource utilization and isolation. Containers, on the other hand, provide a lightweight alternative, delivering portable and efficient execution environments with faster deployment times and lower overhead.

Despite the advantages of Cloud Computing, as software applications have evolved, emerging workloads have begun to present stringent latency requirements. As a result, the downsides of the cloud's centralized model have become more apparent. The physical distance between data centers and end-users has become a significant limitation for meeting expected performance levels, particularly for applications that require real-time processing or low-latency interactions. In response to these limitations, the Edge Computing paradigm has emerged. Aiming to reduce overall latency and improve user experience, Edge Computing extends the cloud by strategically delivering computational resources closer to end-users.

However, while Edge Computing addresses latency-related challenges, it also introduces its own set of limitations, particularly in terms of computational resources. Edge devices often have constrained processing power, memory, and storage capabilities compared to traditional cloud data centers. This resource shortage can hinder certain types of applications from being fully deployed at the edge, leading to the need for more integrated approaches. In this context, Edge and Cloud Computing are seen as complementary paradigms, enabling the development of a layered architecture known as the Edge-to-Cloud Continuum. This continuum is structured as a hierarchical model containing multiple layers of computing resources.

In order to support the deployment of applications across the Edge-to-Cloud Continuum, several technologies from cloud computing environments were inherited. Among these, container-based virtualization plays a prominent role by enabling efficient resource utilization and rapid application deployment. Moreover, software architectures that promote horizontal scalability, such as microservices, are particularly well-suited for the dynamic and distributed nature of this continuum. The combination of microservices and containerization has proven especially effective, as containers can encapsulate individual application services along with their dependencies, ensuring consistent execution across heterogeneous infrastructure layers.

Even so, effectively allocating applications across the multiple layers of the Edge-to-Cloud Continuum presents a diverse set of challenges. The decision-making process must consider not only the performance demands of applications and the overall user experience but also the interests of infrastructure providers, such as energy consumption and operational costs. This problem becomes even more challenging when considering factors like user mobility, heterogeneous application performance requirements, and the dynamic nature of the infrastructure.

Hence, application provisioning in the Edge-to-Cloud Continuum becomes a non-trivial multi-objective optimization problem. As the scale of the environment increases, the dimensionality of the search space also grows significantly, making traditional solutions less effective. Therefore, the development of strategies and algorithms capable of handling the complexity of this continuum is a crucial task. These algorithms must address the specific constraints and requirements of each application while also considering the dynamic nature of the Edge-to-Cloud Continuum.

Despite the growing interest in this field, existing solutions often focus on isolated aspects of the problem, such as provisioning strategies or network management techniques. However, these approaches frequently overlook the potential of holistic methods that integrate multiple interdependent components to achieve better overall performance. In this context, this study proposes Lancer, a framework that combines complementary decision mechanisms to enhance resource management in Edge-to-Cloud computing environments. We argue that integrating these techniques can lead to significant improvements in both application performance and power efficiency.

The rest of this study is organized as follows: Section 2 provides background on the Edge-to-Cloud Continuum and related areas. Section 3 presents the related work. Section 4 details the design of the Lancer framework and its main components. Section 5 describes the performance evaluation methodology and discusses the results. Finally, Section 6 concludes this work and outlines future research directions.

2. Background

The popularization of software applications across many sectors of society has led to the emergence of use cases with strict performance requirements (e.g., autonomous vehicles [1], augmented/virtual reality [2], etc.). In response, certain architectural trends have gained popularity. One such trend is the decoupling of hardware and software layers, which provides greater flexibility in allocating computational resources, enabling multi-tenancy and dynamic application allocation [3] [4].

Another significant trend is the modularization of applications, in which software artifacts are developed as independent components that communicate via specific network protocols (e.g., gRPC[1], REST[2], etc.). This component-based development approach has shown several advantages, including increased flexibility for development teams (which gain the freedom to use different software stacks among components) and selective scalability by allowing specific parts of the system to be scaled according to demand (rather than scaling the entire system uniformly) [5].

The ideas of decoupling hardware and software and modularizing applications are generic enough not to be tied to specific technologies. However, some technologies have naturally stood out in practice to enable these architectural trends. One example is virtualization [6], which introduces abstraction layers between hardware and software, allowing multiple isolated applications to run on the same physical machine.

Virtualization is a key technology for many modern computing models, particularly in distributed infrastructures such as the Edge-to-Cloud Continuum [7], where re-

[1] `https://grpc.io/`

[2] `https://developer.mozilla.org/en-US/docs/Glossary/REST`

sources are heterogeneous, geographically dispersed, and interconnected through shared networks. In this context, Virtual Machines (VMs) and containers are often the dominant virtualization technologies [4].

In addition to enabling the execution of multiple co-hosted applications on the same physical hardware, VMs and containers share the idea of using templates (so-called images). Images contain the binaries and dependencies required by a certain application, allowing multiple instances to be spawned with the same configuration. Although both VM and container images serve the same purpose of facilitating application provisioning, there are important differences in how they operate. VM images typically use a monolithic approach, bundling all necessary components into a single file. In practice, even if two co-hosted VMs share most of their dependencies, even slight differences between them require creating two complete, independent images that do not share common components. The lack of component sharing impacts both provisioning time (since complete images need to be transferred for each instance) and disk space usage (since common components are duplicated on the host).

Unlike VM images, container images usually use a multi-layered file system, where the image content is split into multiple layers that can be independently manipulated by the container system. In practice, the layered file system allows multiple co-hosted containers to share common dependencies (i.e., container layers). As a result, there are two main impacts of using container images instead of VM images. First, faster provisioning times can be achieved when multiple containers share dependencies, as duplicate layer transfers can be avoided. Second, lower disk space demand is observed, as common components are stored only once.

Container images are typically pulled from specialized applications (so-called container registries) that manage image distribution and storage. In practice, when a containerized application needs to be provisioned, the container platform selects a target host, which then requests the required image from a registry. When a containerized application already running on a host needs to be relocated to another host, the process involves the container platform defining the new host for the application and the new host requesting the required image from the registry.

Allocation decisions (e.g., relocating an application from server A to server B) are often triggered to mitigate performance issues (e.g., high delays due to user mobility or resource contention). In such cases, the time required to complete provisioning and data movement directly affects how quickly the system can recover to an acceptable performance state. Although container platforms expose abstractions that simplify dynamic allocation, improving provisioning efficiency is still challenging, particularly when relocation involves moving an application across hosts connected through shared, capacity-limited network links, which is a common scenario in heterogeneous infrastructures such as the Edge-to-Cloud Continuum.

In practice, container relocation comprises multiple interrelated decisions that should be orchestrated altogether. First, what server should be selected as the new host for the containerized application? Neglecting this decision may lead to poor placements that exacerbate performance issues (e.g., relocating an application to a host with limited network connectivity). Once a target host is selected, through which network paths should

the image be transferred from the registry to that host? Poor path selection may lead to transfers over congested links, resulting in prolonged provisioning times. Finally, how should network resources (e.g., bandwidth) be managed for the triggered transfers? Inefficient traffic management may lead to bottlenecks and unfair bandwidth sharing among competing flows. Based on these observations, allocation decisions should not be treated in isolation, as they are interrelated and collectively impact the overall efficiency of the provisioning process.

3. Related Work

In this section, we present a review of the literature findings related to layer-aware microservice management in Edge-to-Cloud Continuum environments. At the end of this section, we summarize the main contributions of these works and identify the gaps that our research aims to address.

Temp et al. [8] proposed a mobility-aware container registry placement strategy that dynamically reallocates container registries whenever provisioning times exceed a given threshold. Moreover, their approach also considers inactive container registries, deprovisioning them to reduce overall resource waste. The authors presented a simulation-based evaluation of their approach, demonstrating that it can reduce average provisioning time delay by 33.19% compared to static registry placement strategies.

Building on this idea, the MAPER strategy, proposed by Temp et al. [9], extended the previous work by incorporating both user mobility and energy efficiency awareness into the container registry management process. To the best of our knowledge, this was the first study to explicitly combine these two optimization goals in edge container registry placement. The study introduced a multi-objective optimization model that performs a two-level sorting process to select the best servers capable of providing the lowest delay while consuming less energy. The authors conducted a simulation-based evaluation, where MAPER outperformed baseline strategies, achieving a 60.03% reduction in average application delay issues.

Tang et al. [10] proposed the Layer Dependency-aware Learning Scheduler (LDLS), a layer-aware strategy that addresses container task scheduling in Mobile Edge Computing (MEC) environments. LDLS uses a reinforcement learning approach combined with factorization machines to capture hidden dependencies among container layers and tasks. By considering container layer granularity, LDLS reduced the average task completion time by 54% compared to baseline strategies that used image-level granularity. The authors conducted evaluations using real-world datasets, demonstrating the effectiveness of their approach in optimizing task scheduling in MEC environments.

To address load imbalance and inefficient resource utilization in Kubernetes-based edge environments, Peng et al. [11] introduced LR^{2}Scheduler, a layer-aware, resource-balanced, and request-adaptive scheduling strategy. LR^{2}Scheduler employs a dynamic scoring mechanism with two levels. First, it implements a node scoring based on container image layer information, enabling the algorithm to select the most suitable nodes. Additionally, it uses a scoring system that adapts to resource demands based on user requests and resource information. With this approach, LR^{2}Scheduler reduced the number of nodes with resource usage above 80% by 50% compared to the default Kubernetes scheduler. In addition, the algorithm also reduced the average download time by 47%.

Shi et al. [12] discusses the challenges associated with edge server failures, mainly focusing on how microservice deployment optimizations can lead to higher system-wide reliability. To address this problem, the authors proposed a two-timescale online management framework that combines reliability enhancements with layer-sharing strategies, enabling better fault tolerance during deployment decisions. Theoretical analysis and extensive simulation-based evaluations demonstrated that the proposed framework achieved a 12.4% increase in deployment reliability and a 28.57% reduction in total service delay when compared to existing counterpart solutions.

Aiming to reduce overall task latency in edge-assisted vehicular networks, Tang et al. [13] developed an Online Container Migration algorithm. This approach considers a multi-user and layer-sharing scenario, where multiple users can request tasks with common layers. OCM is a multi-user and layer-aware container migration algorithm that applies deep reinforcement learning based on policy gradients to make online container migration decisions. Using real-world data traces, the algorithm reduced total task latency by 8% to 30% on average compared to baseline strategies.

Wang et al. [14] tackled the problem of communication overhead between microservices and controllers in traditional single-controller service mesh architectures. The authors introduced a multi-controller service mesh architecture coupled with a novel RIME-ACPO algorithm to optimize service placement decisions. This solution aims to reduce overall system communication latency and deployment cost while dynamically adapting to resource utilization in the edge infrastructure. Experimental results showed that RIME-ACPO consistently outperformed baseline strategies, achieving a 3.03% reduction in user response time and a 0.33% decrease in deployment cost.

In order to improve network bandwidth allocation in Edge environments, Garcia et al. [15] proposed Service-Level Weighted Queuing (SLWQ), a traffic-shaping strategy that expands upon the concept of Weighted Fair Queuing (WFQ). SLWQ assigns weights to network flows based on high-level metrics - such as application SLA - allowing allocations to be adjusted according to specific requirements of each application. The experiments conducted by the authors demonstrated that SLWQ provides faster provisioning times for high-priority applications, overperforming baseline strategies as Equal Share, Max-Min Fairness and Standard WFQ.

Finally, Zeng et al. [16] defined a Profit-Driven Mixed-Integer Nonlinear Programming problem and proposed a Graph-Aware Hybrid Reinforcement Learning algorithm. The proposed approach targets the joint optimization of containerized service placement and resource allocation. GAHRL explicitly considers container image layer sharing and combines two deep reinforcement learning strategies: the Deep Deterministic Policy Gradient network and the Dueling Deep Q-Network. Experiments showed that the proposed algorithm outperformed baseline strategies in maximizing revenue as well as reducing service latency and storage cost.

Despite the significant contributions of these works, most of them focus on one isolated aspect of containerized workload management, such as placement or network shaping. However, application provisioning often involves multiple interrelated decisions that can impact overall performance. In summary, we believe that presented solutions lack holistic approaches that consider multiple interdependent decisions in microservice

provisioning. To address this gap, we propose Lancer, a comprehensive framework that integrates relocation policies, path selection algorithms, and traffic shaping strategies. By combining these mechanisms, Lancer aims to manage containerized workloads in Edge-to-Cloud Continuum environments more effectively than isolated strategies. A comparison of the main contributions of the reviewed works is presented in Table 1.

Table 1. Related work comparison.

Work	Resource Management Task			Target Metrics	
	Positioning	Path Finding	Traffic Shaping	SLA Compliance	Power Saving
Temp et al. [8]	✓	✗	✗	✓	✗
Tang et al. [10]	✓	✗	✗	✗	✗
Shi et al. [12]	✓	✗	✗	✗	✗
Tang et al. [13]	✓	✗	✗	✓	✗
Wang et al. [14]	✓	✗	✗	✓	✓
Temp et al. [9]	✓	✗	✗	✓	✓
Peng at al. [11]	✓	✗	✗	✗	✗
Zeng et al. [16]	✓	✗	✗	✓	✗
Garcia et al. [15]	✗	✗	✓	✓	✗
Our Work	✓	✓	✓	✓	✓

4. Lancer Design

This section presents Lancer, a framework designed to enhance resource management in Edge-to-Cloud computing environments. We combine multiple heuristic strategies, Lancer jointly reduces SLA violations and energy consumption.

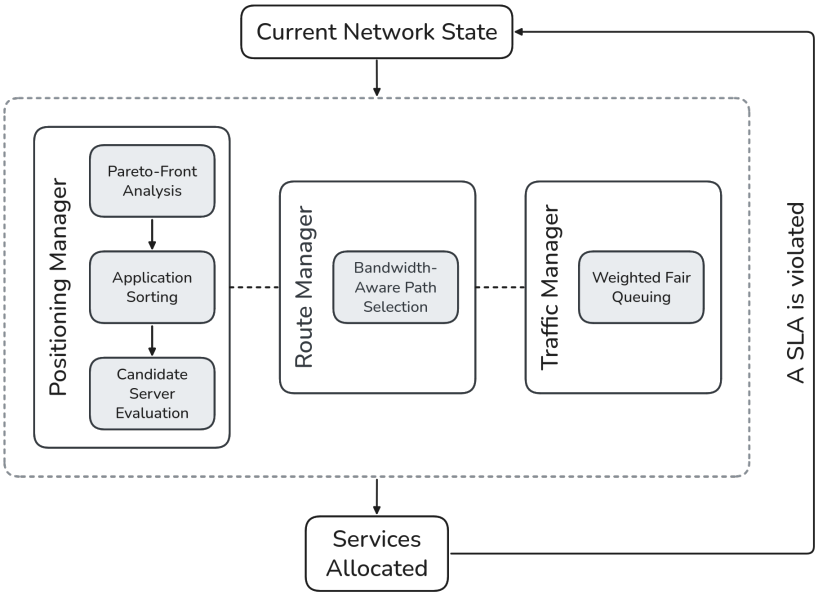


Figure 1. Schematic overview of the Lancer framework.

Lancer operates on three main components (Figure 1): a Positioning Manager that relies on Pareto-front analysis for multi-objective decision-making; a Route Manager that uses a modified Dijkstra algorithm to compute widest paths for container image transfers; and a Traffic Manager module that employs a WFQ strategy to regulate network traffic.

In the remainder of this section, we describe each of Lancer's components.

4.1. Positioning Manager

The Lancer Positioning Manager (LPM) is primarily responsible for application placement and relocation decisions. It employs multi-objective optimization based on Pareto-front analysis to evaluate candidate placements that optimize several attributes simultaneously. Dominance relations are used for this evaluation: one candidate dominates another if it is no worse in all objectives and strictly better in at least one.

A relocation process is triggered only when specific conditions are met. First, an application must be identified as violating its SLA requirements, which can be detected through monitoring mechanisms that track latency metrics. Additionally, the target host must have sufficient resources to accommodate the application and must be different from the current host. Once these requirements are satisfied, the Positioning Manager initiates the relocation procedure.

The Positioning Manager follows a depth-based approach in which applications are provisioned sequentially. Consequently, all services of a given application must be placed before moving to the next. Because resources at the edge are limited, the order in which applications are processed may affect placement quality. Therefore, after identifying all applications violating their SLA thresholds, Lancer's Positioning Manager sorts them according to two criteria:

- **SLA violation severity**: Applications exhibiting higher SLA degradation are less likely to meet their performance requirements. Accordingly, prioritizing their relocation can lead to faster system stabilization.
- **Container layer size**: Applications with larger image sizes naturally favor the propagation of container layers in the network. Consequently, relocating these applications first can lead to more efficient use of network resources and reduced overall transfer times.

Once the relocation order is established, the Positioning Manager evaluates potential placement options for each service of the application. This evaluation ranks candidate hosts according to four minimization criteria:

- **Delay for previous service in the chain**: Minimizing delays between dependent services is crucial for maintaining low end-to-end latency, as such delays accumulate along the chain.
- **Pending layers**: Deploying services on hosts that already contain some of the required container layers reduces deployment time and network overhead.
- **Deployment queue length**: Container runtimes limit the number of concurrent deployments they can process; thus, hosts with shorter deployment queues typically offer faster setup times.
- **Incremental power consumption**: Deploying a new service increases the host's power usage. Hosts with lower incremental consumption are preferred to improve power efficiency.

These criteria are used to compute the dominance count for each candidate host, guiding the selection of the most suitable placement. By evaluating hosts through this multi-objective process, Lancer Pareto manages resource allocation while balancing performance and power efficiency.

4.2. Route Manager

The Lancer Route Manager (LRM) algorithm is responsible for determining the path used to transfer container images between hosts during application deployment or relocation. A simplistic approach to handle this problem consists in employing a Dijkstra's algorithm to compute the shortest path based on link delays. The primary advantage of this method lies in its simplicity and its ability to quickly identify low-latency routes, which can minimize transfer times under low-traffic conditions.

However, in high-traffic scenarios, shortest-path routes may become congested, resulting in increased transfer times and potential SLA violations. Moreover, shortest-path algorithms do not always account for the available bandwidth along the path, an essential factor for large data transfers such as container images. To mitigate these limitations, Lancer incorporates a Bandwidth-Aware (BWA) path-selection strategy that focuses on identifying the path with maximum available bandwidth, commonly referred to as the widest path.

The Bandwidth-Aware Route Manager is a modified version of Dijkstra's algorithm that uses available bandwidth as the primary metric. Instead of minimizing cumulative delay, the algorithm maximizes the minimum bandwidth along the candidate path. This strategy prioritizes routes capable of accommodating large data transfers more efficiently, thereby reducing congestion and improving overall data transfer performance.

4.3. Traffic Manager

The Lancer Traffic Manager (LTM) is responsible for distributing network bandwidth among concurrent flows sharing the same link. Without proper management, some flows may dominate the link, resulting in unfair bandwidth distribution and potentially slowing down the deployment or relocation of other applications. A simplistic way to determine the bandwidth distribution among concurrent flows is giving them an equal bandwidth share or leveraging fairness-oriented algorithms such as the Max-Min Fairness [17], that distributes bandwidth proportionally to the size of each flow.

While these strategies provide a baseline level of fairness, they lack the ability to prioritize flows according to their importance or urgency. To address this limitation, Lancer implements a WFQ strategy that assigns different weights to flows based on predefined criteria. This approach allows more critical flows to receive a larger share of the available bandwidth, thereby accelerating their deployment or relocation.

Lancer's Traffic Manager considers two factors when assigning flows weights:

- **Number of SLA violations**: Flows associated with applications experiencing more frequent SLA violations are assigned higher weights, since prioritizing their transfers can contribute to the overall network stabilization.
- **Remaining data volume**: Flows with smaller remaining data volumes receive higher weights, as completing these transfers quickly helps free network resources for other flows.

As this is a multi-objective approach, the metrics used may not share the same scale. Therefore, a Min-Max Normalization [18] technique is applied to rescale each criterion to a common range before combining them into a single weight. By doing that, Lancer optimizes bandwidth allocation, promoting not only fairness but also responsiveness to the dynamic demands of the network.

5. Performance Evaluation

This section describes the experiments conducted to evaluate Lancer's performance in simulated Edge-to-Cloud Continuum environments. Section 5.1 presents the experimental setup, including the simulation environment, workload characteristics, and evaluation metrics. Section 5.2 then summarizes and discusses the results, comparing Lancer against baseline approaches in each of its main components: the Positioning Manager, Route Manager, and Traffic Manager.

5.1. Methodology

The experiments were conducted using EdgeSimPy [19], a Python-based edge simulator that provides abstractions for various network components. EdgeSimPy was chosen due to its flexibility and fine-grained modeling capabilities for provisioning containerized applications. The simulator was configured to run testing scenarios for one hour, with a time granularity of 1s, which we consider sufficient to capture the dynamics of user mobility and application demand in this environment.

Table 2. Edge server specifications used in the experiments.

Model	CPU	RAM (GB)	Max Power (W)	Static Power (%)
Dell PowerEdge R620	16 cores	24	243	22.2%
HPE ProLiant DL360 Gen9	36 cores	64	276	16.3%
SGI Rackable C2112-4G10	32 cores	32	1387	19.1%
Acer AR585 F1	48 cores	64	559	22.7%

The experimental scenario includes 20 edge servers whose characteristics are summarized in Table 2. Although we could not find public storage specifications, CPU and RAM specifications were based on real server models. These edge servers are interconnected through 900 base stations equipped with network switches, forming a partially connected mesh topology distributed across a 30x30 hexagonal grid. Base-station communication occurs over links with 100 Mbps of bandwidth and 5 ms of delay. The placement of edge servers across the map was determined using a K-means clustering algorithm [20] to ensure an more even distribution.

Table 3. Adopted service demand values.

Specification	CPU (cores)	RAM (MB)
Tiny	1	1024
Small	2	2048
Medium	3	3072
Large	4	4096

The workload consists of 96 mobile users simulated with a pathway mobility model. Each user is initially associated with a random base station and moves in intervals of 60 steps. Each user is linked to an application selected from six predefined application specifications, consisting of 2, 4, or 6 services, each with specific CPU and RAM demands as listed in Table 3. SLA requirements are defined with delay thresholds of 35 ms and 70 ms, representing strict and lenient requirements, respectively.

The applications services are based on 10 of the 150 most popular DockerHub[3] container images, whose characteristics appear in Table 4. The container images are grouped into three representative categories (operating systems, language runtimes, and generic applications) with sizes ranging from 69 MB to 1009 MB. We assume that container images are pulled from a centralized registry.

Table 4. Adopted container image specifications.

Category	Image Name	Size (MB)
Operating Systems	archlinux	154.821
	clearlinux	69.3679
Language Runtimes	haskell	784.243
	elixir	597.899
	swift	1009.91
	python	354.531
Generic Applications	node	389.033
	rails	303.099
	flink	630.182
	storm	430.388

We evaluate Lancer using six key performance metrics. All metrics are collected every simulation step (1 second) throughout the entire execution and summarized using statistical measures such as minimum, maximum, sum, mean, standard deviation, median, and percentiles (90th, 95th, and 99th), with the exception of the *Number of relocations*, which is a scalar metric. The considered metrics are:

- **SLA Violations**: Total number of SLA violations observed during the simulation, indicating how well the system satisfies user performance requirements.
- **SLA Violations while Pulling Layers**: Number of SLA violations occurring specifically during the pulling of container images from the registry to edge servers.
- **Energy Consumption**: Total energy consumed by edge servers, measured in Kilowatt-hour (kWh), reflecting the efficiency of resource management.
- **Relocation Times**: Time required to relocate applications after SLA violations, measured in seconds.
- **Pulling Times**: Time required to pull container images from the registry to edge servers, measured in seconds.
- **Number of Relocations**: Total number of application relocations performed during the simulation.

We compare Lancer decision-making modules against (and in complement to) three baseline strategies. The first, Follow User [21], prioritizes allocating microservices to servers closer to users. In addition, MostAllocated selects the server with the highest existing demand, while LeastAllocated selects the server with the lowest existing demand. We chose Follow User as it is a strong baseline for latency-sensitive microservice provisioning. MostAllocated and LeastAllocated were chosen as they are representative examples of the state-of-practice, as they are incorporated as allocation plugins in the well-known Kubernetes Scheduler[4].

[3] https://hub.docker.com/

[4] `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler`

We adjusted the baseline strategies to align with our context and ensure a fair comparison. These changes include: (i) tracking server assignments during provisioning to avoid redundant relocations of the same service; (ii) verifying all application services are available before triggering relocations hasty decisions based on incomplete service chain information; (iii) filtering candidate servers based on the full service chain's total delay, to meet minimum SLA compliance; and (iv) requiring candidate servers to show improvements in both resource score and delay, compared to the current server, before any relocation is triggered. This prevents relocations that do not yield clear benefits.

5.2. Results

This section presents and discusses the results obtained during Lancer's performance evaluation. The analysis is structured into three segments, each focusing on a key performance metric: SLA Violations, Number of Relocations, and Energy Consumption. For each metric, we compare Lancer's modules against (and in complement to) baseline approaches to highlight its effectiveness in optimizing the proposed metrics.

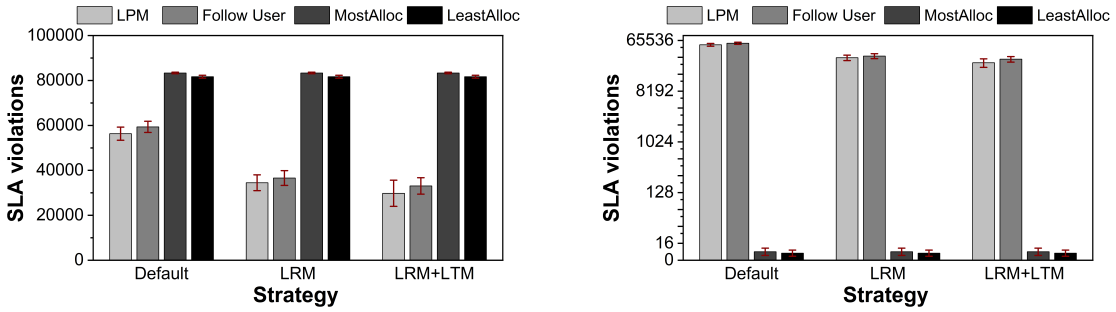
5.2.1. SLA Violations

Figure 2a compares the total number of SLA violations observed for each strategy. We notice that LPM outperforms baseline strategies even when used alone, yielding XX% fewer violations compared to the second-best approach. This improvement can be attributed to two key factors. First, in addition to prioritizing applications with severe SLA violations, LPM also considers container layer sizes during relocation decisions. Although this may seem counterintuitive at first glance - since larger layers may take longer to transfer — it actually leads to more efficient use of network resources. By relocating these applications earlier, LPM propagates more layers through the network, allowing subsequent relocations to benefit from already downloaded layers. This reduces overall transfer times and consequently improves SLA compliance.

Second, LPM's multi-objective optimization approach orders hosts based on an evaluation of several factors, including delay, pending layers, deployment queue length, and incremental energy consumption. This holistic perspective enables more informed relocation decisions, further reducing SLA violations. Moreover, when multiple hosts are non-dominant relative to each other, the final choice is determined using a tiebreaker criterion based on the sum of normalized values of previous service delay and incremental energy consumption. These attributes are prioritized because they have a more direct impact on SLA compliance and energy efficiency.

When LRM is added to the relocation process, we observe even greater performance improvements. This behavior is explained by Figure 2b, which shows that most SLA violations in LPM and Follow User strategies occur during the pulling of container layers. Lancer's BWA strategy significantly reduces these violations by prioritizing paths with higher available bandwidth, thereby avoiding network congestion and accelerating data transfers.

Additionally, when LPM is combined with LRM and LTM, it achieves the lowest number of SLA violations across all tested configurations. This result highlights the



(a) Total SLA violations. **(b) SLA violations when pulling layers.**

Figure 2. SLA violation results.

synergistic effect of Lancer's components: by selecting paths with higher available bandwidth, WFQ strategy can allocate bandwidth more effectively among competing flows. The Follow User strategy also reaches its best performance when both BWA and WFQ are applied, demonstrating that other relocation policies can also benefit from Lancer's traffic management strategies.

LRM and LTM impacts are less evident in the MostAllocated and LeastAllocated strategies due to their inherent characteristics. Both algorithms rely on simple score calculations that primarily consider resource utilization during relocation decisions. As a result, they tend to favor the same limited set of edge servers, which eventually cache most of the required layers.

In the case of MostAllocated, the strategy favors a few high-demand servers, and once a service is placed on one of them, it is less likely to be relocated again. In contrast, the LeastAllocated strategy seeks to balance the load across servers; however, it faces significant challenges in simultaneously meeting SLA requirements and maintaining load balance. Consequently, fewer movements are performed, and services remain tied to the same servers for longer periods. Therefore, path-selection and traffic-shaping strategies are less relevant in these cases.

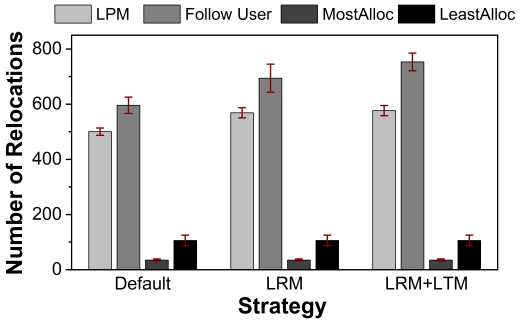


Figure 3. Comparison on the number of relocations.

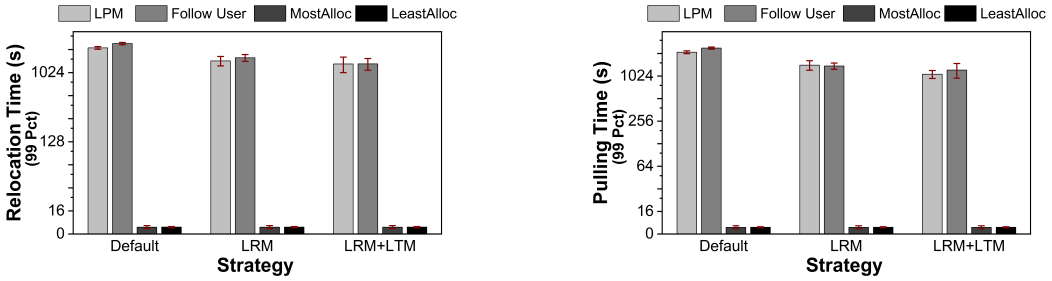
5.2.2. Number of Relocations

The characteristics of the MostAllocated and LeastAllocated strategies also influence the number of relocations performed during the simulation. Figure 3 compares the number of

relocations performed by each strategy. As expected, the MostAllocated strategy achieves the lowest number of relocations, followed by the LeastAllocated strategy.

LPM achieves the second-highest number of relocations, with only the Follow User strategy performing more. This outcome is expected, as the Follow User strategy aims to keep applications as close as possible to users, leading to frequent relocations as users move across the network. In contrast, LPM favors the relocation of more critical applications first, sometimes avoiding unnecessary movements.

In addition, when the BWA and WFQ strategies are incorporated into the relocation process, we observe a significant increase in the number of relocations performed by the LPM and Follow User strategies. This occurs because both BWA and WFQ improve network resource utilization, allowing more relocations without causing excessive delays or congestion. Consequently, applications can be relocated more frequently to better satisfy SLA requirements.



(a) Overall relocation times. **(b) Layer pulling times.**

Figure 4. Relocation time results.

Figure 4a and Figure 4b offers a more detailed perspective on the relocation process. This results align with the previous analysis of SLA violations during layer pulling, as we also observe that most part of the relocation time is spent during layer pulling. Here, LRM plays a crucial role by significantly reducing pulling times when combined with LPM and Follow User algorithms. As a result, we have shorter overall relocation times, which contributes to multiple benefits, including better SLA compliance and reduced energy consumption.

5.2.3. Energy Consumption

Finally, Figure 5 presents the energy consumption for all combinations of traffic-shaping, path-selection, and relocation policies. In this scenario, LPM exhibits the second-lowest energy consumption when combined with the BWA and WFQ strategies. During a relocation process, the service remains active on the source host until relocation is complete, while resources are reserved on the target host from the beginning of the relocation. As a result, both source and target hosts consume energy throughout the entire process. Therefore, reducing relocation times also contributes to lower energy consumption.

As expected, the LeastAllocated strategy yields the lowest energy consumption, since its primary goal is to balance the load across servers, leading to more efficient

resource utilization and reduced overall energy usage. Conversely, the MostAllocated strategy results in the highest energy consumption, as it concentrates workloads on a small number of high-capacity servers, increasing resource utilization and energy demand.

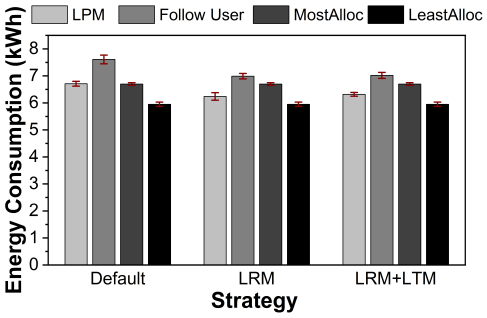


Figure 5. Comparison of energy consumption.

6. Final Remarks

Some trends in the IT ecosystem are encouraging the adoption of emerging technologies and softwarization paradigms. On the infrastructure side, distributed and heterogeneous environments such as the Edge-to-Cloud Continuum are increasingly embraced. On the software side, composite architectures based on containerization and microservices have become a common choice. In such heterogeneous environments, resource allocation becomes a complex task. Consequently, resource allocation decisions must consider multiple potentially conflicting objectives, such as application performance and energy efficiency, while also addressing interdependencies among different decision-making components.

This work presented Lancer, a framework that integrates multiple decision-making mechanisms to enhance resource management in Edge-to-Cloud computing environments. Lancer combines a Pareto-based relocation policy with Bandwidth-Aware path selection and a Weighted Fair Queuing traffic-shaping strategy. By jointly addressing these interrelated components, Lancer was able to achieve significant improvements in both application performance and energy efficiency.

Simulation results demonstrated that each individual component of Lancer contributed to performance gains when compared to established baselines. When combined, these components yielded even greater benefits, optimizing application performance and reducing infrastructure energy consumption by 60,17% and 6,8%, respectively, illustrating how holistic approaches can lead to superior outcomes in complex environments.

For future research directions, we intend to investigate the integration of proactive relocation strategies capable of mitigating SLA violations before they occur. Additionally, we plan to explore the application of other optimization techniques, such as heuristically guided metaheuristics, to further prune the search space and enhance decision-making efficiency. Finally, future work may also consider validating Lancer in real-world testbeds to assess its performance under practical conditions and workloads.

References

[1] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[2] Yushan Siriwardhana, Pawani Porambage, Madhusanka Liyanage, and Mika Ylianttila. A survey on mobile augmented reality with 5g mobile edge computing: Architectures, applications, and technical aspects. *IEEE Communications Surveys & Tutorials*, 23(2):1160–1192, 2021.

[3] Qiang Duan, Shangguang Wang, and Nirwan Ansari. Convergence of networking and cloud/edge computing: Status, challenges, and opportunities. *IEEE Network*, 34(6):148–155, 2020.

[4] Yaser Mansouri and M Ali Babar. A review of edge computing: Features and resource virtualization. *Journal of Parallel and Distributed Computing*, 150:155–183, 2021.

[5] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

[6] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, pages 1–13, 2016.

[7] Dejan Milojicic. The edge-to-cloud continuum. *Computer*, 53(11):16–25, 2020.

[8] Daniel Chaves Temp, Paulo Silas Severo de Souza, Arthur Francisco Lorenzon, Marcelo Caggiani Luizelli, and Fábio Diniz Rossi. Mobility-aware registry migration for containerized applications on edge computing infrastructures. *Journal of Network and Computer Applications*, 217:103676, 2023.

[9] Daniel C Temp, Alexandre AF da Costa, Angelo NC Vieira, Ester S Oribes, Ivan M Lopes, Paulo Silas S de Souza, Marcelo C Luizelli, Arthur F Lorenzon, and Fábio D Rossi. Maper: mobility-aware energy-efficient container registry migrations for edge computing infrastructures. *The Journal of Supercomputing*, 81(1):1–29, 2025.

[10] Zhiqing Tang, Jiong Lou, and Weijia Jia. Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing. *IEEE Transactions on Mobile Computing*, 22(6):3444–3459, 2022.

[11] Wentao Peng, Zhiqing Tang, Jianxiong Guo, Jiong Lou, Tian Wang, and Weijia Jia. Lr2scheduler: layer-aware, resource-balanced, and request-adaptive container scheduling for edge computing. *CCF Transactions on Pervasive Computing and Interaction*, pages 1–15, 2025.

[12] You Shi, Yuye Yang, Changyan Yi, Bing Chen, and Jun Cai. Towards online reliability-enhanced microservice deployment with layer sharing in edge computing. *IEEE Internet of Things Journal*, 2024.

[13] Zhiqing Tang, Fangyi Mou, Jiong Lou, Weijia Jia, Yuan Wu, and Wei Zhao. Multi-user layer-aware online container migration in edge-assisted vehicular networks. *IEEE/ACM Transactions on Networking*, 32(2):1807–1822, 2024.

[14] Zhaoyang Wang, Jinqi Zhu, Jia Guo, and Yang Liu. Microservice deployment based on multiple controllers for user response time reduction in edge-native computing. *Sensors*, 25(10):3248, 2025.

[15] Pedro Henrique Sachete Garcia, Arthur Lorenzon, Marcelo Luizelli, Paulo Silas Severo de Souza, and Fábio Rossi. Wfq-based sla-aware edge applications provisioning. In *Proceedings of the 15th International Conference on Cloud Computing and Services Science - CLOSER*, pages 159–166. INSTICC, SciTePress, 2025.

[16] Chao Zeng, Xingwei Wang, Rongfei Zeng, Shining Zhang, Jianzhi Shi, and Min Huang. Containerized service placement and resource allocation at edge: A hybrid reinforcement learning approach. *Computer Networks*, page 111343, 2025.

[17] D. Bertsekas and R. Gallager. *Data Networks: Second Edition*. Athena Scientific, 2021.

[18] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[19] Paulo S. Souza, Tiago Ferreto, and Rodrigo N. Calheiros. EdgeSimPy: Python-Based Modeling and Simulation of Edge Computing Resource Management Policies. *Future Generation Computer Systems*, 148:446–459, 2023.

[20] J MacQueen. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, volume 1, pages 281–297, 1967.

[21] Hong Yao, Changmin Bai, Deze Zeng, Qingzhong Liang, and Yuanyuan Fan. Migrate or not? exploring virtual machine migration in roadside cloudlet-based vehicular cloud. *Concurrency and Computation: Practice and Experience*, 27(18):5780–5792, 2015.