PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

PAULO SILAS SEVERO DE SOUZA

MINIMIZING LATENCY AND MAINTENANCE TIME DURING SERVER UPDATES ON EDGE COMPUTING INFRASTRUCTURES

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

MINIMIZING LATENCY AND MAINTENANCE TIME DURING SERVER UPDATES ON EDGE COMPUTING INFRASTRUCTURES

PAULO SILAS SEVERO DE SOUZA

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Dr. Tiago Coelho Ferreto
Co-Advisor: Prof. Dr. Rodrigo Neves Calheiros

Porto Alegre
2023

Ficha Catalográfica

S729m Souza, Paulo Silas Severo de

Minimizing latency and maintenance time during server updates on edge computing infrastructures / Paulo Silas Severo de Souza. – 2023.
135 f.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Tiago Coelho Ferreto.
Coorientador: Prof. Dr. Rodrigo Neves Calheiros.

1. Resource Management. 2. Maintenance. 3. Server Updates. 4. Edge Computing. 5. Application Latency. I. Ferreto, Tiago Coelho. II. Calheiros, Rodrigo Neves. III. , . IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

Paulo Silas Severo de Souza

Minimizing Latency and Maintenance Time during Server Updates on Edge Computing Infrastructures

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on August 25th, 2023.

COMMITTEE MEMBERS:

Prof. Dr. Francisco Vilar Brasileiro (PPGCC/UFCG)

Prof. Dr. Luiz Fernando Bittencourt (UNICAMP)

Prof. Dr. César Augusto Fonticielha De Rose (PPGCC/PUCRS)

Prof. Dr. Tiago Coelho Ferreto (PPGCC/PUCRS - Advisor)

I dedicate this work to my beloved wife, Amanda.

“For of Him and through Him and to Him are all things, to whom be glory forever. Amen.”
(The Holy Bible, NKJV, Romans 11:36)

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to the Almighty God for giving me strength, guidance, and wisdom throughout my doctoral journey. Without His blessings and grace, I would have been unable to achieve this significant milestone in my academic and personal life.

I am forever grateful to my wife, Amanda, for her love, support, and understanding. Her encouragement, patience, and sacrifices have been a constant source of inspiration and motivation. I am truly blessed to have her by my side.

Likewise, I would like to thank my family, especially my parents, Getulio and Zeni, my sister Gianine, and my brother-in-law Evandro, for their unconditional love and support. Their encouragement and prayers have been vital.

I am indebted to my advisor, Prof. Dr. Tiago Coelho Ferreto, and my co-advisor, Prof. Dr. Rodrigo Neves Calheiros, for their invaluable mentorship and support. Their expertise, feedback, and encouragement have been critical in shaping my research and improving my writing. I am also grateful to my thesis committee members for their insightful comments and constructive criticisms.

I would like to express my sincere gratitude to my professors and academic mentors, especially Prof. Dr. Fábio Rossi and Prof. Dr. Jaline Mombach, for their invaluable support throughout my educational journey.

I am also indebted to my friends, especially Ângelo Vieira and my laboratory colleagues, for all the support and encouragement. The stimulating conversations and lively discussions have been a source of inspiration and motivation. I am grateful for all the understanding and flexibility during the challenging times of my doctoral journey.

Furthermore, I would like to thank the staff and faculty at the Pontifical Catholic University of Rio Grande do Sul, particularly the School of Technology and the Software Engineering Research Center (CePES), for providing me with the resources, facilities, and opportunities to pursue my doctoral studies.

This work was supported by the PDTI Program, funded by Dell Computadores do Brasil Ltda (Law 8.248 / 91). I also acknowledge the support from the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul.

MINIMIZANDO LATÊNCIA E TEMPO DE MANUTENÇÃO DURANTE ATUALIZAÇÃO DE SERVIDORES EM INFRAESTRUTURAS DE COMPUTAÇÃO NA BORDA

RESUMO

A Computação na Borda oferece baixa latência para aplicações de tempo real, transferindo tarefas de processamento de *data centers* tradicionais na nuvem para a borda da rede, em proximidade às fontes dos dados. À medida que as expectativas sobre a Computação na Borda se amplificam, também aumenta a pressão sobre profissionais de TI responsáveis por planejar e executar manutenções em infraestruturas de borda. Manutenções na borda são essenciais, dado que servidores de borda—especialmente aqueles posicionados em instalações ao ar livre—são expostos a várias vulnerabilidades, incluindo problemas de *hardware* e ameaças de segurança. Para complicar ainda mais a situação, muitas características únicas de infraestruturas de borda, como requisitos estritos de latência das aplicações e a dispersão física dos servidores, dificultam a possibilidade de reutilização de estratégias de manutenção projetadas para a nuvem. Diante deste cenário, esta tese de doutorado busca possibilitar atualizações mais rápidas de servidores de borda, reduzindo o impacto da manutenção no desempenho das aplicações. Para isso, esta tese de doutorado faz as seguintes contribuições: (i) Uma revisão de literatura que organiza a pesquisa existente sobre manutenção direcionada à Computação na Borda e dois paradigmas relacionados (Computação em Nuvem e Internet das Coisas) de acordo com uma nova taxonomia; (ii) Um simulador, chamado EdgeSimPy, que modela vários componentes de infraestruturas de borda e dá suporte a casos de uso de manutenção; (iii) Duas estratégias de manutenção, chamadas Lamp e Laxus, que incorporam a localização dos usuários na tomada de decisões de manutenção para reduzir o impacto de atualizações de servidores de borda na latência das aplicações; e (iv) Uma estratégia de manutenção, chamada Hermes, que toma vantagem do conteúdo compartilhado de imagens de contêineres das

aplicações de borda para reduzir o tempo de manutenção através de realocações otimizadas. Experimentos extensivos mostram que as soluções propostas são capazes de acelerar a atualização de servidores de borda, reduzindo o impacto da manutenção no desempenho das aplicações em comparação com estratégias da literatura.

Palavras-Chave: Gerência de Recursos, Manutenção, Atualização de Servidores, Computação na Borda, Latência de Aplicações.

MINIMIZING LATENCY AND MAINTENANCE TIME DURING SERVER UPDATES ON EDGE COMPUTING INFRASTRUCTURES

ABSTRACT

Edge Computing offers low latency for real-time applications by shifting processing tasks from traditional cloud data centers to the network's edge, near data sources. As expectations about Edge Computing grow, so does the pressure on IT personnel responsible for planning and executing maintenance operations on edge infrastructures. Maintenance at the edge is critical, given that edge servers, especially those installed in outdoor facilities, are exposed to several vulnerabilities, including hardware issues and security threats. To make things even more complicated, many unique characteristics of edge infrastructures, such as tight application latency requirements and the physical dispersion of edge servers, hinder the possibility of reusing maintenance strategies designed for the cloud. In light of the given scenario, this doctoral thesis seeks to enable faster updates of edge servers while reducing maintenance's impact on edge application performance. To this end, this doctoral thesis makes the following contributions: (i) a literature review that organizes existing maintenance research targeting Edge Computing and two related paradigms (Cloud Computing and Internet of Things) according to a novel taxonomy; (ii) a simulation toolkit, called EdgeSimPy, that models various components of edge infrastructures and supports maintenance use cases; (iii) two maintenance strategies, called Lamp and Laxus, that incorporate user location awareness into maintenance decision-making to reduce the impact of server updates on application latency; and (iv) a maintenance strategy, called Hermes, that capitalizes on the shared content of container images of edge applications to reduce maintenance time through optimized relocations. Extensive experiments show that proposed solutions can accelerate edge server updates while reducing the impact of maintenance on edge application performance compared to strategies from the literature.

Keywords: Resource Management, Maintenance, Server Updates, Edge Computing, Application Latency.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1. INTRODUCTION

1.1 Motivation

Cloud computing has been recognized as the reference model for deploying applications through the Internet since the release of popular services managed by big players such as Amazon Web Services (AWS)[1] in 2006 [120] [23]. While the cloud domain was unchallenged for many years, advances in hardware production and telecommunications enabled the emergence of the Internet of Things [53], which introduced new classes of sensor-rich applications whose latency and bandwidth requirements could not be satisfied solely by cloud data centers due to their distance from data sources.

As a response to the challenge of handling the demand for real-time IoT applications, a new paradigm called Edge Computing [132] was introduced. The main idea of Edge Computing is acting as an extension of the cloud that brings computing resources closer to data sources to alleviate the demand upon the Internet's core and to cope with the application's low latency requirements [131]. Despite their contrasting characteristics, cloud and edge infrastructures share some commonalities, including the need for meticulous maintenance strategies, which serve as countermeasures against undesired events that can affect the performance and security of applications [167] [44] [110] [149]. Although maintenance planning is a complex task for both cloud and edge infrastructures, some unique characteristics of the edge increase the difficulty of such a process even further.

To better illustrate the challenges of conducting maintenance at the edge, let us consider an edge server patching scenario where advancing maintenance requires relocating applications to avoid downtime during the updates. The first consideration in such a scenario is that edge infrastructures are often made up of heterogeneous hosts with varying processing and capacity, and edge applications have strict performance requirements [175]. Consequently, there are limited provisioning options that preserve application performance during maintenance work. Furthermore, edge servers are typically connected by public networks without mandatory redundancy and performance guarantees [160] [8]. As a result, network operations are subject to unexpected instability, thereby delaying any maintenance operations that depend on them.

While considerable research advancements have been made to optimize the planning and execution of maintenance operations in cloud data centers, little effort has been directed toward Edge Computing environments. On top of this, the aforementioned maintenance demands of edge infrastructures make the problem more complex than simply reusing existing strategies designed for cloud environments. Consequently, new maintenance ap-

[1] https://aws.amazon.com/

proaches are required to ensure that updates are quickly applied to the edge infrastructure while keeping the impact on application performance as low as possible.

1.2 Objective

This thesis aims to optimize maintenance operations on Edge Computing environments, enabling faster updates of edge components while ensuring that the impact of maintenance work on the performance of edge applications is reduced.

1.3 Hypothesis and Research Questions

We formulate the following hypothesis to guide this doctoral research:

"Driving maintenance decisions according to performance requirements and characteristics of edge applications could enable faster updates of edge components with reduced impact on application performance."

We advocate that resource allocation decisions made during maintenance should take into account the performance requirements of edge applications and the constraints of edge infrastructures when determining "when" and "how" to update components. By adhering to this approach, we believe maintenance strategies could be less intrusive to the performance of applications running at the edge, while ensuring the stability and security of the infrastructure through faster updates.

Guided by the above rationale, we define the following research questions:

- ***Research Question 1:*** *What are the main approaches and metrics of interest in the context of maintenance on Edge Computing infrastructures?*
  - The objective of this research question is twofold. First, it aims to understand what requirements are considered during maintenance operations on Edge Computing environments and how they are expressed through performance indicators. Second, it seeks to catalog the main edge maintenance approaches.
- ***Research Question 2:*** *How can we evaluate research prototypes of maintenance strategies for Edge Computing infrastructures?*
  - This research question seeks to identify the available resources for validating and evaluating maintenance strategies for edge environments, analyzing areas of improvement and potential alternatives to support the maturation process of research prototypes in the field.

- ***Research Question 3:*** *What is the impact of location-aware application relocation during maintenance on Edge Computing infrastructures?*
  - The objective of this research question is twofold. First, it aims to demonstrate that simply reusing maintenance strategies designed for cloud environments during the update of components on edge infrastructures is ineffective due to the incurred degradation of edge application performance. Second, it seeks to understand how location awareness can be incorporated into maintenance decision-making to reduce the impact of maintenance on application performance.
- ***Research Question 4:*** *How can we leverage characteristics of edge applications to reduce maintenance time during edge server updates?*
  - This research question focuses on optimizing edge server updates. In this context, the aim is to identify which characteristics of edge applications impact the maintenance time during edge server updates and how they can be incorporated into maintenance strategies to accelerate such a process.

1.4 Main Contributions

This thesis advances the state of the art through the following contributions:

- It presents a novel taxonomy that organizes existing maintenance strategies targeting physical and logical components of Edge Computing environments and two related paradigms (Cloud Computing and Internet of Things).
- It introduces EdgeSimPy, a simulation toolkit that enables the modeling and simulation of resource management policies for edge infrastructures. In addition to implementing a conceptual model that accurately represents the entire lifecycle of edge applications, EdgeSimPy supports the evaluation of maintenance strategies for various components of edge infrastructures, including edge servers and network devices.
- It proposes three novel maintenance strategies tailored to the needs of the edge. The first two strategies, Lamp and Laxus, incorporate user location awareness into maintenance decision-making to reduce the impact of server updates on application latency. The third strategy, Hermes, leverages the shared content of container images of edge applications to reduce maintenance time through optimized relocations.

1.5 Thesis Organization

Figure 1.1 shows the relationship between the subsequent chapters of this thesis. It is worth noting that portions of the content of subsequent chapters have been partially derived from a series of papers published throughout the doctoral candidature.

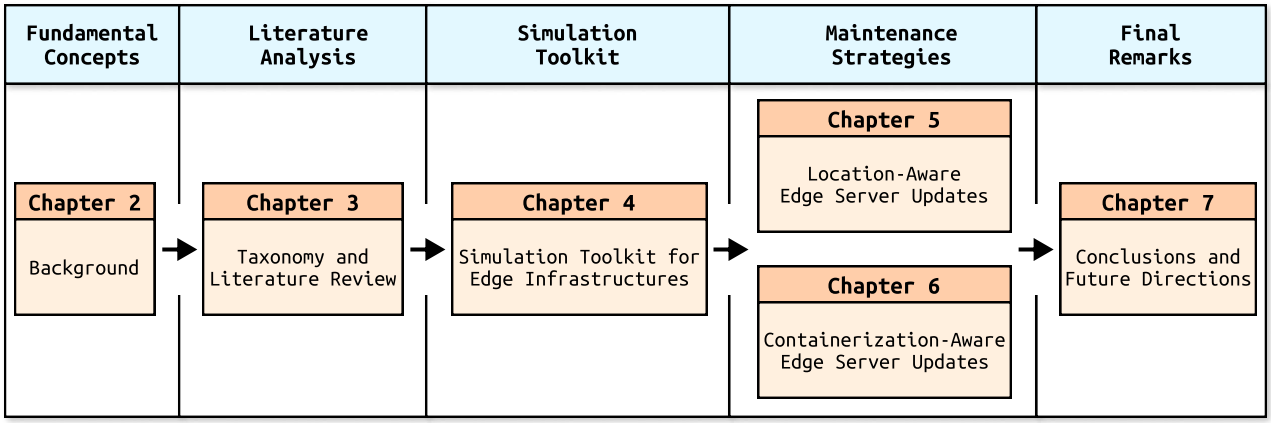


Figure 1.1 – Thesis organization.

The remainder of this thesis is organized as follows:

- Chapter 2 reviews the concepts that shape the foundation of our research, including Cloud Computing, Internet of Things, Edge Computing, and maintenance.
- Chapter 3 presents a review of the literature that organizes maintenance research focusing on Edge Computing and related paradigms according to a novel taxonomy. This chapter addresses Research Question 1 and is partially derived from a research paper under review in the *ACM Computing Surveys* journal.
- Chapter 4 presents a novel simulation toolkit that implements several functional abstractions to model the entities that compose edge infrastructures and provides support for modeling and evaluating maintenance strategies. This chapter addresses Research Question 2, and it is partially derived from [148]:
  - ***Souza, P. S.**; Ferreto, T.; Calheiros, R. N. "EdgeSimPy: Python-based modeling and simulation of edge computing resource management policies", Future Generation Computer Systems, vol. 148–1, November 2023, pp. 446–459.*
- Chapter 5 introduces two maintenance strategies that incorporate user location awareness into maintenance decision-making to reduce edge application latency during edge server updates. This chapter addresses Research Question 3, and it is partially derived from [149]:

  - ***Souza, P. S.**; Ferreto, T. C.; Rossi, F. D.; Calheiros, R. N. "Location-aware maintenance strategies for edge computing infrastructures", IEEE Communications Letters, vol. 26–4, February 2022, pp. 848–852.*

- Chapter 6 expands the work of Chapter 5 through a maintenance strategy that capitalizes on the shared content of container images to reduce maintenance time through optimized application relocations. This chapter addresses Research Question 4.

- Chapter 7 presents the final considerations, including an overview of contributions, a discussion on how established research questions are answered, and a listing of challenges and open questions related to maintenance operations on Edge Computing infrastructures.

2. BACKGROUND

This chapter discusses the core concepts that shape the foundation of this work, including Cloud Computing (Section 2.1), Internet of Things (Section 2.2), and Edge Computing (Section 2.3). In addition, it highlights the interaction between these paradigms and emphasizes the importance of maintenance in such environments (Section 2.4).

2.1 Cloud Computing

Cloud Computing has been established over the years as a *de facto* standard for hosting applications on the Internet [23]. The basic idea behind Cloud Computing is providing users on-demand access to infrastructure, platforms, and software applications through a subscription business model that works on a pay-as-you-go basis [11]. The service-based cloud model has lowered the barrier to entry for entrepreneurs and low-to-midsize companies, who previously had to afford a significant investment to build and manage their own data center infrastructure.

One of the key enablers for the on-demand subscription model offered by the cloud is elasticity, which dynamically fits resources to cope with the demand [62]. When demand increases, elastic cloud systems add resources to services, allowing them to scale according to workload. Cloud scalability occurs by giving a service a larger slice of its host server resources (vertical scaling) or dividing the demand for services across multiple servers (horizontal scaling) [48]. Conversely, underutilized resources are released to reduce the provider's operational costs.

Cloud elasticity is generally enabled by virtualization technology, which decouples software applications from the underlying hardware, granting fine-grained control over computing resources. In addition to scaling applications on-the-fly, virtualization technology allows a single physical instance to host multiple applications simultaneously, improving overall resource usage. Although virtualization is transparent to end-users, it usually takes two possible forms: Virtual Machines (VMs) and containers.

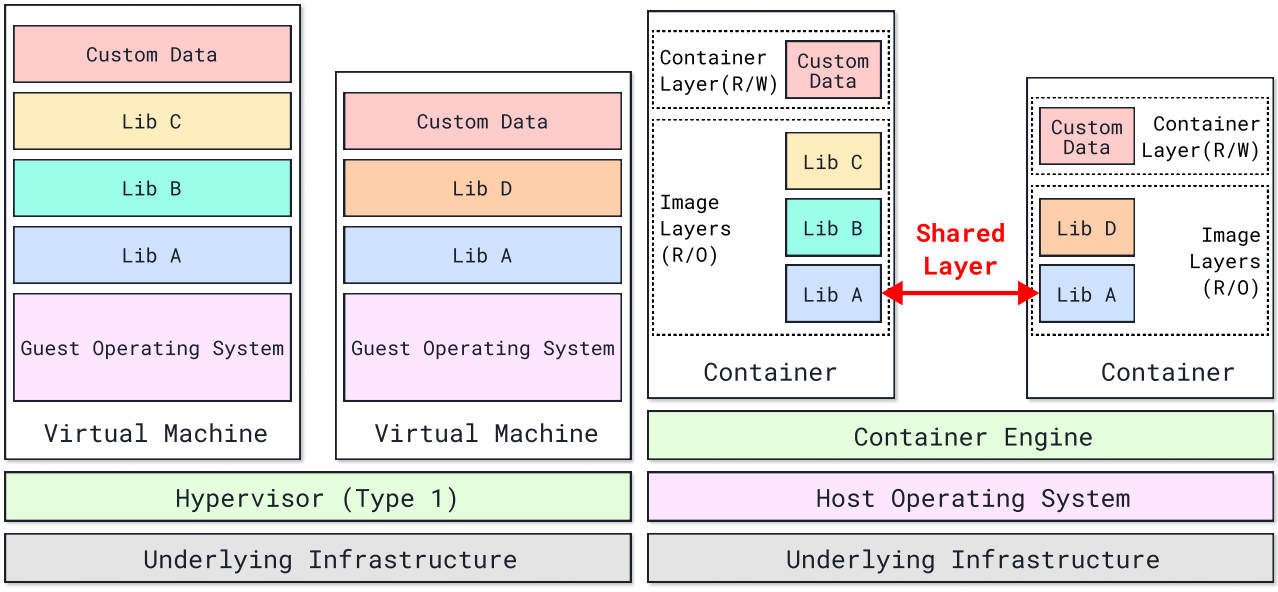
Virtualization also allows developers and infrastructure operators to avoid repetitive tasks during application deployment through template images [161]. At this point, the differences between VMs and containers become more visible. Whereas VM images are monolithic, most modern container solutions follow Docker's[1] lead, splitting container images into read-only layers representing software instructions. This difference may seem subtle, but it affects the entire lifecycle management of applications (i.e., building, deploying, and terminating operations) [166].

[1] https://docs.docker.com/storage/storagedriver/#images-and-layers

When a container needs to be spawned, a new writable layer is created on top of the container image's filesystem. All changes made to the container are stored in that writable top layer, so the image layers remain unchanged regardless of the changes at the application level, which allows containers to share common container layers, reducing the storage and memory overhead [32].

The shared layer structure of container images also enables optimization of provisioning time and network traffic when provisioning containerized applications. This is possible because transferring the complete container image is not mandatory if the target host already retains some of the container's layers. Instead, only the missing layers must be pulled from dedicated repositories for container images called container registries [151]. Conversely, spawning a VM requires the creation of a complete copy of the base image, as no persistence measure is taken to prevent applications from changing instructions from their base images. Consequently, provisioning a VM-based application is generally slower and generates higher disk demand than a container-based application [81].

In addition to the differences in template images, VMs and containers rely on different types of virtualization. In the VM model, a control layer, known as a hypervisor, virtualizes the hardware, giving each VM a dedicated virtual CPU, memory, I/O, and network devices. In the container model, a control layer, known as container runtime, virtualizes the host OS kernel instead of the hardware, so containers share the host Operating System (OS) kernel while isolation is handled at the OS level. Although containerized applications do not need a standalone OS, containers display a lower virtualization overhead than VMs at the cost of weaker isolation among co-hosted instances [138]. Figure 2.1 illustrates the architectural differences between VMs and containers.



(a) Virtual Machines (b) Containers

Figure 2.1 – Architectural differences between typical VM and container deployments.

Although virtualization enables better resource usage, it also raises some performance concerns, as applications running on the same host can degrade each other's performance [117]. In this context, Service Level Agreements (SLAs) emerge as one of the main ways to increase consumer confidence in the performance provided by the cloud [104].

In general, SLAs are contracts signed between stakeholders to formalize that a certain level of Quality of Service (QoS) should be delivered over a certain period, subject to penalties if not met. At scale, SLAs can comprise several key metrics, called Service Level Objectives (SLOs), used to assess service performance (e.g., availability, response time, etc.). Once SLAs are created, stakeholders can check if services remain SLA-compliant by observing the Service Level Indicators (SLIs), which are actual measurements of the SLOs.

As cloud technology matures and new players enter the market, providers set SLAs with increasingly strict performance promises to stand out against the competition. While ever-increasing performance expectations raise customer satisfaction, they also put pressure on IT operations teams, who need to create efficient maintenance strategies to allow data center resources to be repaired and updated to ensure continued compliance with performance and security requirements while causing the least possible service disruption.

2.2 Internet of Things

Despite significant advances in Information and Communications Technology, most of the interaction between physical and electronic components has been dependent on human intervention, restricting the potential of technology to the time, attention, and accuracy constraints of human beings [53]. To avoid this bottleneck, several discussions have pointed to the potential of the Internet of Things [53], which incorporates networking capabilities (Internet) into physical objects (Things), allowing such devices to automatically collect, communicate, and process environmental data supporting intelligent and data-driven decisions.

By encompassing a variety of devices, IoT enables several compelling use cases, from agriculture [41], where smart irrigation reduces water waste and improves harvest efficiency, to personal healthcare [68], where body sensors in patients provide real-time insights to doctors. As IoT embraces pervasiveness, wireless technology is generally preferred for connectivity in IoT deployments, imposing bandwidth and latency constraints. Additionally, IoT sensors and actuators are subject to space, energy, and thermal constraints to ensure that they remain seamlessly integrated into the environment, ultimately favoring design decisions that restrict their computational power.

Although the abundance of data fuels the rich use cases of IoT, it also poses a significant challenge for data processing. As IoT devices are resource-constrained, they must offload data to third-party entities, which perform the processing and send the feedback to IoT devices for on-site decision-making. In addition to making IoT communication costly,

this raises concerns about potential leaks of sensitive user information, such as geographical location and medical diagnoses [73]. This broad attack surface forces IoT operations teams to constantly run after efficient maintenance plans to update devices, safeguarding them from security vulnerabilities [86].

2.3 Edge Computing

Early IoT and mobile applications offloaded computing-intensive tasks to the cloud, where resource-abundant data centers could easily hold the demand [76]. However, the increasing need for high bandwidth and low latency highlighted the negative consequences of the cloud's centralized model, where computing resources are distant from data sources.

Although the many hops necessary for communication between data sources (IoT sensors and actuators, mobile devices, etc.) and cloud data centers incur network round-trip times that conflict with the real-time latency requirements of mobile applications, holding IoT processing in consolidated data centers also leads to high ingress bandwidth demand, reducing overall network performance [133]. Consequently, the decision to process data from such applications in the cloud became more arguable, paving the way for emerging paradigms such as Edge Computing [132].

Edge Computing refers to placing networked computing devices at the Internet's edge, close to end devices. The fundamental idea of decentralizing computing resources to the edge solves two major problems that challenge the cloud model. Although distributing edge servers to different locations avoids bottlenecks at specific hotspots, the network proximity to data sources grants faster application response times [131].

The coordination between edge resources, cloud data centers, and data sources shapes a three-tiered computing model [134], as shown in Figure 2.2. At Tier 1, cloud data centers hold large-scale computing and storage scalability, with stable network infrastructure and robust security mechanisms. Tier 2 represents edge resources distributed near data sources, displaying lower latency than Tier 1 at the cost of limited scalability due to restricted space, cooling, and power supply. Finally, Tier 3 hosts data producers and consumers, including mobile devices and IoT sensors.

Similarly to cloud data centers, edge infrastructures rely on virtualization technology to achieve greater manageability over physical resources. There is no consensus about what type of virtualization should be employed on edge infrastructures. Some researchers argue that VMs and containers should be used depending on deployment needs, allowing a broader spectrum of demands to be managed more efficiently [55].

In such a line of reasoning, containers can fit better when shorter provisioning times and small footprints are needed, while VMs can deliver better security and isolation

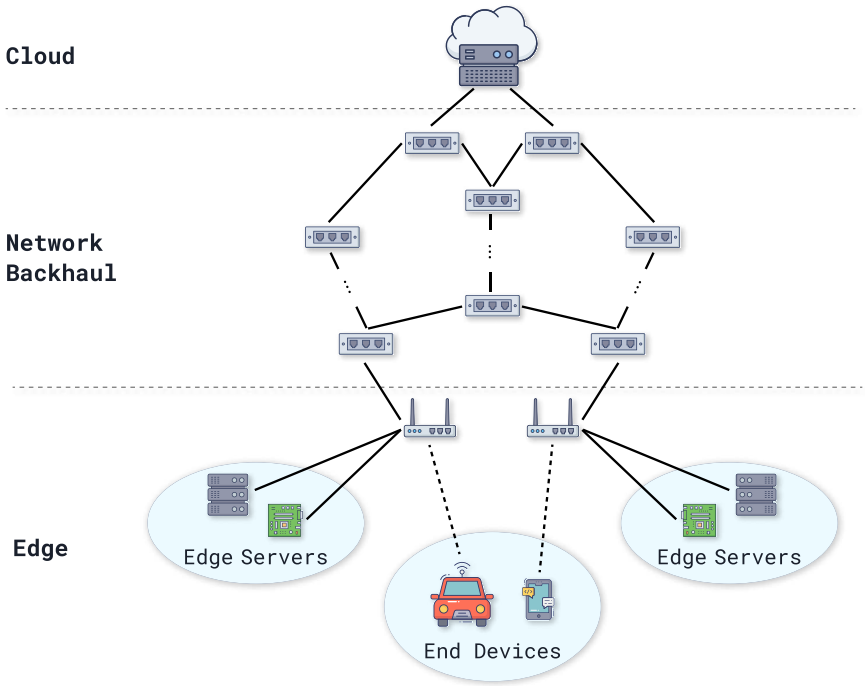


Figure 2.2 – Interplay between cloud data centers, edge sites, and end devices.

in multi-tenant deployments. Despite the potential use cases for both types of virtualization, containers have been taking the lead as the primary architecture for deploying applications on the edge, as their small footprint and low virtualization overhead fit well with the resource constraints of edge infrastructures while also meeting strict provisioning time constraints of edge applications [69].

Edge Computing deployments typically span networked compute nodes dispersed across the environment, as installing large-scale edge data centers is often unfeasible in various scenarios, such as urban centers. While physical dispersion allows edge servers to be located only a few hops from end devices, it also introduces significant technical challenges related to edge IT operations. Once edge servers are deployed outdoors in small-sized facilities, they are inherently exposed to hardware issues (e.g., accelerated aging due to increased temperatures, power outages, physical damage, etc.) and security threats (e.g., network attacks, physical tampering, etc.). In this context, maintenance strategies are critical in ensuring that performance and security issues do not nullify the potential gains of edge infrastructure's network proximity to data sources.

2.4 Maintenance Operations

The IT market is in the middle of a significant shift, with cloud and edge platforms working together to produce actionable insights from the overwhelming amount of data produced in real-time by mobile and IoT applications. To accomplish such a goal, IT operations

teams managing such a three-tiered computing model (Cloud-Edge-IoT) must overcome several operational challenges, such as equipment failures, cyber-attacks, and hardware and software aging. In this context, proper maintenance planning (what to do) and scheduling (when to do it) are critical to maintaining the various software stacks and their underlying infrastructure running smoothly.

Multiple maintenance policies have been proposed over the years. Initially, enterprises invested in corrective maintenance, following a “run-to-failure” strategy where components remain in operation until they fail. Corrective maintenance is based on the assumption that maintenance cost savings are higher than the cost of disruption. However, such an assumption cannot hold in cases where high availability is required. In addition, certain assets present detectable states of degradation (e.g., failing more often with increasing age), enabling less disruptive maintenance decisions than run to failure [103]. Figure 2.3 illustrates a P-F Curve [105], a typical cumulative damage model that represents the progression of misbehavior symptoms towards functional asset failure. In such scenarios, preventive measures can be taken within the so-called P-F Interval, a period between the beginning of misbehavior and the asset crash.

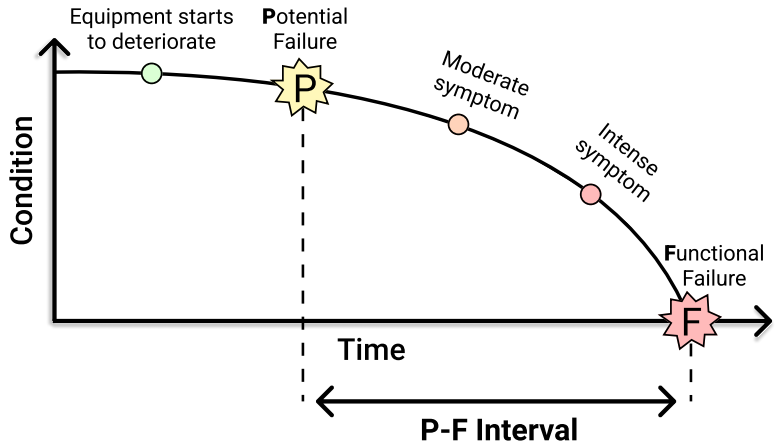


Figure 2.3 – Typical representation of the P-F Curve, a cumulative damage model that represents an asset’s degrading behavior towards functional failure.

The simplest preventive maintenance approach is Schedule-Based Maintenance (SBM) [116], in which assets are repaired at predefined intervals. Given that excessively long maintenance intervals increase the risk of asset failure, SBM is usually performed within significantly short periods—even if this is not always necessary—to avoid functional failures. Although SBM shows a natural advantage over corrective maintenance by avoiding service disruption, it assumes that degradation progresses steadily so that failures do not occur between maintenance actions. In contrast to SBM, Condition-Based Maintenance (CBM) [116] follows a more flexible approach, triggering maintenance actions only when the asset’s state has deteriorated to a certain threshold, optimistically avoiding both delayed and hasty maintenance decisions.

Preventive maintenance assumes that the degradation leaves clues for long enough for maintenance measures to be taken, which does not apply to situations where failures display non-linear progression or occur due to external factors such as cascading errors. Given these limitations, significant efforts have been made to replace proactive maintenance decisions based on thresholds with predictive approaches [144].

Predictive maintenance techniques determine the maintenance schedule by estimating an asset's Remaining Useful Life (RUL) based on its internal attributes and the state of neighboring components. Predictive maintenance techniques are divided into two categories: model-based and data-based methodologies. While the former relies on mathematical models built through the knowledge of engineers, the latter uses statistical and machine learning algorithms to predict the state of the asset through historical data [78]. Figure 2.4 presents a taxonomy based on Silvestri et al. [142] and Kim et al. [78] that categorizes the different maintenance approaches.

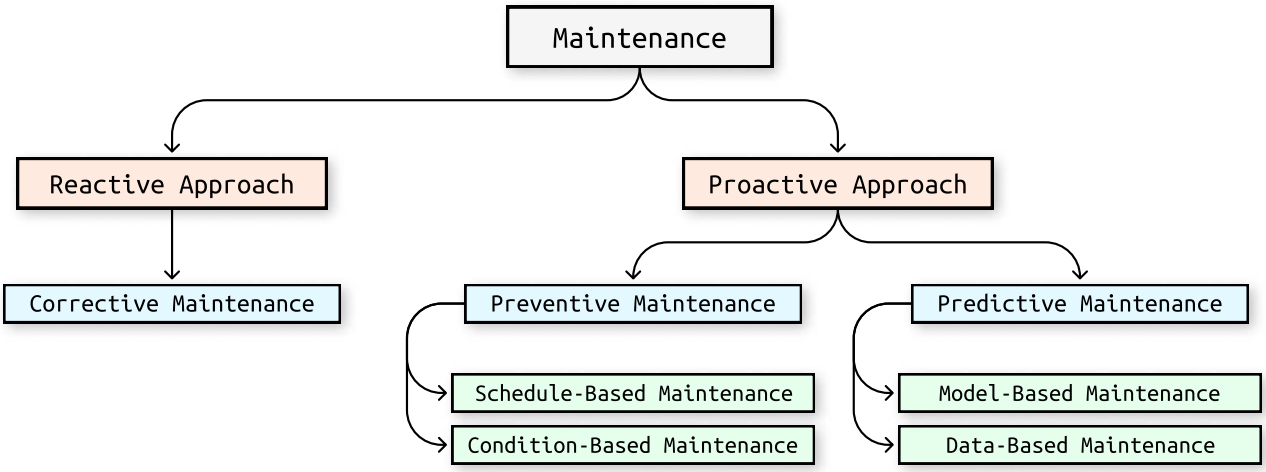


Figure 2.4 – Taxonomy of maintenance approaches [142, 78].

2.5 Closing Remarks

Despite the strategic value of maintenance in preserving the continuous operation of complex ecosystems such as Cloud Computing, Edge Computing, and the Internet of Things, there is a lack of review studies in the field, which raises the barrier to entry for new researchers. To fill this gap, we proposed a novel taxonomy that characterizes the main research efforts in the field according to multiple characteristics. The following chapter details the review methodology, proposed taxonomy, and literature analysis.

3. TAXONOMY AND LITERATURE REVIEW

This chapter reviews maintenance strategies for cloud, edge, and IoT environments. Although this thesis primarily focuses on maintenance operations on Edge Computing, the literature review encompasses investigations in related paradigms, such as Cloud Computing and the Internet of Things, to analyze potential intersections and synergies among existing contributions designed for these different scenarios, thereby providing a comprehensive understanding of the broader landscape.

3.1 Motivation

Despite the contrasting characteristics between cloud, edge, and IoT infrastructures, these environments share some commonalities, including the need for meticulous maintenance strategies, which act as countermeasures against various undesired events that can affect the applications' performance and security (e.g., component failures and cyber-attacks) [167] [44] [110] [149]. In this context, the wide range of tasks encompassed by maintenance work raises significant concerns about the potential damage caused by poor provisioning decisions during such activities.

From a networking perspective, the increased traffic incurred by maintenance-related operations (e.g., patch distributions and application migrations) can quickly saturate the network and cause several side effects, such as packet losses and increased latency. From a computing perspective, maintenance work can affect infrastructure stability by either making components temporarily unavailable (e.g., when patches require device reboots to take effect) or by excessively increasing the concurrent demand under physical resources (e.g., when applications are stacked on a single server while other servers are updated).

The potential side effects of infrastructure maintenance put significant pressure on IT personnel, who must have a deep understanding of it to avoid the undesirable effects of such a resource-intensive activity. Several research efforts have addressed maintenance-related challenges in the Cloud-Edge-IoT ecosystem. However, summarizing and extracting insightful and actionable information from their findings and drawing parallels between multiple works is challenging due to the large number of papers in the literature.

Review papers stand out as valuable sources of systematic knowledge. Overall, review papers can be helpful for the community, as their content can benefit both active members in the field with overviews of the state of the art and indications of research subjects requiring further attention and newcomers with a rich source of learning material.

3.2 Related Surveys

This section analyzes existing maintenance reviews on cloud, edge, and IoT environments. In addition, it highlights how this chapter's review complements their efforts and lists our contributions to the community.

Benestad et al. [17] presented a literature review on software maintenance, focusing on analyzing the objectives and attributes that drive maintenance work targeting software systems throughout their lifecycles. Among the identified gaps, the authors highlighted the need for software maintenance strategies based on theoretical models that could be adapted to different contexts to complement existing approaches, which rely on empirical observations that result in a lack of extensibility.

Abadi et al. [42] addressed maintenance work under a different facet, with a literature review on maintenance in cloud data center facilities. The authors focused on operations tasks such as managing cooling and power supply systems, indicating open challenges related to infrastructure management activities (e.g., developing availability benchmarks for data center facilities).

More recently, some reviews have focused on maintenance research applied to emerging subjects such as Industry 4.0 [84], which envisions a digital transformation in the manufacturing industry through IoT-related technologies. Silvestri et al. [142] surveyed existing maintenance solutions for Industry 4.0, highlighting the need for new diagnostic and prognostic algorithms to integrate intelligence into manufacturing processes. Zonta et al. [179] presented a literature review on predictive maintenance applied to Industry 4.0, introducing a taxonomy that organizes research works and describes several research opportunities, such as using image analysis to evaluate the state of manufacturing equipment and trigger maintenance work accordingly.

Table 3.1 compares the coverage of this work and existing reviews. While existing reviews analyze the literature from specific facets (e.g., scoping the study to certain components, maintenance approaches, and specific scenarios), our work incorporates a broader analysis, covering research work that presents maintenance strategies targeting physical and logical components in cloud, edge, and IoT environments.

Table 3.1 – Comparison of the scope of this chapter's review and previous studies.

Study	Year	Target Components		Covered Paradigms		
		Physical	Logical	Cloud Computing	Edge Computing	Internet of Things
Benestad et al. [17]	2009	✘	✔	✔	✘	✘
Abadi et al. [42]	2020	✔	✘	✔	✘	✘
Silvestri et al. [142]	2020	✔	✔	✘	✘	✔
Zonta et al. [179]	2020	✔	✔	✘	✘	✔
This Review	2023	✔	✔	✔	✔	✔

To the best of our knowledge, none of the existing reviews provides a unified analysis and categorization of the existing academic literature on the various maintenance approaches designed to address the needs of physical and logical components in cloud, edge, and IoT environments.

To fill this gap, this chapter's review makes the following contributions:

- It presents a literature review of academic solutions for addressing the various maintenance needs of the physical and logical components (e.g., replacement of defective equipment and software updates) that compose cloud, edge, and IoT environments.
- It introduces a novel taxonomy that organizes existing maintenance solutions according to diverse characteristics (e.g., target components, maintenance approaches, and target metrics) to reduce the barrier to entry for new researchers in the field.
- It sheds light on several research challenges and opportunities that represent promising directions for future investigations.

3.3 Methodology

This survey follows the research methodology presented by Brereton et al. [21], which divides the review process into three stages, namely Planning, Execution, and Report, as shown in Figure 3.1. Although such a methodology was initially designed for the field of Software Engineering, it provides a generic guide for identifying and synthesizing relevant research studies. As such, it has been adopted in several domains, including areas that intersect this work's scope, such as Cloud Computing [88] and Edge Computing [143].

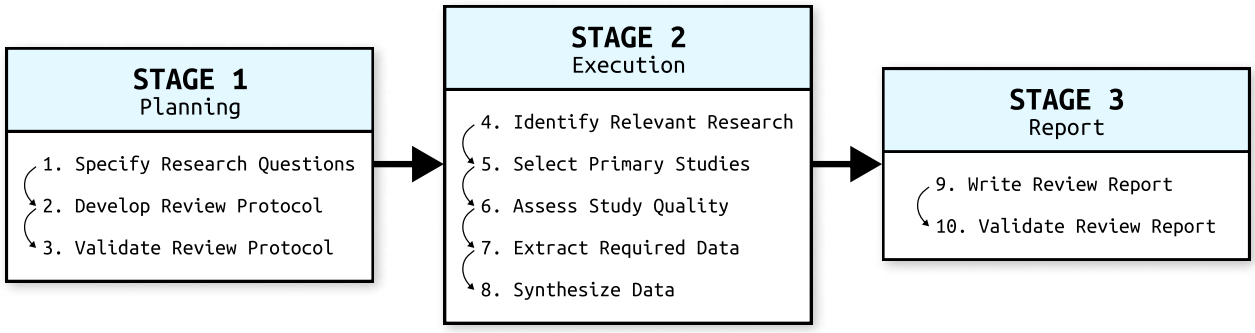


Figure 3.1 – Review methodology presented in Brereton et al. [21].

This review employs three research questions (RQs), presented in Table 3.2. With RQ1, answered in the remainder of this Section, we seek to quantify the number of published works on the addressed research topics. Regarding RQ2, the goal is to identify the relevant characteristics of the selected studies. RQ2 is answered in Section 3.4, which comprises a

taxonomy that organizes the selected research according to its characteristics. Finally, RQ3 aims to identify research opportunities. RQ3 is answered in Section 3.5, highlighting the open challenges within the investigated research subject.

Table 3.2 – List of research questions addressed in this review.

Identifier	Research Question
RQ1	How many papers on maintenance in Cloud Computing, Edge Computing, and the Internet of Things have been published between 2017 and 2022?
RQ2	What are the metrics of interest, strategies, and validation methodologies used to perform maintenance in the evaluated scenarios?
RQ3	What are the challenges and open questions on maintenance in Cloud Computing, Edge Computing, and the Internet of Things?

This review surveys academic studies indexed in three search engines: Association for Computing Machinery (ACM) Digital Library[1], Institute of Electrical and Electronics Engineers (IEEE) Xplore[2], and Scopus[3]. Although ACM and IEEE are two of the largest academic organizations in Computer Science and Electrical Engineering, contributing to the organizing committees of numerous high-impact venues, Scopus provides a large-scale database that indexes several publishers such as Springer[4], ScienceDirect[5], and Wiley[6].

The search string used to retrieve research works in the selected databases is shown in Figure 3.2. As maintenance comprises several activities (e.g., replacement of defective equipment, device patching, software update, etc.), our search string incorporates a group of keywords (i.e., maintenance, patch, repair, update, upgrade, rejuvenation, and recovery) representing the maintenance process. We configured the selected databases to look for the search string in their indexed papers' titles, abstracts, and keywords.

```
("Maintenance" OR "Patch" OR "Repair" OR "Update" OR "Upgrade" OR "Rejuvenation"
OR "Recovery") AND ("Cloud Computing" OR "Edge Computing" OR "Internet of Things")
```

Figure 3.2 – Search string used in the review.

After collecting the papers returned by the search engines, we applied a set of inclusion and exclusion criteria to filter the results. As inclusion criteria, we only considered primary studies (i.e., those that make direct contributions to the field rather than reviewing previous research) with at least four pages written in English. The exclusion criteria eliminate

[1] https://dl.acm.org/
[2] https://ieeexplore.ieee.org/Xplore/home.jsp
[3] https://www.scopus.com/search/form.uri
[4] https://www.springer.com/
[5] https://www.sciencedirect.com/
[6] https://www.wiley.com/

documents that are not regular research papers (e.g., book chapters, technical reports, and patents), duplicates of the same paper, studies that present no explicit validation of proposed solutions, and those not primarily focused on the maintenance of cloud, edge, or IoT devices and applications. Papers targeting Industry 4.0 were also excluded since previous reviews have already surveyed this group of studies (see Section 3.2).

The search string returned 14711 entries. After collecting the initial sample, we started filtering papers according to the methodology shown in Figure 3.3. In the initial filtering stage, inclusion and exclusion criteria filtered papers based on their titles, reducing the number of selected studies to 380. Then, inclusion and exclusion criteria were applied by looking at the paper abstracts, reducing the number of selected studies to 210. Finally, we analyzed the entire manuscript of the remaining studies according to the inclusion and exclusion criteria, reducing the final review list to 42 papers.

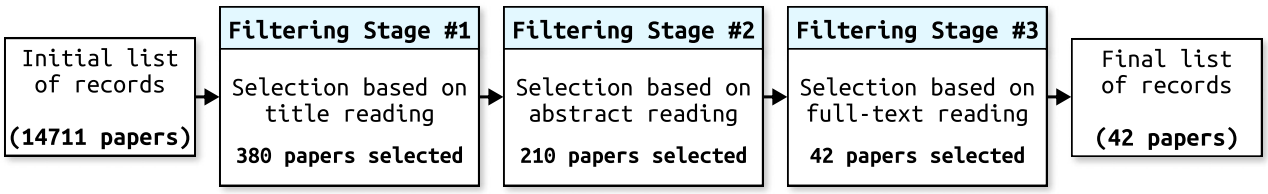


Figure 3.3 – Methodology used to filter papers during the review.

Figure 3.4 answers RQ1 by presenting the distribution of selected papers by target paradigm and year of publication. Despite the growing interest in the subject since 2019, 88.1% of the research efforts have focused on addressing maintenance challenges in cloud and IoT environments, highlighting the initial stage of maintenance research on edge environments. A detailed discussion of existing contributions is presented in Section 3.4.

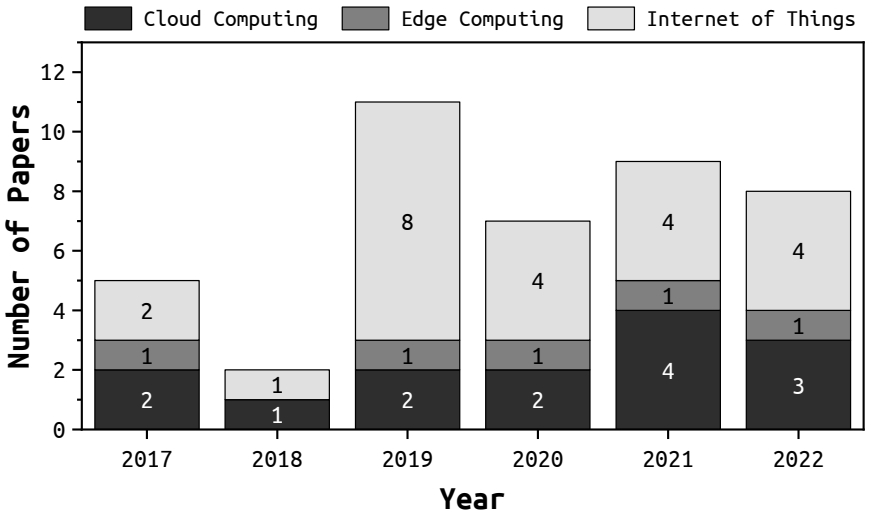


Figure 3.4 – Number of selected papers per year and target paradigm.

3.4 Taxonomy and Survey

This section presents a systematic review of existing maintenance research in the fields of cloud, edge, and IoT. As maintenance in target scenarios comprises various activities, we categorize the surveyed studies according to the taxonomy presented in Figure 3.5, which groups research efforts according to the following characteristics:

- **Target:** Components updated, repaired, or replaced during maintenance.
- **Strategy:** Maintenance strategies employed during the scenarios approached.
- **Technique:** Algorithms and methods used to implement maintenance strategies.
- **Metric:** Performance indicators used to drive allocation decisions during maintenance.
- **Validation:** Evaluation methodologies used to assess proposed solutions.

Our review employs the Faceted Taxonomy model, where observed entities are not constrained to a single branch of the taxonomy [158]. On the contrary, the taxonomy is divided into several facets representing aspects of the observed entities. In this work's scope, a research paper can introduce a corrective maintenance approach (Strategy) based on Reinforcement Learning (Technique) for reducing migration time (Metric) during server updates (Target) in simulated (Validation) data center scenarios. This taxonomy model has been adopted by several reviews in related areas, such as Qu et al. [121] and Liu et al. [93], as it eases the categorization of solutions with common characteristics.

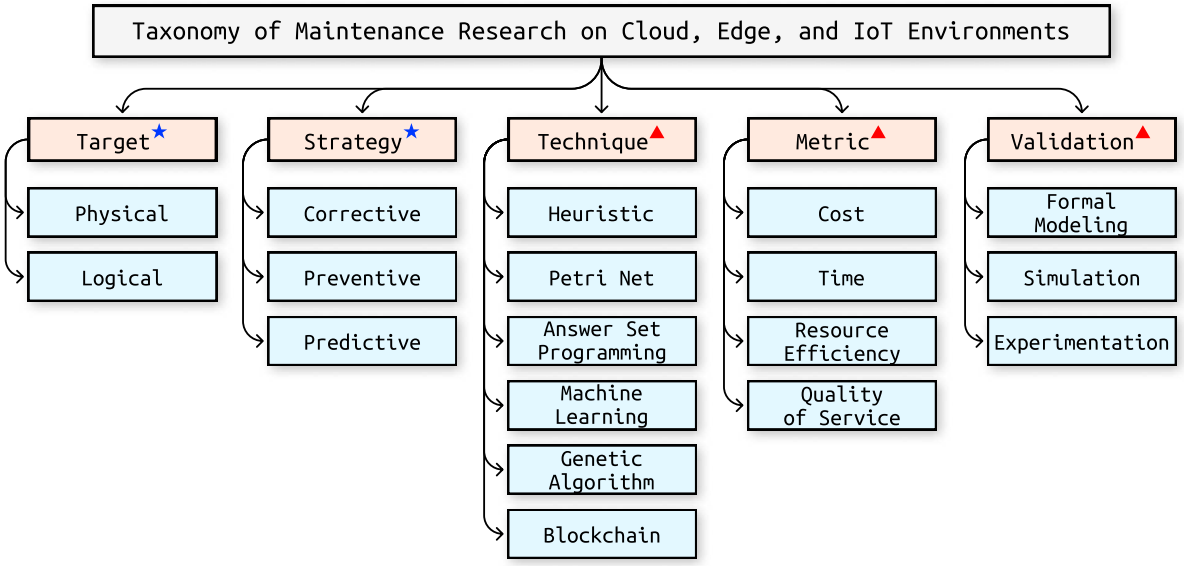


Figure 3.5 – Proposed taxonomy that organizes maintenance research aiming at cloud, edge, and IoT environments. A paper can fit in a single item of categories with the star mark (⋆) and in multiple items of categories with the triangle mark (▲).

The remainder of this section discusses selected maintenance strategies for cloud, edge, and IoT, categorizing them according to the branches of the proposed taxonomy.

3.4.1 Target

The “Target” category indicates the components manipulated in maintenance activities. We divide maintenance targets into physical and logical components, as shown in Figure 3.6. While maintenance work targeting physical components mainly focuses on servers and storage devices, maintenance of logical components comprises low-level control applications (firmware and hypervisors), network applications (routing rules and Virtual Network Functions (VNFs)), and general-purpose applications. Research studies that did not detail the target software were included in the “general-purpose applications” category.

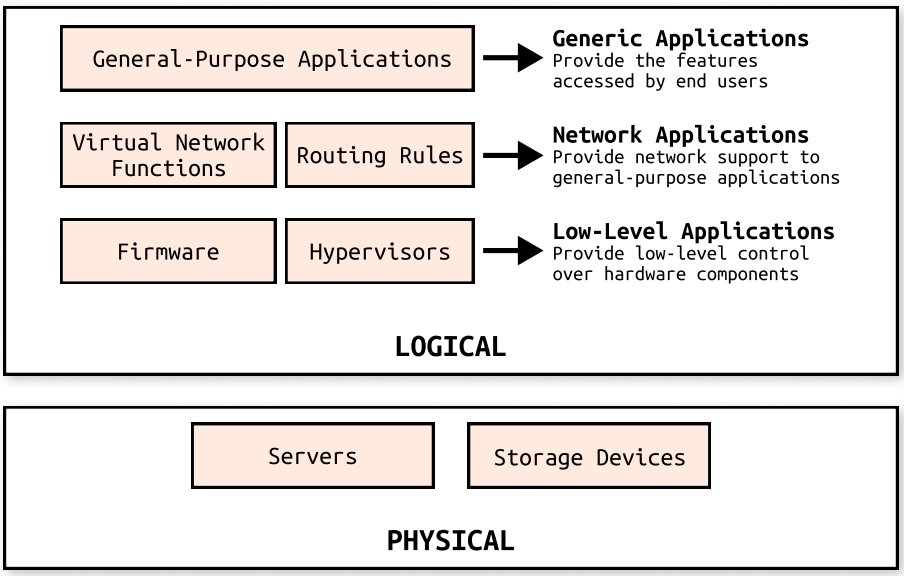


Figure 3.6 – Components targeted by maintenance on cloud, edge, and IoT sites.

Physical Components

Wu et al. [168] was the only study focused on storage-specific challenges during maintenance. The authors considered IoT scenarios composed of mobile nodes that cooperate to process and store data. Although node mobility enhances the network’s flexibility, it raises concerns about fault tolerance, as nodes can move out of the cluster’s coverage area, causing loss of the data they hold. The authors tackled this challenge with a predictive repair strategy based on erasure codes. Once they identify that a node is about to leave the cluster’s coverage, they migrate and reconstruct that node’s data on the other nodes.

Most existing studies on routine server maintenance argue that servers often must be rebooted for updates to take effect [63] [112] [162]. Therefore, the proposed solutions employ rolling update policies, in which servers are grouped into batches according to various criteria, and each batch is updated at a time. Before being updated, servers undergo a draining process, where their applications are relocated to alternative hosts, avoiding application downtime due to server reboots. In this context, some efforts proposed algorithms that orchestrate migrations during maintenance for various purposes, such as reducing maintenance time [174] and avoiding network saturation [63].

In addition to migration planning, some researchers, such as Okuno et al. [112], focused on routine maintenance activities without strict deadlines. As such, these authors presented maintenance scheduling policies that perform maintenance during idle periods to reduce maintenance's impact on applications. Other specific contributions include defining initial application replica placement for scenarios where infrastructure operators cannot migrate applications during server updates due to regulatory constraints [162].

Although routine maintenance typically does not have strict completion deadlines, other events may require fast responses to minimize damage. Wu et al. [167] discussed the impact of power outages on the quality of service of applications in cloud data centers. In this scenario, on the one hand, infrastructure operators must relocate applications hosted by the affected components as early as possible to avoid downtime. On the other hand, poorly planned migrations can prematurely drain the emergency power supply, leading to a complete data center blackout. In response to such a challenge, the authors presented two scheduling algorithms that move applications out from affected servers without unnecessarily increasing the infrastructure's power consumption.

Like power outages, patching security vulnerabilities requires timely decisions to maintain the integrity of the environment. As servers typically need to be rebooted for patches to take effect, preserving application continuity implies relocating them to alternative hosts. Taking into account this scenario, Souza et al. [149] presented two maintenance strategies that make location-based migration decisions, keeping applications near their users to avoid latency increases during server updates on edge infrastructures.

Although virtualization techniques allow applications to be relocated with reduced downtime, Russinovich et al. [127] highlighted that these techniques produce a significant network communication overhead. Accordingly, they employed an approach that replaces relocation techniques by allowing servers to be rebooted faster and without discarding the state of running applications.

Logical Components

In general, existing research on firmware updates focused on optimizing patch distribution for IoT devices. As downloading firmware updates is a prerequisite to start main-

tenance, some studies presented routing strategies to avoid network bottlenecks and allow devices to download patches as soon as possible [89]. In contrast to the traditional update distribution model, where devices download firmware updates from predetermined nodes managed by vendors, several studies employed peer-to-peer architectures [152] [165] [47], where devices can download patches from neighbor nodes, avoiding saturation at specific points of the network and improving fault tolerance in case some of the nodes serving updates fail. As firmware distribution is typically carried out over the Internet, some studies also focused on ensuring the integrity of downloaded updates, especially when they are retrieved from neighbor nodes [83] [12] [61].

Whereas virtualization gives more flexibility to resource management policies, infrastructure operators must keep hypervisors up-to-date to preserve the environment's integrity and performance. One of the primary motivations for updating hypervisors is to avoid software aging, which occurs when hypervisors running for long periods start presenting bugs that can affect applications. Most software rejuvenation approaches have resorted to migrating the running applications from aged hypervisors to healthy ones, which can be co-located [25] or within different hosts within the infrastructure [43] [156].

Although migration techniques allow infrastructure operators to rearrange applications within the infrastructure during maintenance, the number of simultaneous migrations is limited by the resulting network communication overhead. In addition, seamless migration often requires compatibility between hypervisors, narrowing the migration options on heterogeneous infrastructures. Alternatively, some studies focused on in-place upgrade strategies, where hypervisors are patched without migrating applications.

Segalini et al. [137] presented Hy-FiX, an in-place upgrade solution for Kernel-based Virtual Machine (KVM)[7] hypervisors. Hy-FiX combines two checkpoint techniques (*suspend-to-disk* and *suspend-to-RAM*), which dump VM data to disk but keep its state in memory for faster recovery. Additionally, Hy-FiX employs a lazy initialization technique that reduces server reboot time, making in-place upgrades less intrusive for applications. Ngoc et al. [110] introduced a solution called HyperTP, which supports Xen[8] and KVM hypervisors and allows infrastructure operators to choose whether to migrate applications before updating hypervisors (out-of-place upgrade) or temporarily pause applications during patching (in-place upgrade).

The lack of compatibility among network appliances from different vendors and other manageability issues have paved the way for Software-Defined Networking (SDN) [82], which decouples control and data planes and gives more freedom to network operators. In a typical SDN infrastructure, controller nodes manage the entire network, defining, for example, the routing rules that guide packet-forwarding devices. Whenever routing rules change, controllers must broadcast them to all forwarding devices to ensure that the net-

[7] https://www.linux-kvm.org/page/Main_Page
[8] https://xenproject.org/

work is up-to-date. Whereas such a centralized model grants fine-grained control, frequent rule changes can lead to significant network communication overhead. In response, some research efforts presented algorithms that schedule route update flows to avoid network saturation [122] [109]. Other specific contributions related to the maintenance of network services include the repair of deprecated routing rules in scenarios with link failures [14] [108].

In addition to enabling better management of routing rules, SDN provides the underlying features for Network Function Virtualization (NFV) [58], which replaces traditional hardware appliances by hosting Virtual Network Functions (VNFs) such as firewalls and intrusion detection systems on general-purpose servers. Maintenance research efforts targeting VNFs focused primarily on improving their fault tolerance. Raza et al. [153] presented a checkpoint and rollback VNF failure recovery strategy that regularly takes checkpoints and timely rolls back VNFs as failures are detected. Other studies also predict host failures and proactively provision VNFs on alternative servers to avoid downtime [65] [67].

Maintenance research efforts targeting general-purpose applications on cloud and edge sites have focused on repairing failures. Saxena et al. [135] presented an algorithm that proactively identifies and mitigates resource starvation problems, in which applications with time-varying workloads overload their hosts, causing performance degradation and other related issues. Olorunnife et al. [113] proposed a framework that tackles software failures caused by runtime changes in containerized applications. Whenever the proposed framework identifies a failing application, it rebuilds it based on healthy nodes metadata, discarding any settings that might have caused the failure. Other investigations also focused on repairing hang bugs [60] and avoiding aging-related issues [100].

Research works also focused on general-purpose IoT applications committed to scheduling and security challenges in the distribution of updates. Weißbach et al. [164] and Bui et al. [22] proposed patch distribution scheduling strategies that consider software dependencies that add constraints regarding the update order. Regarding security-related efforts, the focus was on defining nodes to distribute patches to their neighbors in peer-to-peer infrastructures [54] [10] [96]. In such a scenario, choosing isolated nodes to distribute patches can delay maintenance. In addition, compromised nodes can propagate malicious software across the infrastructure.

Summary and Takeaways

Existing maintenance work covers repairing, upgrading, and replacing various physical and logical components in cloud, edge, and IoT environments. Most research efforts targeting the maintenance of physical components focused on cloud data centers considering hyper-converged infrastructures [124] [57], which replace storage appliances with servers that bundle both compute and storage capabilities. Consequently, most of the strategies were designed for conducting server maintenance.

As for research on the maintenance of logical components, we can observe specific directions followed by cloud and IoT papers. While cloud research focused on hypervisor maintenance, most IoT research optimized resource usage, cost, and security during patch distribution. The few papers targeting logical components at edge sites presented maintenance strategies for VNFs and general-purpose applications.

3.4.2 Strategy

The category “Strategy” discusses the role of the different maintenance approaches presented in the taxonomy of Figure 2.4 (i.e., corrective maintenance, preventive maintenance, and predictive maintenance) in the research efforts for cloud, edge, and IoT environments. In addition to summarizing the commonalities between solutions under the same maintenance strategy, we present a holistic view that examines the pros and cons of each approach regarding the various maintenance demands in the addressed paradigms.

Corrective Maintenance

Most maintenance research efforts focused on corrective strategies, acting on failure or delegating the decision of “when” to initiate maintenance to third-party entities or events (e.g., the release of security patches or the arrival of new components to replace existing ones). Consequently, proposed solutions have focused mainly on optimizing “how” maintenance takes place, which comprises several decisions, such as defining strategies to reduce maintenance impact on applications [137] [110] and the order of updates for components when they cannot be updated at once (typically due to QoS constraints) [149] [174].

Although some corrective maintenance strategies can advance or postpone maintenance work in a convenient way in scenarios without strict deadlines, they act on the premise that the decision to perform maintenance has already been taken, solely deciding the best schedule for it [112] [174]. In such cases, the decision to perform maintenance often comes from external sources, such as anomaly detection algorithms, which analyze logs and performance metrics to identify problematic events [87] [49]. In computing infrastructures, anomaly detection algorithms rely on monitoring systems (e.g., Zabbix[9], Nagios[10], and Instana[11]) that collect data through monitoring agents installed in the hardware and various software layers (hypervisor, operating system, and applications).

Despite growing evidence about the potential of proactive strategies [179], there is concern about prediction errors, as maintenance work performed unnecessarily can quickly

[9] https://www.zabbix.com/
[10] https://www.nagios.org/
[11] https://www.instana.com/

saturate the infrastructure and degrade application performance. Consequently, corrective maintenance has remained in the view of the research community, especially due to the potential risks incurred by some events that are hard to predict (e.g., power outages [167] and certain cyberattacks [156]). In such cases, maintenance strategies have mainly focused on mitigating the problem that triggered the maintenance as efficiently as possible rather than taking the risk of making wrong maintenance decisions based on inaccurate feedback from proactive approaches.

Preventive Maintenance

Although corrective maintenance is effective in some scenarios, preventive approaches can reduce maintenance costs when handling issues with deterministic behavior, such as software aging. Preventive maintenance strategies mitigate software aging through software rejuvenation techniques, which typically involve gracefully restarting software components to clean up their internal state [66].

As discussed in Section 2.4, preventive maintenance comprises schedule-based and condition-based approaches. While schedule-based maintenance strategies do not require real-time monitoring mechanisms, finding appropriate maintenance intervals is challenging in some situations. On the other hand, condition-based maintenance strategies grant enhanced flexibility but assume that degradation advances slowly enough to enable on-point correction. Consequently, existing preventive maintenance strategies employ schedule-based and condition-based policies for convenience, often combining them when needed.

Fakhrolmobasheri et al. [43] discussed the criticality of hypervisor rejuvenation in virtualized infrastructures, as hypervisor failures intrinsically affect all hosted VMs. In such a scenario, aging occurs due to several factors, such as the lifetime and workload pattern of hosted VMs. The authors employed a condition-based maintenance approach that avoids aging-related failures through a two-threshold policy. Whenever a hypervisor indicates slight aging, the first threshold is triggered, and hosted VMs are migrated if the hosts of other hypervisors have enough available resources. However, since migrations are not mandatory, a hypervisor can reach an advanced aging level where failures are more likely to occur before its virtual machines are relocated. In that case, the second threshold is triggered, and hosted VMs enter the queue for immediate migration regardless of the occupation of alternative hosts, as their hypervisor must be evacuated for rejuvenation as early as possible. The proposed solution also performs migrations to consolidate VMs on the most occupied hosts, switching off idle servers to save power and reverse any aging effects of hosted hypervisors, which makes hypervisors ready for later usage if their servers are restarted.

Meng et al. [100] tackled aging-related issues more broadly, discussing the role of software rejuvenation in preserving the health of general-purpose applications. The authors described that aging progression damages components until it leads to functional failure.

In such a scenario, if threshold-based maintenance policies are able to detect aging properly, rejuvenation is eventually triggered, resetting the accumulated damage. However, if the aging progression pattern is unknown to the thresholds, maintenance takes place only in a corrective fashion after software failure. After discussing the risks of relying solely on threshold-based maintenance policies, the authors implement a hybrid strategy that triggers software rejuvenation at certain damage thresholds or predefined intervals, whichever occurs first. Accordingly, even failures that show an unknown progression that could bypass the thresholds are mitigated by maintenance work performed at fixed intervals.

Predictive Maintenance

Preventive maintenance strategies are based on maintenance intervals (schedule-based maintenance) and damage thresholds (condition-based maintenance), which might be hard to define in some scenarios. In addition, preventive maintenance requires certain assumptions to be true for the system to function correctly. While schedule-based maintenance assumes that damage progression is predictable so that failures do not occur between maintenance runs, condition-based maintenance assumes that damage progresses slowly enough so that the intervals in which thresholds are triggered and functional failures are sufficiently long for maintenance personnel to act. As such assumptions may not be satisfied in certain scenarios, some research works leverage predictive maintenance approaches to forecast the upcoming system state and take more effective measures.

Liu et al. [168] focused on increasing disk fault tolerance in clusters of mobile IoT nodes (e.g., smart vehicles and drones). In such a scenario, mobile nodes can quickly leave the cluster's coverage area, causing loss of the data they store. The authors presented two proactive data repair techniques based on predictions that indicate nodes about to leave the cluster's area (so-called soon-to-fail nodes). More specifically, the first technique replicates data from soon-to-fail nodes into healthy nodes through migration over the network, and the second technique proactively reconstructs data chunks from several healthy nodes through erasure codes. While replication and erasure codes optimize various aspects within the system (i.e., avoiding single points of failure and reducing disk usage), the authors do not detail how node mobility prediction happens, delegating this feature to third-party entities.

Huang et al. [65] discussed the challenges of providing fault-tolerant VNFs on edge infrastructures. In such a scenario, the authors argued that the potential failure surface at the edge spans physical and logical layers in the edge servers and network components, such as switches and links. Additionally, acting reactively upon failure incurs a high operational cost as VNFs provide the core underlying features for end-user applications (e.g., firewall, load balancing), and the lack of these features even for a short period can lead to significant security and performance issues. To address these issues, the authors presented a proactive VNF failover architecture that details the requirements and core components to

provide resilient VNF services at the edge. In addition, Huang et al. [67] presented a novel algorithm that predicts network failures and proactively reprovisions VNF instances on unaffected components to avoid service outages.

Saxena et al. [135] focused on ensuring high availability and fault tolerance for general-purpose applications in cloud data centers. The authors argued that resource saturation represents a significant portion of service outages in cloud environments, often occurring due to unexpected workload variations and over-extended time needed for reprovisioning applications. Therefore, they introduced a resource utilization forecast mechanism that anticipates server overutilization. The proposed solution categorizes applications into two groups, normal and failure-prone, depending on whether their servers have sufficient resources for the near future or they are about to be overloaded, respectively. Then, the resource usage predictions feed migration and replica placement policies that provision failure-prone applications on alternative servers to avoid service outages and improve the data center's resource efficiency.

Summary and Takeaways

Existing maintenance work employed different approaches depending on the addressed scenarios. Corrective solutions typically offload the decision of when to start maintenance to third-party entities (e.g., external analytics tools), focusing on components with low repair costs without strict maintenance completion deadlines or when maintenance triggers are caused by hard-to-predict events (e.g., security patch releases). We can also observe that preventive maintenance is favored when maintenance triggers display well-defined patterns (e.g., software aging). Finally, predictive maintenance is often used when proactiveness is a requirement and monitoring indicators move too fast for preventive approaches to be used. In such cases, predictive methods can forecast the state of the system to support infrastructure operators with actionable insights and sufficient time to work.

3.4.3 Technique

The "Technique" branch classifies the methods and algorithms used within the maintenance strategies surveyed. As some studies propose conceptual maintenance architectures instead of algorithms to perform maintenance, we put them into the "Heuristic" category. To this end, we follow the definition of Romanycia et al. [125], considering heuristic any procedure that adopts rules of thumb to solve a certain problem and, within the scope of this review, that also does not fit into the other methods and techniques discussed.

Heuristic

As maintenance encompasses the coordination of several complex decision-making processes, most of the strategies in the literature employ heuristic procedures to obtain sufficiently good solutions in a reasonable time. Maintenance heuristics generally incorporate groups of predefined rules that trigger various actions during maintenance. Despite the significant differences among solutions due to the varying requirements of the scenarios addressed, maintenance heuristics often have similarities, such as the adoption of sorting policies based on custom cost and score functions to make decisions such as scheduling application migrations and defining the update order of components [63] [27]. Furthermore, some heuristic solutions also make more specific decisions, such as identifying maintenance trigger events such as software bugs [60].

While most maintenance heuristics implemented problem-specific procedures to make accurate decisions, some proposals employed ensemble approaches, which combine multiple techniques to cope with broader scenarios or to increase the solution's robustness (e.g., targeting global optima rather than local optima). Saxena et al. [135] proposed an ensemble maintenance heuristic called OFP-TM. While OFP-TM follows a set of predefined rules during allocation decisions to improve the fault tolerance of cloud applications, it employs Artificial Intelligence models to proactively identify failure-prone applications.

Petri Net

The need to represent the variety of simultaneous processes and events during maintenance has drawn the community's attention to modeling techniques such as Petri Networks (Petri Nets) [107], which provide means for graphically modeling complex processes. Petri Nets comprise groups of places, transitions, and arcs. While places and transitions denote system states and events that move the system from one place to another, respectively, arcs represent relationships between places and transitions. In this setting, tokens denote the Petri Net workflow as its transitions are triggered. Some studies used Petri Nets to model dynamic maintenance systems where concurrent events can occur, subject to precedence and frequency constraints. Specifically, we observe an interest in Petri Nets in model maintenance scheduling algorithms in software rejuvenation scenarios, where concurrent events, such as user requests, directly affect how fast component aging progresses.

Fakhrolmobasheri et al. [43] proposed a hypervisor rejuvenation system based on Stochastic Activity Networks (SANs) [129], an extension of Petri Nets that facilitates the modeling of activities such as user requests whose duration impacts system performance. The proposed SAN model comprises five sub-models representing the arrival of concurrent user requests, application migrations between hypervisors, a shutdown mechanism for idle servers, hypervisor rejuvenation, and aging-related failures. In this setting, several functions

determine appropriate times to rejuvenate the hypervisors to mitigate software failures while reducing the infrastructure's power consumption.

Torquato et al. [156] followed a similar line of reasoning, employing Stochastic Rewards Networks (SRNs) [28], which extend SANs with reward-based functions to measure the reliability of complex systems. The authors presented a maintenance system that schedules application migrations to rejuvenate hypervisors while reducing the vulnerability surface of applications against network attacks.

Answer Set Programming

Maintenance modeling typically encompasses several problems with high computational complexity, such as application migration [140] and process scheduling [40]. As a result, finding optimal solutions to large-scale maintenance problems in a reasonable time is often unfeasible. Although heuristics are typically used as an alternative approach, defining the appropriate rules of thumb for such strategies might be challenging in some maintenance scenarios with several components and dynamic behavior.

In this context, Answer Set Programming (ASP) [98] comes into the spotlight as an efficient alternative that, unlike traditional programming paradigms, which require explicit search instructions, employs Answer Set Solvers to efficiently find solutions through the definition of an objective function and constraints that must be satisfied for a solution to be considered valid [39]. In this way, ASP avoids suboptimal solutions resulting from inaccurate algorithmic instructions.

Okuno et al. [112] employed ASP to formulate a cloud server maintenance problem that focuses on optimizing two conflicting goals: preserving application availability and reducing maintenance time. To this end, the proposed model tries to minimize maintenance time while a set of constraints restricts the migration scheduling during periods where applications are under heavy load. Given the high complexity of defining optimal maintenance scheduling in such a scenario, the proposed solution employs a Divide-and-Conquer approach that partitions maintenance scheduling into subproblems composed of groups of servers and time slots. After solving each subproblem individually, the proposed solution combines the partial solutions, obtaining the optimal maintenance schedule for the entire data center. The authors demonstrated that the proposed solution finds optimal maintenance schedules while displaying reduced time and space complexity compared to the baseline approach that employs ASP without the proposed divide-and-conquer strategy.

Machine Learning

Efficient maintenance planning requires extensive effort, even for domain experts, as resource allocation decisions during maintenance (e.g., component update order defini-

tion or application migration scheduling) can lead to undesirable cascading events that are difficult to track. This problem becomes even more challenging for the maintenance of critical assets, where proactive decisions are required to reduce repair costs and avoid service disruption. In this context, some research efforts advocated the usage of Machine Learning (ML) [74] models to obtain accurate maintenance decisions. Unlike traditional algorithms, ML models can discover hidden patterns and correlations in input data to identify ongoing events or predict upcoming environment states.

Several ML algorithms learn how to approach target problems through a training phase, building their internal rules through identified patterns in historical data. In this context, ML learning algorithms can define biased rules if their starting search point in the training dataset leads to local optima. To overcome such challenges, a branch of ML called Ensemble Learning [36] combines the output of multiple ML algorithms. The diversity of Ensemble methods often grants them higher accuracy than individual models, as they are not restricted to the decisions of a single learning algorithm. Saxena et al. [135] presented an Ensemble algorithm that combines three ML models (Feed-Forward Neural Network, Support Vector Machine, and Linear Regression) to predict failures in cloud applications. Similarly, Huang et al. [65] used a tree-based ensemble algorithm called Random Forest to predict failures that could affect VNFs on edge infrastructures.

Although ML models display great potential for handling non-trivial optimization problems, they typically require significant training data. As such, they fall short when the input data frequently changes or when there are no historical data for training. In response, Reinforcement Learning (RL) [75] emerged as a branch of ML that employs intelligent agents capable of adjusting their internal rules on-the-fly based on reward and penalty systems.

Motivated by the benefits of RL algorithms, Nain et al. [109] employed this type of model to tackle the challenge of coordinating the distribution of routing rules in SDN-supported IoT environments. In such a scenario, while indiscriminately broadcasting updated routing rules to nodes incurs network saturation, excessively delaying the distribution of new rules may expose the network to transmission failures and security vulnerabilities. Consequently, the authors employed an RL algorithm called Q-Learning, which dynamically coordinates the distribution of updated routing rules to ensure that nodes are up-to-date as early as possible without causing network saturation. Using a different scenario, Ying et al. [174] used Reinforcement Learning to schedule application migrations during maintenance in cloud data centers where there is no prior knowledge about the state of the network (e.g., link delays, bandwidth usage, etc.).

Genetic Algorithm

Like declarative paradigms and ML models, metaheuristics [19] are heuristic replacements that define high-level and problem-independent procedures for tackling complex

optimization problems. One of the most known metaheuristic methods is called the Genetic Algorithm (GA) [102], which mimics the biological process described by Darwin's theory of evolution by natural selection. Rather than evolving solutions independently, GAs employ a population-based search strategy that evolves multiple solutions simultaneously according to predefined objectives. One of the main features of GAs is the use of mutation and crossover mechanisms to balance exploration and exploitation when looking for efficient solutions [30]. While exploration suggests visiting new regions in the search space (often far from explored points), exploitation suggests visiting the surroundings of already explored regions.

Souza et al. [149] employed a multiobjective GA called Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [34] to perform server maintenance on edge infrastructures while considering application latency requirements. Unlike traditional GAs, NSGA-II is designed to find Pareto-optimal solutions for multiobjective optimization problems through an elitist approach that favors non-dominated solutions. As the authors adopted a batch-based maintenance model in which the servers drained in a given batch are updated in the next, the application migration schedule affects the server update order. Consequently, the proposed solution finds the best application migration scheduling to reduce maintenance time without neglecting end-user latency requirements.

Blockchain

Patch distribution is a vital maintenance process that involves transferring updated binaries over the network from vendors to outdated nodes. One of the main challenges with patch distribution is ensuring the integrity of patches against potential malware injections into the network. Patch distribution security is even more concerning in peer-to-peer infrastructures, where the attack surface goes beyond the network, as compromised nodes may distribute malware to neighboring nodes. In this context, distributed ledger technology such as Blockchain [178] comes into the spotlight as distributed databases that store transaction records on multiple nodes simultaneously, allowing the implementation of consensus mechanisms to prevent malicious changes during patch distribution.

Several research works have proposed solutions using Blockchain technologies such as Bitcoin[12], Hyperledger Fabric[13], and Ethereum[14] for secure patch distribution. Most of the proposals presented frameworks that leverage the Blockchain's block validation process, where individual transactions (e.g., patch submissions) are verified before being added to the permanent ledger of the Blockchain, reducing the chances of malware distribution on the network [61] [5] [106] [157] [64] [165] [10]. As Blockchain security increases as more nodes join the network, some research efforts also presented incentive strategies that compensate nodes for participating in the Blockchain network as patch distributors [89] [152] [47].

[12]https://bitcoin.org/
[13]https://www.hyperledger.org/
[14]https://ethereum.org/

Summary and Takeaways

Existing maintenance solutions use multiple technologies depending on the addressed scenarios. Most of the solutions employed heuristic procedures to increase flexibility and improve the prototyping speed. We observe a clear preference for ML to develop predictive approaches. In addition, Petri Nets serve as good alternatives for graphically modeling complex maintenance systems, while GAs and ASP facilitate the development of maintenance strategies for scenarios where it is challenging to define algorithmic procedures to find solutions. Finally, most approaches focused on distributing firmware to nodes in IoT environments used Blockchain technology for enhanced security, which is vital as IoT devices are typically installed outdoors, which increases the risk of physical tampering, and connectivity is often provided by public wireless networks, which widens the attack surface.

3.4.4 Metric

The “Metric” category organizes and discusses the metrics used as performance indicators to evaluate maintenance strategies. We group the metrics evaluated into four groups: Cost, Time, Resource Efficiency, and Quality of Service, as depicted in Figure 3.7. A single metric (or variations of it) can be used to evaluate several performance facets, given the wide variety of maintenance scenarios addressed in the reviewed papers.

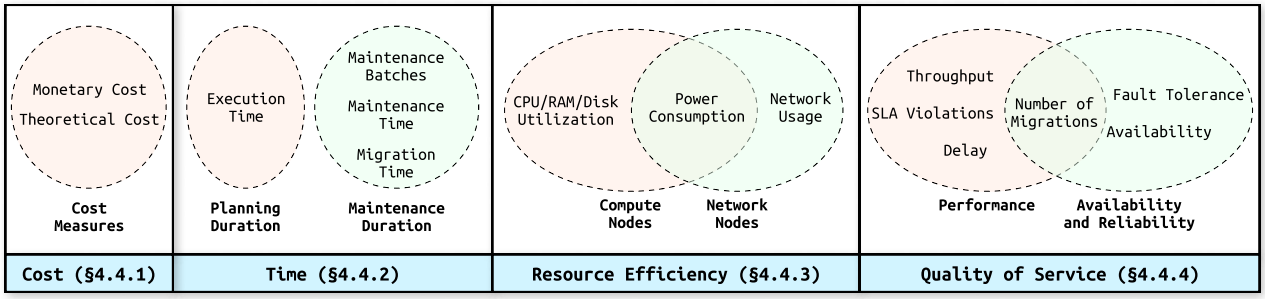


Figure 3.7 – Classification of metrics used to evaluate maintenance strategies.

Cost

Most maintenance research works that employ cost metrics are somehow related to Blockchain’s usage. In Blockchain networks, miners play a critical role in preserving network security by validating each transaction before adding it to the Blockchain ledger [178]. As an incentive, miners are rewarded for the computational effort incurred by transaction validation through network fees charged to Blockchain users [38]. In addition to incentivizing miners’ work, fees help prevent spam attacks, which become costly to implement at scale. Although

network fees are vital in the Blockchain's security ecosystem, some maintenance papers evaluated the cost efficiency of proposed Blockchain systems, as indiscriminately inflated fees can hinder Blockchain's sustainability [89] [10] [5] [47] [106] [157].

In this line of reasoning, Tapas et al. [152] presented P^{4}UIoT, a firmware update distribution system that uses Bitcoin's Lightning Network[15] to increase network throughput during patch distribution without excessively raising network fees. The Lightning Network creates payment channels between parties in the network, allowing cheaper and faster transactions to be exchanged outside the main Blockchain. Once the payment channel is closed, off-chain transactions are consolidated and broadcast to the main Blockchain. As just a single transaction resulting from the payment channel is added to the main Blockchain ledger, the Lightning Network enables reduced traffic on the Blockchain and lower fees.

Huang et al. [67] deviated from the Blockchain's scope, employing cost as a performance indicator during the maintenance of failing VNF deployments comprised of a master and multiple backup instances. While failures in the master VNF instance require selecting a new master among the backup instances and updating the routing path accordingly, failures in the backup instances only require the deployment of replacement backup instances. In this context, the authors presented a failure recovery cost function that considers the VNF recovery time during failures in the master and backup instances.

Time

Performance and security concerns typically place time efficiency as a critical performance indicator for maintenance strategies. This happens because maintenance work comprises costly processes such as application relocations that produce significant side effects such as service disruption and infrastructure saturation. Additionally, maintenance completion often means mitigating undesirable conditions such as performance degradation, component failures, and security vulnerabilities. In general, research efforts measure the time efficiency of maintenance strategies from two perspectives: (i) planning time and (ii) maintenance execution time. Planning time studies the feasibility and scalability of proposed strategies in terms of the time needed to find appropriate maintenance action plans. Maintenance execution time evaluates how much it takes to execute the maintenance plans, either from a global perspective or looking at the duration of different procedures during the maintenance implementation (e.g., patching, application migrations, etc.).

Okuno et al. [112] evaluated the time needed to define the rolling update schedule for cloud servers. In such a scenario, applications must be migrated to alternative hosts before their servers undergo maintenance to avoid downtime. As such, migration decisions can have cascading effects on how maintenance proceeds. As looking for optimal solutions can become impractical due to the large search space, the proposed maintenance scheduler

[15] https://lightning.network/

employs a divide-and-conquer approach that extends an ASP-based model to find quality solutions within a reasonable time. The authors evaluated the execution time of their scheduler in different scenarios with a varying number of servers and applications, comparing the baseline ASP model to the proposed divide-and-conquer approach. During the evaluation, they also measured the time taken by the divide-and-conquer approach to solving the smallest and largest scheduling subproblems, identifying possible performance bottlenecks.

In a similar line of reasoning, Bui et al. [22] used planning runtime as a performance indicator during the maintenance of composite IoT applications. The authors compared the running time of a proposed heuristic against the CPLEX mathematical solver in five scenarios with different numbers of devices, demonstrating that the proposed heuristic could find near-optimal solutions orders of magnitude faster than the CPLEX solver.

Research works evaluated maintenance execution time through two approaches: (i) the number of batches and (ii) the duration of events during maintenance. Maintenance modeling generally encompasses several events that can occur simultaneously and potentially affect each other. As representing such interactions incurs high computational effort at scale, batch-based maintenance modeling emerges as a lightweight alternative.

Although the batch-based model allows measuring the duration of specific events during maintenance (see Souza et al. [149] for example), some studies employ simplified models where maintenance time is evaluated by the number of batches needed to update all target components. Following this reasoning, Hou et al. [63] used batch-based modeling to evaluate the time efficiency during edge server updates. In this scenario, each batch incorporates the selection of the servers that will undergo maintenance and the migration of applications on those servers to alternative hosts before the update starts. The authors assessed the effectiveness of migration decisions in reducing the number of batches needed to complete maintenance.

Wang et al. [162] used the number of batches to evaluate maintenance plans in a more restrictive scenario, where regulatory rules prevent applications from migrated at runtime. In response to this constraint, the authors leveraged replica placement strategies to avoid service disruption during maintenance. By spreading replicas across the infrastructure, maintenance can advance gradually without affecting application availability as long as at least one replica of each application is hosted outside the group of servers updated in each batch. In such a scenario, the resulting number of maintenance batches indicates the effectiveness of replica placement strategies.

Although the number of batches provides a notion of time, most research efforts evaluated maintenance strategies by unraveling maintenance time according to the duration of various events, such as application migrations, recovery, patching, and other specific operations. In addition to indicating how time-consuming relocations are during maintenance [149], migration time is also used to measure application downtime [25], which is related to the quality of service metrics discussed in Section 3.4.4.

There are different approaches to measuring the update time depending on the scenario and the techniques used. While some investigations discussed the generic benefits of shorter update times, such as performance improvements [149], other works indicated specific gains, such as reduced attack surface (security patches) [110] [27], and shortened service disruption (failure correction updates) [113] [168] [135] [60] [127].

Specific update time measures are seen in some maintenance scenarios, such as in-place hypervisor updates, where the time to store/restore application states and restart the servers is also considered [137] [110] [127]. Other operations comprised within the maintenance time include the computations performed to decode encrypted patches [12] [54] and the time required to release patches on Blockchain networks [10] [5].

Resource Efficiency

As maintenance encompasses several processes that might cause resource saturation in the network and compute nodes, some research work considered resource efficiency as a performance indicator while evaluating maintenance strategies.

Network resource efficiency is predominantly assessed during broken link recovery in SDN infrastructures, where control packets with updated routing rules must be distributed to forwarding devices to isolate failing resources and avoid packet losses. In such scenarios, decisions disregarding the dispatch order and the bandwidth reserved for propagating control packets can render damaging effects such as network starvation. Consequently, resource efforts targeting this scenario evaluated the resource efficiency of maintenance strategies with respect to the number of links and bandwidth used during the distribution of control packets to identify over-consuming approaches [122] [14].

Regarding compute node resource efficiency, there is a preference for reducing memory consumption during the maintenance of low-level applications (i.e., firmware and hypervisors). While firmware update approaches aimed to minimize memory usage in operations such as patch propagation, downloading, decompression, integrity check, and update installation [83] [44] [106], hypervisor repair papers focused on minimizing the memory requirements of in-place updates and application migrations [137] [110].

In addition to raw compute and network usage, power consumption was also considered to assess resource efficiency in scenarios where energy-draining operations could impact service continuity and sustainability goals. Wu et al. [167] discussed the impact of migration decisions on energy efficiency during data center evacuation under power outages. Although data centers typically have abundant power supplies, this situation is reversed during power outages, where infrastructure operators must define application migration plans to evacuate affected data centers while spare power supply systems are still working. In such a scenario, inefficient migration decisions can drain the backup energy supply and cause various issues, such as data loss and hardware failures due to improper halting.

Fakhrolmobasheri et al. [43] discussed how large-scale cloud data centers represent a substantial portion of society's electricity usage and how this impacts greenhouse gas emissions. As such, the authors included power efficiency as a maintenance target for hypervisor rejuvenation work. Other research efforts focused on minimizing maintenance power consumption in IoT environments, where expensive operations such as Blockchain computations and network transfers can quickly drain the battery life of computing devices [106] [5] [108].

Quality of Service

As maintenance encompasses several resource-intensive activities, there is concern about their possible degradation in the quality of service for end users. In this context, network-related metrics such as delay and throughput come into the spotlight as key qualities of service indicators.

Banikhalaf et al. [14] evaluated the application delay during route repairs in vehicular networks, where wireless connectivity and vehicle movement amplify the potential impact of improper route changes. He et al. [61] approached a different scenario, evaluating the delay of transactions that verify the integrity of firmware patches on Blockchain networks, in which increased response times can result from bottlenecks both on the network and on computing nodes.

Other efforts evaluated the impact of network-hungry operations such as patch distribution [64] [165] and routing rules propagation [109] [108] on application throughput. Given that several events during maintenance can undermine the applications' behavior during maintenance (e.g., migrations, software failures, etc.), affecting the end-users' quality of service, some maintenance strategies were also evaluated regarding their impact on application reliability and service continuity.

Maintenance works that included fault tolerance metrics in their evaluation were primarily concerned with preventing failures or mitigating their effects on the environment. He et al. [60] concentrated on the repair of hang bugs, which are difficult to identify as they undermine the system's responsiveness without causing explicit failures. Taking this into account, the authors evaluated the effectiveness of repair strategies with respect to the number of partially and completely fixed hang bugs.

Similarly, Fakhrolmobasheri et al. [43] evaluated the impact of hypervisor rejuvenation strategies on preventing software failures. During their evaluation, the authors employed several fault tolerance metrics, such as the number of hypervisor failures, the ratio of serving VMs to accepted requests, and the number of VM failures. Torquato et al. [156] evaluated the impact of migrations performed to mitigate hypervisor aging on application availability. Other research efforts discussed the potential downsides of migrations more generically, highlighting their implications on application performance and availability [135] [63].

Some works also evaluated the impact of maintenance on the quality of service in terms of both application performance and service continuity. Souza et al. [149] concentrated on server updates on edge infrastructures, where maintenance decisions, such as server update order and application migrations, can undermine the availability and performance of applications. Whereas ineffective update order decisions can result in unnecessary migrations, which cause temporary service disruption, poorly planned migrations can place applications on edge servers far from users, resulting in communication delays that conflict with the existing SLAs, which represent the performance expectations of end-users in terms of application delay. Some research efforts also discussed the performance and service continuity implications of maintenance activities such as in-place updates and application migrations during hypervisor repairs [137] [25] [110].

Summary and Takeaways

Maintenance typically aims to mitigate or prevent critical issues, such as security vulnerabilities and infrastructure failures, where poor decisions can lead to severe consequences. As such, the quality of maintenance strategies is often measured through cost metrics that consider several factors, from monetary charges to failure costs. Several research efforts also evaluate maintenance strategies regarding time-related metrics such as patching time and the number of batches/rounds required to complete the maintenance.

Although maintenance helps preserve the performance and security of target environments, it encompasses several resource-intensive activities that can cause harmful effects on the infrastructure if not accompanied by proper planning and execution. In this context, resource usage and QoS-related metrics are also considered to assess the feasibility and efficiency of maintenance strategies. Whereas resource efficiency includes the demand of compute nodes (i.e., CPU, memory, and disk), the network occupation (ingress/egress traffic and the number of used links), and the infrastructure's power consumption, QoS is typically represented through network-related metrics (e.g., delay and throughput) and service continuity (e.g., availability and the number of failures). Some research works also consider the number of migrations in QoS assessments since applications often suffer from downtime and degraded network performance during migrations.

3.4.5 Validation

The category "Validation" indicates the validation approaches used to evaluate maintenance solutions. We divide the validation approaches into three groups: (i) formal modeling, (ii) simulation, and (iii) empirical experimentation. While simulation is self-

explanatory, formal modeling comprises the adoption of mathematical models and formal analyses, and empirical experimentation implies conducting experiments on practical testbeds.

Formal Modeling

Formal methods enable the analysis and evaluation of research prototypes without requiring practical implementations, allowing cost savings related to building and maintaining real testbeds or developing simulation tools.

Hu et al. [64] used a formal analysis to validate their approach to secure delivery of firmware updates to IoT devices. The authors introduced several theorems demonstrating their system's effectiveness in preventing various cyberattacks, such as denial of service and the distribution of malware pretending to be regular firmware updates. In addition, they presented a computational complexity analysis to compare the proposed solution with an approach from the literature. Okuno et al. [112] also employed a formal method during their evaluation, representing the scenario addressed with *clingo* [51], an ASP system that enables the modeling of large-scale combinatorial problems as logic programs.

Simulation

Although formal modeling methods are suitable for early-stage research projects, they typically require the adoption of many high-level abstractions to find reasonable solutions in a feasible time. In this context, simulation comes into the scene as an alternative that allows for rapid prototyping with more realistic conceptual models and lower implementation costs. Most existing research efforts that present simulation-based evaluations used custom simulators [100] [167] [27] [63] [122] [22] [10] [65] [162] [67] [135] [168] [174]. Although this facilitates the modeling of specific scenarios, it usually undermines performance comparisons, especially since the source code of custom simulators is often private. Despite the predominance of custom simulators whose source code is not publicly available, some research work used open-source simulation frameworks. In general, maintenance simulations use three types of simulation tools: (i) general-purpose simulators, (ii) network simulators, and (iii) maintenance-related simulators.

Malik et al. [96] modeled the coordination of patch distributions in software-defined vehicular networks using AnyLogic[16], a general-purpose simulation framework that provides a flexible programming interface that supports three simulation models. Agent-Based Simulation Modeling, Discrete Event Simulation Modeling, and System Dynamics Simulation Modeling. While some works employed general-purpose simulators such as AnyLogic aiming at flexible support for multiple types of conceptual models, others have chosen simulation tools with more specific purposes.

[16]https://www.anylogic.com/

Fakhrolmobasheri et al. [43] modeled a hypervisor rejuvenation scenario with the Möbius [31] simulator, which focuses on modeling and analyzing stochastic processes such as Markov chains and Petri Nets. Möbius adopts a discrete-event simulation model with flexible monitoring capabilities that enable measuring several metrics (e.g., reliability, availability, performance, and security) at varied intervals (e.g., specific time points, over periods of time, or when the system reaches steady state). Torquato et al. [156] took a similar approach, modeling a hypervisor rejuvenation scenario with TimeNET [52], which focuses on providing flexible support to Petri Net simulations (e.g., evaluation of models with nonexponentially distributed transition firing delays and support to Colored Stochastic Petri Nets).

As several maintenance-related activities involve intense network communication (e.g., patch distribution and application migrations), some research works employed network simulators during their evaluations. Banikhalaf et al. [14] considered a route repair scenario where the IoT infrastructure is composed of moving vehicles. The authors used NS-2 [2] along with SUMO[17] to validate the proposed strategy. While NS-2 simulates various aspects of the wireless network infrastructure, such as routing and traffic control, SUMO enables modeling vehicle mobility based on real and synthetic traces.

Nain et al. [109] [108] addressed a different scenario, employing the Cooja simulator [114] to perform network route updates in low-power and lossy IoT networks, where network devices and transmission modes are resource-constrained. While NS-2 addresses network simulation from a more general perspective (e.g., supporting both wired and wireless networks), Cooja focuses on simulating Wireless Sensor Networks (WSNs), providing built-in support for emulating real hardware platforms. In addition to supporting several network technologies, such as the Routing Protocol for Low Power and Lossy Networks, Cooja provides fine-grained control over network stacks and real-time monitoring of various metrics, such as network throughput and power consumption.

Although general-purpose and network simulators provide suitable features for some maintenance scenarios, they lack support to model specific maintenance problems, highlighting the need for specialized simulators. Souza et al. [149] modeled an edge server maintenance scenario using EdgeSimPy[18], a simulation toolkit designed to facilitate the prototyping of resource management policies on Edge Computing infrastructures through system models that accurately represent various elements, including the lifecycle of containerized applications. Although EdgeSimPy can model maintenance work involving multiple components (e.g., edge servers, applications, and network devices), other simulation toolkits concentrate on specific maintenance scenarios.

An example of a specialized maintenance simulator is FUOTASim [1], used by Anastasiou et al. [5] to model firmware updates in IoT environments. In addition to simu-

[17] https://sourceforge.net/projects/sumo/
[18] https://edgesimpy.github.io/

lating the behavior of wireless network protocols such as LoRaWAN[19], FUOTASim eases the monitoring of several metrics, such as patch distribution time and power consumption.

Experimentation

Empirical experimentation consists of validating research prototypes in testbeds with characteristics similar to production environments, which enables the identification of real-world challenges and requirements. In addition to approximating research outcomes to practical applications, many research works present update/repair frameworks that rely on optimizations upon hardware-specific behaviors that are difficult to simulate accurately, which makes empirical experimentation even more attractive.

One of the main challenges in this context is related to the cost of building testbeds at scale, which involves the acquisition of several components (e.g., compute and network devices, cooling and power supply systems, among others) and continuous environment supervision. As a result, we observe that most existing academic efforts performed empirical tests on small-sized testbeds comprised of a few servers or personal computers equipped with networking and virtualization technologies [164] [12] [61] [54] [83] [60] [165] [113] [137] [25] [110] [47] [44] [106] [157] [153].

In contrast, some contributions included practical tests on mid-to-large infrastructures. For example, Leiba et al. [89] and Tapas et al. [152] used IoT testbeds composed of 48 and 50 single-board computers, respectively, while Russinovich et al. [127] evaluated their proposed server update approach by performing driver updates in Microsoft Azure's data centers with millions of servers.

Summary and Takeaways

Existing maintenance research efforts employ various validation approaches according to the addressed scenario. In general, few maintenance strategies are validated through formal modeling methods. This is mainly because maintenance scenarios often include nondeterministic phenomena caused by cascading events, which can be challenging to model and evaluate at scale. In contrast, several works use simulation tools to represent conceptual models with significant levels of detail at low implementation costs. We also observe that most simulated evaluations use custom simulators, highlighting the absence of a *de facto* standard maintenance simulation toolkit. In addition, many existing research efforts perform empirical experiments, as proposed solutions leverage hardware-specific behaviors that are difficult to model artificially.

[19] https://lora-alliance.org/about-lorawan/

3.5 Closing Remarks

Despite the promising use cases for the unified IoT-Edge-Cloud model, IT personnel face the challenge of implementing efficient maintenance strategies to preserve the environment's performance and security, which resorts to understanding the numerous requirements, demands, and challenges of such a complex ecosystem. While the abundant resources from the academic literature could help in that regard, extracting actionable insights from their findings and drawing parallels between multiple works is challenging due to the massive number of existing papers. In this context, we observed a need for review papers providing an overview of existing solutions and indications of open challenges to lower the barrier to entry for new researchers.

This chapter presented a comprehensive survey of maintenance research aimed at cloud, edge, and IoT environments. As the scope of our study intersected multiple research subjects, we proposed a taxonomy that provides a logical organization of existing works based on various characteristics, such as maintenance approaches, techniques used, and metrics of interest.

Although our review covers three paradigms (edge, cloud, and IoT), this thesis focuses primarily on optimizing maintenance on edge infrastructures. That said, throughout our literature analysis, we have identified some research topics concerning edge maintenance that require further investigation. The identified gaps constitute the primary motivation for the research work detailed in the subsequent chapters, where we coordinate ideas from the other paradigms explored in this review (i.e., cloud and IoT) with novel strategies to optimize maintenance work within Edge Computing infrastructures.

The next chapters address the following identified challenges:

- **Modeling and simulation of maintenance at the edge:** We have observed that simulation has been the primary validation approach for maintenance strategies as it allows for the cost-effective evaluation of research prototypes in large-scale environments. Despite the widespread use of simulation, most research efforts resort to general-purpose simulation tools, which fall short in modeling maintenance-specific operations such as patching and component versioning. **This research gap is addressed in Chapter 4**, where we introduce EdgeSimPy, an Edge Computing simulation tool written in Python that models various features of the edge environment (e.g., infrastructure power consumption and application composition), while also supporting use cases such as edge server maintenance, applications, and network devices.
- **Location-aware application relocations during maintenance:** Existing maintenance strategies that consider scenarios where applications must be relocated during maintenance are designed for cloud data centers, where application relocation is performed

by observing if the application demand could be met by a candidate host, regardless of that new host location within the data center, given the robust network capacities available. This situation is inversed on Edge Computing environments, as edge servers are interconnected by less reliable networks, and applications have strict latency requirements. **This research gap is addressed in Chapter 5**, where we introduce Lamp and Laxus, two maintenance strategies that mitigate latency issues during maintenance by incorporating user-location awareness into application relocations.

- **Optimized container provisioning during maintenance:** We have observed that existing maintenance strategies relocate applications during maintenance using VM-based techniques such as cold migration. However, modern computing platforms are increasingly adopting containers as the preferred virtualization technology over VMs. Although the architectural differences between containers and VMs may seem subtle, they significantly influence how applications are provisioned, as discussed in Section 2.1. One of the key differences is that container images typically employ layered file systems, which enable software instructions to be shared amongst container images, making room for significant reductions in the application provisioning time when servers already possess parts of container images. As initiatives in the literature rely on the VM model, they cannot leverage the shared content of container images to speed up relocations performed during maintenance to reduce the overall maintenance time. **This research gap is addressed in Chapter 6**, where we introduce Hermes, a maintenance strategy that reduces maintenance time during edge server updates through efficient relocations of containerized applications.

It is worth noting that some of our contributions presented in subsequent chapters (i.e., Lamp, Laxus, and EdgeSimPy) have been cataloged in our literature review due to variations in their individual manuscript publication timelines.

4. SIMULATION TOOLKIT FOR EDGE INFRASTRUCTURES

In the previous chapter (Section 3.5), the first identified challenge was enabling the modeling and simulation of maintenance operations at the edge. This chapter addresses such a challenge through EdgeSimPy, a simulation framework written in Python that incorporates various functional abstractions for the entities that compose edge infrastructures and supports several use cases, including maintenance operations targeting physical and logical components.

First, this chapter presents a motivation for EdgeSimPy (Section 4.1) and a comparison against existing simulators (Section 4.2). Then, it describes EdgeSimPy's architecture (Section 4.3) and checks the correctness of EdgeSimPy's core features implementation (Section 4.4). Finally, it presents two use cases for EdgeSimPy (Section 4.4).

4.1 Motivation

While proximity to data sources grants Edge Computing a natural advantage over the cloud, it also introduces significant technical constraints. As deploying large-scale edge data centers in urban centers is typically not feasible, edge sites often comprise groups of devices with reduced computing power distributed across small physical spaces with limited power and cooling supply [160]. As such, ensuring efficient use of resources, which is more critical than ever, becomes challenging. Therefore, efficient resource management policies must be developed and tested to ensure that they achieve the expected goals.

In addition to the cost and time required to prototype resource management policies, the distributed nature of edge infrastructures adds additional barriers to empirical experimentation of new resource management policies. Examples of such barriers are network instability and power outages. Such challenges favored the rise of several edge simulators that promise to close the gap between conceptual research and prototyping (see Section 4.2). Despite such initiatives, researchers and practitioners still face barriers when designing edge resource management policies, as existing simulators do not provide fine-grained modeling of edge infrastructures, which comprises a variety of processes such as application provisioning, network flow scheduling, server maintenance, and others.

To overcome this challenge, this chapter introduces EdgeSimPy, a Python-based simulator for modeling and evaluating resource management policies on Edge Computing environments. EdgeSimPy ships with a modular architecture with several functional abstractions for edge servers, network devices, and applications. EdgeSimPy embodies a novel conceptual model that replicates the application provisioning method of widely used

platforms such as Docker[1], allowing seamless integration with repositories such as DockerHub[2]. In addition to demonstrating the effectiveness of EdgeSimPy through an in-depth verification that checks the correctness of the simulator implementation, we describe two case studies available in the literature [146, 149] showing EdgeSimPy in action in different large-scale scenarios.

The contributions of this chapter are the following:

- We provide a high-level interface that leverages the features of well-known modeling solutions such as Mesa [77] and NetworkX [56] to facilitate the development of resource management policies and simulator extensions (Section 4.3.1).
- We propose a conceptual model that accurately represents the lifecycle of edge applications by replicating the behavior of widely used platforms like Docker (Section 4.3.3).
- We implement multiple system models to simulate several features of edge environments, such as infrastructure power consumption, application composition, service workload variations, and user mobility (Sections 4.3.2–4.3.4).
- We perform a verification that checks the correctness of EdgeSimPy's core features implementation by comparing the simulation results to the expected outputs (Section 4.4).
- We demonstrate EdgeSimPy's utility through two case studies based on peer-reviewed papers [146] [149] that have employed the simulator in different scenarios (Section 4.5).

4.2 Related Simulators

During the past decade, simulation tools such as CloudSim [24] and GreenCloud [79] have been widely adopted to accelerate the development and validation of resource management strategies for cloud data centers. Similarly, the dawn of Edge Computing has motivated the development of several edge simulators. This section starts with an overview of simulation tools for Edge Computing (Section 4.2.1). Then, it highlights the differences and contributions of EdgeSimPy over existing simulators (Section 4.2.2).

4.2.1 Edge Computing Simulators

Sonmez et al. [145] highlight the limitations of cloud and network simulators when modeling the characteristics of edge environments. While cloud simulators such as CloudSim

[1] https://www.docker.com/
[2] https://hub.docker.com/

lack user mobility and wireless support, network simulators are not focused on modeling edge servers and users. Accordingly, the authors present EdgeCloudSim, a simulator that implements user mobility, edge device power modeling, and network management, facilitating the prototyping of placement strategies on Edge Computing environments.

Qayyum et al. [119] argue that edge simulators abstract network characteristics, restricting the options for designing new resource management strategies. The authors introduce a new simulator, FogNetSim++, which models the power consumption of edge devices, supports various communication protocols (e.g., MQTT and CoAP), and simulates the handover of mobile users in the network. Also, the authors present a use case that shows FogNetSim++'s effectiveness in modeling edge placement and scheduling policies.

Puliafito et al. [118] discuss the critical role of application migration in preserving low latency for users despite their mobility and how edge simulators lack the features needed to evaluate migration decisions. Based on these observations, the authors propose MobFogSim, a simulator that models the application migration process based on user mobility (including attributes such as speed and moving direction) and the coverage area of access points spread in the environment.

Amarasinghe et al. [4] present a novel simulator called ECSNeT++, which is focused on the deployment and processing of stream-based applications on Edge Computing environments. Unlike other simulators, ECSNeT++ enables fine-grained resource management for streaming applications, where allocation policies can determine how tasks are processed at the core level on edge devices (where each CPU core has a processing queue), aiming to reduce the network delay and power consumption of edge devices.

Lera et al. [90] introduce YAFS, an edge simulator focused on evaluating allocation decisions for composite applications (i.e., applications composed of multiple components) on edge infrastructures. In YAFS, routing policies coordinate the communication of the application across the network, allowing modules of a given application to be allocated on different edge devices. The authors discuss several use cases, including scenarios with dynamic scheduling of application components, infrastructure failures, and user mobility.

Jha et al. [71] discuss the complexity of the Cloud-Edge-IoT ecosystem, which involves allocating resources across heterogeneous devices using multiple network protocols (e.g., LoRa and Zigbee) and messaging protocols (e.g., CoAP and MQTT). After highlighting the lack of edge simulators considering the protocols of the Cloud-Edge-IoT ecosystem, the authors present the IoTSim-Edge simulator. The simulator allows the prototyping of edge resource management strategies based on the energy consumption of edge devices, application composition, user mobility, and communication protocols.

Alwasel et al. [3] make a case for Osmotic Computing, a paradigm that involves workload migration between cloud data centers and edge devices based on performance and security events. The authors present a novel simulator, IoTSim-Osmosis, focused on Osmotic Computing scenarios, which implements a variety of models for data transmission,

energy consumption, and application performance. Finally, the authors present a use case where IoTSim-Osmosis is used to model strategies that optimize performance, energy consumption, and budget in Cloud-Edge scenarios.

Mahmud et al. [95] highlight the lack of support for real datasets on edge simulators, which limits the evaluation of resource management policies on edge environments comprising composite applications. Then, the authors introduce iFogSim2, a simulator with native support to real datasets that incorporates modeling and simulation of service migration in multi-tier infrastructures (e.g., Cloud-Edge), user mobility, service orchestration, and edge devices clustering. They present use cases that demonstrate the effectiveness of iFogSim2 in simulating several resource allocation scenarios at the edge.

4.2.2 Discussion

Despite the significant research interest in simulation modeling focused on Edge Computing environments, existing edge simulators lack native support for managing the lifecycle of containerized applications. At first, extending existing simulators may seem trivial, as most of them are built with programming stacks that facilitate the inclusion of new features. However, in this case, several changes may be needed to allow evaluation of container management strategies at the edge, such as including several new entities (e.g., container registries, images, layers, etc.) and modeling the container provisioning process from scratch, as it differs significantly from the VM model, as discussed in Section 2.1.

EdgeSimPy models aspects of edge computing that are also supported by existing simulators, such as user mobility, energy consumption (for computing and network devices), and application composition. Complementarily, it provides functional abstractions that are not covered by existing edge simulators (e.g., container registries, container images, and container layers), enabling the simulation of the lifecycle of containerized applications (e.g., placement, scheduling, migration, update, and removal operations). Table 4.1 summarizes the main differences between EdgeSimPy and related simulators regarding built-in features.

Table 4.1 – Built-in features supported by existing simulators and EdgeSimPy.

Simulator	**Power Modeling**		**Network**	**User**	**Application**	**Container**
	Servers	**Network**	**Routing**	**Mobility**	**Composition**	**Lifecycle**
EdgeCloudSim [145]	✘	✘	✘	✔	✘	✘
FogNetSim++ [119]	✔	✔	✔	✔	✘	✘
MobFogSim [118]	✔	✘	✘	✔	✔	✘
ECSNeT++ [4]	✔	✔	✔	✔	✔	✘
YAFS [90]	✔	✘	✔	✔	✔	✘
IoTSim-Edge [71]	✔	✘	✔	✔	✔	✘
IoTSim-Osmosis [3]	✔	✘	✔	✘	✔	✘
iFogSim2 [95]	✔	✘	✔	✔	✔	✘
EdgeSimPy (this work)	✔	✔	✔	✔	✔	✔

In addition to its built-in functionalities, EdgeSimPy aims to contribute to the community by providing more comprehensive support for specific use cases, as shown in Table 4.2. Although specific simulators such as MobFogSim and iFogSim2 support the modeling of service migration strategies, they focus predominantly on VM-based migration models. For instance, although MobFogSim provides a ContainerVM class, such an entity follows the VM migration model, where containers are relocated from a source to a destination server, overlooking the shareable structure of container images and the role of container registries.

Table 4.2 – Use cases supported by existing simulators and EdgeSimPy.

Simulator	Service Migration	Network Flow Scheduling	Container Registry Management	Maintenance Operations
EdgeCloudSim [145]	✘	✘	✘	✘
FogNetSim++ [119]	✘	✔	✘	✘
MobFogSim [118]	✔	✘	✘	✘
ECSNeT++ [4]	✘	✔	✘	✘
YAFS [90]	✘	✘	✘	✘
IoTSim-Edge [71]	✘	✔	✘	✘
IoTSim-Osmosis [3]	✘	✔	✘	✘
iFogSim2 [95]	✔	✔	✘	✘
EdgeSimPy (this work)	✔	✔	✔	✔

Unlike MobFogSim and iFogSim2, EdgeSimPy's has abstractions for container-related entities (i.e., container registries, container images, and container layers), which allows it to support the design of relocation strategies for containerized applications, widening EdgeSimPy potential use cases. For instance, as EdgeSimPy represents container registries as domain-specific services, it supports the design of provisioning policies that manage such entities aiming at optimizations in the lifecycle of containerized instances. In addition, EdgeSimPy facilitates modeling maintenance operations for physical resources (e.g., servers, network devices) and applications, which is not supported by existing simulators. An in-depth analysis of EdgeSimPy's suitability in two case studies and a comparison with existing simulators in these scenarios is provided in Section 4.5.

4.3 Simulator Architecture

EdgeSimPy's primary design goal is to support researchers interested in evaluating resource management strategies for the edge. For example, EdgeSimPy models different resource allocation decisions (e.g., placement, migration, scheduling, and maintenance) while considering the infrastructure's heterogeneity (power consumption, resource capacity), users (mobility, access profile), and applications (composition, performance requirements). Figure 4.1 shows a sample EdgeSimPy simulation scenario where entities interact with each other and different resource management decisions affect the environment.

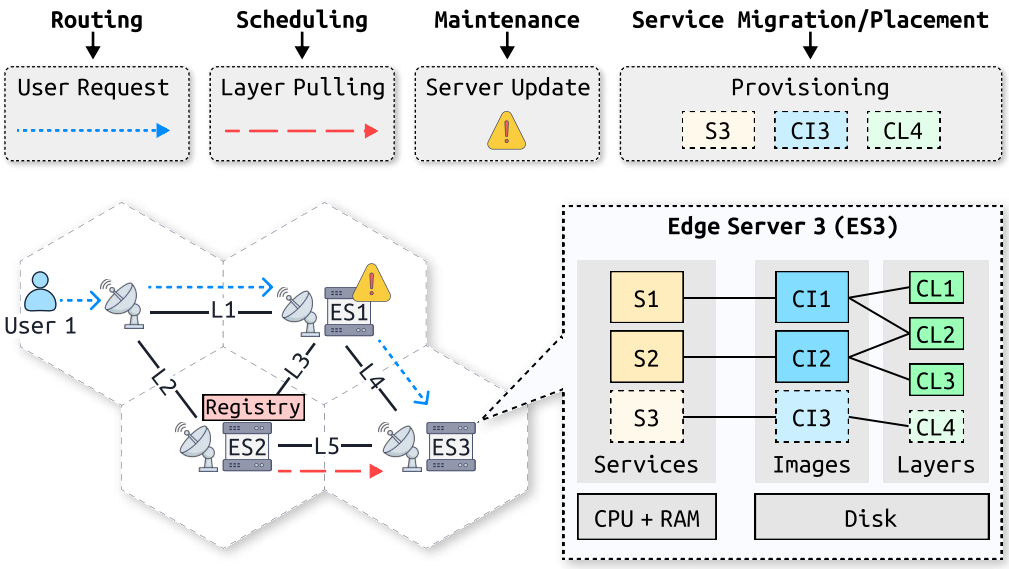


Figure 4.1 – Sample EdgeSimPy simulation scenario. While “L" and “ES" denote the network links and edge servers, “S", “CI", and “CL" represent the services, container images, and container layers.

Before starting the simulation, EdgeSimPy expects an input file defining the simulated scenario. EdgeSimPy input files are written in JavaScript Object Notation (JSON) format, which has a human-friendly file structure [111] and is widely adopted in various software domains [115], which facilitates EdgeSimPy integration with other tools. EdgeSimPy input files organize the metadata of each simulated entity into a well-defined structure made up of two distinct information groups: attributes and relationships. Attributes refer to the internal characteristics of entities, such as edge server capacity, network link bandwidth, and application delay, among others. Relationships represent the associations between entities (e.g., a service’s host or a user’s applications). By adhering to this predefined structure, EdgeSimPy can automatically identify entity input metadata and construct the simulated scenario, even in cases where custom attributes and relationships have been specified. Figure 4.2 shows a sample EdgeSimPy input file with the metadata of an application entity.

Simulated entities can carry geospatial metadata, which facilitates the integration of datasets containing real or synthetic coordinates into EdgeSimPy and allows the modeling of events such as user mobility. By default, EdgeSimPy uses the map model proposed by Aral et al. [9], which divides the environment into hexagonal cells.

Once the simulation starts, EdgeSimPy triggers a monitoring mechanism that stores snapshots of the entity’s state at the end of each time step. Simulation logs are stored in MessagePack[3], a binary serialization format designed to be a faster and smaller alternative to JSON [159]. Instead of writing data to disk each time step, EdgeSimPy stores the simulation output at configurable intervals, reducing the I/O pressure during the simulation.

[3]https://msgpack.org/index.html

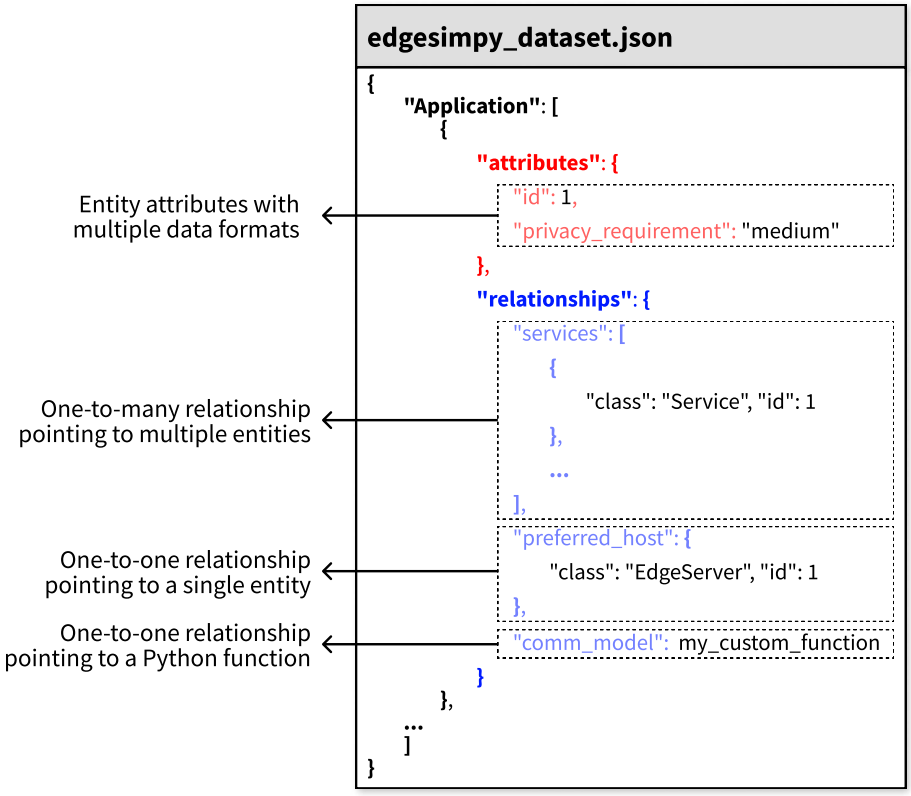


Figure 4.2 – Sample EdgeSimPy input file.

EdgeSimPy's flexibility stems from a modular architecture (depicted as a block diagram in Figure 4.3), which divides the functional abstractions into four layers (Core, Physical, Logical, and Management). Each abstraction is self-contained to streamline the integration of new features and algorithms.

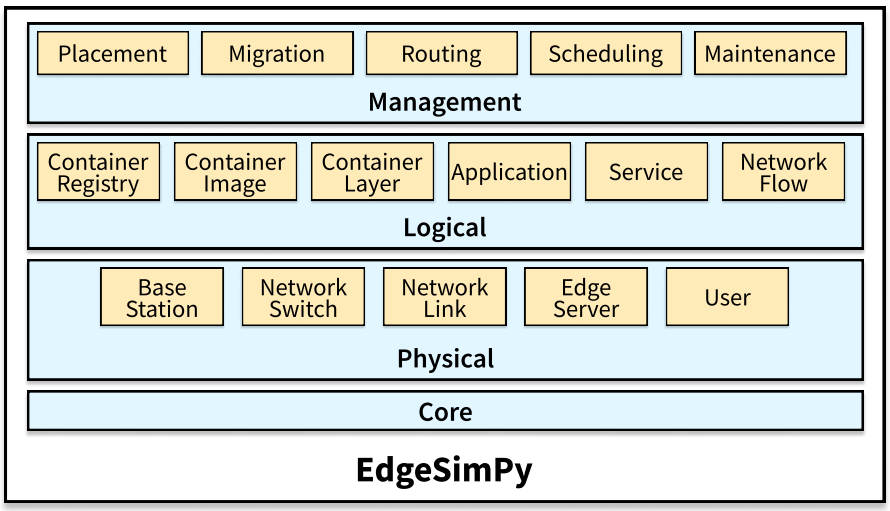


Figure 4.3 – EdgeSimPy architecture.

4.3.1 Core Layer

Edge computing environments can be seen as complex systems with several entities that interact with each other in a non-linear way. This ecosystem may assemble emergent phenomena [136], which are the product of interactions between entities. An example of an emerging phenomenon at the edge is infrastructure saturation generated by a series of poor allocation decisions. While detecting these events is key to avoiding erroneous allocation decisions, understanding its causes through the attributes of individual entities is challenging as it arises from the interaction between multiple entities.

One of the most popular techniques for studying emergent phenomena is Agent-Based Modeling (ABM) [94] [20], which represents the world through a bottom-up approach, where independent entities (called agents) interact with each other and the environment over time according to communication and decision-making rules. Most ABM-based simulation systems adopt the Fixed-Increment Time Advance (FITA) strategy [85], which advances the simulation clock in fixed increments of the time delta (Δ). If multiple events are scheduled for the same time step t, they are considered to have occurred concurrently at the end of t.

FITA-based simulators have two main properties: Δ's granularity and the agents' activation regime [85]. Δ's granularity affects the accuracy of the simulation output and the simulation time—lower Δ yields higher simulation accuracy, as it reduces the number of events computed simultaneously at the cost of a longer runtime. The activation regime defines the order in which the simulator computes simultaneous events at each time step, which can affect the simulation output, especially if simultaneous events influence each other.

EdgeSimPy employs the ABM approach, representing entities of edge scenarios as interactive agents. The features that comprise the simulation core (i.e., agent modeling, activation regime, and time advance) are managed by Mesa [77], a well-known ABM framework that ships several built-in modules that encompass the building, analysis, and visualization of ABM simulations. EdgeSimPy's simulation workflow is depicted in Figure 4.4.

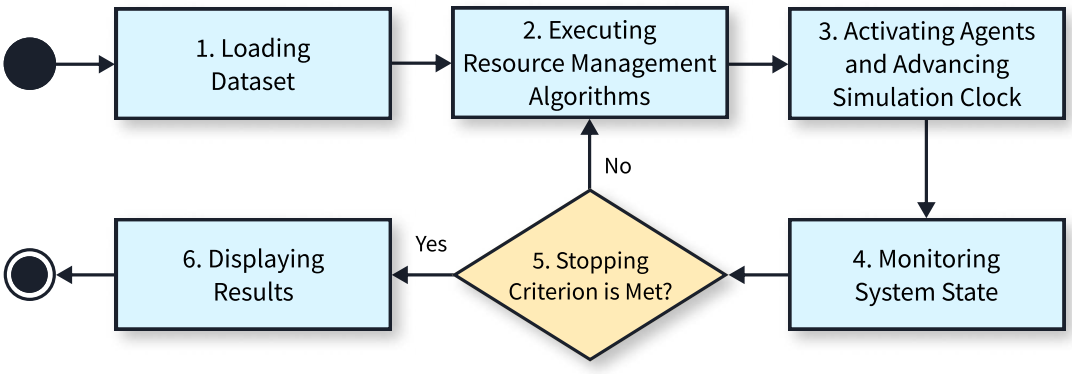


Figure 4.4 – EdgeSimPy's simulation workflow.

When EdgeSimPy is started, it loads input data from JSON files or Python dictionaries, spawning simulated entities accordingly (Figure 4.4, step 1). After loading the scenario, EdgeSimPy starts an iterative process until a user-defined stopping criterion is met. At each time step, EdgeSimPy executes user-defined resource management policies, calls the activation regime to update agent state, increases the simulation clock, and collects logs of the state of the system, respectively (Figure 4.4, steps 2–4).

Once the stopping criterion is met, EdgeSimPy stops the simulation and displays the collected metrics and logs (Figure 4.4, step 6). Thanks to EdgeSimPy's decoupled architecture, it is possible to define custom stopping criteria, resource management algorithms, and personalized routines for collecting and exhibiting simulation metrics.

4.3.2 Physical Layer

The Physical layer contains functional abstractions for users and resources that comprise the edge infrastructure. Regardless of their distinct functions, all components in the Physical layer have a *coordinates* attribute, which carries the component's geospatial information. Physical entities that provide networking capabilities leverage the features of NetworkX [56], a well-known graph library for manipulating complex networks that ships several built-in methods (e.g., shortest path and community finding).

The remainder of this section discusses the components in the Physical layer.

Base Stations

Base Stations act as gateways in the edge network, providing wireless connectivity for seamless communication between users and edge servers. EdgeSimPy assumes that the base stations cover the entire map area so that users always have connectivity regardless of location. As such, the set of coordinates of the base stations comprehends the whole available area for user transit, so users cannot be in a position that represents a different coordinate from all base stations, as they would not have network connectivity. Base stations on EdgeSimPy embody multiple customizable attributes, such as energy consumption and wireless latency, allowing various scenarios to be modeled.

In addition to providing wireless connectivity, EdgeSimPy automatically handles user handoff between base stations based on user mobility patterns. Accordingly, EdgeSimPy allows for a realistic simulation of users moving through the edge network and the associated changes in connectivity as they transition from one base station to another. Additionally, base stations can be equipped with network switches for wired connectivity and network flow management and edge servers for hosting services, ensuring an adaptable model for edge environments.

EdgeSimPy can also be customized to support map models where a single base station covers multiple coordinates, and attributes such as wireless latency are affected by the distance between the user's and base station's positions. This allows the modeling and analysis of various connectivity scenarios and their impact on edge application performance.

Network Switches

Network switches provide wired connectivity between infrastructure components (e.g., base stations and edge servers) and manage data flow in the network. These components ship multiple configurable parameters, such as chassis types and varying numbers of ports with specific delay and bandwidth properties. In EdgeSimPy, network switches are modeled as nodes in the graph representing the network topology.

Although network switches are used as network nodes by default, EdgeSimPy allows other entities to be modeled for this purpose, enabling a variety of networking scenarios to be simulated. For example, cars can act as topology nodes in vehicular networks, serving the environment as intermediary data exchange and communication entities.

EdgeSimPy assumes that user requests are protected by QoS policies so that one user's request does not negatively affect others. On the other hand, more demanding data transfers between edge servers (e.g., service migrations) are modeled as network flows, sharing the bandwidth of network links. The duration of a network flow depends on the bandwidth of the links it spans over and the resource allocation policy of the network switches. Whenever a network flow starts or ends, EdgeSimPy runs a flow scheduling algorithm to update the occupation of the involved links, redistributing the available bandwidth, if applicable.

Since a network flow can use a path with links containing different bandwidth demands, EdgeSimPy normalizes the bandwidth available to a network flow to the lowest available bandwidth between the links in its path. The Max-Min Fairness algorithm [18] is used as the default flow scheduling algorithm, dividing the network bandwidth proportionally to the flow demand. As the flow scheduling logic is fully encapsulated, it is possible to define custom flow scheduling algorithms without burden.

During simulation, the power consumption of the network varies according to the resource usage and the technical features of the network switches. By default, EdgeSimPy includes the network power models proposed by Conterato et al. [29] and Riviriego et al. [123]. However, the behavior of power models is fully encapsulated, allowing new models to be implemented without requiring changes to the simulator's core.

Edge Servers

Edge servers are used to host services. EdgeSimPy assumes that the edge infrastructure is virtualized so that edge servers can host multiple services simultaneously. In

addition to capacity parameters such as CPU, RAM, and disk storage, edge servers can also have performance parameters like Million Instructions Per Second (MIPS), which allows for an alternative representation of server performance and resource allocation. It is possible to define the distribution of MIPS among applications on a server based on custom criteria, allowing the modeling of advanced phenomena such as resource contention among co-hosted applications.

Technical specifications such as hardware resources and resource usage affect the power consumption of edge servers during simulation. EdgeSimPy incorporates three built-in generic power consumption models (LinearPowerModel, QuadraticPowerModel, CubicPowerModel) [16], which assume that edge server power consumption grows according to their CPU usage following linear, quadratic, and cubic functions, respectively. EdgeSimPy's energy modeling enables the implementation of advanced features, such as temporarily turning off edge servers to save energy. As the properties of power models are fully encapsulated, EdgeSimPy supports custom power models for edge servers.

As edge servers have static coordinates, they are immobile by default. Nevertheless, EdgeSimPy can be extended to assign mobility models to edge servers, allowing the representation of mobile devices with computing capabilities, such as drones or Single-Board Computers (SBCs) connected to automobiles.

Users

As part of the Physical layer, users have a *coordinates* attribute that defines their location on the map. Users can remain in the same position during the entire simulation or move according to the mobility models. By default, EdgeSimPy incorporates two mobility models, Random and Pathway [70], which can be easily replaced by other synthetic models or real mobility traces.

Users and applications are linked by a many-to-many relationship, which allows a user to access multiple applications or even an application to be accessed jointly by multiple users. Following this design principle, each user has properties that define their delay and availability requirements for each application they access. As each entity is self-contained, adding new user requirements such as security and budget is possible, which opens room for prototyping custom allocation strategies.

Each user has his access pattern, specifying when he will call his applications and how long each access will last. By default, EdgeSimPy incorporates two user access pattern templates, Random and Circular. While the former arbitrarily defines when and for how long the user will access their applications, the latter establishes a pattern that repeats indefinitely. Figure 4.5 illustrates four users using the Random and Circular access patterns.

User access patterns are defined through independent classes, allowing the definition of custom patterns without burden. This functionality allows EdgeSimPy to model

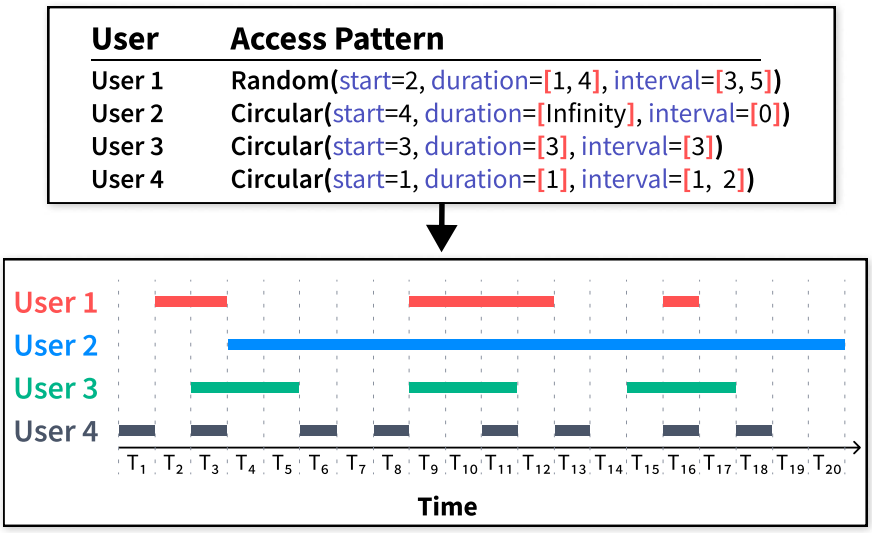


Figure 4.5 – EdgeSimPy's built-in user access patterns.

different workloads, from streaming to batch processing applications and serverless functions. As a user's access can be intermittent, allocation policies may choose to deprovision applications during idle periods, which can be beneficial in terms of resource saving or harmful if there are long application provisioning times after the user's request (as in the case of serverless functions facing cold starts [141]).

4.3.3 Logical Layer

The Logical layer comprises functional abstractions for applications running on the edge infrastructure. Despite supporting VMs, EdgeSimPy adopts containerization as the default virtualization model (see details in Chapter 2). Consequently, abstractions for container registries, container images, and container layers are provided. The rest of this section describes the components of the Logical layer.

Applications

Applications are modeled as abstract entities that represent data flows between services. In this way, the application services are allocated within the infrastructure, rather than the applications themselves. As self-contained entities, applications can receive custom attributes to support the modeling of specific scenarios (e.g., QoS priority and budget).

EdgeSimPy calculates the latency of an application based on the time it takes to visit the servers that host all its services. The default implementation of the application in EdgeSimPy assumes that the data flow starts at the application users and passes through all its services sequentially. In addition to the built-in application communication behavior,

EdgeSimPy supports custom communication patterns to model multiple software architectures, from monoliths to microservices [45] and stream applications [33]. It is also possible to specify custom communication policies among the services that compose an application based on various criteria, such as inter-service latency and edge server characteristics.

Services

The services in EdgeSimPy are modeled as container instances within the infrastructure. While a service's disk demand corresponds to the size of the layers that comprise its container image, its CPU and memory demand describe the computational resources required by the service instance, and therefore are unrelated to the service's image. Each service also has a state attribute that defines whether it is stateless or stateful.

Services leverage a layered file system model to separate service components and ensure that changes in one containerized service do not affect other co-hosted services using the same base image. This containerization approach provides an efficient and isolated environment for hosting services in the edge infrastructure. The layered file system model used by EdgeSimPy services is shown in Figure 4.6.

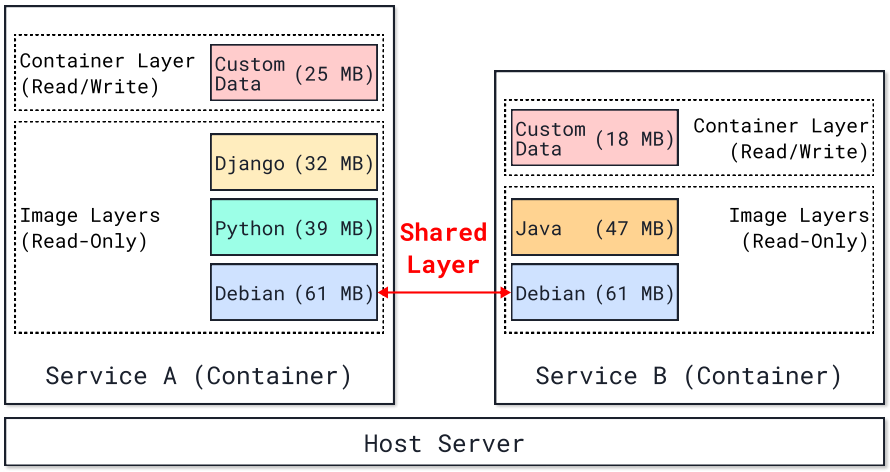


Figure 4.6 – Layered file system used by service in EdgeSimPy.

The layered file system used by services divides the service image into two parts: the container and image layers. In this setting, each service has its container layer with its runtime files and user session data. As the container layer is not shared with co-hosted services, it has read-write permissions. Conversely, image layers hold read-only permissions as they provide static files, libraries, and dependencies, and are shared among co-hosted services. This separation enables a streamlined relocation process and ensures that services can be efficiently transferred across edge servers without affecting other instances or interfering with shared resources.

The state attribute plays a crucial role in the way the service is relocated. Stateless services maintain no user session data or runtime state, allowing them to be relocated

without downtime, as their container layer can be easily discarded on the target host and recreated on the destination host. In contrast, stateful services require the transfer of both image layers and the container layer containing user session data, resulting in a brief downtime while the service's state is transferred to the destination host.

Container Registries

Container registries are the main component when allocating a service in the edge infrastructure, as the service's container image is pulled from it to the destination host. A container registry is a containerized service built on top of a registry image that embeds image distribution and storage functionality. Thus, a container registry also has its own CPU and memory requirements to perform image distribution processing.

EdgeSimPy supports the definition of custom policies for selecting from which container registries container images are pulled to the edge servers. This feature fosters the design of resource management strategies that optimize service provisioning considering factors such as network usage and the location of container registries. Additionally, as container images follow a layered file system model, EdgeSimPy allows resource management strategies that leverage multiple container registries to download the layers of a given image.

EdgeSimPy also allows one to define how many layers of a particular image are downloaded simultaneously to a given host. This level of control enables the design of resource management strategies that optimize service provisioning by balancing multiple metrics of interest, such as network usage, provisioning time, and resource availability.

As container registries are modeled as domain-specific services that distribute container images across the edge infrastructure, EdgeSimPy enables dynamic provisioning of container registries as it does with regular services. This feature allows users to adapt the deployment of the container registry to changing network conditions, resource requirements, or application demands, ensuring efficient image distribution and optimized resource utilization in the edge environment.

Container Images

Container images embed the basic functionality for services. Each image has a *tag* attribute representing its version, allowing the modeling of scenarios where new image versions are released during the simulation. Like applications, container images are modeled as abstract entities, so they have no resource requirements by themselves. Instead, the disk demand for a given container image results from the aggregated size of its layers.

EdgeSimPy models container images according to the Open Container Initiative (OCI) [4] format, a well-known standard for container images. By adhering to the OCI format,

[4] https://opencontainers.org/

EdgeSimPy can incorporate metadata from real-world container images from repositories like DockerHub, providing a more accurate representation of services in the edge infrastructure. In addition to providing compatibility with real image traces, EdgeSimPy's container image format allows the definition of service provisioning constraints to model edge-specific scenarios. For instance, it is possible to define datasets with container image specifications that narrow the service provisioning options based on the architecture and hardware capabilities of available hosts.

Container Layers

The container layers represent the instructions aggregated into container images. Each container layer represents a set of files added or modified from the previous layer. In EdgeSimPy, container layers can be identified by a digest attribute, enabling hosts to check the integrity of downloaded container layers.

Each container layer has attributes representing its software instruction and disk size. As container images in EdgeSimPy adhere to a layered filesystem model, co-hosted services can share read-only image data, resulting in considerable disk savings. This model makes room for the design of layer-aware resource management strategies that could reduce application provisioning time by selectively choosing hosts that already possess the necessary service layers.

4.3.4 Management Layer

In addition to modeling the behavior of several entities of Edge Computing scenarios, EdgeSimPy provides fine-grained control over network and edge server resources. Consequently, it facilitates prototyping various resource management policies, such as:

- **Service Placement:** Defining the placement of application services in the infrastructure represents a vital decision to ensure the efficient use of resources. By supporting the definition of custom user access patterns, EdgeSimPy enables the modeling of online and offline placement strategies [176, 154]. In offline placement scenarios, the allocation policy has *a priori* knowledge about all the applications it needs to provision. In online placement scenarios, application provisioning requests occur at runtime, and provisioning policies must allocate application services on demand.
- **Service Migration:** Given the strict latency requirements of applications running on the edge infrastructure, user mobility may require relocation of services at runtime. To support this functionality, EdgeSimPy supports the modeling of allocation policies with fine-grained control over the migration process. In this way, migration policies can

define which edge servers should host the services, from which container registries the layers should be pulled, and which network paths should be used in this process.

- **Maintenance:** Updates for applications and physical devices are often released to add new features, fix bugs, or mitigate security vulnerabilities. As components of the Physical and Logical layers have versioning attributes, EdgeSimPy supports the modeling of maintenance scenarios, incorporating decisions such as the order in which components are updated.
- **Network Flow Scheduling:** In large-scale edge scenarios, events such as user mobility can trigger the provisioning of multiple applications simultaneously. However, the lack of network management can allow large flows to indiscriminately saturate the network, causing the starvation of smaller flows and the reduction in the overall performance of the network [139] [155]. Accordingly, EdgeSimPy allows the definition of scheduling strategies for network flows that can control the priority of each flow based on objectives modeled through built-in or custom attributes.

The entities comprising EdgeSimPy's architecture shape a robust platform for modeling various Edge Computing scenarios, where different resource allocation policies can be prototyped and validated without burden. In addition, each component is designed to work independently, which facilitates the inclusion of new attributes and entities to the simulator, extending the breadth of potential EdgeSimPy use cases. The next section describes the verification process used to demonstrate the correctness of EdgeSimPy implementation.

4.4 Verification

Simulation is known to allow the analysis of real-world phenomena with lower complexity and cost than empirical experimentation [170]. As modeling all the details and behaviors of a real-world system might be infeasible given the high complexity involved, simulators usually make assumptions and abstractions about the real world. Although these can reduce the simulation complexity, they inherently add inaccuracies to the model [130].

One of the most critical tasks in simulation studies is checking if a simulator delivers acceptable accuracy levels given the assumptions and abstractions it implements. This task generally involves two processes: validation and verification [13]. While validation determines whether the conceptual model accurately represents the real world, verification checks whether the conceptual model's implementation is correct.

The conceptual model adopted in EdgeSimPy is based on well-known abstractions representing user mobility, network scheduling, and application provisioning, which have already been formalized and discussed in previous work [70] [50] [80]. As EdgeSimPy adopts such abstractions without modification, model validation is beyond the scope of this work.

The remainder of this section presents a verification that demonstrates the correctness of the implementation of EdgeSimPy's conceptual model. The verification process is conducted using two well-known techniques, Animation and Tracing [170], which display the simulated entities' behavior over time, allowing us to check the simulator's logic correctness.

4.4.1 Scenario Description

Figure 4.7 depicts the edge infrastructure used in EdgeSimPy's verification. We consider three NVIDIA Jetson TX2 boards that represent edge servers. Each edge server has 4 CPU cores, 8 GB of RAM, and 64 GB of storage (CPU and memory values are obtained from Süzen et al. [150]). Edge servers can download at most three container layers at once, following Docker's default configuration to avoid network congestion[5]. The edge network comprises nine links with heterogeneous capacities (for simplicity sake, L1-L4 have a bandwidth capacity of 9 Mbps while L5–L9 have 2.5 Mbps).

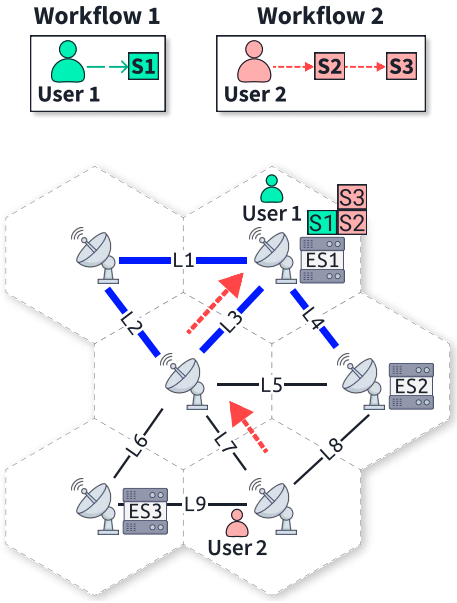


Figure 4.7 – Overview of the infrastructure considered in EdgeSimPy's verification. Symbols "L", "ES", and "S" denote the infrastructure's network links, edge servers, and services.

In this scenario, two users move according to the Pathway model [70], each accessing one of the two existing applications. As shown in Figure 4.8, Application 1 has a stateless service (S1), while Application 2 comprises a stateless service (S2) and a stateful service (S3). Services S1 and S2 use the same image with a single layer (L1), while S2 has an image with four layers (L1–L4). All services are initially on edge server ES1, which also hosts a container registry demanding 1 CPU core, 1 GB of memory, and 10 MB of disk.

[5] https://docs.docker.com/engine/reference/commandline/pull/#concurrent-downloads

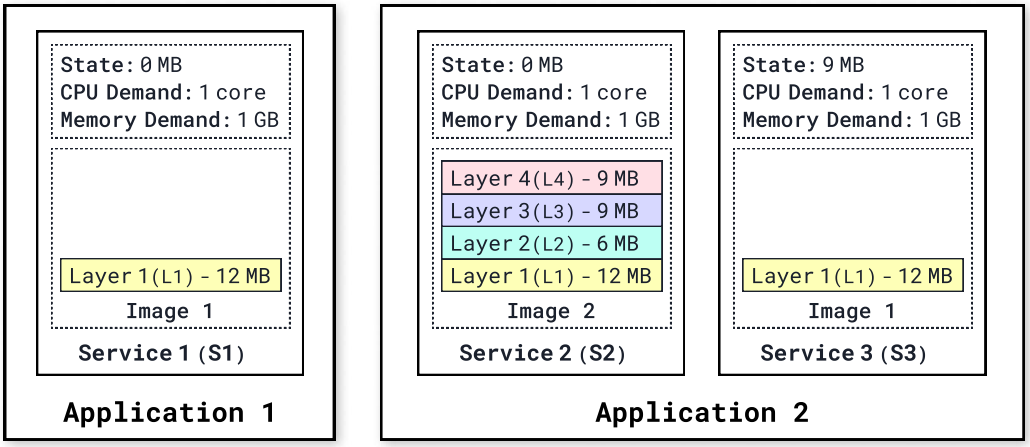


Figure 4.8 – Application specifications used in EdgeSimPy's verification.

In the first time step, all services start to be moved out of edge server ES1. This behavior is defined through a simple resource allocation strategy to demonstrate the provisioning process. Specifically, S1 is moved to edge server ES3 through links L3 and L6, while S2 and S3 are moved to edge server ES2 through link L4. The reallocation of the three services takes six time steps. For simplicity, each time step corresponds to 1 second, and the Max-Min Fairness algorithm [50] is used as the network flow scheduling policy.

The remainder of this section presents a discussion that demonstrates that EdgeSimPy reproduces the expected behavior. For such, we draw a parallel between the result of the EdgeSimPy simulation, presented in Figures 4.9-4.10 and Table 4.3, and the conceptual model adopted within the simulator (i.e., provisioning services according to the container lifecycle and sharing bandwidth based on the Max-Min Fairness algorithm [50]).

Table 4.3 – Edge servers state throughout EdgeSimPy's verification simulation.

Step	Instance	Demand			Services	Waiting Queue	Download Queue	Layers
		CPU	RAM	Disk				
T_1	1	4	4096	46	S1, S2, S3			Registry, L1, L2, L3, L4
	2	2	2048	36		L4	L1, L2, L3	
	3	1	1024	12			L1	
T_2	1	4	4096	46	S1, S2, S3			Registry, L1, L2, L3, L4
	2	2	2048	36		L4	L1, L3	L2
	3	1	1024	12			L1	
T_3	1	4	4096	46	S1, S2, S3			Registry, L1, L2, L3, L4
	2	2	2048	36			L1, L4	L2, L3
	3	1	1024	12			L1	
T_4	1	4	4096	46	S1, S2, S3			Registry, L1, L2, L3, L4
	2	2	2048	36				L2, L3, L1, L4
	3	1	1024	12			L1	
T_5	1	2	2048	46	S1, S3			Registry, L1, L2, L3, L4
	2	2	2048	36	S2			L2, L3, L1, L4
	3	1	1024	12			L1	
T_6	1	1	1024	46				Registry, L1, L2, L3, L4
	2	2	2048	36	S2, S3			L2, L3, L1, L4
	3	1	1024	12	S1			L1

4.4.2 Verification Discussion

Figure 4.9 shows the progress of the network flows spawned by migrations. Service S1 is migrated over links L3 and L6, which have different bandwidths (9 Mbps and 2.5 Mbps, respectively). EdgeSimPy equalizes the bandwidth available for service provisioning with the bandwidth of the link with the lowest capacity. Consequently, layer L1, which corresponds to the image of service S1, is transferred at 2.5 Mbps during time steps 1–5.

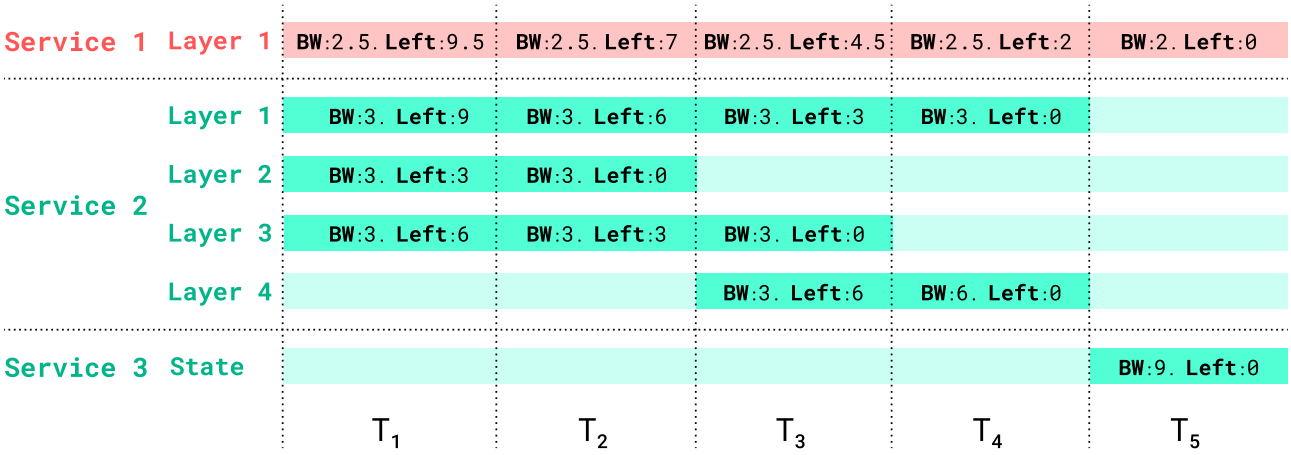
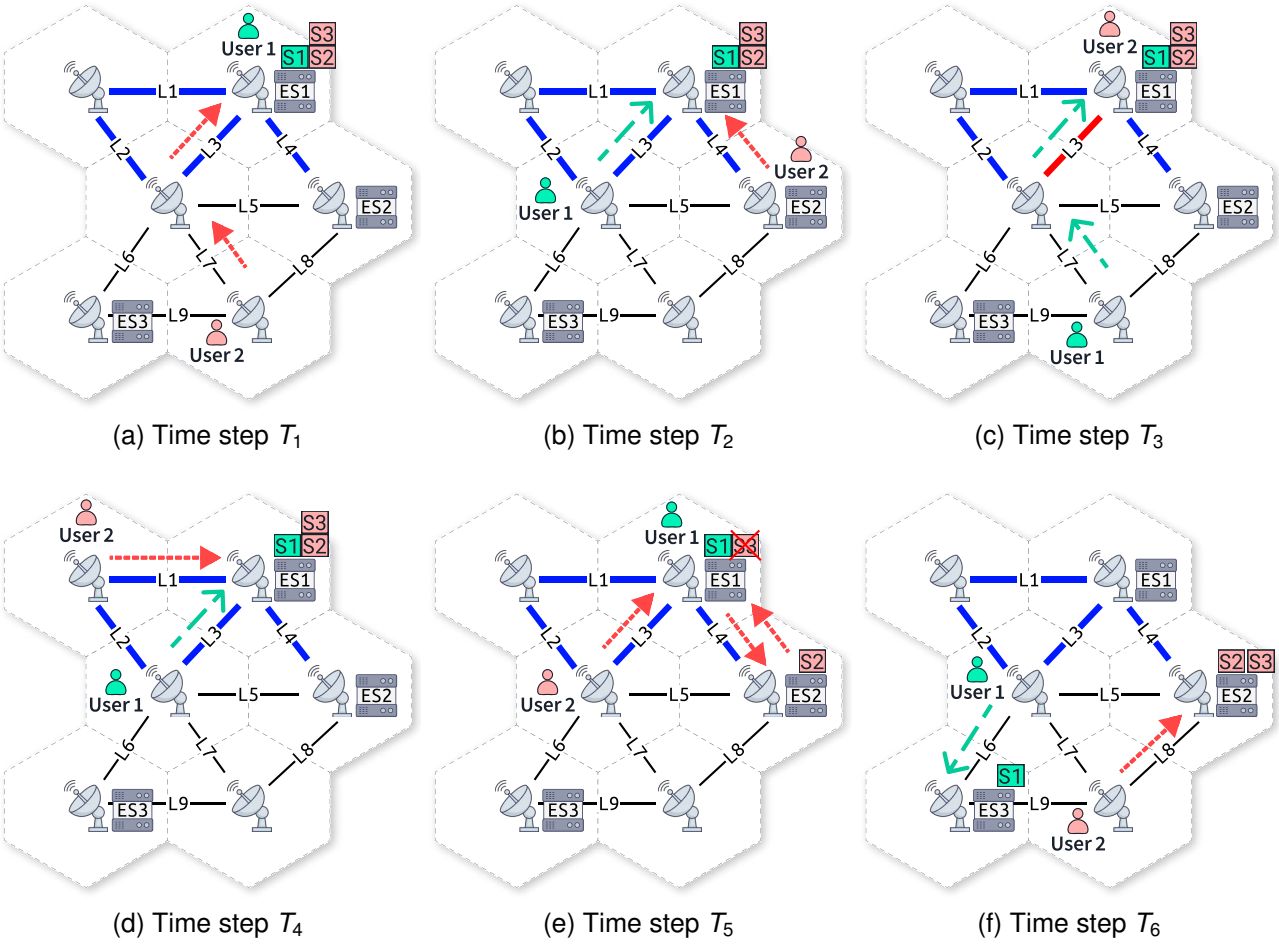


Figure 4.9 – Network flows used to transfer container layers and service states among servers. For conciseness, “BW" denotes the bandwidth available for the network flows, and “Left" denotes the remaining data that will be transferred in subsequent time steps.

Services S2 and S3 are moved from edge server ES1 to edge server ES2 through the same path, sharing the bandwidth of link L4. As the images of S2 and S3 are based on layer L1, ES2 does not pull L1 twice. Despite the layer sharing optimization, the edge server ES2 can only pull three layers at once, forcing layer L4 to wait during time steps T_1 and T_2 until the number of active downloads of ES2 decreases as the transfer of layer L2 ends.

At time step T_4, the Max-Min Fairness algorithm distributes 3 Mbps and 6 Mbps to the active flows of layers L1 and L4, respectively, instead of giving them an equal bandwidth share. This happens because Max-Min Fairness divides the available bandwidth proportionally to the size of the network flows. As layer L1’s flow only needs 3 Mbps to finish, the 1.5 Mbps leftover is offered to L4’s flow, which can be transferred at 6 Mbps.

The differences between stateless and stateful service provisioning are noticeable in time step T_5. Once all layers of S3’s image are present in edge server ES2, S3 is stopped on its source host ES1, and its state is transferred to ES3. EdgeSimPy keeps S3 unavailable during time step T_5 while its state is transferred, as shown in Figure 4.10. Further information on service provisioning is presented in Table 4.3, which details resource demand and provisioned layers on the edge servers throughout the simulation.



(a) Time step T_1 (b) Time step T_2 (c) Time step T_3

(d) Time step T_4 (e) Time step T_5 (f) Time step T_6

Figure 4.10 – Dynamics of each time step of EdgeSimPy's verification. Dashed arrows represent the network paths used for communication between users and services.

4.5 Case Studies

This section presents two case studies from research papers that used EdgeSimPy as a validation platform for resource management on edge infrastructures. An in-depth discussion of each case study can be found in the original publications [146] (Section 4.5.1) and [149] (Section 4.5.2). Consequently, this section focuses on describing the motivation, scenarios, and adjustments made to EdgeSimPy for the simulations rather than discussing the results. These case studies illustrate how EdgeSimPy can be easily extended to comprehend various resource management scenarios.

4.5.1 Case Study 1: Application Migration

As Edge Computing research matures, large-scale infrastructure providers are starting to move toward a new service model called Edge-as-a-Service (EaaS), which har-

nesses the physical proximity of the edge in an infrastructure capable of providing the speed necessary to meet the demand of latency-sensitive applications [26] [72]. Just as cloud services have led a technological shift over the past decade, EaaS offerings display great potential to become the “next big thing" in the IT industry.

This case study explored the possibility of federated edges, in which coalitions of EaaS providers are created to improve profits and meet application performance expectations [6]. In such a scenario, microservice-based applications must be migrated according to user mobility while respecting the privacy requirements of specific microservices.

Whereas infrastructure providers are willing to share resources to reduce application latency bottlenecks, users exhibit distinct levels of trust with different providers, which restricts allocation options for services with special privacy requirements. Performance evaluation compared a novel algorithm with three migration strategies from the literature on the number of migrations and SLA violations, which consider predefined latency thresholds of applications and the privacy requirements of microservices.

EdgeSimPy’s base architecture does not include a functional abstraction for infrastructure providers. Therefore, to support this case study, a “*provider*” attribute is added to edge servers to identify their infrastructure providers, and a “*trusted_providers*” attribute is added to users with the providers they trust. During simulation, both parameters are checked to arrange edge servers according to the level of trust between users and infrastructure providers. In addition, a “*privacy_requirement*” attribute is added to the services to specify their level of privacy requirement.

The simulated scenario consisted of an infrastructure with 60 edge servers managed by two infrastructure providers and 240 services with heterogeneous capacity, privacy, and latency requirements. The results obtained demonstrated that certain migration decisions, that is, prioritizing services with special privacy requirements from applications with tight latency demands, can reduce privacy leaks by up to 7.95% at federated edges without sacrificing application latency [146].

4.5.2 Case Study 2: Edge Server Maintenance

Edge infrastructures typically comprise computing resources deployed near end users, as they help to provide low latency to applications [132]. At the same time, proximity also forces infrastructure dispersion, as the deployment of large-scale data centers close to urban centers may not be feasible [134]. Although the distribution of computing resources allows them to be close to end users, it also introduces IT operation challenges.

In case of edge resources cannot be deployed on a large scale within centralized facilities, communication between nodes is typically based on public networks, which in-

creases the chance of instability caused by outages and a lower bandwidth [8]. In addition, edge devices deployed outdoors are more prone to physical problems and security threats. In such a scenario, infrastructure operators must invest in maintenance strategies to mitigate possible security threats and infrastructure failures.

This case study focused on an edge server maintenance scenario. This case assumed that the patches require the edge servers to be rebooted for the changes to take effect. Consequently, before an edge server can be updated, the applications it hosts must be relocated to another server to avoid downtime (this process is called server draining). As such, maintenance strategies need to schedule the server update order and define new hosts for the applications hosted by the servers that will be updated.

This case study imposed two main objectives on maintenance strategies: reducing maintenance time and application latency bottlenecks. From a security perspective, maintenance must be completed as soon as possible, as patches can correspond to critical security updates, so the infrastructure remains vulnerable until all edge servers are updated. At the same time, applications hosted on the infrastructure are latency sensitive, so migration decisions performed to drain servers must keep applications close enough to their users to ensure that latency remains low.

The default implementation of edge servers in EdgeSimPy does not have an attribute related to maintenance. Therefore, to support this case study, an "*updated*" attribute is added to edge servers to denote their updated status (this attribute is False by default). An entity "*Patch*" is also created with the time required to complete the update. Edge servers and patches are bounded through a relationship attribute. Finally, a method "*update*()" is implemented within the edge server entity, representing the patching process. Once an edge server starts to be updated, it is tagged as unavailable for its patch duration, and when the update period ends, the edge server's update status is changed accordingly. In this case study, the simulation continues until all edge servers are updated.

This case study evaluation compared two novel algorithms with two strategies from the literature regarding maintenance time, the number of migrations, latency requirement violations, and vulnerability surface (which quantifies how long edge servers remain outdated during maintenance). The simulated scenario comprises an infrastructure with 40 edge servers hosting 90 applications with heterogeneous capacity and latency requirements. The results showed that user-location-aware migration decisions could reduce latency requirement violations by 30.67% on average without extending maintenance time [149].

4.5.3 Discussion

This section motivates the adoption of EdgeSimPy to address the needs of the described case studies, highlighting how using it is more practical than the existing simulators.

The first case study focused on the migration of composite applications within an edge environment composed of multiple infrastructure providers. While many of the existing edge simulators offer support for application composition (e.g., IoTSim-Edge, YAFS, and iFogSim2), they lack functional abstractions for infrastructure providers. Additionally, they do not provide features for managing the lifecycle of containers, which is the preferred virtualization technology for composite applications.

The second case study focused on edge server maintenance. Unlike EdgeSimPy, existing edge simulators do not provide the features for modeling maintenance operations, which involves incorporating version attributes to differentiate updated devices from outdated devices and functions to represent the update process.

Although EdgeSimPy's built-in entity attributes had to be extended for conducting the simulations, EdgeSimPy allowed such changes to be included out of the box without requiring modifications to its base features. This was possible because EdgeSimPy automatically identifies custom entity attributes, provided that its input format is followed. This feature mitigates the risk of human-induced errors arising from changes to the simulator's base features.

The ease with which EdgeSimPy allows adding custom attributes and entities is an advantage over existing simulators, which generally require the creation or extension of classes to represent such attributes and entities. For example, ECSNeT++ and FogNetSim++ are built on top of OMNeT++, a discrete-event simulator written in C++. In OMNeT++, custom entities can be modeled using C++ classes called modules. Despite the flexibility of creating classes for new entities to suit specific needs, entity instances are bound by the attributes previously defined in their class definitions, meaning they cannot incorporate unique attributes not initially present in their class constructors. This constraint is also observed in EdgeCloudSim [145], MobFogSim [118], IoTSim-Edge [71], IoTSim-Osmosis [3], and iFogSim2 [95], which extend the functionality of CloudSim [24] and require the extension of built-in classes or the creation of new ones to represent the necessary features.

YAFS [90] shows the closest resemblance to EdgeSimPy in extensibility, as it is also written in Python, which is highly flexible in extending classes. Nevertheless, as YAFS lacks a well-defined format for importing and exporting entities, custom components must be built manually rather than automatically imported through dataset files, as compatible serialization formats (e.g., JSON and XML) cannot recognize advanced data structures used, for example, to define relationships between objects. Additionally, manual instructions must be sent to the simulator regarding how entities are updated over the simulation time and how entity metrics should be gathered. Once the import of custom entities becomes a manual process, YAFS suffers from the same issue as other Java and C++ based simulators. Although this behavior may not be an issue for small tests, it hinders the efficient execution of large-scale batch executions.

4.6 Lessons Learned

Several reflections have emerged during EdgeSimPy's development, providing valuable insights for researchers and practitioners interested in developing simulation tools.

Selecting Python as the base language for EdgeSimPy has enabled users to benefit from a broad ecosystem of libraries, especially those related to emerging fields such as Artificial Intelligence. One key takeaway from our experience developing EdgeSimPy that could benefit future simulator developers involves carefully considering the base programming language, especially regarding the language's popularity within the community and the number of available libraries that users could leverage when using the developed simulator. This approach aims to broaden the simulator's utility, enabling users to harness community-supported algorithms and thus potentially reducing the need for manual implementation.

Throughout our internal testing, we also found that EdgeSimPy's native support for Jupyter Notebooks[6] considerably eases code sharing, as online platforms like Google Colaboratory[7] and MyBinder[8] facilitate real-time collaboration among EdgeSimPy users, eliminating the need for local asset installation. With this in mind, we advise researchers interested in simulator development to select a base programming language and design the simulator with user interactivity as the primary requirement. Leveraging interactive computing platforms can enhance the user experience through an environment that supports real-time collaboration and learning, where users can run experiments with different configurations, observe the results in real-time, and adjust their approaches accordingly.

Another lesson we can share with researchers interested in simulator development is the importance of designing a framework with a decoupled architecture. In EdgeSimPy, we have observed that its strength primarily comes from this highly decoupled structure, which includes class-based modularization and standardized input format, which facilitate the implementation of new entities and attributes, thereby supporting users with specific simulation requirements. This degree of decoupling further translates into a high level of configurability, where users can adjust simulation parameters out-of-the-box, including the simulation tick rate and system models (e.g., user mobility, power consumption of compute and networking devices, network bandwidth sharing control, etc.).

Finally, it should be noted that appropriate calibration of simulator parameters is critical to obtaining an accurate representation of real-world experimental setup. The calibratable parameters include the simulation tick rate and the configuration of the system model to match the scenario being modeled. For example, this may involve defining a geographically accurate map, setting up a realistic power consumption model, or mimicking

[6] https://jupyter.org/
[7] https://colab.research.google.com/
[8] https://mybinder.org/

actual user mobility patterns. Additionally, the selection of metrics to be collected during the simulation is another crucial factor that must be carefully calibrated.

4.7 Closing Remarks

Aside from the development challenges, experimental research at the edge can be expensive and time-consuming as it involves building and instrumenting an infrastructure with many interconnected devices. Furthermore, the distributed nature of the edge makes room for several external factors (e.g., network instability) to affect the reproducibility and reliability of results, undermining the extraction of insights needed to mature prototypes. Consequently, simulation has been considered the main approach to accelerate the validation of prototypes in the early stages at a reduced cost [128].

Existing Edge Computing simulators incorporate various features for modeling user mobility, application composition, and energy consumption of the edge infrastructure. However, they fail to provide fine-grained control over container provisioning, which has been acknowledged as the primary choice for deploying applications at the edge.

To fill this gap, we presented EdgeSimPy, a novel simulator that models the lifecycle of containerized applications through several functional abstractions that replicate the behavior of container runtimes like Docker. In addition, EdgeSimPy features a flexible input format that allows users to define custom parameters for simulated entities out-of-the-box, extending the simulator's built-in capabilities without modifying its core features. EdgeSimPy's source code can be accessed via GitHub[9]. Furthermore, a tutorial library[10] with several practical examples has been developed to help researchers and practitioners effectively integrate EdgeSimPy into their work.

The following chapter presents two maintenance strategies, modeled and evaluated with EdgeSimPy, that optimize maintenance operations at the edge, addressing the second identified challenge described in Section 3.5 regarding performing location-aware application relocations during maintenance at the edge.

[9] https://github.com/EdgeSimPy/EdgeSimPy
[10] https://github.com/EdgeSimPy/edgesimpy-tutorials

5. LOCATION-AWARE EDGE SERVER UPDATES

The second challenge identified during our review of the literature (Section 3.5) was to perform location-aware application relocations during edge maintenance. This chapter addresses such a challenge through two maintenance strategies (Lamp and Laxus), which incorporate user location awareness into migration decisions performed during edge maintenance, reducing latency issues during edge server updates.

First, this chapter discusses the motivations that highlight the literature gap related to edge maintenance (Section 5.1). Then, it introduces the system model (Section 5.2) and the design details of the proposed strategies (Section 5.3). Finally, it presents a performance evaluation that demonstrates the effectiveness of proposed solutions compared to existing maintenance algorithms (Section 5.4).

5.1 Motivation

Edge Computing infrastructure operators have the responsibility of keeping the edge infrastructure up and running, which is not trivial as resources are scarce, and the network infrastructure is less robust than its cloud counterpart. Worse still, attacks targeting the edge are becoming more and more frequent [171]. For example, the Mirai virus [7] orchestrated a Distributed Denial of Service attack against edge servers that led to downtime in more than 178000 domains.

During maintenance in cloud data centers, operators can evacuate servers by relocating hosted applications to other servers, typically observing if the demand for applications could be met, regardless of the new server location within the data center [174]. However, edge servers may have heterogeneous hardware configurations, meaning that some hosts in the edge infrastructure may not deliver analogous performance for applications. Additionally, applications executed on edge environments usually have tight locality and latency constraints, which narrow the candidate hosts that could accommodate them while delivering acceptable response times.

There has been considerable prior work on cloud maintenance [177] [174] [147]. While some strategies could be adapted to edge computing, they overlook users' locations when relocating applications. Although this characteristic does not significantly impact cloud data centers, it can generate a significant increase in application latency on edge environments. Therefore, migration techniques that consider users' location [91] can be adapted to maintenance scenarios, ensuring that the impact of maintenance on applications' performance remains as low as possible.

5.2 System Model

This section describes the edge maintenance scenario that is approached in this work. First, we describe the elements of the edge infrastructure. Then, we formulate the steps that comprise the maintenance process. Table 5.1 summarizes the notations.

Table 5.1 – Summary of main notations used in this chapter.

Symbol	Description
b_f	Wireless latency of base station $\mathcal{B}_f$
ξ_u	Delay of network link $\mathcal{L}_u$
$\aleph_u$	Bandwidth capacity of network link $\mathcal{L}_u$
ρ_i	Capacity of edge server $\mathcal{S}_i$
η_i	Demand of edge server $\mathcal{S}_i$
μ_i	Update status of edge server $\mathcal{S}_i$
$\wp_i$	Update time of edge server $\mathcal{S}_i$
∂_i	Sanity check time of edge server $\mathcal{S}_i$
λ_j	Demand of application $\mathcal{A}_j$
ω_j	Delay of application $\mathcal{A}_j$
$\hbar_j$	Delay threshold of application $\mathcal{A}_j$
$x_{i,j}$	Matrix that represents the application placement
$\Upsilon(\mathcal{A}_j, \mathcal{S}_i)$	Set of links used to migrate $\mathcal{A}_j$ to $\mathcal{S}_i$

We represent the environment as in Aral et al. [9], dividing the map into several hexagonal cells. Edge infrastructure comprises a set of interconnected base stations $\mathcal{B}$ equipped with edge servers $\mathcal{S}$, positioned in each map cell. While base stations provide wireless connectivity to a set of users $\mathcal{U}$, edge servers host user applications $\mathcal{A}$. We represent a base station as $\mathcal{B}_f \leftarrow \{b_f\}$, where b_f is $\mathcal{B}_f$'s wireless latency, and a network link as $\mathcal{L}_u \leftarrow \{\xi_u, \aleph_u\}$, where the attributes ξ_u and $\aleph_u$ represent $\mathcal{L}_u$'s latency and bandwidth.

We model an edge server as $\mathcal{S}_i = \{\rho_i, \eta_i, \mu_i, \wp_i, \partial_i\}$. Attributes ρ_i and η_i represent $\mathcal{S}_i$'s capacity and demand, respectively. More specifically, η_i is the sum of the demand of all applications hosted by $\mathcal{S}_i$. The update status of $\mathcal{S}_i$ is denoted by μ_i, which is set to 1 when $\mathcal{S}_i$ is updated and to 0 otherwise. Patching $\mathcal{S}_i$ takes $\wp_i$ units of time. After patching $\mathcal{S}_i$, we execute sanity checks that validate its integrity after the update, which takes ∂_i units of time.

An application has the following properties $\mathcal{A}_j = \{\lambda_j, \omega_j, \hbar_j\}$. Here, λ_j represents the application's demand, which is considered when choosing which server will host it. The latency ω_j of application $\mathcal{A}_j$ is calculated as in Eq. 5.1, considering the wireless latency of the base station used by $\mathcal{A}_j$'s user (denoted as $\mathcal{B}_f$) summed to the aggregated latency of the network links used to communicate $\mathcal{A}_j$ to its user.

As $\mathcal{A}_j$'s user moves around the map, a handoff process switches him from one base station to another. In such a scenario, if $\mathcal{A}_j$'s user is not connected to the same base station whose server hosts $\mathcal{A}_j$, the connection between them is made by routing $\mathcal{A}_j$'s data through a set of network links called θ. We define the set of links θ through the Dijkstra shortest

path algorithm [37] using the link latencies as weight. An SLA violation occurs when the application's latency ω_j exceeds its latency threshold $\hbar_j$. The application placement is given by a binary matrix x, where $x_{i,j}$ receives 1 if $\mathcal{S}_i$ hosts $\mathcal{A}_j$ and 0 otherwise.

$$\omega_j \leftarrow b_f + \sum_{v=1}^{|\theta|} \xi_v \tag{5.1}$$

The addressed maintenance scenario focuses on updating each server $\mathcal{S}_i \in \mathcal{S}$, where the maintenance process is divided into a set of batches $\mathcal{Q} = \{\mathcal{Q}_1, \mathcal{Q}_2, ..., \mathcal{Q}_{|\mathcal{Q}|}\}$ as in the model by Zheng et al. [177]. Maintenance continues in several batches until $\sum_{i=1}^{|\mathcal{S}|} \mu_i = |\mathcal{S}|$, which means that all servers have been updated. Our goal is to update servers as soon as possible while performing as few migrations as possible and avoiding SLA violations.

In our modeling, servers need to be rebooted for patches to take effect. Also, only servers that host no applications can be updated as a means of avoiding application downtime. At each maintenance batch $\mathcal{Q}_e \in \mathcal{Q}$, all outdated edge servers that host no applications are updated. Then, migrations take place, relocating applications out of the remaining outdated servers so that they can be updated in the next batch (this migration process is called "server draining" as the goal of migrations is "emptying" the outdated servers)[1]. This process is repeated for a number of maintenance batches until all servers are updated.

We assume that there is no shared storage in the infrastructure. Therefore, migrating an application $\mathcal{A}_j$ to an edge server $\mathcal{S}_i$ implies transferring $\mathcal{A}_j$'s demand λ_j from its current host to $\mathcal{S}_i$ through a set of links $\Upsilon(\mathcal{A}_j, \mathcal{S}_i) \subseteq \mathcal{L}$ that interconnect the edge servers' base stations. We define $\Upsilon(\mathcal{A}_j, \mathcal{S}_i)$ through the Dijkstra shortest path algorithm (link bandwidths are used as weight) [37]. In this context, the time it takes to migrate $\mathcal{A}_j$ to $\mathcal{S}_i$ is given by the ratio between $\mathcal{A}_j$'s demand λ_j and the bandwidth available for migration, as shown in Eq. 5.2. As the edge network may be heterogeneous, the set of links $\Upsilon(\mathcal{A}_j, \mathcal{S}_i)$ may have different bandwidth capacities. Accordingly, the lowest available bandwidth between the links in $\Upsilon(\mathcal{A}_j, \mathcal{S}_i)$ is considered the actual bandwidth for the migration.

$$\kappa(\mathcal{A}_j, \mathcal{S}_i) = \frac{\lambda_j}{min\{\aleph_u \mid u \in \Upsilon(\mathcal{A}_j, \mathcal{S}_i)\}} \tag{5.2}$$

5.3 Location-Aware Edge Server Maintenance

This section presents two maintenance strategies that incorporate user location awareness into migration decisions. The first strategy, Lamp, introduces cost functions that decide when and how the servers are drained (Section 5.3.1). The second strategy, Laxus, uses a genetic algorithm to make Pareto-optimal migration decisions (Section 5.3.2).

[1] Migrations are performed sequentially as in Zheng et al. [177].

5.3.1 Lamp

At the beginning of each maintenance batch, Lamp updates all outdated servers that do not host applications (Alg. 1, lines 3–6). Then, if there are still outdated servers in the infrastructure, it starts to migrate applications to drain the servers that still need to be updated (Alg. 1, lines 8–20).

Algorithm 1: Lamp maintenance strategy.

```
1  while There are outdated servers in S do
2      F ← Outdated servers in S
3      foreach F_i ∈ F do
4          if F_i hosts no application then
5              Update F_i
6              Remove F_i from F
7      if F is not empty then
8          Sort the elements of F by Eq. 5.3 (asc.)
9          T ← {}
10         foreach server F_i ∈ F do
11             N ← Applications in F_i sorted by demand (desc.)
12             Y ← S − {T ∪ F_i}
13             if checkCapacity(Y, N) = |N| then
14                 foreach application N_j ∈ N do
15                     Y ← Servers in Y sorted by Eq. 5.4 (asc.)
16                     foreach server Y_i ∈ Y do
17                         if ρ_i − η_i ≥ λ_j then
18                             Migrate N_j to Y_i
19                             break
20                 T ← T ∪ {F_i}
```

To select the order in which servers are drained, Lamp sorts outdated servers according to a cost function ϖ (Eq. 5.3), which considers the normalized capacity, demand, and time required to update outdated servers (we normalize variables that have different scales with Min-Max Normalization [59] to ensure that equations are not unbalanced). While patching larger servers means being able to accommodate a larger number of applications on updated servers early, prioritizing less occupied servers with shorter patching time ensures that the infrastructure has up-to-date resources early.

$$\varpi(\mathcal{S}_i) \leftarrow norm\left(\frac{1}{\rho_i}\right) + norm\left(\eta_i\right) + norm\left(\wp_i + \partial_i\right) \tag{5.3}$$

Before performing any migration to drain a given server $\mathcal{S}_i$, Lamp calls a method described in Alg. 2 to check if the other servers that are not being drained in the current batch have enough capacity to host all the applications hosted by $\mathcal{S}_i$ (Alg. 1, line 13). In this way, it avoids migrations that will not result in servers being drained in the current batch, shortening the duration of maintenance batches, which leads to servers being updated early.

Algorithm 2: Lamp's server capacity checking method.

```
1  Function checkCapacity(Y, N):
2      Y' ← List of servers in Y
3      N' ← List of applications in N
4      ϰ ← 0
5      foreach N'_j ∈ N' do
6          foreach Y'_i ∈ Y' do
7              if ρ_i − η_i ≥ λ_j then
8                  Host application N'_j on edge server Y'_i
9                  ϰ ← ϰ + 1
10                 break
11     return ϰ
```

After ensuring that a server $\mathcal{S}_i$ can be drained in the current maintenance batch, Lamp uses a cost function ϕ (Eq. 5.4) to sort the servers that can accommodate each of the applications hosted by $\mathcal{S}_i$ according to their update status, occupation, and latency to the user that accesses the application being migrated, represented by $\Re$ (Alg. 1, line 15). In this way, Lamp tries to guarantee that applications are placed on up-to-date servers close enough to users to avoid SLA violations. Finally, Lamp goes through the ordered list of candidate servers, migrating applications to the first server found with enough capacity to host them (Alg. 1, lines 14–19).

$$\phi(\mathcal{S}_i) \leftarrow (1 - \mu_i) + norm\,(\rho_i - \eta_i) + norm(\Re) \tag{5.4}$$

5.3.2 Laxus

Laxus starts each maintenance batch by updating outdated edge servers that are not hosting applications (Alg. 3, lines 3–6). After that, if there are still outdated edge servers, Laxus performs migrations in an attempt to drain those servers (Alg. 3, lines 8–12). Laxus makes migration decisions through a function called *NSGAII*, a metaheuristic called Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [35]. We use NSGA-II as it effectively finds Pareto-optimal solutions for several multi-objective problems with an $O(MN^2)$ time complexity, where M represents the number of objectives and N is the population size [35].

Genetic encoding[2]**.** The NSGA-II algorithm used by Laxus creates chromosomes representing migration plans to drain outdated servers. Each chromosome $\mathcal{P}_v$ of population $\mathcal{P}$ is a vector of size $|\mathcal{V}|$, where $\mathcal{V}$ denotes the set of applications hosted by outdated servers. Each index j of $\mathcal{P}_v$ represents one of the applications in $\mathcal{V}$, and the value of each index represents the suggested edge server to host these applications. Thus, $\mathcal{P}_{v,j} \leftarrow \mathcal{S}_i$ indicates that $\mathcal{V}_j$ should migrate to $\mathcal{S}_i$. No migration is performed if $\mathcal{S}_i$ already hosts $\mathcal{V}_j$.

[2] At the beginning of the NSGA-II's execution, $\mathcal{P}$ is generated randomly.

Algorithm 3: Laxus maintenance strategy.

```
while There are outdated servers in 𝒮 do
    F ← Outdated servers in 𝒮
    foreach server F_i ∈ F do
        if F_i hosts no application then
            Update F_i
            Remove F_i from F
    if F is not empty then
        𝒲 ← NSGAII(𝒮, 𝒱)
        υ ← Placement ∈ 𝒲 w/ the lowest ϱ (Eq. 5.10)
        for k ← 1 to |υ| do
            if edge server υ_k does not host application 𝒱_k then
                Migrate 𝒱_k to υ_k
```

Constraints and fitness functions. In our modeling, a solution is only considered valid if $\sum_{i=1}^{|\mathcal{S}|} [\eta_i > \rho_i] = 0$, which means that the demand of none of the edge servers exceeds its capacity. We use three fitness functions α, β, and γ that aim to minimize: (i) the number of outdated servers hosting applications; (ii) the migration cost; and (iii) the number of SLA violations, respectively.

The first fitness function α (Eq. 5.5) quantifies the effectiveness of $\mathcal{P}_v$ in draining outdated servers. As maintenance continues until all edge servers are updated, solutions that drain a larger number of servers per batch tend to finish the maintenance early. In addition, in many maintenance scenarios, updating servers as soon as possible is essential. For instance, when servers must receive security patches, draining more of them sooner implies that these can be updated early, reducing the infrastructure's attack surface [147]. Therefore, α accounts for the number of outdated edge servers hosting applications.

$$\alpha(\mathcal{P}_v) \leftarrow \sum_{j=1}^{|\mathcal{P}_v|} 1 - \mu_{\mathcal{P}_{v,j}} \tag{5.5}$$

The second fitness function β (Eq. 5.6) quantifies the migration cost imposed by $\mathcal{P}_v$. First, β considers the average migration time (given by $\tau(\mathcal{P}_v)$, in Eq. 5.7) so that solutions that perform long migrations are penalized. In addition, β also considers the function $\zeta(\mathcal{P}_v)$ (Eq. 5.8), which penalizes solutions that make undesired migrations. More specifically, function $\zeta(\mathcal{P}_v)$ treats as undesired those migrations that meet at least one of the following criteria: (i) the current server of the application being migrated hosts other applications that will not be relocated (meaning that, despite the migration, that server will not be drained in that maintenance batch); (ii) the application is being migrated to an outdated server.

$$\beta(\mathcal{P}_v) \leftarrow \tau(\mathcal{P}_v) \cdot max(1, \zeta(\mathcal{P}_v)) \tag{5.6}$$

$$\tau(\mathcal{P}_v) \leftarrow \frac{1}{|\mathcal{P}_v|} \sum_{j=1}^{|\mathcal{P}_v|} \kappa(\mathcal{V}_j, \mathcal{P}_{v,j}) \tag{5.7}$$

$$\zeta(\mathcal{P}_v) \leftarrow \sum_{j=1}^{|\mathcal{P}_v|} \left[x_{\mathcal{P}_{v,j},j} = 0\right] \cdot \left(\left[\{\mathcal{S}_i \in \mathcal{S} | x_{i,j} = 1\} \subset \mathcal{P}_v\right] + 1 - \mu_{\mathcal{P}_{v,j}}\right) \tag{5.8}$$

The third fitness function γ (Eq. 5.9) quantifies the number of SLA violations produced by allocation decisions made by a solution $\mathcal{P}_v$. Thus, solutions that overlook users' location and migrate applications to servers too distant from users so that the access latency exceeds the application SLAs are penalized for sacrificing the delivered quality of service.

$$\gamma(\mathcal{P}_v) \leftarrow \sum_{j=1}^{|\mathcal{P}_v|} \left[\omega_j > \hbar_j\right] \tag{5.9}$$

Non-dominated sorting. After assigning fitness scores to the population, individuals are arranged on different fronts based on their dominance over other individuals in the population. In addition, each individual receives a score called crowding distance [35], which corresponds to the distance of the individual to its neighboring solutions on the Pareto front. Once the population is arranged on different fronts and the crowding distance of each individual is calculated, the population for the next generation is chosen based on the fronts structure (individuals on the first fronts and with larger crowding distances are prioritized). After this sorting process, only the $|\mathcal{P}|$ best individuals are selected to be part of the population $\mathcal{P}$. We define a fixed number of generations ψ as the stopping criterion. Therefore, the evolution process is repeated until the maximum number of generations ψ is reached.

Pareto-optimal selection. Instead of looking for a single optimal solution in the search space, NSGA-II returns a Pareto Set $\mathcal{W}$ comprising non-dominated solutions in the Pareto Front. Accordingly, as soon as the algorithm's stopping criterion is reached, we must choose which Pareto-Optimal solution will be employed in the maintenance process. For this, we evaluate each solution $\mathcal{W}_v \in \mathcal{W}$ according to a cost function ϱ (Eq. 5.10), selecting the solution with the lowest ϱ (Alg. 3, line 9).

$$\varrho(\mathcal{W}_v) \leftarrow \mathit{norm}(\alpha(\mathcal{W}_v)) + \mathit{norm}(\beta(\mathcal{W}_v)) + \mathit{norm}(\gamma(\mathcal{W}_v)) \tag{5.10}$$

5.4 Performance Evaluation

This section details the experiments conducted to validate the proposed strategies (Lamp and Laxus). First, we discuss our methodology (Section 5.4.1). Then, we detail the

sensitivity analysis executed to find the best set of parameters for Laxus (Section 5.4.2). Finally, we present the results achieved against the baseline strategies (Section 5.4.3).

5.4.1 Experiments Description

We consider an edge computing infrastructure with 40 edge servers interconnected by a Barabási-Albert network topology [15] with links containing latency = {5, 10} and bandwidths = {5, 10}. We assume that servers have heterogeneous capacities = {200, 250}. We update the edge servers with two patches with duration = {250, 350}. Each patch type has sanity checks with duration = {300, 400}. We consider a set of 90 users distributed randomly across the environment and 90 applications with demands = {20, 40, 60} and latency SLAs = {45, 90}[3]. The initial application placement is defined according to Alg. 4.

Algorithm 4: Initial application placement heuristic.

```
1 A' ← List of applications in A arranged randomly
2 foreach A'_j ∈ A' do
3 |   U_r ← User that accesses application A'_j
4 |   S' ← List of edge servers in S sorted by latency from U_r (asc.)
5 |   foreach edge server S'_i ∈ S' do
6 |   |   if ρ_i − η_i ≥ λ_j then
7 |   |   |   Host application A'_j on edge server S'_i
8 |   |   |   break
```

To the best of our knowledge, we are the first to update edge servers while considering the application latency requirements. Therefore, we compare Lamp and Laxus with two maintenance strategies designed to update servers in cloud data centers. The first baseline strategy, called Greedy Least Batch (GLB) [177], aims to reduce the number of maintenance batches needed to update a group of servers. The second baseline strategy, called Salus [147], focuses on security patch scenarios, where the main goal is to reduce the period during which servers are still outdated (i.e., vulnerable to attacks) during maintenance. We chose these baseline strategies because they also model the maintenance process in batches while considering most of the performance metrics we evaluate.

We compare the strategies analyzed in terms of maintenance time, number of migrations, Vulnerability Surface (VS) [147] (which quantifies how long servers remain outdated during maintenance), and number of SLA violations. While the first three metrics assess specific maintenance goals, the number of SLA violations measures the impact of migrations on the quality of service delivered to end-users.

We conducted our experiments on an Ubuntu 20.04.1 LTS host with 8 CPU cores and 32 GB of RAM. The virtual machine was configured with Python 3.7.10 (PyPy 7.3.5 with

[3]Network, edge server, and application parameters are assigned uniformly.

GCC 7.3.1 20180303). We strive to follow reproducible research and open science principles in our investigation. The companion material hosted in a public GitHub repository[4] contains the source code, dataset, and instructions to reproduce our results.

5.4.2 Sensitivity Analysis

Among the compared strategies, Laxus is the only one that contains configurable parameters. Without loss of generality, we define the population size $|\mathcal{P}|$ = 120 and mutation probability to $\frac{1}{|\mathcal{V}|}$. We conducted a sensitivity analysis to determine the best settings between different generations ψ = {100, 200, 300, ..., 1000} and crossover probabilities = {25%, 50%, 75%, 100%}. After testing each combination σ among these parameters, we choose the best configuration based on a score function $\delta(\sigma)$ (Equation 5.11), which considers the normalized sum of each evaluated metric.

$$\delta(\sigma) \leftarrow \mathit{norm}\left(\sigma^{\mathit{time}}\right) + \mathit{norm}\left(\sigma^{\mathit{migr}}\right) + \mathit{norm}\left(\sigma^{\mathit{vs}}\right) + \mathit{norm}\left(\sigma^{\mathit{viol}}\right) \tag{5.11}$$

Figure 5.1 presents the δ score for each number of generations ψ considering the best crossover probability among the tested configurations. The best configurations found were ψ = {800, 900} with crossover probability = 100%. We used ψ = 800 with crossover probability = 100% when comparing Laxus against the other strategies.

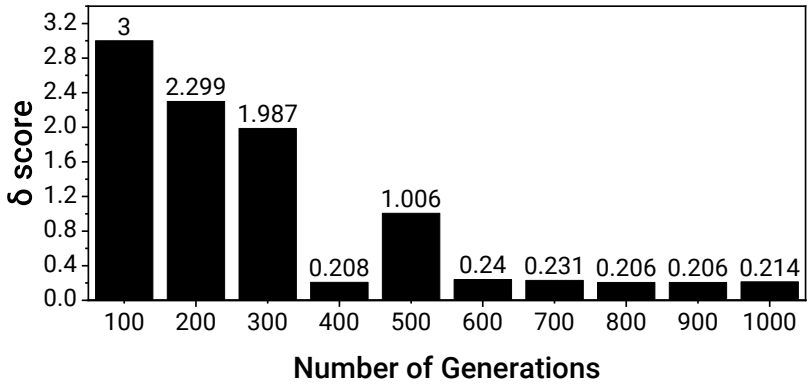


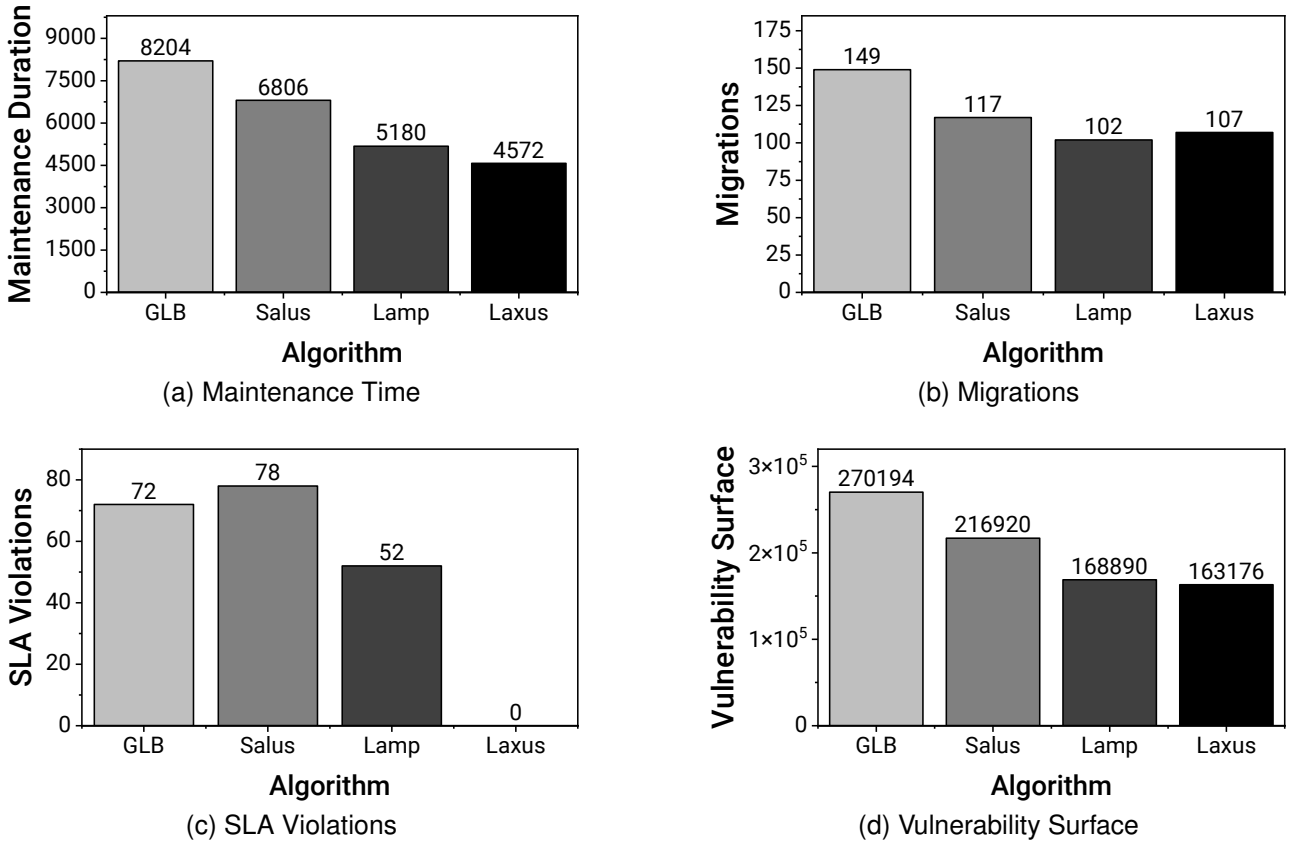
Figure 5.1 – Sensitivity analysis of Laxus parameters.

5.4.3 Comparison with Baseline Heuristics

Figure 5.2(a) shows the results in terms of maintenance time. We can observe that Laxus updates servers faster than other strategies, followed by Lamp, Salus, and GLB,

[4]Experiment assets: https://github.com/paulosevero/lamp_laxus.

respectively. The total time spent on migrations during maintenance was the factor that most influenced the maintenance time during the experiments. While Salus and GLB spent 4556 and 5954 units of time with migrations, Laxus and Lamp completed all their migrations in only 2322 and 2930 units of time, respectively.



(a) Maintenance Time

(b) Migrations

(c) SLA Violations

(d) Vulnerability Surface

Figure 5.2 – Comparison between Lamp, Laxus and baseline approaches.

The reduction in the total time spent on migrations results from Laxus and Lamp's location awareness. As these strategies avoid placing applications far away from their users, they perform fewer long migrations than the other strategies. This characteristic is reflected in the average migration time of Laxus and Lamp (8.18 and 9.61 respectively) compared to GLB and Salus (14.91 and 15.29 respectively).

Figure 5.2(b) shows the number of migrations performed by the strategies during the tests. As we can see, GLB is the one that performs the most migrations during maintenance. This behavior occurs because GLB is the only strategy that does not take into account the demand of applications when performing migrations, which opens space for potential waste of resources compared to other strategies that prioritize migrating larger applications. Although Laxus does not order applications by demand directly, its ability to evolve over a number of generations allows it to find the best packing of applications to drain servers with the lowest migration cost.

Figure 5.2(c) shows the number of SLA violations that occurred while executing the compared strategies. We can observe that the baseline strategies (GLB and Salus)

obtained similar results regarding the number of SLA violations (72 and 78, respectively), as they are both designed to perform maintenance on cloud data centers, performing migrations regardless of users' locations.

Unlike cloud data centers, the edge infrastructure is distributed so that migrations between edge servers distant from one another can significantly affect the application's latency. As Lamp considers the distance between servers and users when performing migrations, it reduces the number of SLA violations by 27.78% and 33.33% compared to GLB and Salus, respectively. Still, Laxus achieves the best results, completing maintenance without any SLA violation. Such gains come from Laxus' ability to test several placement alternatives as it evolves to select the configuration that ultimately achieves the best results.

Figure 5.2(d) presents the results on Vulnerability Surface, which evaluates the time required by maintenance strategies to update servers. As we can see, GLB shows the worst results by overlooking servers' exposure during maintenance. Salus, which strives to avoid unnecessary migrations and update servers as soon as possible, manages to reduce the Vulnerability Surface by 19.72% compared to GLB.

The proposed strategies reduce the Vulnerability Surface by 23.46% on average compared to Salus. This shows the importance of performing short migrations when carrying out security patches on edge computing infrastructures, allowing servers to be updated early. Laxus achieves the best results among the evaluated strategies, updating 30 of the 40 edge servers in the infrastructure 39.65% faster than Lamp, 49.38% faster than Salus, and 61.91% faster than GLB. These gains are primarily due to the reduced migration time.

5.5 Closing Remarks

This chapter presented two maintenance strategies, Lamp and Laxus, which incorporate user location awareness into migration decisions to reduce the impact of maintenance on application latency. While Lamp uses cost functions that reduce application latency issues during maintenance by 31% while performing 23% fewer migrations than existing strategies, Laxus adopts a genetic algorithm that updates the edge infrastructure while causing no application latency issue, performing only 5% more migrations than Lamp.

The following chapter extends this chapter's work on edge server updates through a more realistic system model and a maintenance strategy that leverages the shared content of container images of edge applications to reduce maintenance time through optimized relocations, addressing the third identified challenge described in Section 3.5 regarding optimized container provisioning during maintenance.

6. CONTAINERIZATION-AWARE EDGE SERVER UPDATES

The third challenge identified during our literature review (Section 3.5) was optimizing the provisioning process of containerized applications to reduce maintenance time. This chapter addresses this challenge through a maintenance strategy called Hermes. Hermes extends our work on edge server updates described in the previous chapter, capitalizing on the shared content of container images to reduce maintenance time through optimized application relocations.

First, this chapter revisits the motivation to reduce maintenance time (Section 6.1). Then, it introduces the adopted system model (Section 6.2) and the Hermes design details (Section 6.3). Finally, it presents a performance evaluation that shows the effectiveness of Hermes in reducing maintenance time compared to baseline strategies (Section 6.4).

6.1 Motivation

When maintenance involves applying updates that require server reboots, affected applications must be relocated to alternative servers to ensure service continuity. During this process, deciding on target servers for applications that must be relocated involves considering several factors. Considering Edge Computing infrastructures, where applications have stringent latency requirements, it is necessary to assess the relocation's impact on the quality of service delivered to end-users. Also, it is vital to analyze how relocations contribute to maintenance progression, as prolonged relocations can delay server maintenance, which becomes even more critical in scenarios where maintenance is aimed at correcting security vulnerabilities, and patches must be applied as fast as possible.

Although existing maintenance strategies [112] [156] [25] [174], including our work in Chapter 5, have exploited specific virtualization-based approaches during server updates at the edge, they present some limitations that motivate further research. First, the proposed strategies employ theoretical models assuming sequential relocations, overlooking the required coordination of concurrent relocations within the network, which distances them from practical implementations. Secondly, relocations occur according to the VM model. While there is a motivation for using VMs in specific scenarios, containers have taken the lead as the prime architecture for deploying applications at the edge.

As discussed in Section 2.1, there are considerable differences between relocating VM-based and container-based applications. Since VM images are monolithic, VM-based applications are migrated directly from a source server to a target. On the contrary, relocating container-based applications involves pulling their container images from image repositories called container registries to the target servers and optionally transferring application infor-

mation (e.g., user session data and runtime state) from the source server to the target server in cases where applications are stateful [151] [172]. Additionally, since co-hosted containers share common layers of their respective images, provisioning a containerized application on a server already possessing layers that constitute its container image is faster, as fewer data needs to be downloaded from container registries.

This chapter's research builds on the observation that existing maintenance strategies rely on conceptual models grounded in the VM paradigm. As such, they neglect the composition of container images stored on edge servers during the decision-making process for application relocation, thereby missing opportunities to reduce relocation times.

To fill this gap, we introduce a novel conceptual model that formalizes maintenance operations where applications residing on outdated edge servers are relocated to alternative hosts using a containerization procedure that mirrors the behavior of well-known container platforms like Docker. In addition, we propose a novel maintenance strategy that reduces maintenance time through efficient application relocations that consider the degree of container layer sharing among applications that require relocation and on the set of container layers downloaded from edge servers. Finally, we demonstrate the effectiveness of our strategy through a set of simulated experiments against baseline approaches that overlook the architectural characteristics of containerized applications during maintenance work.

6.2 System Model

This section describes the edge server maintenance scenario addressed. First, we detail the core elements of the edge infrastructure. Then, we describe the various steps that constitute the maintenance process, including the specification of the application provisioning process and their performance requirements, alongside our optimization objectives. Table 6.1 summarizes the notations.

We represent the geographical positioning of elements according to the map model presented by Aral et al. [9], which divides the map regions into hexagonal cells. Although such a model abstracts discrete locations within the same hexagonal cell and assumes nonintersection among different cells, it provides a suitable representation for typical cellular networks [9]. The edge infrastructure leverages the base stations that compose the cellular network and are positioned in each map cell. In this setting, base stations are interconnected by network links and can be optionally equipped with edge servers. Whereas base stations provide wireless connectivity to users, network links allow wired communication among edge servers across different base stations.

Each edge server is modeled as $\mathcal{E}_i = \{c_i, r_i, d_i, p_i\}$, where c_i, r_i, and d_i represent $\mathcal{E}_i$'s CPU, RAM, and disk capacity specifications, respectively. In addition to capacity attributes, each edge server $\mathcal{E}_i$ has an attribute p_i that describes the time required to patch it and a

Table 6.1 – Summary of notations used in this chapter.

Symbol	Description
$\mathcal{Q}$	Set of maintenance batches
$\mathcal{B}$	Set of base stations
$\mathcal{E}$	Set of edge servers
$\mathcal{N}$	Set of network links
$\mathcal{A}$	Set of applications
$\mathcal{L}$	Set of container layers
$\mathcal{F}$	Set of network flows
$\Omega(\mathcal{Q}_q)$	Duration of maintenance batch $\mathcal{Q}_q$
c_i	$\mathcal{E}_i$'s CPU capacity
r_i	$\mathcal{E}_i$'s RAM capacity
d_i	$\mathcal{E}_i$'s Disk capacity
p_i	$\mathcal{E}_i$'s Patch Time
$\xi(\mathcal{E}_i)$	$\mathcal{E}_i$'s Update state
$\delta(\mathcal{E}_i)$	$\mathcal{E}_i$'s Downloaded layers
$\partial(\mathcal{E}_i)$	$\mathcal{E}_i$'s Download queue
$\lambda(\mathcal{E}_i)$	$\mathcal{E}_i$'s Waiting queue
$\gamma(\mathcal{E}_i, \mathcal{Q}_q)$	$\mathcal{E}_i$'s capacity limit compliance check
b_u	$\mathcal{F}_u$'s source
e_u	$\mathcal{F}_u$'s target
o_u	$\mathcal{F}_u$'s path
t_u	$\mathcal{F}_u$'s total size
$\zeta(\mathcal{F}_u)$	Amount of data from $\mathcal{F}_u$ already downloaded
g_f	$\mathcal{L}_f$'s latency
h_f	$\mathcal{L}_f$'s Bandwidth capacity
$\phi(\mathcal{T})$	Shortest path finder
$\Psi(\mathcal{P})$	Path's accumulated delay
w_j	$\mathcal{A}_j$'s user's base station
z_j	$\mathcal{A}_j$'s user's latency SLA
x_j	$\mathcal{A}_j$'s image layers
s_j	$\mathcal{A}_j$'s state
m_j	$\mathcal{A}_j$'s CPU demand
n_j	$\mathcal{A}_j$'s RAM demand
$\beta(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q)$	$\mathcal{A}_j$'s SLA compliance check
$\alpha(\mathcal{E}_i, \mathcal{A}_j, \mathcal{Q}_q)$	Application placement at maintenance batch $\mathcal{Q}_q$
l_k	$\mathcal{L}_k$'s size

function $\xi(\mathcal{E}_i)$ that returns its update status (i.e., 1 if $\mathcal{E}_i$ is updated and 0 otherwise) at any point during maintenance work. In our modeling, the patching procedure involves rebooting edge servers for updates to take effect. As shutting down applications, even temporarily, should be avoided to preserve the QoS delivered for end-users, only edge servers hosting no applications can be updated. Consequently, maintenance work progresses in a rolling update fashion, where groups of edge servers are gradually evacuated and updated in turns called maintenance batches [177]. Let $\mathcal{Q}$ be the set of maintenance batches required for patching a group of edge servers. For simplicity, we assume that a maintenance batch $\mathcal{Q}_q \in \mathcal{Q}$ lasts until all events it comprises are terminated, and its duration can be obtained through a function $\Omega(\mathcal{Q}_q)$. Figure 6.1 illustrates the adopted batch duration representation.

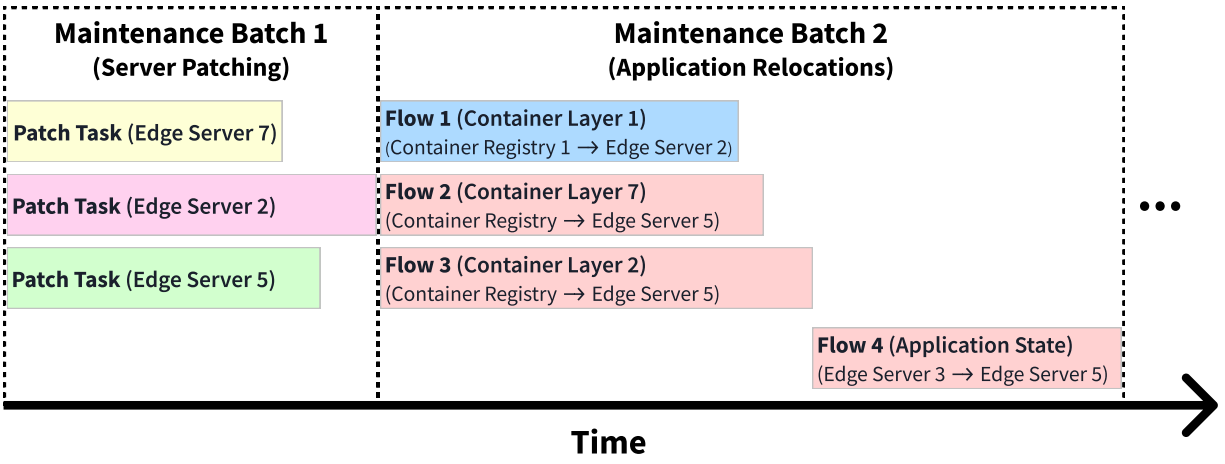


Figure 6.1 – Adopted maintenance batch duration model.

Maintenance progression is heavily influenced by the time necessary to relocate applications housed by outdated servers to alternative hosts. In this context, relocating applications to updated hosts is preferable, as any application relocated to an outdated edge server would need to be moved at least once more when that new host needs to be evacuated to receive its patch. Additionally, our model assumes the existence of no shared storage within the infrastructure. As such, when an application needs to be relocated, its image is pulled to the target host from a container registry, and the application state data are migrated from the source to the target server, if the application is stateful.

Popular container platforms such as Docker[1] set a limit on the number of container layers that edge servers can download simultaneously. Based on that, we define three functions associated with the container layers of a given edge server: the container layers that the edge server has downloaded already ($\delta(\mathcal{E}_i)$), the container layers currently being downloaded ($\partial(\mathcal{E}_i)$), and the container layers waiting to be downloaded ($\lambda(\mathcal{E}_i)$).

When an application is assigned to an edge server, layers from its container image that are absent on that edge server and need to be pulled from a container registry are either added to the edge server's download queue or waiting queue, depending on the edge server's status. More specifically, once the edge server reaches its maximum number of layers being downloaded simultaneously, new layers are added to the waiting queue until the layers from the download queue are successfully downloaded. Otherwise, layers are immediately added to the download queue and start being pulled immediately.

Data transfers across the network infrastructure are modeled as network flows, where each network flow is represented as $\mathcal{F}_u = \{b_u, e_u, o_u, t_u\}$. Here, b_u and e_u represent $\mathcal{F}_u$'s source and target edge servers, respectively, o_u is the network path used to connect b_u and e_u, and t_u represents $\mathcal{F}_u$'s total size (e.g., the size of the application state or the container layer to be transferred). Beyond these static attributes of network flows, we define

[1] https://docs.docker.com/engine/reference/commandline/pull/#concurrent-downloads

a helper function, $\zeta(\mathcal{F}_u)$, to verify the amount of data from $\mathcal{F}_u$ that has been successfully downloaded until any given point in time. Accordingly, a network flow $\mathcal{F}_u$ remains active until $\zeta(\mathcal{F}_u) = t_u$. The time it takes to complete the transfer of a network flow $\mathcal{F}_u$ depends on the bandwidth provided by the links it spans and the resource allocation policy of the network switches, which can adjust the available bandwidth for network flows based on changing factors such as network load.

The duration of network flows (and, consequently, the provisioning time of applications) is inherently affected by the characteristics of the network links involved in the data transfer. In our modeling, a network link is represented by $\mathcal{L}_f = \{g_f, h_f\}$, where g_f and h_f represent $\mathcal{L}_f$'s latency and overall bandwidth capacity, respectively. To facilitate the computation of network operations, we define two additional helper functions. The first function, $\phi(\mathcal{T})$, returns the shortest network path that connects a set $\mathcal{T}$ with any two elements within the infrastructure (e.g., users, applications, edge servers, etc.). The second function, $\Psi(\mathcal{P})$ (Eq. 6.1), calculates the accumulated delay of any network path $\mathcal{P}$ obtained using $\phi(\mathcal{T})$.

$$\Psi(\mathcal{P}) = \sum_{N_f \in \phi(\mathcal{T})} g_f \tag{6.1}$$

An application is represented as $\mathcal{A}_j = \{w_j, z_j, x_j, s_j, m_j, n_j\}$. The first two attributes, w_j and z_j, are related to the application's user: while w_j references the base station to which $\mathcal{A}_j$'s user is connected, z_j describes the latency requirement of $\mathcal{A}_j$'s user. As maintenance decisions to evacuate outdated edge servers may imply assigning applications to edge servers at varying network distances to their users, observing the impact of relocations over application latency is desirable.

We obtain the latency of an application $\mathcal{A}_j$ by combining functions $\phi(T)$ and $\Psi(P)$ to find the shortest path between $\mathcal{A}_j$'s user and $\mathcal{A}_j$'s host server and then calculate the accumulated delay of such path. We also use a helper function $\beta(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q)$ (Eq. 6.2) to check if $\mathcal{A}_j$'s current latency meets its latency requirement in a maintenance batch $\mathcal{Q}_q$, given that $\mathcal{A}_j$ is hosted by an edge server $\mathcal{E}_i$. In our modeling, SLA violations refer to periods where $\mathcal{A}_j$'s latency exceeds its user's latency requirement (i.e., when $\beta(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q) = 1$).

$$\beta(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q) = \begin{cases} 1 & \text{if } \Psi(\phi(\{w_j, \mathcal{E}_i\})) > z_j \\ 0 & \text{otherwise.} \end{cases} \tag{6.2}$$

As we focus on containerized deployments, attribute x_j of an application $\mathcal{A}_j$ references the list of container layers that constitute $\mathcal{A}_j$'s container image, with each of these container layers represented as $\mathcal{I}_k = \{l_k\}$, where l_k stands for $\mathcal{L}_k$'s size. Since containerized applications can be stateless or stateful, attribute s_j carries the size of $\mathcal{A}_j$'s state, set to zero if $\mathcal{A}_j$ is stateless. Finally, attributes m_j and n_j represent $\mathcal{A}_j$'s CPU and RAM demand, respectively. The application's placement can be checked at any point during maintenance

using the helper function $\alpha(\mathcal{E}_i, \mathcal{A}_j, \mathcal{Q}_q)$, described in Eq. 6.3. As edge servers have limited capacity, we use a helper function $\gamma(\mathcal{E}_i, \mathcal{Q}_q)$ to check if the aggregated demand of the applications hosted by an edge server $\mathcal{E}_i$ in a maintenance batch $\mathcal{Q}_q$ does not exceed $\mathcal{E}_i$'s overall capacity (i.e., $\gamma(\mathcal{E}_i, \mathcal{Q}_q) = 1$ if $\mathcal{E}_i$ is overloaded and 0 otherwise).

$$\alpha(\mathcal{E}_i, \mathcal{A}_j, \mathcal{Q}_q) = \begin{cases} 1 & \text{if edge server } \mathcal{E}_i \text{ hosts application } \mathcal{A}_j \text{ at maintenance batch } \mathcal{Q}_q \\ 0 & \text{otherwise.} \end{cases} \tag{6.3}$$

We consider a twofold objective function (Eq. 6.4), which minimizes the number of SLA violations and the overall maintenance duration (i.e., the sum of the duration of all batches required to patch all outdated edge servers), subject to two constraints. The first constraint (Eq. 6.5) ensures that each application is provisioned only on one edge server in a given maintenance batch $\mathcal{Q}_q \in \mathcal{Q}$. The second constraint (Eq. 6.6) ensures that no edge server is overloaded. As the problem objectives can assume values on different scales, they are normalized using the Min-Max Normalization method [59].

$$\textbf{Min: } \sum_{q=1}^{|\mathcal{Q}|} norm(\Omega(\mathcal{Q}_q)) + norm(\beta(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q)) \tag{6.4}$$

Subject to:

$$\sum_{i=1}^{|\mathcal{E}|} \alpha(\mathcal{E}_i, \mathcal{A}_j, \mathcal{Q}_q) = 1,\ \forall k \in \{1, ..., |\mathcal{A}|\},\ \forall q \in \{1, ..., |\mathcal{Q}|\} \tag{6.5}$$

$$\sum_{i=1}^{|\mathcal{E}|} \gamma(\mathcal{E}_i, \mathcal{Q}_q) = 0,\ \forall q \in \{1, ..., |\mathcal{Q}|\} \tag{6.6}$$

6.3 Containerization-Aware Edge Server Updates

This section presents Hermes, a maintenance strategy that leverages characteristics of containerized deployments to reduce latency SLA violations and maintenance time during server updates on Edge Computing infrastructures.

Hermes is designed to perform maintenance according to a batch-based model. Based on that, it divides maintenance work into two groups of activities: server patching (Alg. 5, lines 4–6) and application relocations (Alg. 5, lines 8–20), which are scheduled for separate maintenance batches and occur in turns until all target edge servers have been updated. In this context, application relocations aim to drain (i.e., evacuate) outdated servers so that they can be patched in the next maintenance batch.

Algorithm 5: Hermes maintenance strategy.

```
1  Q ← {}
2  while There are outdated servers in E do
3      Q_q ← New maintenance batch
4      S ← Outdated edge servers in E hosting no applications
5      foreach edge server S_i ∈ S do
6          Update S_i
7      if S is empty then
8          D ← Outdated edge servers sorted by Eq. 6.7 (desc.)
9          T ← {}
10         foreach edge server D_i ∈ D do
11             N ← Applications hosted by D_i sorted by Eq. 6.11 (desc.)
12             Y ← All edge servers except D_i and elements of T
13             if checkCapacity(Y, N, Q_q) = |N| then
14                 foreach application N_j ∈ N do
15                     Y' ← Edge servers in Y sorted by Eq. 6.14 (desc.)
16                     foreach edge server Y'_i ∈ Y' do
17                         if edge server Y'_i has enough free resources to host N_j then
18                             Relocate application N_j to edge server Y'_i
19                             break
20             T ← T ∪ {D_i}
```

The decision-making made by Hermes to patch edge servers is straightforward. After grouping outdated edge servers that host no applications into a list S (Alg. 5, line 4), Hermes iterates over edge servers in S, initiating their patching process (Alg. 5, lines 5–6). At this stage, Hermes does not sort the servers in S due to three assumptions about the employed maintenance model. First, the servers being updated cannot interfere with each other's patching process. Second, the order of elements in S does not affect the speed of patching operations. Third, the subsequent maintenance batch only starts when all servers in S are updated.

In maintenance batches reserved for application relocations, the first decision made by Hermes is to select the order in which outdated edge servers will be drained (Alg. 5, line 8). As edge infrastructures are typically resource-constrained, sorting outdated edge servers can imply selecting which servers will be drained and updated in the subsequent maintenance batch and which outdated edge servers will only be drained in later maintenance batches. As such, defining the order in which outdated edge servers are drained affects the entire maintenance workflow.

The order in which outdated servers are drained is determined according to a score function $\aleph(\mathcal{E}_i, \mathcal{Q}_q)$, presented in Eq. 6.7, which favors outdated edge servers that predominantly present three characteristics. First, Hermes prioritizes edge servers with larger capacity to ensure early availability of larger updated computational capacity to host applications (Eq. 6.8). Given that we consider a multidimensional representation of edge server capacity with values on different scales (i.e., while CPU is represented by the number of cores,

RAM and disk are represented in gigabytes), the overall edge server capacity is normalized using geometric mean. Second, Hermes prioritizes edge servers that host applications with smaller state sizes based on smaller container images, assuming that draining these servers would be faster given the lower network traffic necessary to provision their applications on alternative hosts (Eq. 6.9). Third, Hermes prioritizes edge servers that host applications with less strict delay SLAs (more specifically, the average latency SLA of hosted applications), as in the early stages of maintenance, there are fewer updated resources and, consequently, lower chances of ensuring that all applications on the edge server being drained could be relocated to updated edge servers near their users (Eq. 6.10).

$$\aleph(\mathcal{E}_i, \mathcal{Q}_q) = norm(\aleph_{cap}(\mathcal{E}_i)) + norm(\aleph_{img}(\mathcal{E}_i, \mathcal{Q}_q)) + norm(\aleph_{sla}(\mathcal{E}_i, \mathcal{Q}_q)) \tag{6.7}$$

$$\aleph_{cap}(\mathcal{E}_i) = \sqrt[3]{c_i \cdot r_i \cdot d_i} \tag{6.8}$$

$$\aleph_{img}(\mathcal{E}_i, \mathcal{Q}_q) = \frac{1}{max\left(1,\ \sum_{j=1}^{|\mathcal{A}|}\left(s_j + \sum_{\mathcal{L}_k \in x_j} l_k\right) \cdot \alpha(\mathcal{E}_i, \mathcal{A}_j, \mathcal{Q}_q)\right)} \tag{6.9}$$

$$\aleph_{sla}(\mathcal{E}_i, \mathcal{Q}_q) = \frac{\sum_{j=1}^{|\mathcal{A}|} z_j \cdot \alpha(\mathcal{E}_i, \mathcal{A}_j, \mathcal{Q}_q)}{max\left(1, \sum_{j=1}^{|\mathcal{A}|} \alpha(\mathcal{E}_i, \mathcal{A}_j, \mathcal{Q}_q)\right)} \tag{6.10}$$

Hermes iterates over the ordered list of outdated edge servers to define their draining plans (Alg. 5, lines 10-20). At this stage, Hermes sorts the applications hosted by each edge server to be drained according to a function $\beth(\mathcal{A}_j)$, presented in Eq. 6.11, to define the relocation order (Alg. 5, line 11). The application sorting function considers two criteria. The first criterion prioritizes applications with more strict SLAs (Eq. 6.12). This criterion aims to prevent applications with more flexible SLAs from occupying resources that could be used to avoid SLA violations of applications with stricter SLAs. The second sorting criterion favors applications with lower CPU and RAM demand to promote better use of updated resources by consolidating applications into a reduced set of updated edge servers (Eq. 6.13).

$$\beth(\mathcal{A}_j) = norm(\beth_{sla}(\mathcal{A}_j)) + norm(\beth_{dem}(\mathcal{A}_j)) \tag{6.11}$$

$$\beth_{sla}(\mathcal{A}_j) = \frac{1}{z_j} \tag{6.12}$$

$$\beth_{dem}(\mathcal{A}_j) = \sqrt{m_j \cdot n_j} \tag{6.13}$$

Before starting relocating applications, Hermes uses a checking function, described in Alg. 6, to verify if all applications of the outdated edge server to be drained could be provisioned in alternative hosts (Alg. 5, line 13). If so, Hermes proceeds with drainning the outdated edge server. Otherwise, Hermes moves to the next host in the list of outdated edge servers to be drained without performing any relocations. This checking procedure avoids unnecessarily extending the maintenance batch duration due to relocations that are insufficient to evacuate a given outdated edge server.

Algorithm 6: Hermes edge server capacity checking method.

```
1  Function checkCapacity(Y, N, Q_q):
2      Y' ← List of edge servers in Y
3      N' ← List of applications in N
4      ϰ ← 0
5      foreach N'_j ∈ N' do
6          foreach Y'_i ∈ Y' do
7              if edge server Y'_i has enough free resources to host application N'_j then
8                  Host application N'_j on edge server Y'_i
9                  ϰ ← ϰ + 1
10                 break
11     return ϰ
```

If Hermes determines that it is feasible to drain a server in the current maintenance batch, it begins the draining procedure by iterating over the server's application list. While choosing a new host for an application $\mathcal{A}_j$ that needs to be relocated, Hermes performs a sorting procedure (Alg. 5, line 15) that prioritizes updated hosts to speed up maintenance progression and safeguard applications when patches fix vulnerability issues. As a tie-breaking measure, each candidate edge server to host $\mathcal{A}_j$ is sorted according to a function $\Upsilon(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q)$ (Eq. 6.14), which favors candidate edge servers that predominantly show three characteristics.

The first tie-breaking criterion in $\Upsilon(\mathcal{A}_j, \mathcal{E}_i)$ observes the network distance between the candidate edge servers and the application's user, prioritizing edge servers close enough to the application's user to avoid an SLA violation (Eq. 6.15). The second criterion favors edge servers that possess a larger amount of the container layers that compose the application's container image (we look into each edge server's downloaded layers list and their download and waiting queues), as they are likely to be able to provision the application faster (Eq. 6.16). The third criterion prioritizes edge servers with shorter download and waiting queues (Eq. 6.17), with the aim of preventing long waiting times during application relocation due to network contention (server's download queue) and download limit constraints imposed by container runtimes (server's waiting queue).

$$\Upsilon(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q) = \Upsilon_{sla}(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q) + norm(\Upsilon_{img}(\mathcal{A}_j, \mathcal{E}_i)) + norm(\Upsilon_{queues}(\mathcal{E}_i)) \tag{6.14}$$

$$\Upsilon_{sla}(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q) = \frac{1}{\beta(\mathcal{A}_j, \mathcal{E}_i, \mathcal{Q}_q)} \tag{6.15}$$

$$\Upsilon_{img}(\mathcal{A}_j, \mathcal{E}_i) = \sum_{\mathcal{L}_k \in x_j} \begin{cases} l_k & \text{if } l_k \in \delta(\mathcal{E}_i) \vee l_k \in \partial(\mathcal{E}_i) \vee l_k \in \lambda(\mathcal{E}_i) \\ 0 & \text{otherwise.} \end{cases} \tag{6.16}$$

$$\Upsilon_{queues}(\mathcal{E}_i) = \frac{1}{|\partial(\mathcal{E}_i)| + |\lambda(\mathcal{E}_i)|} \tag{6.17}$$

Finally, Hermes iterates on the ordered list of candidate servers, relocating applications to the first edge server with enough available capacity to host the application (Alg. 5, lines 16–19). The workflow of edge server patching and application relocations is repeated until all edge servers are updated.

6.4 Performance Evaluation

This section details the experiments carried out to demonstrate the effectiveness of Hermes in reducing maintenance time and latency SLA violations during server updates on Edge Computing infrastructures. First, Section 6.4.1 describes the dataset, baseline strategies, and evaluated metrics. Then, Section 6.4.2 presents a sensitivity analysis used to fine-tune the parameters of one of the baseline strategies. Finally, Section 6.4.3 discusses the results obtained.

6.4.1 Experiments Description

The maintenance scenario considered involves updating 25 heterogeneous edge servers, whose specifications are presented in Table 6.2. Although we could not find public storage specifications, CPU and RAM specifications are based on real servers [99]. Although capacity specifications are evenly distributed among the 25 edge servers, we assume that all edge servers take 120 seconds to be updated. The edge servers are interconnected by 169 base stations equipped with network switches, forming a partially-connected mesh topology distributed across a 13x13 hexagonal grid map. The base station communication is done through links with 100 Mbps of bandwidth and 5 ms of latency.

Table 6.2 – Edge server capacity specifications [99].

Model	**CPU Cores**	**RAM**	**Disk**
Dell PowerEdge R620	16	24 GB	128 GB
SGI Rackable C2112-4G10	32	32 GB	128 GB

Edge servers host 60 containerized applications, each accessed by a single user. We assume that container images are pulled into the edge infrastructure from a centralized container registry located in the cloud. In this setup, network links with 40 Mbps of bandwidth connect edge servers and the container registry to simulate network congestion between the core network and the edge infrastructure, which is consistent with data provided by previous studies, such as Ye et al. [173]. Since the positioning of elements within the infrastructure influences the application latency, edge servers and users are positioned at random positions on the map. In addition, applications are initially provisioned on random edge servers positioned close enough to their users so as not to violate their SLAs.

The 60 applications in our dataset are based on uniformly distributed specifications for demand, SLAs, and container images. As we did not find publicly available specifications for container capacity limits, we assume five demand specifications (Table 6.3). As for latency SLAs, we use specifications from 3GPP[2], representing remote surgery (15 ms) and collaborative gaming (20 ms) applications, which are potential Edge Computing use cases.

Table 6.3 – Application demand specifications.

Specification	CPU Cores	RAM
Tiny	1	1 GB
Small	2	2 GB
Medium	4	4 GB
Large	6	6 GB
Extra Large	8	8 GB

Applications are built based on 10 of the 150 most popular DockerHub[3] container images. These container images, whose specifications are shown in Table 6.4, were selected based on the following observations. First, they form the backbone for many real-world software use cases, including web applications, databases, and data processing systems. Second, the selected container images collectively form a distribution where one-third of the container layers are shared across multiple container images, which, according to Fu et al. [46], is a baseline behavior on large-scale containerized deployments. Without loss of generality, we assume that the generic application images are the only stateful, i.e., with state sizes greater than zero (*Flink* = 100 and *Couchbase* = 200).

Our experiments are carried out in EdgeSimPy [148], detailed in Chapter 4. In addition to incorporating functional abstractions for the various components that compose the edge infrastructure (e.g., edge servers, network switches, and base stations), EdgeSimPy provides a fine-grained model of the provision of containerized applications, which is key for representing the addressed scenario. We set EdgeSimPy's tick rate to 1 s and employ the Max-Min Fairness algorithm [18] to determine bandwidth shares for concurrent network flows generated by application relocations performed during maintenance.

[2]https://www.etsi.org/deliver/etsi_ts/122200_122299/122261/16.16.00_60/ts_122261v161600p.pdf
[3]https://hub.docker.com/

Table 6.4 – Container image specifications.

Category	Name	# of Layers	Size of Unique Layers	Size of Shared Layers	Avg. Layer Size
Operating Systems	Debian	1	0 MB	47.2559 MB	47.2559 MB
	CentOS	1	79.6491 MB	0 MB	79.6491 MB
	Ubuntu	1	28.1653 MB	0 MB	28.1653 MB
	Fedora	1	65.1558 MB	0 MB	65.1558 MB
Language Runtimes	Python	7	27.8841 MB	332.543 MB	51.4896 MB
	Perl	5	14.9323 MB	332.543 MB	69.495 MB
	Erlang	5	0 MB	544.541 MB	109.095 MB
	Elixir	6	6.04197 MB	544.541 MB	91.9191 MB
Generic Applications	Flink	5	537.725 MB	0 MB	107.718 MB
	CouchBase	3	632.935 MB	0 MB	210.98 MB

To the best of our knowledge, no maintenance strategy before Hermes focused on reducing the maintenance time at the edge by optimizing the relocation of containerized applications while reducing latency SLA violations. Therefore, our evaluation compares Hermes with three strategies, Lamp (Chapter 5), Salus [147], and Greedy Least Batch (GLB) [177], which partially optimize the objectives addressed in our work. Although Lamp is a maintenance strategy described earlier to reduce latency SLA violations during server updates at the edge (see Section 5.3.1), Salus and GLB are maintenance strategies designed to update servers as fast as possible in cloud data centers (i.e., they overlook the impact of relocations on application latency).

As none of the baseline strategies chosen from the literature reduces maintenance time through optimized relocations of containerized applications, our comparison also considers a metaheuristic called Non-Dominated Sorting Genetic Algorithm (NSGA-II) [35], configured to find Pareto-optimal application relocation plans for each maintenance batch where there are no outdated edge servers ready to be patched (i.e., hosting no application) according to the objective function described in Section 6.2. Although this approach does not guarantee the optimal solution since it iteratively assembles the overall solution batch by batch, it provides a good baseline for the other evaluated heuristic strategies (i.e., Lamp, GLB, Salus, and Hermes).

As we seek to follow the principles of reproducible research and open science, companion materials are publicly available on GitHub[4], including the dataset, source code, and instructions to reproduce our results.

6.4.2 Sensitivity Analysis

The NSGA-II algorithm employed uses a predefined number of generations as stopping criterion. Given that the ideal number of generations may vary according to the evaluated scenario, we performed a sensitivity analysis to determine the optimal choice for this

[4]https://github.com/paulosevero/hermes/

parameter. Without loss of generality, we define the population size as 300, the mutation probability as 10%, and the crossover probability as 100%. The algorithm was configured with the Uniform Crossover and Polynomial Mutation methods. The initial population was set using a Random-Fit Algorithm defined by Souza et al. [146]. Although the used Random-Fit algorithm does not make maintenance-specific decisions, it ensures that the initial population comprises only valid solutions (in our case, relocation plans compliant with Eq. 6.5 and Eq. 6.6, i.e., which do not provision an application on multiple hosts simultaneously and respect the capacity limits of edge servers).

We assess 30 number of generations, ranging from 100 to 3000, increasing in increments of 100. Each combination of parameters was evaluated using the objective function defined in Section 6.2, which seeks to minimize the sum between the normalized maintenance time and the normalized number of SLA violations achieved by the solutions. As indicated in Figure 6.2, the NSGA-II algorithm achieved the best results with 1700 and 1800 generations. Consequently, we used 1700 generations in the evaluation of Section 6.4.3.

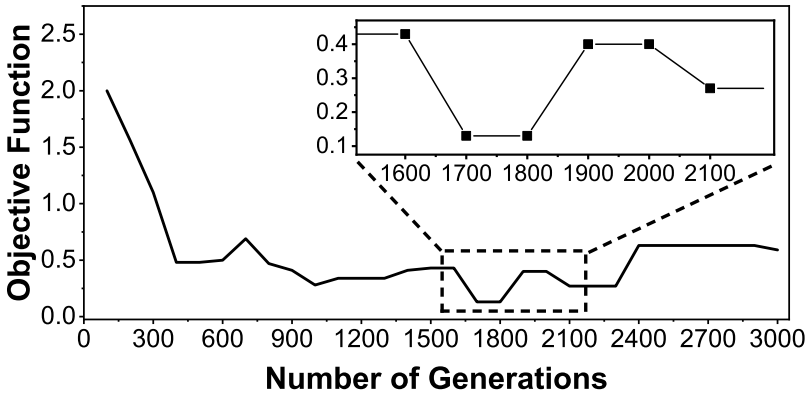


Figure 6.2 – NSGA-II sensitivity analysis.

6.4.3 Comparison with Baseline Algorithms

Execution Time. Figure 6.3 presents the execution time (in seconds) of the evaluated algorithms. Results are displayed on a log2-scale to facilitate visualization of values with varying magnitudes. Overall, the heuristic algorithms (e.g., GLB, Salus, Lamp, and Hermes) exhibited similar execution times. In contrast, NSGA-II took, on average, 6561 times longer to execute than the other strategies. NSGA-II's longer execution time is due to the number of operations it had to perform to find its final outcome. More specifically, for each maintenance batch where application relocations were needed, NSGA-II had to execute simulations with multiple action plans in addition to performing selection, crossover, and mutation operations until Pareto-optimal solutions were found.

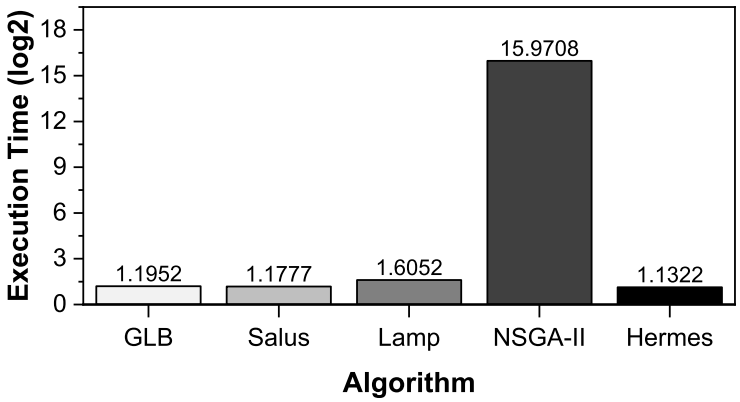
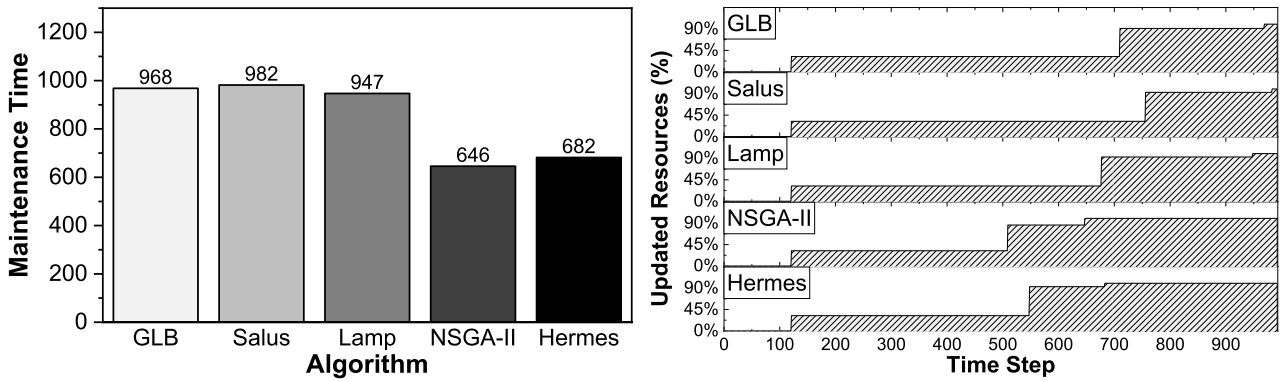


Figure 6.3 – Execution time analysis.

The difference in execution time between NSGA-II and the other strategies provides a strong rationale for the use of heuristics in the addressed scenario. While the evaluated heuristics had slightly inferior results compared to NSGA-II regarding the evaluated metrics (as will be discussed in more detail in this section), they compensated for these losses with a significantly reduced execution time. Although not relevant in some situations, reduced execution time may be a critical requirement when maintenance plans must be devised within a short time frame (e.g., when patches to fix security vulnerabilities must be applied).

Maintenance Time. Figure 6.4(a) displays the results concerning the time required by the evaluated strategies to update all edge servers within the infrastructure. Hermes achieved a result only 5.57% worse than NSGA-II, being able to reduce the maintenance time 29.38% on average compared to GLB, Salus, and Lamp, which showed very similar results. Given that evaluated strategies adhere to a rolling upgrade model, excessive delays in specific maintenance batches can conceal fast progress in the rest of the maintenance work if we look only at the overall maintenance time. Therefore, our maintenance time evaluation also includes an analysis of how fast the evaluated strategies manage to update the computational resources within the infrastructure (Figure 6.4(b)).



(a) Overall maintenance time

(b) Maintenance progression over time

Figure 6.4 – Maintenance time results.

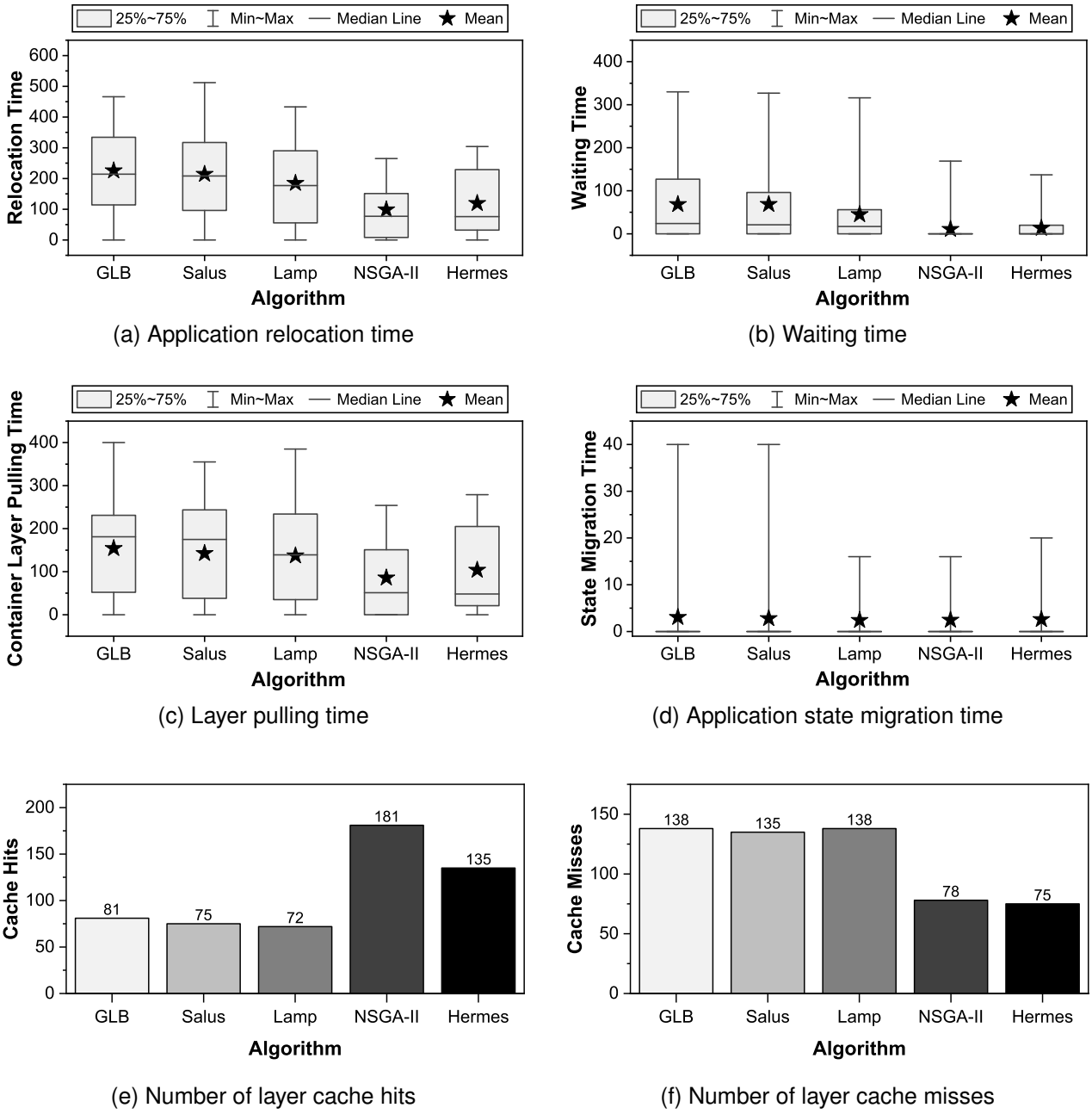
Figure 6.4(b) shows the maintenance progression over time, allowing observation of the speed with which edge servers are updated. Since the patching time for the edge servers is equal to all strategies, Figure 6.4(b) allows us to identify the impact of relocation decisions on the maintenance progression and, consequently, on the overall maintenance time. GLB, Salus, and Lamp spent significantly more time relocating applications than Hermes and NSGA-II, which on average could update 54% of the infrastructure resources 166 time steps faster than GLB, Salus, and Lamp. The prolonged period in which most of the infrastructure is outdated is alarming, especially when outdated resources pose vulnerabilities to the infrastructure (e.g., when hosts can be used to spread malware to their peers).

Figure 6.5 presents a more in-depth analysis of the application relocations conducted by the evaluated strategies. As we can observe in Figure 6.5(a), all strategies exhibited highly variable relocation times. This occurred as the analyzed scenario includes applications with container images with highly varied sizes, making some relocations inevitably longer, regardless of the effectiveness of specific strategy decisions.

In general, NSGA-II and Hermes were more effective during application relocations, reducing the average relocation time by 47.62% compared to GLB, Salus, and Lamp. The primary reason for the reduced relocation time achieved by NSGA-II and Hermes is their ability to take advantage of the shared content of container images to accelerate application provisioning. As NSGA-II is a metaheuristic and, therefore, does not make fixed decisions to achieve its objectives, the following relocation time analysis discusses which decisions Hermes took allowed it to achieve such gains compared to GLB, Salus, and Lamp.

Figure 6.5(b) presents waiting time results, which account for the period in which the container layers that compose the container images used by the applications being relocated need to wait before being downloaded due to restrictions on the number of simultaneous downloads imposed by the container runtimes. As we can see, Hermes reduced the average waiting time for application relocations by 78.61% compared to GLB, Salus, and Lamp. This was possible because when choosing candidate hosts for applications that need to be relocated, in addition to prioritizing updated edge servers to progress maintenance more quickly, Hermes adopts a set of sorting criteria that includes favoring those updated edge servers with smaller download and waiting queues (Alg. 5, line 15).

Figure 6.5(c) presents the pulling time results, which account for the period in which the container layers are downloaded from the container registry. Hermes reduced the average download time of container layers by 28.27% compared to GLB, Salus, and Lamp, as it focuses on relocating applications to edge servers with a larger amount of container layers that form the container images of the applications (Alg. 5, line 15). As a result, application relocations performed by Hermes required the download of fewer container layers, as shown in Figures 6.5(e) and 6.5(f), which present the number of container layer cache hits and misses incurred from relocations performed during maintenance. In this context, while



(a) Application relocation time

(b) Waiting time

(c) Layer pulling time

(d) Application state migration time

(e) Number of layer cache hits

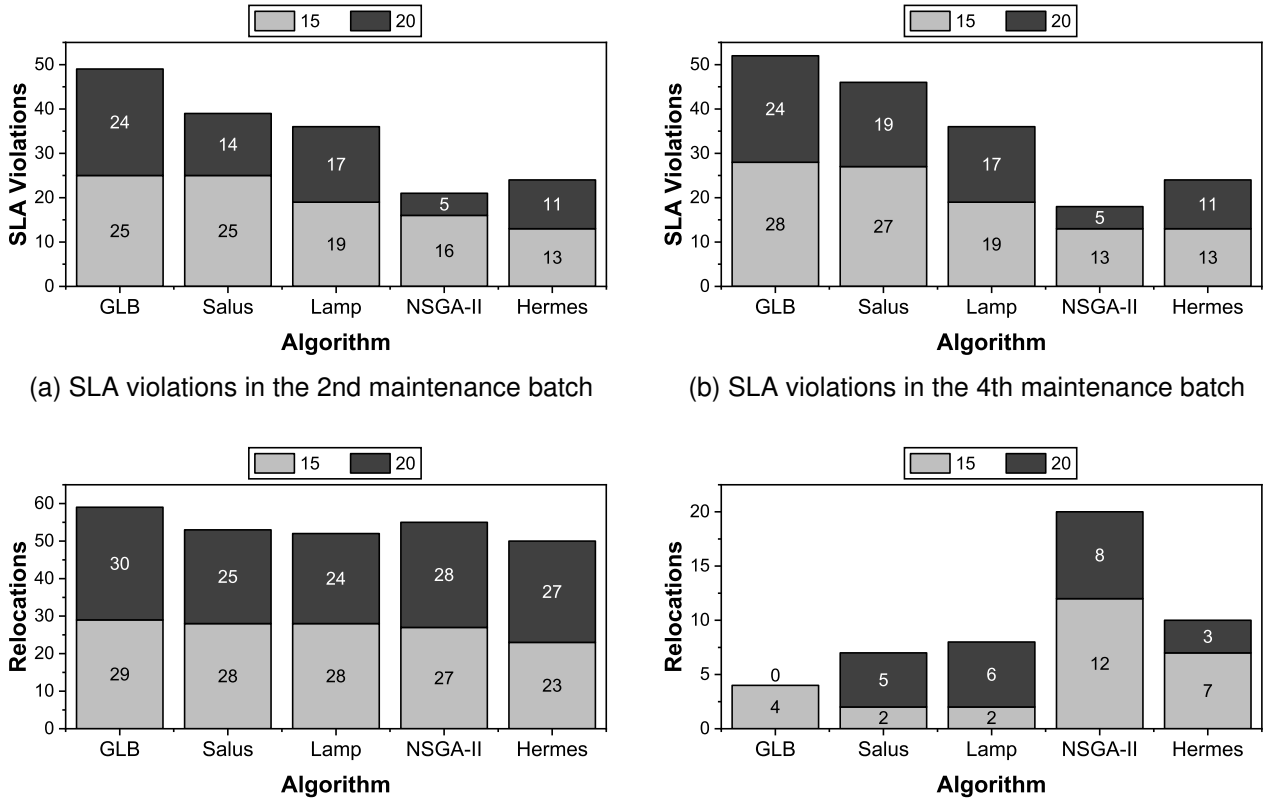
(f) Number of layer cache misses

Figure 6.5 – Application relocation time results.

cache hits account for the layers that were already present at their target hosts and did not have to be downloaded from the container registry, cache misses mean the opposite.

Figure 6.5(d) presents the state migration time results. Among the application relocation stages, the state migration process was the one that showed the least differences between the evaluated strategies. The smaller difference in the state migration results occurred for some reasons. First, only Flink and CouchBase instances were stateful, accounting for only 12 of the 60 applications, meaning most relocations did not require state migration. Second, unlike container layers, state data is not shared among applications, so none of the maintenance strategies makes specific decisions to reduce the state migration time.

Latency SLA Violations. Figure 6.6 presents the results related to the number of latency SLA violations incurred by the assessed strategies. Although being 22% worse than NSGA-II, Hermes achieved an average reduction of 43% in the number of SLA violations compared to the other heuristics evaluated. As expected, GLB and Salus showed the worst results regarding SLA violations, given their focus on maintenance in cloud data centers, where the impact of relocations on application latency is not considered, given the robust network capabilities available to all cloud hosts.



(a) SLA violations in the 2nd maintenance batch

(b) SLA violations in the 4th maintenance batch

(c) Relocations in the 2nd maintenance batch

(d) Relocations in the 4th maintenance batch

Figure 6.6 – Latency SLA violations and number of relocations per SLA (15 ms and 20 ms).

It should be noted that, in the adopted conceptual model, the two operations conducted during maintenance (i.e., edge server patching and application relocations) are separated into distinct maintenance batches and take turns until maintenance completion. In this context, the compared strategies make only different decisions during the maintenance batches reserved for application relocations. Based on this, our analysis of SLA violations will focus solely on the second and fourth maintenance batches, where application relocations took place, as in the maintenance batches reserved for edge server patching, the number of SLA violations did not change compared to their previous relocation batch.

The reduction in the number of SLA violations achieved by Hermes was achieved through two main decisions. First, when setting the draining order of the outdated edge servers, Hermes prioritizes those hosting applications with less strict SLAs (Alg. 5, line 8). This ordering criterion is based on the rationale that, especially at the beginning of main-

tenance, there may be fewer updated resources available in proximity to users and, therefore, the chances of SLA violations occurring are higher. We can observe the impact of this decision by looking at the number of application relocations performed in each maintenance batch, displayed in Figures 6.6(c) and 6.6(d). Compared to other heuristic strategies, Hermes relocated 18.82% fewer applications with the most strict SLA specification at the beginning of maintenance (i.e., the second batch of maintenance), achieving an average reduction of 43.48% in the number of SLA violations for these applications compared to all baseline heuristics and a reduction of 31.58% compared to Lamp, which is also designed for maintenance at the edge but does not include application SLAs when deciding the order of edge server draining and application relocations.

Even if postponed, relocating applications with more strict SLAs is necessary for their hosts to be drained and updated. At this stage, the second key decision made by Hermes that contributed to reducing SLA violations comes into play. When Hermes decides to drain a particular edge server, it first relocates applications with more strict SLAs to prevent updated resources close to such applications from being unnecessarily occupied by applications with less strict SLAs (Alg. 5, line 11). Based on such a decision, Hermes also reduced SLA violations by 46.27% on average during the second round of relocations during maintenance (i.e., the fourth batch of maintenance), as shown in Figure 6.6(b).

6.5 Closing Remarks

Reducing maintenance time during server updates has been the target of several research efforts [177] [147] [174] [149]. Motivations to reduce maintenance time are numerous. For instance, server updates often relate to critical aspects such as security vulnerability mitigation. In addition, maintenance work usually requires application relocations, which increase the load on the infrastructure and, depending on application characteristics, can cause service disruption, affecting the quality of service offered to end-users.

Despite existing efforts, during our survey on existing maintenance research, we discussed that existing maintenance strategies do not take advantage of the shared content of images from containerized applications, missing optimization opportunities that could reduce maintenance time (see Section 3.5). Based on such observations, this chapter presented a novel maintenance strategy called Hermes, which extends our work on edge server updates described in Chapter 5, reducing the maintenance time during server updates on edge infrastructures by leveraging the shared content of containerized application images during relocations carried out during maintenance work. Our experiments show that Hermes reduces maintenance time by 29.38% on average compared to baseline strategies.

The work presented in this chapter concludes the set of contributions of this thesis. First, Chapter 4 introduced a simulation toolkit with support for edge maintenance oper-

ations. Then, Chapter 5 and this chapter presented maintenance strategies that reduce maintenance time during edge server updates while alleviating the impact of maintenance work on the performance of edge applications. These contributions address the research challenges identified by our review of the literature (Section 3.5).

The following chapter presents our final remarks, including a discussion on how our contributions answer the Research Questions established in the introduction and a list of future research directions.

7. CONCLUSIONS AND FUTURE DIRECTIONS

This chapter presents the final considerations of this thesis. First, it recapitulates the motivation for this work, answers the research questions defined in Chapter 1, and summarizes the main contributions (Section 7.1). Then, it concludes by listing future research directions (Section 7.2).

7.1 Overview and Discussion

Advances in telecommunications and the emergence of the Internet of Things have introduced new classes of mobile and sensor-rich applications with latency and bandwidth requirements that could not be satisfied by the traditional Cloud Computing model, given the significant distance between data sources and cloud data centers. The evident limitations of the cloud's centralized model led to the rise of a new computing paradigm called Edge Computing, where computing devices, often called edge servers, are dispersed in strategic positions at the Internet's edge, in close proximity to end devices.

Whereas decentralizing the infrastructure helped mitigate congestion at specific high-traffic zones, it introduced substantial technical challenges related to IT operations. As edge servers are often deployed in small-scale facilities with limited cooling and power supply, they are inherently susceptible to various hardware issues. In addition, the typical edge infrastructure's lack of robustness makes room for a broad spectrum of security threats, including network attacks and physical tampering, especially when edge servers are located in outdoor facilities. In this context, maintenance strategies are critical in ensuring that performance and security concerns do not undermine the edge's inherent advantages. This thesis has focused on optimizing edge maintenance operations by coordinating maintenance requirements (e.g., reducing component update times) with the performance demands of edge applications and the resource constraints of edge infrastructures.

The central hypothesis defined in this thesis was based on the rationale that maintenance strategies should be optimized according to the unique characteristics of edge infrastructures. We assumed that improved decisions such as "when" to update components and "how" to relocate applications to advance maintenance could reduce maintenance time while keeping the impact on application performance to a minimum. To organize our research work, we divided the defined hypothesis into four research questions. Based on the research work detailed throughout the chapters of this thesis, the answers to the established research questions are the following:

- ***Research Question 1:*** *What are the main approaches and metrics of interest in the context of maintenance on Edge Computing infrastructures?*

  - Maintenance operations comprise a broad spectrum of activities depending on the scenario at hand. Research efforts generally focus on correcting or preventing critical issues, such as security vulnerabilities and infrastructure failures. In such instances, maintenance follows the rolling upgrade model, progressing gradually. When the target components are edge servers, affected applications are relocated to alternative hosts to avoid downtime. In addition to directly influencing maintenance progression (as servers must be evacuated before being updated), relocation decisions can also impact application performance. As such, we observe that several studies evaluate the effectiveness of maintenance strategies according to their impact on the quality of service exhibited by the applications. Alternatively, some authors consider resource efficiency metrics during their evaluations, acknowledging that maintenance operations can impose significant stress on edge infrastructures.

- ***Research Question 2:*** *How can we evaluate research prototypes of maintenance strategies for Edge Computing infrastructures?*

  - We have observed that existing maintenance research efforts employ various validation approaches depending on the addressed scenario. In general, simulation is the preferred option, as it allows for cost-effective evaluations of research prototypes in large-scale environments. Moreover, we noticed that most simulated evaluations use custom simulators, highlighting the lack of a widely-accepted maintenance simulation toolkit. Given this context, we developed EdgeSimPy, a Python-based simulation toolkit designed for modeling and evaluating resource management policies, including maintenance of physical resources (e.g., edge servers and network devices) and applications on edge infrastructures. Unlike existing simulators, EdgeSimPy integrates several functional abstractions and a novel conceptual model that accurately represents the lifecycle of edge applications and ensures seamless integration with real application traces.

- ***Research Question 3:*** *What is the impact of location-aware application relocations during maintenance on Edge Computing infrastructures?*

  - While edge infrastructures typically comprise distributed hosts often interconnected via public networks without mandatory redundancy and performance guarantees, edge applications have tight latency requirements. Consequently, the quality of service exhibited by applications can be severely degraded if they are moved to edge servers too far from their users. In this context, we identified that the absence of location awareness in application relocations is one of the main factors that prevent the use of maintenance strategies designed for the cloud during the update of edge components. In response, we proposed two maintenance strategies, Lamp and Laxus, that incorporate user location awareness into application

relocation decisions conducted during edge server maintenance. The results indicated that Lamp and Laxus could reduce latency issues by an average of 65.5% compared to existing maintenance strategies designed for cloud data centers.

- ***Research Question 4:*** *How can we leverage characteristics of edge applications to reduce maintenance time during edge server updates?*
  - Most research efforts on server maintenance assume that servers must be rebooted for updates to take effect. In this context, the shutdown of applications, even temporarily, is avoided to preserve the quality of service delivered to end-users, so servers must be evacuated through relocation techniques before undergoing maintenance. As a result, relocation decisions become a significant factor influencing total maintenance time. During our literature review, we observed that existing maintenance strategies employed migration strategies designed for VM-based applications, overlooking the emergence of containerization as the leading architecture for deploying edge applications and consequently missing the relocation optimization opportunities offered by containerization. To fill this gap, we designed Hermes, a maintenance strategy that capitalizes on the shared content of container images to reduce maintenance time through optimized application relocations. Our experiments showed that Hermes can reduce maintenance time by 29.38% on average compared to baseline strategies that perform application relocations based on the VM model.

Based on the research work presented in this thesis, our conclusion reinforces our hypothesis that maintenance strategies that implement intelligent decisions based on the performance requirements and characteristics of edge applications can achieve significant reductions in edge component update times, while reducing the impact of maintenance work on application performance.

In summary, this thesis has made the following contributions:

- A taxonomy that provides a systematic organization of maintenance research targeting Edge Computing environments and two related paradigms (Cloud Computing and the Internet of Things) based on various characteristics, including maintenance approaches, techniques used, and metrics of interest (Chapter 3).
- A simulation toolkit that implements several functional abstractions for edge infrastructure components and incorporates a conceptual model that replicates the provisioning method of widely used containerization platforms, supporting several use cases, including the maintenance of computing and network devices at the edge (Chapter 4).
- Three maintenance strategies that reduce application latency and maintenance time during edge server updates by integrating user location and containerization awareness into the maintenance decision-making process (Chapters 5–6).

7.2 Future Directions

This thesis has addressed several challenges related to coordinating maintenance operations on Edge Computing infrastructures. Nevertheless, we observe that many other key challenges require further investigation.

7.2.1 Optimized Prioritization of Maintenance Decisions

Prioritization policies play a vital role during maintenance operations on large-scale infrastructures, where the order in which events occur can affect the quality of maintenance through cascading phenomena. In such a scenario, scheduling algorithms can determine the order of various decisions (e.g., component updates and application migrations) based on multiple factors, including the criticality of affected components within the infrastructure and the application workload.

There are several use cases where maintenance prioritization policies can enhance the effectiveness of updates on edge infrastructures. For example, server groupings can prioritize edge sites with known vulnerabilities. Additionally, during critical security updates, edge servers from sandbox environments can be assigned a lower priority than production edge servers, as hosts from sandbox environments can be temporarily disabled without affecting critical applications, while delaying the update of production environments can facilitate vulnerability propagation.

Existing maintenance efforts use prioritization policies to support the scheduling of patch distribution [109], application migration [174], and component updates [149]. However, these policies focus primarily on generic goals such as reducing maintenance time, leaving other factors, such as infrastructure security, out of the scheduling decision-making. New robust prioritization policies considering parameters such as the edge infrastructure organization (e.g., sandbox and production environments) and workload characteristics (e.g., how many users and applications depend on affected components) should be developed to enhance the effectiveness of maintenance work. Also, another promising approach would be the design of novel metrics that characterize the impact of prioritization policies on the quality of maintenance decisions.

7.2.2 Inter-Service Communication Awareness

Application composition architectures have been gaining traction within the IT community as loosely coupled services expand the provisioning possibilities compared to tra-

ditional monoliths [92]. One of the advantages of composite applications is the enhanced ability to handle workload fluctuations efficiently. As services from composite applications are deployed independently, infrastructure operators can adjust the computational capacity available to the services based on the specific demand of each application component, which is typically not feasible for monolithic applications, that are deployed as single software units. Given that edge infrastructures can comprise resource-constrained hosts, the ability to break down an application into multiple small components is beneficial as it allows multiple heterogeneous edge servers to cooperate to deliver desirable performance levels.

Despite the benefits of application composition, its distributed nature introduces some challenges that must be considered during maintenance operations, especially since application relocations are often required to advance maintenance. One of the noteworthy challenges in relocating composite applications is the narrowed provisioning options: services from composite applications must be provisioned close to each other to avoid network bottlenecks, especially on edge infrastructures, where poor positioning of services can lead to significant latency increases, given that network instability can occur without prior notice [163]. Although some initiatives consider maintenance scenarios with composite applications [22], the proposed solutions do not focus on implementing optimized provisioning decisions. Therefore, there is a need for more sophisticated edge maintenance strategies that consider the performance requirements of composite applications to mitigate the impact of maintenance work on the quality of service delivered to end users.

7.2.3 Mitigation of Resource Contention

Virtualization is at the core of most modern computing platforms, providing increased provisioning flexibility and fine-grained control of physical resources [97]. One of the main features virtualization provides is multiplexing, which allows multiple applications to run on top of the same physical resources [101]. Whereas multiplexing widens the provisioning options, it introduces performance concerns related to resource contention (also known as performance interference), where co-located applications, especially those that rely on the same type of resource (e.g., CPU cache and I/O), degrade each other performance [169].

Although performance interference is not exclusive to the edge, the resource constraints of the edge servers and the strict performance requirements of edge applications amplify the problem at the edge. The side effects of performance interference at the edge are particularly alarming during maintenance operations, where widely-used approaches such as rolling upgrade strategies, often result in stacking applications on updated hosts as maintenance progresses [149]. Although some maintenance strategies successfully alleviate specific infrastructure issues such as network bottlenecks and application downtime [110] [174], there is a lack of study on the potential performance interference induced

by maintenance decisions. Therefore, more research is needed on interference-aware allocation approaches that can accommodate the demands of maintenance scenarios and alleviate the impact on application performance during maintenance operations.

7.2.4 Energy Consumption Reduction

Despite existing efforts to reduce power consumption during maintenance, existing solutions that address this concern are predominantly focused on IoT environments, acting under the motivation that IoT nodes have limited power supply [22] [109], and overlooking the fact that edge servers can also operate under power supply constraints. In general, existing edge maintenance solutions primarily focus on optimizing maintenance goals (e.g., reducing maintenance time) and reducing the impact of maintenance over application performance, so they are not concerned with ensuring that maintenance work meets sustainability goals.

Although some initiatives have incorporated power-saving considerations into maintenance decision-making [126], they resort to simple approaches like switching off idle edge servers that become idle during maintenance. While deactivating idle hosts during maintenance is effective in cloud data centers due to the abundance of computing resources, it may yield limited benefits in resource-constrained edge infrastructures where the number of idle edge servers during maintenance can be negligible.

Therefore, there is a need for novel energy-aware maintenance strategies tailored to the specific needs and constraints of edge infrastructures to ensure that maintenance work aligns with sustainability goals as closely as possible. Potential research directions include adopting alternative power-saving techniques such as Dynamic Voltage and Frequency Scaling (DVFS), which can leverage fluctuating workloads of edge applications to reduce the energy consumption of edge servers, and combining conventional energy sources (i.e., fossil fuels and nuclear power) with renewable energy (e.g., solar and wind energy).

REFERENCES

[1] Abdelfadeel, K.; Farrell, T.; McDonald, D.; Pesch, D. “How to make firmware updates over lorawan possible”. In: International Symposium on World of Wireless Mobile and Multimedia Networks, 2020, pp. 16–25.

[2] Altman, E.; Jimenez, T. “Ns simulator for beginners”, *Synthesis Lectures on Communication Networks*, vol. 5, January 2012, pp. 1–184.

[3] Alwasel, K.; Jha, D. N.; Habeeb, F.; Demirbaga, U.; Rana, O.; Baker, T.; Dustdar, S.; Villari, M.; James, P.; Solaiman, E.; Ranjan, R. “Iotsim-osmosis: A framework for modeling and simulating iot applications over an edge-cloud continuum”, *Journal of Systems Architecture*, vol. 116, June 2021, pp. 1–13.

[4] Amarasinghe, G.; de Assunção, M. D.; Harwood, A.; Karunasekera, S. “Ecsnet++ : A simulator for distributed stream processing on edge and cloud environments”, *Future Generation Computer Systems*, vol. 111, October 2020, pp. 401–418.

[5] Anastasiou, A.; Christodoulou, P.; Christodoulou, K.; Vassiliou, V.; Zinonos, Z. “Iot device firmware update over lora: The blockchain solution”. In: International Conference on Distributed Computing in Sensor Systems, 2020, pp. 404–411.

[6] Anglano, C.; Canonico, M.; Castagno, P.; Guazzone, M.; Sereno, M. “A game-theoretic approach to coalition formation in fog provider federations”. In: International Conference on Fog and Mobile Edge Computing, 2018, pp. 123–130.

[7] Antonakakis, M.; April, T.; Bailey, M.; Bernhard, M.; Bursztein, E.; Cochran, J.; Durumeric, Z.; Halderman, J. A.; Invernizzi, L.; Kallitsis, M.; et al.. “Understanding the mirai botnet”. In: USENIX Security Symposium, 2017, pp. 1093–1110.

[8] Aral, A.; Brandić, I. “Learning spatiotemporal failure dependencies for resilient edge computing services”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 32–7, December 2020, pp. 1578–1590.

[9] Aral, A.; De Maio, V.; Brandic, I. “Ares: Reliable and sustainable edge provisioning for wireless sensor networks”, *IEEE Transactions on Sustainable Computing*, vol. 7–4, January 2021, pp. 761–773.

[10] Arbabi, M. S.; Shajari, M. “Decentralized and secure delivery network of iot update files based on ethereum smart contracts and blockchain technology”. In: Annual International Conference on Computer Science and Software Engineering, 2019, pp. 110–119.

[11] Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; et al.. “A view of cloud computing”, *Communications of the ACM*, vol. 53–4, April 2010, pp. 50–58.

[12] Asokan, N.; Nyman, T.; Rattanavipanon, N.; Sadeghi, A.-R.; Tsudik, G. “Assured: Architecture for secure software update of realistic embedded devices”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37–11, October 2018, pp. 2290–2300.

[13] Balci, O. “Principles and techniques of simulation validation, verification, and testing”. In: Winter Simulation Conference, 1995, pp. 147–154.

[14] Banikhalaf, M.; Manasrah, A. M.; AlEroud, A. F.; Hamadneh, N.; Qawasmeh, A.; Al-Dubai, A. Y. “A reliable route repairing scheme for internet of vehicles”, *International Journal of Computer Applications in Technology*, vol. 61–3, September 2019, pp. 229–238.

[15] Barabási, A.-L.; Albert, R. “Emergence of scaling in random networks”, *Science*, vol. 286–5439, October 1999, pp. 509–512.

[16] Beloglazov, A.; Buyya, R. “Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers”, *Concurrency and Computation: Practice and Experience*, vol. 24–13, September 2012, pp. 1397–1420.

[17] Benestad, H. C.; Anda, B.; Arisholm, E. “Understanding software maintenance and evolution by analyzing individual changes: a literature review”, *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21–6, September 2009, pp. 349–378.

[18] Bertsekas, D. P.; Gallager, R. G.; Humblet, P. “Data networks”. Prentice-Hall International New Jersey, 1992, vol. 2, 486p.

[19] Blum, C.; Roli, A. “Metaheuristics in combinatorial optimization: Overview and conceptual comparison”, *ACM Computing Surveys*, vol. 35–3, September 2003, pp. 268–308.

[20] Bonabeau, E. “Agent-based modeling: Methods and techniques for simulating human systems”, *Proceedings of the National Academy of Sciences*, vol. 99, May 2002, pp. 7280–7287.

[21] Brereton, P.; Kitchenham, B. A.; Budgen, D.; Turner, M.; Khalil, M. “Lessons from applying the systematic literature review process within the software engineering domain”, *Journal of Systems and Software*, vol. 80–4, April 2007, pp. 571–583.

[22] Bui, N. H.; Nguyen, K. K.; Pham, C.; Cheriet, M. “Energy efficient software update mechanism for networked iot devices”. In: Global Communications Conference, 2019, pp. 1–6.

[23] Buyya, R.; Srirama, S. N.; Casale, G.; Calheiros, R.; Simmhan, Y.; Varghese, B.; Gelenbe, E.; Javadi, B.; Vaquero, L. M.; Netto, M. A.; et al.. “A manifesto for future generation cloud computing: Research directions for the next decade”, *ACM Computing Surveys*, vol. 51–5, November 2018, pp. 1–38.

[24] Calheiros, R. N.; Ranjan, R.; Beloglazov, A.; De Rose, C. A.; Buyya, R. “Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms”, *Software: Practice and Experience*, vol. 41, January 2011, pp. 23–50.

[25] Cerveira, F.; Barbosa, R.; Madeira, H. “Mitigating virtualization failures through migration to a co-located hypervisor”, *IEEE Access*, vol. 9, July 2021, pp. 105255–105269.

[26] Chen, N.; Yang, Y.; Zhang, T.; Zhou, M.-T.; Luo, X.; Zao, J. K. “Fog as a service technology”, *IEEE Communications Magazine*, vol. 56–11, September 2018, pp. 95–101.

[27] Cheng, S.-M.; Chen, P.-Y.; Lin, C.-C.; Hsiao, H.-C. “Traffic-aware patching for cyber security in mobile iot”, *IEEE Communications Magazine*, vol. 55–7, July 2017, pp. 29–35.

[28] Ciardo, G.; Trivedi, K. S. “A decomposition approach for stochastic reward net models”, *Performance Evaluation*, vol. 18, July 1993, pp. 37–59.

[29] Conterato, M. d. S.; Ferreto, T. C.; Rossi, F.; Marques, W. d. S.; de Souza, P. S. S. “Reducing energy consumption in sdn-based data center networks through flow consolidation strategies”. In: Symposium on Applied Computing, 2019, pp. 1384–1391.

[30] Črepinšek, M.; Liu, S.-H.; Mernik, M. “Exploration and exploitation in evolutionary algorithms: A survey”, *ACM Computing Surveys*, vol. 45–3, July 2013, pp. 1–33.

[31] Daly, D.; Deavours, D. D.; Doyle, J. M.; Webster, P. G.; Sanders, W. H. “Möbius: An extensible tool for performance and dependability modeling”. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, 2000, pp. 332–336.

[32] Darrous, J.; Lambert, T.; Ibrahim, S. “On the importance of container image placement for service provisioning in the edge”. In: International Conference on Computer Communication and Networks, 2019, pp. 1–9.

[33] de Assuncao, M. D.; da Silva Veith, A.; Buyya, R. “Distributed data stream processing and edge computing: A survey on resource elasticity and future directions”, *Journal of Network and Computer Applications*, vol. 103, February 2018, pp. 1–17.

[34] Deb, K.; Pratap, A.; Agarwal, S.; Meyarivan, T. “A fast and elitist multiobjective genetic algorithm: Nsga-ii”, *IEEE Transactions on Evolutionary Computation*, vol. 6–2, April 2002, pp. 182–197.

[35] Deb, K.; Pratap, A.; Agarwal, S.; Meyarivan, T. “A fast and elitist multiobjective genetic algorithm: Nsga-ii”, *IEEE Transactions on Evolutionary Computation*, vol. 6–2, April 2002, pp. 182–197.

[36] Dietterich, T. G. “Ensemble methods in machine learning”. In: International Workshop on Multiple Classifier Systems, 2000, pp. 1–15.

[37] Dijkstra, E. W.; et al.. “A note on two problems in connexion with graphs”, *Numerische Mathematik*, vol. 1, December 1959, pp. 269–271.

[38] Dotan, M.; Pignolet, Y.-A.; Schmid, S.; Tochner, S.; Zohar, A. “Survey on blockchain networking: Context, state-of-the-art, challenges”, *ACM Computing Surveys*, vol. 54–5, May 2021, pp. 1–34.

[39] Eiter, T.; Ianni, G.; Krennwallner, T. “Answer set programming: A primer”. In: Reasoning Web International Summer School, 2009, pp. 40–110.

[40] El-Rewini, H.; Ali, H. H.; Lewis, T. “Task scheduling in multiprocessing systems”, *Computer*, vol. 28–12, December 1995, pp. 27–37.

[41] Elijah, O.; Rahman, T. A.; Orikumhi, I.; Leow, C. Y.; Hindia, M. N. “An overview of internet of things (iot) and data analytics in agriculture: Benefits and challenges”, *IEEE Internet of Things Journal*, vol. 5–5, October 2018, pp. 3758–3773.

[42] Fadaeefath Abadi, M.; Haghighat, F.; Nasiri, F. “Data center maintenance: applications and future research directions”, *Facilities*, vol. 38–9/10, April 2020, pp. 691–714.

[43] Fakhrolmobasheri, S.; Ataie, E.; Movaghar, A. “Modeling and evaluation of power-aware software rejuvenation in cloud systems”, *Algorithms*, vol. 11–10, October 2018, pp. 1–15.

[44] Fan, Y.-H.; Wang, M.-Q.; Li, Y.-B.; Hu, K.; Li, M.-Z. “A secure iot firmware update scheme against scpa and dos attacks”, *Journal of Computer Science and Technology*, vol. 36–2, March 2021, pp. 419–433.

[45] Faticanti, F.; De Pellegrini, F.; Siracusa, D.; Santoro, D.; Cretti, S. “Throughput-aware partitioning and placement of applications in fog computing”, *IEEE Transactions on Network and Service Management*, vol. 17–4, September 2020, pp. 2436–2450.

[46] Fu, S.; Mittal, R.; Zhang, L.; Ratnasamy, S. “Fast and efficient container startup at the edge via dependency scheduling”. In: Workshop on Hot Topics in Edge Computing, 2020, pp. 1–7.

[47] Fukuda, T.; Omote, K. “Efficient blockchain-based iot firmware update considering distribution incentives”. In: Conference on Dependable and Secure Computing, 2021, pp. 1–8.

[48] Galante, G.; de Bona, L. C. E. “A survey on cloud computing elasticity”. In: International Conference on Utility and Cloud Computing, 2012, pp. 263–270.

[49] Garg, S.; Kaur, K.; Kumar, N.; Kaddoum, G.; Zomaya, A. Y.; Ranjan, R. “A hybrid deep learning-based model for anomaly detection in cloud datacenter networks”, *IEEE Transactions on Network and Service Management*, vol. 16–3, July 2019, pp. 924–935.

[50] Gebali, F. “Analysis of computer and communication networks”. Springer Science & Business Media, 2008, 669p.

[51] Gebser, M.; Kaminski, R.; Kaufmann, B.; Schaub, T. “Multi-shot asp solving with clingo”, *Theory and Practice of Logic Programming*, vol. 19, January 2019, pp. 27–82.

[52] German, R.; Kelling, C.; Zimmermann, A.; Hommel, G. “Timenet: a toolkit for evaluating non-markovian stochastic petri nets”, *Performance Evaluation*, vol. 24–1-2, November 1995, pp. 69–87.

[53] Gubbi, J.; Buyya, R.; Marusic, S.; Palaniswami, M. “Internet of things (iot): A vision, architectural elements, and future directions”, *Future Generation Computer Systems*, vol. 29–7, September 2013, pp. 1645–1660.

[54] Gupta, H.; Van Oorschot, P. C. “Onboarding and software update architecture for iot devices”. In: International Conference on Privacy, Security and Trust, 2019, pp. 1–11.

[55] Ha, K.; Abe, Y.; Eiszler, T.; Chen, Z.; Hu, W.; Amos, B.; Upadhyaya, R.; Pillai, P.; Satyanarayanan, M. “You can teach elephants to dance: Agile vm handoff for edge computing”. In: Symposium on Edge Computing, 2017, pp. 1–14.

[56] Hagberg, A.; Swart, P.; S Chult, D. “Exploring network structure, dynamics, and function using networkx”, Technical Report, Los Alamos National Laboratory, 2008, 5p.

[57] Halabi, S. “Hyperconverged Infrastructure Data Centers: Demystifying HCI”. Cisco Press, 2019, 544p.

[58] Han, B.; Gopalakrishnan, V.; Ji, L.; Lee, S. “Network function virtualization: Challenges and opportunities for innovations”, *IEEE Communications Magazine*, vol. 53–2, February 2015, pp. 90–97.

[59] Han, J.; Pei, J.; Kamber, M. “Data mining: concepts and techniques”. Elsevier, 2011, 703p.

[60] He, J.; Dai, T.; Gu, X.; Jin, G. “Hangfix: automatically fixing software hang bugs for production cloud systems”. In: ACM Symposium on Cloud Computing, 2020, pp. 344–357.

[61] He, X.; Alqahtani, S.; Gamble, R.; Papa, M. “Securing over-the-air iot firmware updates using blockchain”. In: International Conference on Omni-Layer Intelligent Systems, 2019, pp. 164–171.

[62] Herbst, N. R.; Kounev, S.; Reussner, R. “Elasticity in cloud computing: What it is, and what it is not”. In: International Conference on Autonomic Computing, 2013, pp. 23–27.

[63] Hou, W.; Li, W.; Guo, L.; Sun, Y.; Cai, X. “Recycling edge devices in sustainable internet of things networks”, *IEEE Internet of Things Journal*, vol. 4–5, July 2017, pp. 1696–1706.

[64] Hu, J.-W.; Yeh, L.-Y.; Liao, S.-W.; Yang, C.-S. “Autonomous and malware-proof blockchain-based firmware update platform with efficient batch verification for internet of things devices”, *Computers & Security*, vol. 86, September 2019, pp. 238–252.

[65] Huang, H.; Guo, S. “Proactive failure recovery for nfv in distributed edge computing”, *IEEE Communications Magazine*, vol. 57–5, March 2019, pp. 131–137.

[66] Huang, Y.; Kintala, C.; Kolettis, N.; Fulton, N. D. “Software rejuvenation: Analysis, module and applications”. In: International Symposium on Fault-Tolerant Computing, 1995, pp. 381–390.

[67] Huang, Z.; Huang, H. “Proactive failure recovery for stateful nfv”. In: International Conference on Parallel and Distributed Systems, 2020, pp. 536–543.

[68] Islam, S. R.; Kwak, D.; Kabir, M. H.; Hossain, M.; Kwak, K.-S. “The internet of things for health care: a comprehensive survey”, *IEEE Access*, vol. 3, June 2015, pp. 678–708.

[69] Ismail, B. I.; Goortani, E. M.; Ab Karim, M. B.; Tat, W. M.; Setapa, S.; Luke, J. Y.; Hoe, O. H. “Evaluation of docker as edge computing platform”. In: IEEE Conference on Open Systems, 2015, pp. 130–135.

[70] Jardosh, A.; Belding-Royer, E. M.; Almeroth, K. C.; Suri, S. “Towards realistic mobility models for mobile ad hoc networks”. In: Annual International Conference on Mobile Computing and Networking, 2003, pp. 217–229.

[71] Jha, D. N.; Alwasel, K.; Alshoshan, A.; Huang, X.; Naha, R. K.; Battula, S. K.; Garg, S.; Puthal, D.; James, P.; Zomaya, A.; Dustdar, S.; Ranjan, R. “Iotsim-edge: A simulation framework for modeling the behavior of internet of things and edge computing environments”, *Software: Practice and Experience*, vol. 50–6, June 2020, pp. 844–867.

[72] Jindal, A.; Aujla, G. S.; Kumar, N. “Survivor: A blockchain based edge-as-a-service framework for secure energy trading in sdn-enabled vehicle-to-grid environment”, *Computer Networks*, vol. 153, April 2019, pp. 36–48.

[73] Jing, Q.; Vasilakos, A. V.; Wan, J.; Lu, J.; Qiu, D. “Security of the internet of things: Perspectives and challenges”, *Wireless Networks*, vol. 20–8, June 2014, pp. 2481–2501.

[74] Jordan, M. I.; Mitchell, T. M. “Machine learning: Trends, perspectives, and prospects”, *Science*, vol. 349–6245, July 2015, pp. 255–260.

[75] Kaelbling, L. P.; Littman, M. L.; Moore, A. W. “Reinforcement learning: A survey”, *Journal of artificial intelligence research*, vol. 4, May 1996, pp. 237–285.

[76] Kantarci, B.; Mouftah, H. T. “Sensing services in cloud-centric internet of things: A survey, taxonomy and challenges”. In: International Conference on Communication Workshop, 2015, pp. 1865–1870.

[77] Kazil, J.; Masad, D.; Crooks, A. “Utilizing python for agent-based modeling: The mesa framework”. In: International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation, 2020, pp. 308–317.

[78] Kim, D.; Lee, S.; Kim, D. “An applicable predictive maintenance framework for the absence of run-to-failure data”, *Applied Sciences*, vol. 11–11, June 2021, pp. 1–17.

[79] Kliazovich, D.; Bouvry, P.; Khan, S. U. “Greencloud: a packet-level simulator of energy-aware cloud computing data centers”, *The Journal of Supercomputing*, vol. 62–3, November 2012, pp. 1263–1283.

[80] Knob, L. A. D.; Kayser, C. H.; de Souza, P. S. S.; Ferreto, T. “Enforcing deployment latency sla in edge infrastructures through multi-objective genetic scheduler”. In: International Conference on Utility and Cloud Computing, 2021, pp. 1–9.

[81] Kominos, C. G.; Seyvet, N.; Vandikas, K. “Bare-metal, virtual machines and containers in openstack”. In: Conference on Innovations in Clouds, Internet and Networks, 2017, pp. 36–43.

[82] Kreutz, D.; Ramos, F. M.; Verissimo, P. E.; Rothenberg, C. E.; Azodolmolky, S.; Uhlig, S. “Software-defined networking: A comprehensive survey”, *Proceedings of the IEEE*, vol. 103, December 2014, pp. 14–76.

[83] Langiu, A.; Boano, C. A.; Schuß, M.; Römer, K. “Upkit: An open-source, portable, and lightweight update framework for constrained iot devices”. In: International Conference on Distributed Computing Systems, 2019, pp. 2101–2112.

[84] Lasi, H.; Fettke, P.; Kemper, H.-G.; Feld, T.; Hoffmann, M. “Industry 4.0”, *Business & Information Systems Engineering*, vol. 6, June 2014, pp. 239–242.

[85] Law, A. M.; Kelton, W. D.; Kelton, W. D. “Simulation modeling and analysis”. Mcgraw-hill New York, 2015, vol. 5, 776p.

[86] Lee, B.; Lee, J.-H. “Blockchain-based secure firmware update for embedded devices in an internet of things environment”, *The Journal of Supercomputing*, vol. 73–3, September 2017, pp. 1152–1167.

[87] Lee, E. K.; Viswanathan, H.; Pompili, D. “Model-based thermal anomaly detection in cloud datacenters”. In: International Conference on Distributed Computing in Sensor Systems, 2013, pp. 191–198.

[88] Lehrig, S.; Eikerling, H.; Becker, S. “Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics”. In: International ACM SIGSOFT Conference on Quality of Software Architectures, 2015, pp. 83–92.

[89] Leiba, O.; Bitton, R.; Yitzchak, Y.; Nadler, A.; Kashi, D.; Shabtai, A. “Iotpatchpool: Incentivized delivery network of iot software updates based on proofs-of-distribution”, *Pervasive and Mobile Computing*, vol. 58, August 2019, pp. 1–21.

[90] Lera, I.; Guerrero, C.; Juiz, C. “Yafs: A simulator for iot scenarios in fog computing”, *IEEE Access*, vol. 7, July 2019, pp. 91745–91758.

[91] Liang, Z.; Liu, Y.; Lok, T.-M.; Huang, K. “Multi-cell mobile edge computing: Joint service migration and resource allocation”, *IEEE Transactions on Wireless Communications*, vol. 20–9, April 2021, pp. 5898–5912.

[92] Linthicum, D. S. “Practical use of microservices in moving workloads to the cloud”, *IEEE Cloud Computing*, vol. 3–5, November 2016, pp. 6–9.

[93] Liu, X.; Buyya, R. “Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions”, *ACM Computing Surveys*, vol. 53–3, May 2020, pp. 1–41.

[94] Macal, C. M.; North, M. J. “Tutorial on agent-based modeling and simulation”. In: Winter Simulation Conference, 2005, pp. 14–pp.

[95] Mahmud, R.; Pallewatta, S.; Goudarzi, M.; Buyya, R. “ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments”, *Journal of Systems and Software*, vol. 190, August 2022, pp. 1–17.

[96] Malik, A. W.; Rahman, A. U.; Ahmad, A.; Santos, M. M. D. “Over-the-air software-defined vehicle updates using federated fog environment”, *IEEE Transactions on Network and Service Management*, vol. 19–4, June 2022, pp. 5078–5089.

[97] Mansouri, Y.; Babar, M. A. “A review of edge computing: Features and resource virtualization”, *Journal of Parallel and Distributed Computing*, vol. 150, April 2021, pp. 155–183.

[98] Marek, V. W.; Truszczyński, M. “Stable models and an alternative logic programming paradigm”. In: *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K. R.; Marek, V. W.; Truszczynski, M.; Warren, D. S. (Editors), Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, chap. 17, pp. 375–398.

[99] Materwala, H.; Ismail, L.; Shubair, R. M.; Buyya, R. “Energy-sla-aware genetic algorithm for edge–cloud integrated computation offloading in vehicular networks”, *Future Generation Computer Systems*, vol. 135, October 2022, pp. 205–222.

[100] Meng, H.; Zhang, X.; Zhu, L.; Wang, L.; Yang, Z. “Optimizing software rejuvenation policy based on cdm for cloud system”. In: Conference on Industrial Electronics and Applications, 2017, pp. 1850–1854.

[101] Mergen, M. F.; Uhlig, V.; Krieger, O.; Xenidis, J. “Virtualization for high-performance computing”, *ACM SIGOPS Operating Systems Review*, vol. 40–2, April 2006, pp. 8–11.

[102] Mitchell, M. “An Introduction to Genetic Algorithms”. Cambridge, MA, USA: MIT Press, 1998, 139p.

[103] Mobley, R. “An Introduction to Predictive Maintenance”. Elsevier Science, 2002, 438p.

[104] Mogul, J. C.; Wilkes, J. “Nines are not enough: Meaningful metrics for clouds”. In: Workshop on Hot Topics in Operating Systems, 2019, pp. 136–141.

[105] Moubray, J. “Reliability-centered Maintenance”. Industrial Press, 2001, 424p.

[106] Mtetwa, N. S.; Tarwireyi, P.; Sibeko, C. N.; Abu-Mahfouz, A.; Adigun, M. “Blockchain-based security model for lorawan firmware updates”, *Journal of Sensor and Actuator Networks*, vol. 11, January 2022, pp. 1–39.

[107] Murata, T. “Petri nets: Properties, analysis and applications”, *Proceedings of the IEEE*, vol. 77–4, April 1989, pp. 541–580.

[108] Nain, Z.; Musaddiq, A.; Qadri, Y. A.; Kim, S. W. “History-aware adaptive route update scheme for low-power and lossy networks”. In: International Conference on Information and Communication Technology Convergence, 2021, pp. 1830–1834.

[109] Nain, Z.; Musaddiq, A.; Qadri, Y. A.; Nauman, A.; Afzal, M. K.; Kim, S. W. “Riata: A reinforcement learning-based intelligent routing update scheme for future generation iot networks”, *IEEE Access*, vol. 9, May 2021, pp. 81161–81172.

[110] Ngoc, T. D.; Teabe, B.; Tchana, A.; Muller, G.; Hagimont, D. “Mitigating vulnerability windows with hypervisor transplant”. In: European Conference on Computer Systems, 2021, pp. 162–177.

[111] Nurseitov, N.; Paulson, M.; Reynolds, R.; Izurieta, C. “Comparison of json and xml data interchange formats: a case study.” In: International Conference on Computer Applications in Industry and Engineering, 2009, pp. 157–162.

[112] Okuno, S.; Iikura, F.; Watanabe, Y. “Maintenance scheduling for cloud infrastructure with timing constraints of live migration”. In: International Conference on Cloud Engineering, 2019, pp. 179–189.

[113] Olorunnife, K.; Lee, K.; Kua, J. “Automatic failure recovery for container-based iot edge applications”, *Electronics*, vol. 10–23, December 2021, pp. 1–19.

[114] Osterlind, F.; Dunkels, A.; Eriksson, J.; Finne, N.; Voigt, T. “Cross-level sensor network simulation with cooja”. In: Conference on Local Computer Networks, 2006, pp. 641–648.

[115] Pezoa, F.; Reutter, J. L.; Suarez, F.; Ugarte, M.; Vrgoč, D. “Foundations of json schema”. In: International Conference on World Wide Web, 2016, pp. 263–273.

[116] Prajapati, A.; Bechtel, J.; Ganesan, S. “Condition based maintenance: a survey”, *Journal of Quality in Maintenance Engineering*, vol. 18–4, October 2012, pp. 384–400.

[117] Pu, X.; Liu, L.; Mei, Y.; Sivathanu, S.; Koh, Y.; Pu, C. “Understanding performance interference of i/o workload in virtualized cloud environments”. In: International Conference on Cloud Computing, 2010, pp. 51–58.

[118] Puliafito, C.; Gonçalves, D. M.; Lopes, M. M.; Martins, L. L.; Madeira, E.; Mingozzi, E.; Rana, O.; Bittencourt, L. F. “Mobfogsim: Simulation of mobility and migration for fog computing”, *Simulation Modelling Practice and Theory*, vol. 101, May 2020, pp. 1–25.

[119] Qayyum, T.; Malik, A. W.; Khan Khattak, M. A.; Khalid, O.; Khan, S. U. “Fognetsim++: A toolkit for modeling and simulation of distributed fog environment”, *IEEE Access*, vol. 6, October 2018, pp. 63570–63583.

[120] Qian, L.; Luo, Z.; Du, Y.; Guo, L. “Cloud computing: An overview”. In: International Conference on Cloud Computing, 2009, pp. 626–631.

[121] Qu, C.; Calheiros, R. N.; Buyya, R. “Auto-scaling web applications in clouds: A taxonomy and survey”, *ACM Computing Surveys*, vol. 51–4, July 2018, pp. 1–33.

[122] Ren, W.; Sun, Y.; Luo, H.; Guizani, M. “Bllc: A batch-level update mechanism with low cost for sdn-iot networks”, *IEEE Internet of Things Journal*, vol. 6, September 2018, pp. 1210–1222.

[123] Reviriego, P.; Hernández, J. A.; Larrabeiti, D.; Maestro, J. A. “Performance evaluation of energy efficient ethernet”, *IEEE Communications Letters*, vol. 13–9, September 2009, pp. 697–699.

[124] Rolik, O.; Telenyk, S.; Zharikov, E. “Management of services of a hyperconverged infrastructure using the coordinator”. In: International Conference on Computer Science, Engineering and Education Applications, 2018, pp. 456–467.

[125] Romanycia, M. H.; Pelletier, F. J. “What is a heuristic?”, *Computational Intelligence*, vol. 1, January 1985, pp. 47–58.

[126] Rubin, F.; Souza, P.; Ferreto, T. “Reducing power consumption during server maintenance on edge computing infrastructures”. In: Symposium on Applied Computing, 2023, pp. 691–698.

[127] Russinovich, M.; Govindaraju, N.; Raghuraman, M.; Hepkin, D.; Schwartz, J.; Kishan, A. “Virtual machine preserving host updates for zero day patching in public cloud”. In: European Conference on Computer Systems, 2021, pp. 114–129.

[128] Salama, M.; Elkhatib, Y.; Blair, G. “Iotnetsim: A modelling and simulation platform for end-to-end iot services and networking”. In: International Conference on Utility and Cloud Computing, 2019, pp. 251–261.

[129] Sanders, W. H.; Meyer, J. F. “Stochastic activity networks: formal definitions and concepts”. In: School organized by the European Educational Forum, 2000, pp. 315–343.

[130] Sargent, R. G. “Verification and validation of simulation models”. In: Winter Simulation Conference, 2010, pp. 166–183.

[131] Satyanarayanan, M. “The emergence of edge computing”, *Computer*, vol. 50, January 2017, pp. 30–39.

[132] Satyanarayanan, M.; Bahl, P.; Caceres, R.; Davies, N. “The case for vm-based cloudlets in mobile computing”, *IEEE Pervasive Computing*, vol. 8–4, October 2009, pp. 14–23.

[133] Satyanarayanan, M.; Gao, W.; Lucia, B. “The computing landscape of the 21st century”. In: International Workshop on Mobile Computing Systems and Applications, 2019, pp. 45–50.

[134] Satyanarayanan, M.; Klas, G.; Silva, M.; Mangiante, S. “The seminal role of edge-native applications”. In: International Conference on Edge Computing, 2019, pp. 33–40.

[135] Saxena, D.; Singh, A. K. “Ofp-tm: an online vm failure prediction and tolerance model towards high availability of cloud computing environments”, *The Journal of Supercomputing*, vol. 78–6, January 2022, pp. 8003–8024.

[136] Sayama, H. “Introduction to the modeling and analysis of complex systems”. Open SUNY Textbooks, 2015, 478p.

[137] Segalini, A.; Lopez Pacheco, D.; Urvoy-Keller, G.; Hermenier, F.; Jacquemart, Q. “Hy-fix: Fast in-place upgrades of kvm hypervisors”, *IEEE Transactions on Cloud Computing*, vol. 10–4, December 2022, pp. 2679–2690.

[138] Sharma, P.; Chaufournier, L.; Shenoy, P.; Tay, Y. C. “Containers and virtual machines at scale: A comparative study”. In: International Middleware Conference, 2016, pp. 1–13.

[139] Shi, J.; Gurewitz, O.; Mancuso, V.; Camp, J.; Knightly, E. W. “Measurement and modeling of the origins of starvation in congestion controlled mesh networks”. In: International Conference on Computer Communications, 2008, pp. 1633–1641.

[140] Shrivastava, V.; Zerfos, P.; Lee, K.-W.; Jamjoom, H.; Liu, Y.-H.; Banerjee, S. “Application-aware virtual machine migration in data centers”. In: International Conference on Computer Communications, 2011, pp. 66–70.

[141] Silva, P.; Fireman, D.; Pereira, T. E. “Prebaking functions to warm the serverless cold start”. In: International Middleware Conference, 2020, pp. 1–13.

[142] Silvestri, L.; Forcina, A.; Introna, V.; Santolamazza, A.; Cesarotti, V. “Maintenance transformation through industry 4.0 technologies: A systematic literature review”, *Computers in Industry*, vol. 123, December 2020, pp. 1–16.

[143] Singh, J.; Singh, P.; Gill, S. S. “Fog computing: A taxonomy, systematic review, current trends and research challenges”, *Journal of Parallel and Distributed Computing*, vol. 157, November 2021, pp. 56–85.

[144] Sipos, R.; Fradkin, D.; Moerchen, F.; Wang, Z. “Log-based predictive maintenance”. In: International Conference on Knowledge Discovery and Data Mining, 2014, pp. 1867–1876.

[145] Sonmez, C.; Ozgovde, A.; Ersoy, C. “Edgecloudsim: An environment for performance evaluation of edge computing systems”. In: International Conference on Fog and Mobile Edge Computing, 2017, pp. 39–44.

[146] Souza, P.; Crestani, A.; Ferreto, T.; Rossi, F. “Latency-aware privacy-preserving service migration in federated edges”. In: International Conference on Cloud Computing and Services Science, 2022, pp. 288–295.

[147] Souza, P.; Ferreto, T. “A heuristic algorithm for minimizing server maintenance time and vulnerability surface on data centers”, Master’s Thesis, Graduate Program in Computer Science — Pontifical Catholic University of Rio Grande do Sul, 2020, 48p.

[148] Souza, P. S.; Ferreto, T.; Calheiros, R. N. “EdgeSimPy: Python-based modeling and simulation of edge computing resource management policies”, *Future Generation Computer Systems*, vol. 148, November 2023, pp. 446–459.

[149] Souza, P. S.; Ferreto, T. C.; Rossi, F. D.; Calheiros, R. N. “Location-aware maintenance strategies for edge computing infrastructures”, *IEEE Communications Letters*, vol. 26–4, February 2022, pp. 848–852.

[150] Süzen, A. A.; Duman, B.; Şen, B. “Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn”. In: International Congress on Human-Computer Interaction, Optimization and Robotic Applications, 2020, pp. 1–5.

[151] Tang, Z.; Lou, J.; Jia, W. “Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing”, *IEEE Transactions on Mobile Computing*, vol. 22–6, January 2022, pp. 3444–3459.

[152] Tapas, N.; Yitzchak, Y.; Longo, F.; Puliafito, A.; Shabtai, A. “P4uiot: Pay-per-piece patch update delivery for iot using gradual release”, *Sensors*, vol. 20–7, April 2020, pp. 1–27.

[153] Taqi Raza, M.; Tan, Z.; Tufail, A.; Muhammad Anwar, F. “Lte nfv rollback recovery”, *IEEE Transactions on Network and Service Management*, vol. 19–3, June 2022, pp. 2468–2477.

[154] Tian, H.; Wu, J.; Shen, H. “Efficient algorithms for vm placement in cloud data centers”. In: International Conference on Parallel and Distributed Computing, Applications and Technologies, 2017, pp. 75–80.

[155] Toprak, C.; Turker, C.; Erman, A. T. “Detection of dhcp starvation attacks in software defined networks: A case study”. In: International Conference on Computer Science and Engineering, 2018, pp. 636–641.

[156] Torquato, M.; Maciel, P.; Vieira, M. “A model for availability and security risk evaluation for systems with vmm rejuvenation enabled by vm migration scheduling”, *IEEE Access*, vol. 7, September 2019, pp. 138315–138326.

[157] Tsaur, W.-J.; Chang, J.-C.; Chen, C.-L. “A highly secure iot firmware update mechanism using blockchain”, *Sensors*, vol. 22–2, January 2022, pp. 1–20.

[158] Usman, M.; Britto, R.; Börstler, J.; Mendes, E. “Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method”, *Information and Software Technology*, vol. 85, May 2017, pp. 43–59.

[159] Vanura, J.; Kriz, P. “Perfomance evaluation of java, javascript and php serialization libraries for xml, json and binary formats”. In: International Conference on Services Computing, 2018, pp. 166–175.

[160] Wang, J.; Feng, Z.; George, S.; Iyengar, R.; Pillai, P.; Satyanarayanan, M. “Towards scalable edge-native applications”. In: Symposium on Edge Computing, 2019, pp. 152–165.

[161] Wang, K.; Rao, J.; Xu, C.-Z. “Rethink the virtual machine template”, *ACM SIGPLAN Notices*, vol. 46–7, March 2011, pp. 39–50.

[162] Wang, L.; Ramasamy, H. V.; Harper, R. E. “Scheduling physical machine maintenance on qualified clouds: What if migration is not allowed?” In: International Conference on Cloud Computing, 2020, pp. 485–492.

[163] Wang, S.; Guo, Y.; Zhang, N.; Yang, P.; Zhou, A.; Shen, X. “Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach”, *IEEE Transactions on Mobile Computing*, vol. 20–3, December 2019, pp. 939–951.

[164] Weißbach, M.; Taing, N.; Wutzler, M.; Springer, T.; Schill, A.; Clarke, S. “Decentralized coordination of dynamic software updates in the internet of things”. In: World Forum on Internet of Things, 2016, pp. 171–176.

[165] Witanto, E. N.; Oktian, Y. E.; Lee, S.-G.; Lee, J.-H. “A blockchain-based ocf firmware update for iot devices”, *Applied Sciences*, vol. 10–19, September 2020, pp. 1–22.

[166] Wu, C.-P.; Suresh, M. A.; Da Silva, D. “Container lifecycle management for edge nodes: poster”. In: Symposium on Edge Computing, 2017, pp. 1–2.

[167] Wu, W.; Wang, J.; Lu, K.; Qi, W.; Shan, F.; Luo, J. “Providing service continuity in clouds under power outage”, *IEEE Transactions on Services Computing*, vol. 13–5, July 2017, pp. 930–943.

[168] Wu, Y.; Liu, D.; Tan, Y.; Duan, M.; Luo, L.; Wang, W.; Chen, X. “Lfpr: A lazy fast predictive repair strategy for mobile distributed erasure coded cluster”, *IEEE Internet of Things Journal*, vol. 10, January 2022, pp. 704–719.

[169] Xavier, M. G.; De Oliveira, I. C.; Rossi, F. D.; Dos Passos, R. D.; Matteussi, K. J.; De Rose, C. A. “A performance isolation analysis of disk-intensive workloads on container-based clouds”. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2015, pp. 253–260.

[170] Xiang, X.; Kennedy, R.; Madey, G.; Cabaniss, S. “Verification and validation of agent-based scientific simulation models”. In: Agent-Directed Simulation Conference, 2005, pp. 47–55.

[171] Xiao, Y.; Jia, Y.; Liu, C.; Cheng, X.; Yu, J.; Lv, W. “Edge computing security: State of the art and challenges”, *Proceedings of the IEEE*, vol. 107–8, June 2019, pp. 1608–1631.

[172] Xu, B.; Wu, S.; Xiao, J.; Jin, H.; Zhang, Y.; Shi, G.; Lin, T.; Rao, J.; Yi, L.; Jiang, J. “Sledge: Towards efficient live migration of docker containers”. In: International Conference on Cloud Computing, 2020, pp. 321–328.

[173] Ye, Y.; Li, S.; Liu, F.; Tang, Y.; Hu, W. “Edgefed: Optimized federated learning based on edge computing”, *IEEE Access*, vol. 8, November 2020, pp. 209191–209198.

[174] Ying, C.; Li, B.; Ke, X.; Guo, L. “Raven: Scheduling virtual machine migration during datacenter upgrades with reinforcement learning”, *Mobile Networks and Applications*, vol. 27, February 2022, pp. 303–314.

[175] Zhao, H.; Deng, S.; Liu, Z.; Yin, J.; Dustdar, S. “Distributed redundant placement for microservice-based applications at the edge”, *IEEE Transactions on Services Computing*, vol. 15–3, August 2020, pp. 1732–1745.

[176] Zhao, L.; Lu, L.; Jin, Z.; Yu, C. “Online virtual machine placement for increasing cloud provider’s revenue”, *IEEE Transactions on Services Computing*, vol. 10–2, June 2015, pp. 273–285.

[177] Zheng, Z.; Wang, J.; Ren, J.; Hou, W.; Wang, J. “Least maintenance batch scheduling in cloud data center networks”, *IEEE Communications Letters*, vol. 18–6, April 2014, pp. 901–904.

[178] Zheng, Z.; Xie, S.; Dai, H. N.; Chen, X.; Wang, H. “Blockchain challenges and opportunities: A survey”, *International Journal of Web and Grid Services*, vol. 14–4, October 2018, pp. 352–375.

[179] Zonta, T.; Da Costa, C. A.; da Rosa Righi, R.; de Lima, M. J.; da Trindade, E. S.; Li, G. P. “Predictive maintenance in the industry 4.0: A systematic literature review”, *Computers & Industrial Engineering*, vol. 150, December 2020, pp. 1–17.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br