Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (37^{th}cycle)

Distributed Orchestration in a Computing Continuum

By

Stefano Galantino

******

Supervisor(s):
Prof. Fulvio Risso, Supervisor
Dr. Antonio Manzalini, Co-Supervisor

Doctoral Examination Committee:
Prof. Andrea Detti, Referee, Università di Roma Tor Vergata
Prof. Yiannis Verginadis, Referee, Athens University of Economics and Business
Prof. Guido Marchetto, Politecnico di Torino
Dr. Nuria Molner Siurana, Universitat Politècnica de València
Prof. Panagiotis Trakadas, National and Kapodistrian University of Athens

Politecnico di Torino
2025

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Stefano Galantino
2025

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Although extremely intense, these last three years have shaped me both professionally and personally. Now, this thesis marks the end of a journey that would not have been possible without the support of many incredible people, each of whom contributed in their own way to help me reach this point.

First, I would like to express my deepest gratitude to my parents and my sister, who have always supported me through the ups and downs, understanding me even when I am not very communicative—making everything more difficult for them.

A special thank you also to all my friends for always providing a pleasant and often much-needed distraction from work.

Special thanks go to all the people from Saint Louis University, to Professor Flavio Esposito for his insightful ideas and the collaborations we have had (and hopefully more to come), and to all the incredible people in the research lab who welcomed me and made me feel at home. I would also like to extend my deepest gratitude to two amazing people, Edgar and Raven, who always pushed me to become a better version of myself.

I also want to thank TIM S.p.A. for allowing me to pursue this career by sponsoring my PhD, and Antonio Manzalini, my TIM supervisor, for the fruitful exchanges we had before and during the PhD.

Thanks to all the FLUIDOS partners for the fruitful collaborations over the last three years.

Last but not least, I'd like to thank my supervisor, Fulvio Risso. While he challenged me relentlessly, he never withheld his support and helped me grow far more than I would have otherwise.

Abstract

Distributed computing has evolved significantly with the emergence of paradigms like cloud, edge, and fog computing, which bring computing capabilities closer to the end-user to address modern application demands for low latency, geographical distribution, and resource efficiency. Despite these advancements, the current state-of-the-art solutions are predominantly siloed, limiting the seamless integration of resources across diverse infrastructures. This dissertation builds on the concept of the *computing continuum*, an architectural paradigm that eliminates these silos by unifying resources across the compute layers into a single, dynamic, and transparent computational environment. The work is situated within the European Horizon project FLUIDOS (Flexible, scaLable, secUre, and decentralIseD Operating System), which builds the foundation of this research.

The FLUIDOS architecture builds upon Kubernetes and Liqo to establish a decentralized meta-operating system capable of dynamic resource sharing and intent-based orchestration. This dissertation presents key innovations in the design and implementation of the computing continuum, focusing on the REAR (Resource Advertisement and Reservation) protocol, which facilitates resource discovery and negotiation across heterogeneous environments. The REAR protocol introduces a flexible mechanism for dynamic resource advertisement, ensuring that resources can be utilized based on application-specific intents, energy consumption patterns, and cost models.

The thesis further investigates cost-aware task allocation, presenting a novel scheduling framework that minimizes application deployment costs while meeting performance requirements. Experimental evaluations demonstrate that the proposed framework significantly enhances resource utilization, reduces costs, and maintains compliance with user-defined constraints. Additionally, the research addresses energy efficiency within the continuum, proposing an energy-aware orchestration

mechanism that dynamically adjusts workload placement based on power consumption and carbon footprint considerations. This approach integrates carbon-aware scheduling strategies, enabling the execution of tasks in environmentally sustainable locations and timeframes.

Resilience is another critical dimension explored in this work. The dissertation introduces a fault-tolerant orchestration framework designed to ensure the high availability of mission-critical applications, such as electrical grid monitoring. By dynamically redistributing workloads during network failures or infrastructure outages, the framework maintains service continuity, demonstrating its efficacy through real-world experiments conducted in collaboration with RSE (Ricerca sul Sistema Energetico). These experiments highlight the capability of the computing continuum to support distributed and fault-tolerant applications in complex and resource-constrained environments.

This dissertation provides a comprehensive exploration of the computing continuum, encompassing its architectural foundations, enabling technologies, and key application scenarios. The FLUIDOS framework exemplifies how distributed orchestration can address the challenges of scalability, sustainability, and reliability in modern computing. By integrating innovative protocols like REAR, cost-aware and energy-efficient task allocation strategies, and resilience mechanisms, the computing continuum emerges as a transformative paradigm for future distributed systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last years, containerization has increasingly gained popularity as a lightweight solution to package applications in an interoperable format [1], independently of the target infrastructure. This uniform substratum paved the way for the cloud-native revolution, with novel applications shifting their focus from single servers to entire data centers, and where dedicated orchestrators manage the lifecycle of microservice applications. In fact, according to the NIST definition of the cloud computing:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [2]

As of today, Kubernetes has emerged as the de-facto open-source framework for container orchestration, bridging the semantic gaps across competing infrastructure providers. With the rise of the edge and fog computing paradigms [3–5] as solutions accounting for geographical closeness, reduced latency, and improved privacy, the same approaches are being progressively extended towards smaller data centers at the network border, benefiting from uniform primitives to foster service agility.

Despite the emergence of common interfaces for applications orchestration being key towards a real edge to cloud continuum [6, 7], industry-standard approaches handle each infrastructure as a multitude of (connected) isolated silos instead of a unique virtual space. This leads to a sub-optimal fragmented view of the overall available resources, preventing the seamless deployment of fully distributed applica-

tions. Indeed, edge data centers cannot depend on a single centralized control plane, for resiliency (i.e., preventing failure propagation in case of network partitioning) and performance reasons, as orchestration platforms typically suffer if nodes are geographically spread over high-latency WANs [8–10]. This trend finds its roots in scalability concerns, in the hybrid-cloud (i.e., the combination of on-premises and public cloud) and multi-cloud approaches, which aim for high availability, geographical distribution, and cost-effectiveness, while granting access to the breadth of capabilities offered by competing cloud providers. Additionally, non-technical requirements such as law regulations, mergers and acquisitions, physical isolation policies, and separation of concerns contribute to the proliferation of clusters.

Fragmentation also hinders the potential dynamism in the workload placement [11–13], forcing each application to be assigned upfront to a specific infrastructure. No resource compensation is ever possible, hence preventing jobs from transparently moving from an overloaded cluster, e.g., due to unexpected spikes of requests, to another one, underused and potentially offering better performance. At the same time, the deployment of complex applications composed of multiple microservices, each one with specific requirements (e.g., low latency, high computational power, access to specialized hardware, ...), as well as the enforcement of proper geographical distribution and high-availability policies, requires the interaction with different infrastructures. However, this prevents relying on a single point of control, which would coordinate the deployment of arbitrarily complex applications across the entire resource continuum, no matter how many nodes and clusters it is composed of.

Therefore, this dissertation aims to bridge the gap between recent advancements in cloud computing and the need to extend the computing domain, overcoming the limitations of the previously described siloed approach. Specifically, the work of this thesis has been partially carried out as part of the European project FLUIDOS[1] (Flexible, scaLable, secUre, and decentralIseD Operating System), which advocates the opportunity for a novel architectural paradigm, called *liquid computing* in the seminar paper [14], which builds upon and extends the well-established cloud and edge computing approaches towards an endless computing continuum. Overall, the resulting computing domain abstracts away the specificity of each cluster, presenting to the final users, either actively participating as actors or simply renting off-the-shelf

[1] https://fluidos.eu/

services, a unique and borderless pool of available resources, the so-called big cluster. Thanks to this abstraction, applications are no longer constrained in a specific silo, but free to fly in the entire infrastructure, selecting the most appropriate location depending on its requirements (e.g., a user-facing service may be replicated at the edge to account for low latency, while another might be constrained to European infrastructures to comply with GDPR), and the available resources, while retaining full compatibility (hence, models, tools, and commands) with vanilla cloud-based solutions (i.e., Kubernetes).

At the core of this work is FLUIDOS, a European research initiative that introduces a decentralized meta-operating system to enable dynamic and flexible resource management across heterogeneous computing environments. The discussion begins with an exploration of the types of applications that can benefit from such an infrastructure, including latency-sensitive services, energy-aware computations, large-scale data processing, and privacy-preserving applications. By providing a unified abstraction over geographically distributed computing resources, FLUIDOS allows workloads to be dynamically migrated and optimized, adapting to network conditions, user demand, and resource availability. The main research challenges stemming from the creation of FLUIDOS continuum can then be summarized as follows:

- **Decentralization**: The computing continuum requires autonomous decision-making by each node to manage its own boundaries and resource acquisition. Decentralized orchestration strategies must then be implemented to ensure the peer-to-peer-based nature of the continuum.
- **Data Privacy and Sovereignty**: With data moving across diverse administrative domains, maintaining compliance with regulations like GDPR is crucial. Entities in the continuum must then know whether the processing, and most importantly, the corresponding data, is compliant with the requested regulations.
- **Cross-Domain Authentication and Authorization**: The multi-tenant nature of the continuum complicates identity and access management. To this end, strict authentication/authorization policies must be enforced to prevent unauthorized access to the available resources.

- **Extension Beyond Kubernetes**: Kubernetes is one of the enabling technologies in FLUIDOS, however, it is insufficient for many resource-constrained, non-Linux devices in the continuum (e.g., IoT nodes). The orchestration capabilities must then extend to such devices, supporting communication protocols like MQTT and HTTP.

A key enabler of this architecture is the REAR (Resource Advertisement and Reservation) protocol, which facilitates intent-based resource negotiation within the computing continuum. Unlike traditional cloud-based scheduling mechanisms, REAR introduces a flexible and scalable approach for dynamically discovering, advertising, and reserving resources based on application-specific requirements. This protocol enables cost-aware and energy-efficient task placement, allowing applications to select the most appropriate execution environment seamlessly while maintaining compliance with latency constraints, regulatory policies, and sustainability goals.

With the foundational infrastructure in place, the discussion shifts towards task orchestration strategies in the computing continuum. One of the key aspects explored is cost-aware workload allocation, where a centralized scheduling algorithm optimizes resource selection to minimize the perceived cost of application deployment. This approach takes into account pricing models, infrastructure constraints, and network conditions, ensuring that applications are deployed in a cost-effective yet performant manner.

Beyond cost optimization, the research also investigates energy-efficient orchestration, recognizing that energy consumption is a critical concern in large-scale distributed computing. The work begins with a comprehensive energy assessment of the computing continuum, analyzing power consumption patterns across different computing layers. Building upon this, an energy-aware allocation mechanism is introduced, leveraging dynamic workload migration techniques to optimize resource utilization based on energy availability, device efficiency, and workload demand. This approach enables carbon-aware computing strategies, ensuring that tasks are executed in locations where the environmental impact is minimized.

The resilience of the computing continuum is also a key aspect explored in this work, particularly in the context of mission-critical applications such as electrical grid monitoring. In such scenarios, ensuring high availability and fault tolerance is paramount, as disruptions in real-time monitoring can have severe consequences.

To address this, a resilient task orchestration framework is proposed, designed to handle network failures, infrastructure outages, and unexpected workload fluctuations. By dynamically redistributing computing tasks across available resources, this framework enhances system reliability and ensures continuous service availability. This research was conducted in collaboration with RSE (Ricerca sul Sistema Energetico), where real-world experiments demonstrated how the computing continuum can support fault-tolerant and distributed applications for energy system monitoring.

Bringing all these elements together, this dissertation presents a holistic approach to building and optimizing the computing continuum. It demonstrates how FLUIDOS, intent-based resource negotiation, cost-aware scheduling, energy-efficient task placement, and resilient orchestration collectively contribute to a scalable, adaptive, and sustainable computing infrastructure, showcasing some of the possible usage scenarios of the continuum.

1.1 Summary of contributions

This dissertation is structured into four main chapters, each addressing a distinct research area within the computing continuum. Specifically, the contributions of each section can be summarized as follows:

- **Chapter 2**. Presents FLUIDOS, describing its architectural components and the proprietary REAR protocol, which facilitates resource negotiation within the continuum. The discussion begins with an analysis of the types of applications that can benefit from such an infrastructure and then delves into the FLUIDOS software stack and its core functionalities.
- **Chapter 3**. Assuming the existence of a FLUIDOS-enabled computing continuum, this chapter focuses on task orchestration strategies. Specifically, it presents a centralized cost-based allocation algorithm aimed at minimizing the perceived cost of application deployment, while meeting the requirements for the application deployment in terms of computing and networking.
- **Chapter 4**. The focus then shifts toward energy efficiency, beginning with a preliminary assessment of the computing continuum to analyze the relationship between device performance and the energy required to sustain it. The

chapter then introduces a distributed energy-aware orchestration approach that optimizes task placement based on energy consumption considerations, while preventing information disclosure when needed.

- **Chapter 5**. Investigates the resiliency of the computing continuum, demonstrating its capacity to support the execution of critical monitoring applications for electrical grids. Specifically, unexpected events such as network failures, infrastructure outages, and unexpected workload fluctuations have been considered pivotal in the design and validation of the infrastructure.
- **Chapter 6**. Draws conclusions and defines future directions stemming from the current work.

1.1.1 Previously published work

This thesis includes previously published and co-authored works. In particular, Chapter 2 is adapted from the work presented in [15], in the public deliverables of the FLUIDOS project [16, 17] and one yet unpublished work. [2] Chapter 3 is an adaptation of [18]. Chapter 4 is adapted from the preliminary evaluation in [19] and the consequent allocation strategy detailed in [20], carried out with the research group of prof. Flavio Esposito at Saint Louis University, Saint Louis (MO). Finally, Chapter 5 is an adaptation of [21] and a work not yet published, which extends the previous one, both of which were conducted in collaboration with RSE (Ricerca sul Sistema Energetico). [3]

[2] Although the content of this dissertation reflects only the outcomes of the FLUIDOS Work Package 2, for which I served as a Work Package Leader, some of the ideas stem from discussions with other Work Packages to present a cohesive vision for the computing continuum. Therefore, I would like to extend my gratitude to the other FLUIDOS partners.

[3] I would also like to extend my gratitude to Elisa Albanese and Luca Zuanazzi, who both contributed to the unpublished work, but not to the already published one.

Chapter 2

Build the Continuum Infrastructure

The traditional siloed approach to computing, characterized by isolated and domain-specific solutions, presents significant challenges in addressing the diverse and dynamic requirements of modern applications. This approach limits flexibility, scalability, and interoperability, which are crucial for applications such as real-time processing, data-intensive analytics, privacy-centric computations, and resource-constrained operations. By contrast, the computing continuum envisions a seamless integration of heterogeneous computing layers, enabling dynamic and efficient resource utilization across geographical and operational boundaries. Despite its potential to revolutionize modern computing, the lack of a universally accepted definition hinders its standardization and adoption.

2.1 Main contributions

To bridge this gap, this work proposes in [15–17] a refined definition of the computing continuum, grounded in three fundamental transparency properties essential for constructing the continuum abstraction: *deployment*, *communication*, and *resource availability*. These properties draw inspiration from the concept of liquid computing [14] and have been further developed within the context of the FLUIDOS Horizon Europe project. FLUIDOS, an acronym for Flexible, scaLable, secUre, and decentralIseD Operating System, is a decentralized meta-operating system that leverages the orchestration capabilities of Kubernetes. This is augmented by the multi-cloud and multi-cluster abstractions provided by Liqo, enabling FLUIDOS

nodes to seamlessly share computational resources across continuum members while consistently enforcing the three transparency properties. Beyond establishing the continuum infrastructure, FLUIDOS also manages applications deployed within this environment, striving to maximize the user-defined execution goals.

Drawing from established theory (e.g., Silberschatz et al. [22]), an Operating System is traditionally required to provide the following core functionalities: (*i*) *Program execution*, (*ii*) *I/O operations*, (*iii*) *File-system manipulation*, (*iv*) *Communication*, (*v*) *Resource allocation*, (*vi*) *Accounting*, and (*vii*) *Protection and security*. Although FLUIDOS operates as a decentralized OS within the computing continuum, it ensures these foundational functionalities are preserved. Specifically, *program execution*, *I/O operations*, and *file-system manipulation* are inherently supported by Kubernetes' orchestration capabilities. However, *communication*—while partially supported by Kubernetes (limited to intra-cluster communication)—requires augmentation via Liqo's multi-cluster abstraction to enable seamless interaction across continuum nodes. Finally, the functionalities of *resource allocation*, *accounting*, and *protection and security* are implemented through the FLUIDOS control logic, which builds upon Kubernetes and Liqo to provide robust, decentralized operational guarantees within the continuum. This holistic approach ensures that FLUIDOS not only aligns with traditional OS principles but also extends their applicability to a decentralized, multi-cluster environment, or, in broader terms, the continuum.

2.2 Motivations and use cases

Here, we outline several applications that are restricted by the traditional siloed-based computing landscape. Such applications typically feature extremely diversified requirements that cannot be addressed by each silo independently.

Real-time Applications. Real-time applications require the immediate processing of data to make decisions within seconds or, more commonly, milliseconds [23]. To ensure delays do not disrupt functionality, these applications must operate at the far edge of the network, where data is typically produced and where latency can be minimized. In addition, high availability of resources is a critical concern, preventing the single point of failure scenario, and ensuring that the infrastructure guarantees a high degree of resiliency for the deployed applications in case of unexpected disruptive events. For instance, if a hosting cluster becomes unreachable, a new healthy

instance of the application must be deployed on a nearby site. *The computing continuum must allow the creation of geographically distributed infrastructures to ensure fault-tolerance while meeting the communication requirements of applications.*

Data-Intensive Applications. Data-intensive applications generate or rely on vast amounts of data that may need processing, filtering, and storage [24]. Simultaneously, such applications often require hierarchical processing and are often supplemented with analytics tools for statistical data analysis or machine learning models for inference on the generated data. These tools demand significant computing resources, which are typically unavailable at the far edge but readily accessible in cloud infrastructures. *The computing continuum must seamlessly integrate resources across different layers of computing to meet these diverse requirements.*

Privacy and Security-Critical Applications. Privacy and security-critical applications handle sensitive data that must adhere to strict privacy regulations. As a result, they need localized processing to ensure data does not leave a secure boundary of the organization, or in general, that the data is stored and processed according to specific regulations. This all relates to the concept of *data spaces* [25, 26], which refers to building complex federated ecosystems for data sharing to offer technical infrastructure and governance models to enable participants to pool, access, process, use and share data trustfully and transparently. *The computing continuum must be able to implement the data spaces abstraction to ensure trustworthiness in the shared data across the infrastructure.*

Highly Mobile and Decentralized Applications. Similarly to the handover process in telecommunication in which cellular transmission (voice or data) is transferred from one base station (cell site) to another without losing connectivity, this kind of application deal with the mobility of users, and, consequently, the mobility of the application following the same user [27]. *The computing continuum must allow the seamless integration of geographically dispersed and potentially heterogeneous clusters to ensure continuity of operations despite the mobility of both users and processing.*

Resource-Constrained Applications. Resource-constrained applications often operate on devices with limited computational power, memory, or battery life [28]. These limitations necessitate task offloading to more powerful nodes, such as edge servers or the cloud, to either enhance application performance or enable its execution. Similarly, battery constraints require adaptive strategies to balance performance and

energy consumption. *The computing continuum must support the creation of dynamic infrastructures capable of meeting the ever-changing needs of applications.*

2.3 The pathway to the computing continuum

In the following, we outline the most relevant technologies to build and maintain the continuum, differentiating the *production-ready* (Section 2.3.1), and the *research-oriented* projects (Section 2.3.2).

2.3.1 Production-ready projects

This section presents the most promising open-source projects for the creation of the continuum abstraction, along with considerations on the applicability and limitations of each.

KubeEdge. KubeEdge [29] is an open-source platform that extends Kubernetes orchestration to edge devices, enabling unified management of containerized applications across cloud and edge environments. It provides capabilities for application deployment, device management, and metadata synchronization between cloud and edge nodes. KubeEdge facilitates efficient interaction with devices and sensors connected to edge nodes by supporting HTTP-based and MQTT-based workloads, enabling localized data collection and processing while maintaining synchronization with the cloud for further computation and storage. However, KubeEdge has been designed for specific communication patterns (i.e., edge-to-cloud and cloud-to-edge). Therefore, it lacks the proper network fabric needed for broader multi-cluster scenarios (e.g., edge-to-edge).

Karmada. Karmada (Kubernetes Armada)[1] is an open-source system designed to manage applications across multiple Kubernetes clusters and cloud environments without requiring changes to the applications. By leveraging Kubernetes-native APIs and advanced scheduling capabilities, Karmada provides centralized multi-cluster management, automating deployment, high availability, failure recovery, and traffic scheduling across hybrid and multi-cloud scenarios. While Karmada facilitates cluster management under a master-worker model, it does not allow for equal-level

[1] https://karmada.io

federation of clusters (i.e., peer-to-peer), as one cluster must always lead. Moreover, Karmada lacks a built-in network fabric for inter-cluster communication, relying instead on external tools like Submariner[2].

Liqo. Liqo [14, 30] is an open-source project that enables dynamic and seamless Kubernetes multi-cluster management across heterogeneous environments, including on-premises, cloud, and edge infrastructures. Liqo introduces four primary capabilities to facilitate multi-cluster management: peering, offloading, network fabric, and storage fabric. Peering establishes relationships between clusters, enabling secure communication through cross-cluster VPN tunnels. Offloading leverages a virtual node abstraction to distribute workloads across remote clusters while maintaining full Kubernetes API compliance. The network fabric enables inter-cluster pod communication, even in overlapping network spaces, while the storage fabric postpones storage binding to optimize data locality.

However, Liqo lacks in flexibility, as the peering process must be performed (*i*) manually from the infrastructure operator, and (*ii*) based on apriori knowledge (i.e., no discovery of resources if provided). For this reason, Liqo has been selected as enabling technology of FLUIDOS, extending the functionalities well beyond such limitations, and, within this context, providing an automated scaling process of the infrastructure and dynamic resource discovery in the continuum.

Cilium Custer Mesh. Cilium Cluster Mesh[3], developed by Isovalent, is an open-source solution that creates a network mesh for seamless communication between pods across multiple Kubernetes clusters. However, it requires all clusters to coordinate IP addresses, as it does not support overlapping IPs or NAT-based address translation, and mandates the use of the Cilium CNI in all clusters, limiting interoperability with other CNIs, or, in general, with multi-owner infrastructures.

Rancher Fleet. Rancher Fleet[4] is a GitOps-based container management engine designed to provide scalability, visibility, and control over Kubernetes clusters. It supports traditional Kubernetes deployment strategies from Git repositories, converting all resources dynamically into Helm charts for consistency and auditability. Fleet uses custom resource definitions (CRDs) and controllers to manage GitOps workflows across single or multiple clusters, enabling centralized application man-

[2] https://submariner.io
[3] https://isovalent.com/labs/cilium-cluster-mesh
[4] https://fleet.rancher.io

agement and high availability configurations. While Fleet offers robust tools for managing multiple clusters from a single interface, it does not unify them into a single logical entity, as clusters remain isolated with no shared network, service, or storage fabric.

2.3.2 Research-oriented projects

This section analyzes research-oriented solutions, which includes both pure research ideas and experimental papers.

Cloudlets. Mobile cloud computing addresses the resource limitations of mobile devices by offloading computational tasks to the cloud, but high WAN latency makes it unsuitable for real-time applications. To overcome this, cloudlets (trusted, resource-rich devices near mobile users [31]) enable low-latency execution by rapidly deploying virtual machines for specific tasks [32]. Cloudlets can operate as ad-hoc networks, dynamically adjusting to node availability, or as elastic virtualized infrastructures that scale resources as needed. Still, the approach lacks of a dedicated network fabric to scale the applicability to multi-cluster environments.

FLEDGE. FLEDGE [33] is a container orchestration system designed for resource-constrained edge devices while maintaining compatibility with Kubernetes. It integrates a modified Virtual Kubelet and a VPN to connect edge devices with Kubernetes clusters, eliminating the need for the CNI layer. The FLEDGE agent, deployed on edge devices, handles both pod networking and namespace configuration.

FLEDGE is a Kubernetes-compatible container orchestrator designed to manage containers on low-resource edge devices. Unlike a standalone Kubernetes cluster, the edge devices do not run a full Kubernetes setup. Instead, they rely on management from a cloud-based Kubernetes cluster. It lacks a proper network fabric needed for broader multi-cluster scenarios (e.g., edge-to-edge).

Decentralized Kubernetes Federation Control Plane. In their position paper [34], L. Larsson et *al.* propose a distributed and decentralized control plane for the Kubernetes federation to support thousands of clusters and disconnected scenarios. The approach replaces the central KubeFed controller with distributed federation controllers in each cluster, enabling local management of federated resources while maintaining global consistency through a shared database of conflict-free replicated data types (CRDTs). The shared CRDT database ensures conflict-free data distri-

bution and allows clusters to redistribute workloads dynamically based on resource availability.

EUCloudEdgeIoT.eu meta-os projects. The meta-OS projects funded by the EUCloudEdgeIoT.eu initiative aim to develop foundational platforms that enable seamless orchestration across the Cloud-Edge-IoT continuum. In addition to FLUIDOS, five other Meta-OS projects are funded under this initiative: *aerOS* [5], *ICOS* [6], *NebulOuS* [7], *NEMO* [8], *NEPHELE* [9]. Each project tackles the challenge of distributed computing and resource management with a different architectural vision and technical focus. The consortium's objective is to tackle the most relevant open issues in the adoption of the continuum from multiple perspectives, and ensure consistent exploitation of these projects' outcomes to help regain European competitiveness in core internet infrastructures.

2.4 Enabling technologies

The following section provides a more in-depth analysis of the enabling technologies for the creation of the computing continuum, as envisioned in the FLUIDOS project. Specifically, Kubernetes (Section 2.4.1) automate the execution and lifecycle management of applications, and Liqo (Section 2.4.2) allows the creation of the liquid computing (a.k.a. computing continuum) abstraction on top of Kubernetes.

2.4.1 Kubernetes

Kubernetes is the de-facto standard open-source system for automating the deployment, scaling, and management of applications in cloud environment. By abstracting the underlying hardware and infrastructure, Kubernetes enables the efficient management of containerized workloads across multiple hosts, facilitating the operation in heterogeneous environments.

[5] https://aeros-project.eu/
[6] https://icos-project.eu/
[7] https://nebulouscloud.eu/
[8] https://meta-os.eu/
[9] https://nephele-project.eu/

The platform organizes containers (i.e., applications) into logical units known as "pods", which represent the smallest deployable units. These pods can be dynamically scheduled across a cluster of machines, and Kubernetes ensures efficient resource utilization by automatically managing service discovery, load balancing, and fault tolerance. Through its native capabilities for horizontal scaling and self-healing, Kubernetes enables systems to maintain a high degree of availability and resilience, even under varying operational conditions. Additionally, Kubernetes employs a declarative model for system configuration and management. Users define the desired state of the system through high-level configuration files, and Kubernetes continuously monitors and adjusts the system to meet these specifications. This declarative approach simplifies the complexity of operating large-scale distributed systems, allowing for more predictable and efficient management of workloads while minimizing manual intervention.

In our vision, each cluster participating in the continuum hosts a distinct Kubernetes cluster. Each Kubernetes control plane would then be responsible of managing the locally deployed applications, ensuring the functionality of the entire system. It is worth mentioning that multiple versions of Kubernetes are available, targeting also low power devices (e.g., k3s).

2.4.2 Liqo

Despite the multiple functionalities offered, Kubernetes has been designed for a cloud environment and does not offer native support for multi-cluster, i.e., interconnecting geographically distributed computing facilities. To this end, Liqo, an open-source platform designed to enable multi-cluster Kubernetes environments, allows for seamless resource sharing and workload distribution across different Kubernetes clusters. By leveraging Liqo, multiple independent clusters can be interconnected, enabling them to exchange computing resources dynamically and operate as a unified system. This facilitates greater flexibility in managing distributed applications, as workloads can be spread across clusters based on resource availability, geographical considerations, or performance requirements.

At the core of Liqo's functionality is its ability to extend Kubernetes' native capabilities to operate efficiently across heterogeneous infrastructures. Through a mechanism called "virtual node peering" (also shortened in *peering*), Liqo integrates

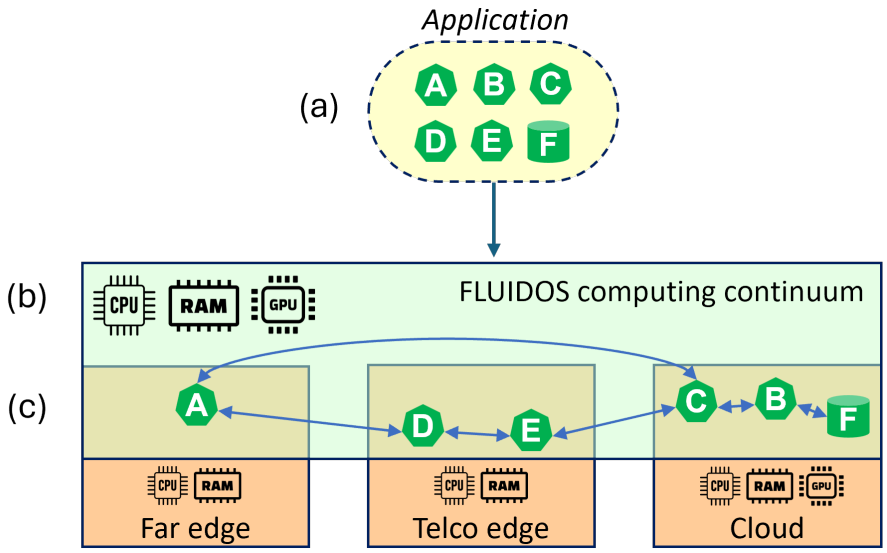


Fig. 2.1 The computing continuum pillars enforced by FLUIDOS.

remote clusters as virtual nodes within a local Kubernetes cluster, enabling the transparent scheduling of workloads on remote clusters without modifying the application's code. This approach ensures that resources, such as computing and storage, can be dynamically provisioned and shared across multiple clusters in real-time, optimizing resource utilization.

Furthermore, Liqo provides an automated mechanism for load balancing and failover across clusters, enhancing the resilience and scalability of Kubernetes-based systems. By abstracting the complexities of multi-cluster management, it allows for more efficient use of distributed resources while maintaining the same declarative management approach found in native Kubernetes environments. in fact, from a control perspective, a cluster A that oversees cluster B and C (*peered* with cluster B and C using the Liqo terminology) can delegate the execution of a given application to cluster B and move it to cluster C if cluster B becomes unreachable or if it is no longer able to guarantee the required Quality of Service (QoS) for the same workload.

2.5 The computing continuum Pillars

At first glance, the computing continuum appears to be a practical reality in the present day, requiring no significant investments in technology or research. For example, numerous software applications are already using multiple components that are set up and run in various places (such as data collection at the far edge, data compilation at the telco edge, and in-depth data analysis in the cloud), which seems to suggest that a computing continuum is already in place. This section outlines three distinctive features we anticipate from the computing continuum, which current methods do not fulfill.

2.5.1 Deployment transparency

When an application composed of various microservices is set up in the current silo-based computing environment, each part must be deliberately configured to be placed in a specific location, such as an edge data center versus the cloud. The location for each component is thus predetermined and fixed; changes to the location of any component are not permitted without initiating a new deployment phase. As a result, any potential dynamic optimization that could be performed during operation is complicated, requiring a comprehensive orchestrator to relocate all necessary components to their new optimal positions, a capability not available with existing technology. In contrast, within the computing continuum, an intent-based interface ensures that each microservice is launched in the most suitable location and allows for dynamic adjustments as needed (see Figure 2.1, part *a*). Therefore, DevOps processes are streamlined, as all services benefit from a unified deployment and control point, while the "magic" of the computing continuum ensures services start in the optimal location based on the requirements of the service and the current state of the infrastructure.

2.5.2 Resource availability transparency

With today's technology, a microservice is restricted to using only the resources within its own cluster. This limitation applies both during normal operations, such as service initiation and during updates, like automatic scaling. As a result, a service might face disruptions due to an inability to access available resources located in

other areas of the continuum, despite their availability. This issue is not particularly significant in cloud data centers, where resource scarcity is unlikely. Even in smaller clusters, which utilize only a fraction of a data center's resources, resizing the cluster by adding or removing worker nodes can easily mitigate the problem. However, the challenge becomes more pronounced with the limited resource pools found at the edge, where typically only a few servers are available. The option to incorporate additional physical resources from nearby nodes is the only way to expand capacity under such constraints.

The computing continuum addresses this challenge by facilitating the creation of a virtual computing space that extends across multiple physical domains (see Figure 2.1, part *b*). This allows a service initiated within this virtual space to utilize resources from across the virtual domain, regardless of their physical locations. Thus, within the computing continuum, a service can scale fluidly based on resource availability across the entire virtual infrastructure. For instance, it could result in one instance running at the telco edge and another in a cloud data center, effectively erasing the traditional, rigid boundaries of clusters.

2.5.3 Communication transparency

The way microservices communicate varies depending on whether they are part of the same communication domain, such as a Kubernetes cluster, or not. For example, communications within a cluster are typically permitted by default, whereas communications from outside the cluster are generally blocked. Moreover, a service that allows internal cluster communications uses different mechanisms (like Kubernetes' *ClusterIP* service) compared to those needed for external communications (such as Kubernetes' *NodePort* or *LoadBalancer* services). This means services need to be specifically set up to interact with each other based on their location, adding complexity to any potential redeployment efforts. It's important to note that certain technologies can help mitigate this issue, for instance, when microservices communicate through message brokers (like Kafka, which utilizes publish/subscribe mechanisms). However, this solution requires all microservices to adopt these specific communication mechanisms, which may not always be feasible due to the limitations of the publish/subscribe model (like potential latency issues) or because some applications don't use this technology and rely on other protocols (such as HTTP or gRPC).

The concept of the computing continuum introduces a solution by creating a virtual cluster that extends over multiple physical clusters, with applications operating within this virtual environment (see Figure 2.1, part *c*). In this setup, a virtual network layer facilitates all microservice communications, ensuring smooth interaction regardless of each microservice's physical location. As a result, communication between services within this virtual space is as straightforward as if they were located in the same physical cluster, eliminating the need for complex and potentially error-prone configuration, regardless of whether they are deployed within the same or in separate clusters.

2.6 REAR protocol

Within the context of the European project FLUIDOS, we argued that the effective utilization of resources within a continuum is contingent upon their recognition and exposure as a unified pool. The dynamicity of the continuum can therefore be fostered by allowing devices to advertise/purchase (possibly) any type of resource, ranging from traditional computing resources (e.g., VMs, slices of Kubernetes clusters), to sensors, actuators, and volumes of data. As a result, devices can dynamically join the continuum, advertising the locally available resources and purchasing remote resources when needed, having access to the entire catalog of possibilities.

This work proposes REAR (REsource Advertisement and Reservation) [15], a novel protocol designed to address the challenges associated with resource advertisement and reservation in the computing continuum. REAR aims to provide a flexible and scalable framework that enables entities within the continuum to advertise their resources and facilitate the reservation of those assets by consumers or applications.

REAR addresses three key objectives. (*i*) *Standardization*, defining common interfaces and messages for resource advertisement and reservation to promote interoperability and compatibility across heterogeneous computing environments. (*ii*) *Efficiency*, to optimize resource allocation and utilization by allowing devices to enrich the resource description including internal energy metrics, latency considerations, and cost models. (*iii*) *Security and Trust*, incorporating mechanisms for authentication, authorization, and secure communication to ensure the integrity and confidentiality of resource transactions.

In the following, we will delve into the design principles, architectural components, and operational workflows of our proposed protocol. Additionally, we will discuss the potential applications and benefits of adopting this protocol in various domains, including edge computing, the Internet of Things (IoT), and cloud computing. By presenting this protocol, we aim to contribute to the ongoing efforts aimed at realizing the full potential of the computing continuum by enabling efficient resource management and utilization.

2.6.1 Motivations

Resource discovery plays a pivotal role in distributed continuum infrastructures to prevent underutilized resources and inefficient allocation policies. In the following, we outline the optimizations enabled by the continuum's resource aggregation and highlight the requirements for REAR.

Support for intent-based allocation

Intelligent, data-driven applications can leverage the continuum to find the optimal placement and satisfy user-specified service level requirements, often referred to as *intents*. Intent-driven orchestration is becoming a popular approach in several scenarios beyond workload management [35], for instance, to express user-level requirements for network configurations [36] or to model network security requirements [37]. Nevertheless, the integration within real-world scenarios is even more challenging due to the complexity of modern system architectures, whether programmable network switches or Kubernetes.

Such enhanced capabilities require approaches to gather information not only on existing (local) resources but also on those offered by other participants in the continuum. This enables the optimal utilization of available resources accounting for requirements beyond traditional computing (i.e., CPU, Memory) such as economics, latency, and security/compliance for the deployment and the lifecycle management of applications. The REAR protocol extends the boundaries of the local compute resources to enable intent-based allocation policies. Moreover, such dynamicity allows users to define the desired application architecture using intents, delegating to the infrastructure the task of retrieving and connecting services to the corresponding data sources.

Support for carbon-aware allocation

Increasing energy demand propelled by recent technological trends including the cloud computing [38, 39] underscore a critical dilemma. While the societal and environmental benefits of computing make its expansion both inevitable and desirable, the sustainability of this growth is questionable due to diminishing returns on hardware efficiency gains [40]. This scenario necessitates the exploration of new paradigms for carbon-aware computing.

A promising decarbonization strategy for a computing continuum advocates for workload scheduling that intelligently shifts computational jobs in space and time to capitalize on locations and periods of lower-carbon electricity availability [41–43]. This location-based approach ensures a genuine reduction in the carbon footprint of a computing continuum, bypassing the pitfalls of market-based strategies by focusing on the true carbon intensity of the grid mix at the time and place of usage. In addition, hardware-embodied emissions can be included in the picture as a way to provide a much more sustainable computing continuum [44, 45]. These are the emissions from the hardware manufacturing phase and they complement the operational emissions described above to provide a more holistic overview of the carbon footprint. Recent advancements in this field [46] demonstrate that it is both possible and essential to include embodied emissions in carbon-aware optimization efforts.

In light of these considerations, the motivation for integrating carbon-aware allocation mechanisms within REAR becomes even more compelling. By embedding such strategies into the fabric of the computing continuum, we not only aim to enhance operational efficiency and collaboration across heterogeneous computing resources but also to pioneer a sustainable approach in such a fluidified cloud-edge continuum.

Support for security features

The REAR protocol is seen as a fundamental component of the computing continuum, facilitating the dynamic (dis)aggregation of assets (resources, data, or capabilities) across various domains. This renders the traditional notion of a static security perimeter obsolete, adhering to the zero-trust approach, where neither asset, provider, or consumer is inherently trusted. Rather, continuous verification, authentication,

authorization, and monitoring of all assets is required [47]. In this context, security is not only an integral component of REAR but is also communicated through it.

In particular, our vision entails mutual authentication and authorization for each party involved in the advertisement and reservation process, enabling authorized access to the offered assets. Given the dynamic nature of such a multi-party scenario, recent advancements in the field suggest a shift towards Decentralised Identifiers [48] and Verifiable Credentials [49] as a lightweight alternative to traditional centralized authentication. Some research endeavors also explore integrating this concept with DLT technology, which can serve as a registration authority and authorization scheme for distributed services [50].

At the same time, the REAR protocol assumes a crucial role in conveying the security attributes delivered by the advertised assets. Let us consider a scenario where a cluster offers a pool of computing resources for others to expand. In this instance, the advertised pool might provide dynamic network policy control, restricting communication solely to services within the pool and refining the network perimeter in real time. Additionally, the advertised features may encompass hardware security capabilities, such as hardware-backed Trusted Execution Environments (TEEs) on compute nodes, to achieve a higher degree of workload confidentiality and integrity (which is particularly relevant for sensitive workloads), and to give tenants the ability to attest the environment (or enclaves) where their workloads run. Furthermore, assets could be equipped with proactive and reactive protection services, such as threat detection and mitigation systems or cyber deception tools, thus providing insights into attackers' tactics and techniques.

2.6.2 Related Work

Reservation protocols play a key role in multi-user applications, networking, and distributed systems, managing access to resources and ensuring efficient and fair resource allocation. The Resource reSerVation Protocol (RSVP) represents a foundational approach in computer networks, designed to manage resource reservations in Quality of Service (QoS) enabled networks for efficient data traffic delivery [51]. MRSVP [52] has extended the functionality to the context of which mobile devices perform reservations based on their current and future locations and RNAP [53], which integrates economic factors into the reservation process. Additionally, RSVP-

TE (Resource Reservation Protocol-Traffic Engineering) introduces traffic engineering capabilities, enabling explicit path establishment for data traffic to optimize network utilization [54]. In our perspective, these protocols primarily address network-centric parameters, overlooking the nuanced multi-dimensionality of computing resources (e.g., CPU, RAM, etc.) and the heterogeneity inherent in modern computing platforms (a.k.a. the As-A-Service model).

In the realm of distributed systems, the Service Negotiation and Acquisition Protocol (SNAP) [55] and subsequent SLA negotiation mechanisms [56] present methodologies for establishing QoS agreements, emphasizing the importance of flexible, bilateral negotiation frameworks. Such advancements illustrate the effort to refine SLA negotiation, catering to the dynamic needs of clients and servers within distributed architectures. Furthermore, authors in [57] also describe a brokering architecture that can make advance resource reservations for SLAs, and also the need to consider security and privacy for managing the distributed services [50].

The advent of 5G technology introduces new dimensions to this landscape, offering telecommunication operators unprecedented opportunities to leverage their network and computing infrastructures [58, 59]. With 5G, the requirements for Edge infrastructure intensify [60], necessitating protocols that support seamless application deployment across diverse Telco providers and facilitate the federation of Operator Platforms [61]. Despite the potential, current proposals face limitations, including (*i*) a lack of resource price discovery mechanisms, (*ii*) limited support for dynamic environments, and (*iii*) a focus on containerized applications that may not fully capture the generality of offered resources/services.

This evolving landscape underscores the imperative for innovative reservation protocols that can address the multifaceted challenges of modern computing environments. Such protocols must not only accommodate the complex interplay of network and computing resources but also adapt to the rapidly changing dynamics introduced by advancements like 5G, thereby ensuring efficient, fair, and economically viable resource allocation in distributed systems.

2.6.3 Architecture

REAR has been designed around the concept of *Node*, i.e., a unique computing environment, under the control of a single administrative entity. The node can be

composed of one or more machines and modeled with a common, extensible set of primitives that hide the underlying details while maintaining the possibility to export the most significant distinctive features (e.g., the availability of specific services, and peculiar HW capabilities). In addition, nodes belonging to the same administrative entities can be logically grouped in *Domains*, allowing for enhanced aggregation policies when advertising available resources. In this respect, a given set of resources can be restricted to be advertised only within the same domain, i.e., only the nodes belonging to the same domain can purchase them. Alternatively, the resources can be made available to all participants in the continuum.

Such a hierarchical infrastructure allows for two different interactions among nodes involved, generically referred to as *consumers* and *providers*, depending on the respective role: (*i*) a *Horizontal* interaction enables the creation of a resource exchange process among peers, which can share their resources and services, or part of them, based upon a set of policies. Horizontal interactions are carried out according to a *peer-to-peer* paradigm, hence without the need for any centralized entity that controls and supervises the entire process. (*ii*) a *Vertical* interaction introduces new concepts such as aggregation and hierarchical scaling into the picture. Third-party brokering entities can provide endpoints that customers can browse and query to obtain aggregated views of the resources available in one (or more) domain. This, in turn, would allow the creation of the "unique pool of resources" enabled by the continuum. It is worth mentioning that such interaction can be recursively implemented to replicate multi-level hierarchical aggregations.

In the REAR protocol, each node has two different components: a *resource importer* and a *resource exporter*. The former is responsible for the discovery of available resources. Since the number (and type) of resources in the continuum infrastructure can be potentially huge, the component is also in charge of filtering the available options based on the data models presented in Section 2.6.4. The latter advertises the node's available resources to the other members of the continuum. This approach is still compliant with the hierarchical model, as the resource exporter can perform the advertisement of the resources of the nodes for which it acts as a broker. In addition, a dedicated *contract manager* component keeps track of the purchased resources in the form of a contract and exposes an endpoint that enables further logic to be implemented to verify the Service Level Agreement (SLA) mentioned in the contract to ensure that it is, in fact, being upheld.

2.6.4 Data Model

The REAR data model outlines the various types of resources advertised in the continuum. This model includes the definition of two terms: *Flavor* and *FlavorType*. The *Flavor* encompasses the set of information shared among all possible resources, while the *FlavorType* is a pointer to another dedicated structure that specifies the unique characteristics of each resource. [10] The complexity of the data model requires a formal, semantic definition of the various components and their relationships. To enable such reasoning, we created two ontologies[11] according to the standard defined by the W3C consortium, characterizing the relationships between K8S entities, the REAR data model, its relationship with Kubernetes, and the various properties of the *FlavorTypes* described later.

Flavor

The *Flavor* data model provides a structured way to represent and manage different kinds of computing resources and it includes all the information that is independent of the chosen *FlavorType*, hence facilitating interoperability and standardization across various systems and platforms, enabling efficient resource advertisement and reservation within the computing continuum. The *Flavor* data model includes the following information:

- *FlavorID*: A unique identifier for the flavor.
- *ProviderID*: A unique identifier for the provider of the flavor, which can be different from the owner in case this flavor is being advertised by a broker.
- *Location*: Information about the location of the flavor described using the triplet <latitude, longitude, altitude>.
- *NetworkPropertyType*: The type of network property ensured by the provider (e.g., 5G, WiFi, Ethernet).
- *Price*: Information about the price of the flavor, including amount, currency, and billing time (e.g., daily, monthly).

[10] https://github.com/fluidos-project/REAR-data-models
[11] https://github.com/fluidos-project/fluidos-ontology

- *Owner*: Information about the owner of the flavor, including domain, node ID, IP address.
- *FlavorType*: A reference to a specific FlavorType schema, allowing for defining details specific to each flavor type.

FlavorType

Currently, five *FlavorTypes* are defined in REAR, which provides flavor-specific data models that describe the computational resources to be purchased. New *FlavorTypes* data models can be added to support more use cases.

K8Slice. It identifies a Kubernetes cluster, capturing both its hardware characteristics and various policies related to its deployment and usage within Kubernetes environments. When a K8Slice flavor is purchased, the remote resources can be seen as a logical extension of the local resources to deploy general-purpose workloads (with Liqo.io). A K8Slice includes the following main information:

- *Characteristics*: Defines the hardware characteristics of the Kubernetes flavor, including CPU, GPU, Memory, Storage, and the number of pods that can be deployed.
- *Properties*: Specifies additional properties of the Kubernetes flavor, including the expected inter-cluster latency, [12] the list of security standards supported (e.g., GDPR, ISO/IEC 27002), and the carbon footprint expressed in terms of embodied and operational emissions.
- *Policy*: Defines policies related to the Kubernetes flavor, including the fact that multiple instances of the same FlavorType can be aggregated into a single entity, or partitioned to obtain only a subset of the resources.

VM. The VM FlavorType provides an option to acquire computing resources in the form of Virtual Machines (VMs). The VM FlavorType shares most of the characteristics of the K8Slice previously described, including the possibility to describe the architecture of the node on which the VM is running as well as information on the

[12] At the time of writing this dissertation, we assume the inter-cluster latency to be computed either based on prior knowledge, or based on network utility tools (e.g., ping), executed by the provider towards the consumer.

operating system on the VM. Despite being similar, it is important to model both FlavorTypes separately as the usage of resources differs. For instance, a K8Slice needs to communicate with the K8S API server, while a VM requires an SSH connection.

Service. The *Service* FlavorType enables providers to advertise services, following the Software-as-a-Service model (SaaS). Since it is almost impossible to provide a generic description suitable for all possible services, the *Service* FlavorType data model follows the same pattern used for the Flavor and FlavorType data models: the *Service* FlavorType provides a high-level description of the Service, including all the specifications that are independent of the actual service, whereas the *ServiceType* details the service-specific characteristics. Specifically, the *Service* FlavorType describes:

- *Name* and *Description*: Respectively, the name and the human-readable description of the service.
- *Tags*: A keyword list that summarizes a service's properties.
- *Plan*: The plan for the service (e.g., Enterprise, Trial).
- *Latency*: The expected latency with the consumer.
- *ServiceType*: The reference to the characteristics of the specific ServiceType. For instance, a *PostgreSQL* ServiceType can include the number of transactions per second or the number of tables that the user can create.

Sensor. The Sensor FlavorType allows for Sensors to be advertised and (possibly) shared among multiple consumers in the continuum. The Sensor FlavorType is characterised by:

- *SensorType*: The type of sensor (e.g., light, humidity).
- *SensorModel* and *SensorManifacturer*: Additional information on the sensor.
- *SamplingRate*: The frequency of the measurements.
- *Accuracy*: The expected accuracy of the measurements.
- *MeasurementUnit* and *SamplingRateUnit*: The unit of measure for both the measurements and the sampling rate.

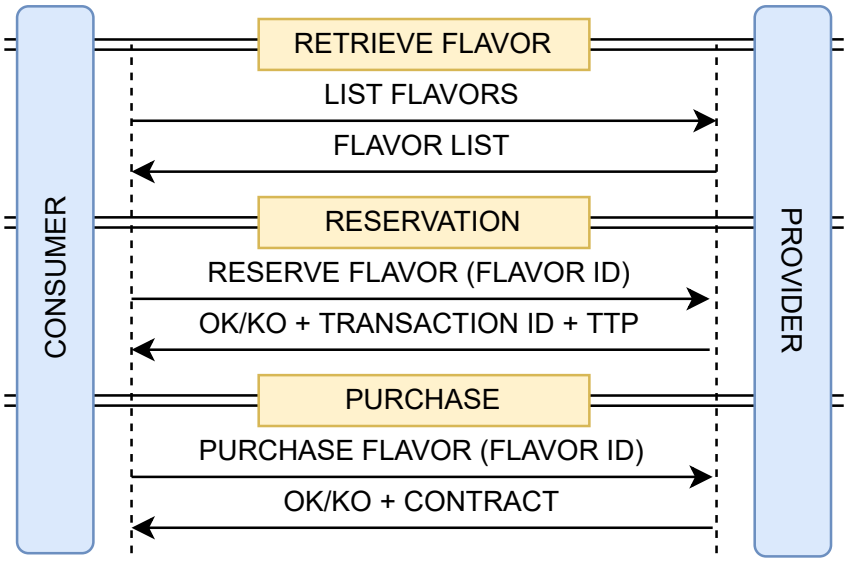


Fig. 2.2 Simplified REAR workflow.

- *AccessType*: How the measurements can be accessed from the consumer (e.g., HTTP, MQTT).

Data. The *Data* FlavorType allows datasets to be shared and advertised between participants of the continuum. This means that REAR enables a form of data sharing as recommended by various EU-funded initiatives such as GAIA-X Data Spaces[13] and IDSA Reference architecture [62]. The Data FlavorType is characterized by:

- *Name* and *Description*: Respectively, the name and the human-readable description of the dataset.
- *Tags*: keyword list summarizing the dataset properties.
- *License*: The license(s) regulating the utilisation of data.
- *Plan*: Dataset usage plan (e.g., Free, Enterprise, Trial).
- *Format*: The format of the dataset (e.g., CSV, TSV, JSON).

2.6.5 REAR Workflow

REAR defines several messages, which can be classified as either *required* or *optional* (an example is depicted in Figure 2.2).

List Flavor (required)

This message is sent by the consumer to probe the available flavors offered by a given provider. Since different FlavorTypes can be offered by a single provider, the consumer can filter out possible resources by specifying the desired characteristics in the *List Flavor* message, using the FlavorType data model described earlier. For example, if the consumer wants to purchase VMs, it can retrieve the list of possible VMs offered by a provider by specifying characteristics of the VM FlavorType, such as 2 CPUs and 4GB of RAM. The provider will then reply with a list of VMs that match the requirements, if any.

The *List Flavor* message thus contains the requested FlavorType with the desired characteristics and some form of identification for the consumer with the tuple $<$*ConsumerID*, *Region*$>$ to allow the provider to generate a (possibly) customized offer for the consumer.

Reserve Flavor (required)

Once the consumer knows the Flavors and their IDs, the *Flavor* reservation process is performed through the *Reserve Flavor* message, sent by the consumer to inform the provider about its willingness to reserve a specific flavor. Specifically, the consumer/provider interaction can be summarized as follows. After the client has collected the list of available *Flavors* offered by the provider, it notifies the intention of reserving a specific flavor by sending the *Reserve Flavor* message, specifying the ID of the *Flavor* to be reserved. To verify the consumer identity, the *Reserve Flavor* message must also include an authentication token that will be then validated by a Trusted third-party authentication and authorization service. Once received, the provider checks if the flavor is still available and if so it replies with a summary of the reservation process including the *TransactionID* and the *Time To Purchase* (TTP), i.e., the time by which the Flavor must be purchased. This allows reserved Flavors

[13] https://gaia-x.eu/what-is-gaia-x/deliverables/

to be released in case either the consumer becomes unreachable, or the subsequent purchase process exceeds a predefined threshold. If the *Flavor* is not available a 404 error message is sent to the consumer.

Purchase Flavor (required)

The *Purchase Flavor* message is sent by the consumer upon receipt of the provider's response during the reservation phase to complete the purchase of an offered flavor. To do so, the consumer sends the *Purchase Flavor* message including the *TransactionID* (obtained with the *Reserve Flavor*) and the identification token to the provider. If authorized, the consumer will then be prompted to a payment service (either external or managed by the provider) and, if successful, a copy of the *Contract* is returned to the consumer, detailing the purchase and the information required to access the purchased resource (e.g., IP address, API endpoint).

Subscribe / Refresh / Withdraw Flavor (optional)

Given that each offered flavor may not be always available, the consumer can notify the intention to receive continuous updates on a specific set of *Flavors* using the *Subscribe Flavor* message. This internally triggers the creation of a stateful channel between the consumer and the provider, which asynchronously sends back updates for any change in the specified *Flavor* using the *Refresh Flavor* message.

If the *Flavor* is no longer available, the provider can tear down the communication channel related to the specific *Flavor* and notify the consumer that the *Flavor* can no longer be purchased using the *Withdraw Flavor* message.

2.6.6 REAR Enrollment and Authentication

The REAR protocol foresees a peer-to-peer interaction between the *customer* (i.e., the node requesting a given resource) and the *provider* (i.e., the node offering the resources). Both parties must be authenticated to prevent unauthorized access to the resources available in the continuum. At its core, REAR relies on a Decentralized identifiers (DIDs) system to provide a globally unique identifier to verify the various actors involved in the REAR message exchange and remove the need for a centralized

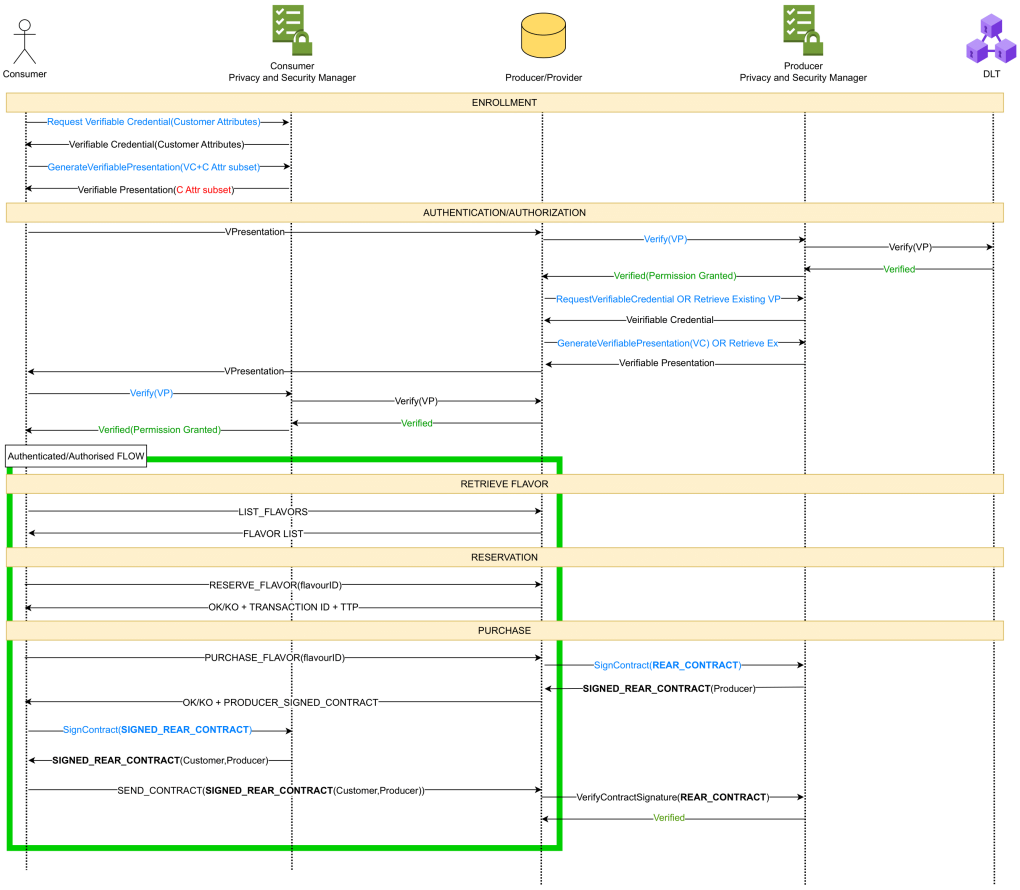


Fig. 2.3 Complete REAR workflow, including authentication.

registry. Each entity in the continuum is represented by a set of attributes, constituting what is called a *Verifiable Credential* (VC).

To participate in the resource exchange process in the continuum, nodes must perform the enrolment towards a Trusted Issuer to obtain a VC. The VC contains all the attributes describing a given node, however, not all of them must be provided when purchasing a given resource. In fact, a given provider might require only a subset of them (e.g., only university and department are required). To this end, a *Verifiable Presentation* (VP) can be requested through the local Privacy and Security Manager (PSM), an entity acting as a Trusted Issuer, including only the subset of requested attributes and preventing information disclosure.

Figure 2.3 details the complete REAR workflow, including also the authentication phase. Upon connecting to a FLUIDOS infrastructure a customer is requested to

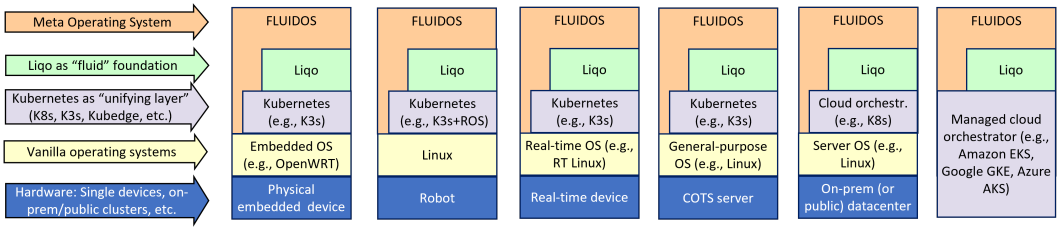


Fig. 2.4 The FLUIDOS software stack.

provide a set of attributes to retrieve the Verifiable Credentials (i.e., the digital proof of identity recognized in FLUIDOS). The Verifiable Credential can then be used to request the related Verifiable Presentation (VP) containing only a subset of the attributes to be used for future resource negotiation.

Before initiating any resource negotiation, the customer and the provider mutually exchange the respective VP to authenticate, following a two-step approach: the customer sends the VP to the provider, which validates it using its own Privacy and Security Manager and a Distributed Ledger Technology (DLT). Upon validating the VP, the provider requests a VP from its Privacy and Security Manager (if not already available in its digital wallet) and sends it to the customer, which, in turn, validates it. Therefore, the Privacy and Security Manager acts as a Policy Decision Point, whereas the REAR protocol acts as a Policy Enforcement Point, generating the contract and granting access to the resources. If this preliminary authentication phase is successful, all the subsequent messages will be authenticated/authorized.

2.7 The FLUIDOS Architecture

The FLUIDOS software stack is composed of four main layers (see Figure 2.4): (*i*) Vanilla operating systems abstract the underlying hardware capabilities. Currently, FLUIDOS is compatible with all major Linux distributions, as well as some embedded OSs to bring the continuum to the far edge of the network. (*ii*) Kubernetes supports workload execution and introduces a uniform layer on top of different infrastructures, regardless of whether they are end-user devices or larger cloud/edge data centers. (*iii*) Liqo, which brings in a multi-cluster abstraction on top of Kubernetes, enables seamless offloading of workloads from one cluster to another. At the same time, it handles all the additional aspects required to make this process transparent

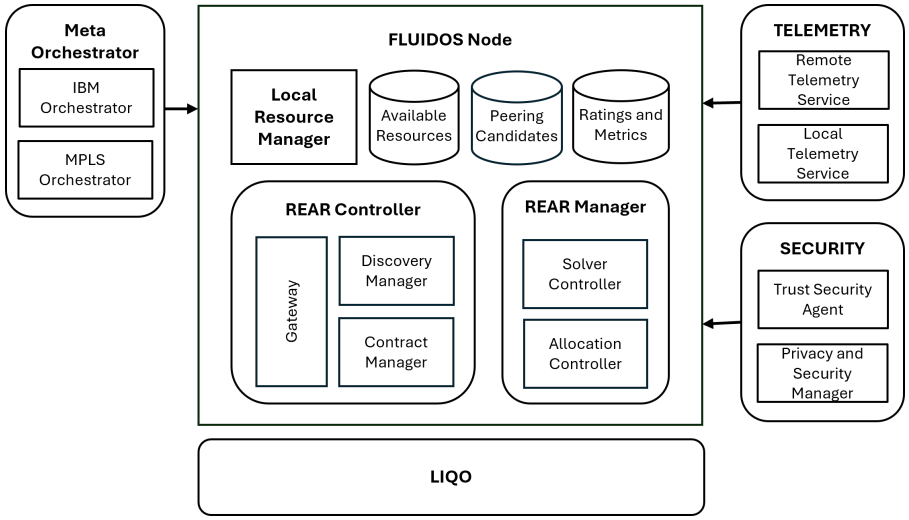


Fig. 2.5 FLUIDOS node core components.

from both the users and the applications' point of view. (*iv*) FLUIDOS implements the full Meta Operating System capabilities.

The core of FLUIDOS is the *node*, which represents the abstraction of a compute resource in the continuum (see Figure 2.5). FLUIDOS nodes can interact *vertically* and/or *horizontally*. The vertical (north/south) interaction introduces new concepts such as aggregation and hierarchical scaling into the picture. A FLUIDOS domain can be created by aggregating multiple nodes, hence exporting a virtual space that is the union of the resources (and services) of the composing nodes. On the other hand, the horizontal (east-west) interaction enables the creation of a FLUIDOS domain among peers, which can share their resources and services, or part of them, based upon a set of policies (e.g., sharing from node X but not from node Y).

FLUIDOS nodes communicate with each other using REAR (REsource Advertisement and Reservation) [15], a novel protocol designed to address the challenges associated with resource advertisement and reservation in the computing continuum. REAR aims to provide a flexible and scalable framework that enables entities within the continuum to advertise their resources and facilitate the reservation of those assets by consumers or applications. Currently, five different classes of assets are supported: Kubernetes slices (i.e., a slice of computing resources of a Kubernetes cluster), VMs, Services (i.e., implementing the so-called Software-as-a-Service), Data, and Sensors. Building upon the negotiation with the REAR protocol, FLUIDOS then enforces the creation and management of the agreed-upon resources in the node. FLUIDOS

nodes rely on the *Network Manager* to discover other FLUIDOS nodes and maintain their metadata. This discovery mechanism ensures that nodes can identify potential resource providers and consumers, allowing them to participate effectively in the REAR protocol for resource advertisement and reservation. The *Network Manager* currently supports two technologies: multicast and Distributed-Hash-Table-based (DHT)[14]. The former targets Local Area Networks (LANs), enabling the discovery of nodes connected to the same physical network, e.g., robots or networks of drones connected to the same 5G cell. In contrast, the latter targets Wide Area Networks (WANs), allowing FLUIDOS to support geographically distributed infrastructures.

All this serves as an enabling technology for the *meta-orchestrator*, the FLUIDOS component in charge of dynamically adjusting the shape of the local cluster, borrowing resources from other FLUIDOS nodes, in response to the submitted workload from the user. Specifically, the meta-orchestrator receives requests to deploy an application in the form of *intents*, which specify the desired characteristics for the execution of the application. The meta-orchestrator will then trigger the REAR protocol to identify the most suitable cluster to satisfy the user intent.

Finally, once resource negotiation is complete, a dedicated *telemetry service* continuously monitors whether the Quality of Experience (QoE) of the exchanged resources aligns with the agreed-upon transaction. Simultaneously, a *security manager* enforces additional security measures to ensure workload isolation and compliance with predefined security policies.

Note: the entire codebase of FLUIDOS is Open Source and available on GitHub.[15]

2.7.1 Research Challenges

In the following, we outline the major research challenges steaming from the continuum, highlighting also how FLUIDOS manages such additional complexities to ensure the computing pillars described in Section 2.5.

Decentralization. The computing continuum inherently involves a highly decentralized infrastructure, where computing resources span across cloud, edge, and IoT devices. As a result, each entity in the continuum must be able to autonomously

[14]DHT repository: https://gitlab.com/pi-lar/neuropil.
[15]https://github.com/fluidos-project

decide how to shape the boundaries of their own continuum, borrowing the most appropriate resources, based on the managed intents. FLUIDOS leverages decentralized orchestration strategies by combining the REAR protocol and the meta-orchestrator to enforce a peer-to-peer-based interaction for resource management. However, a centralized approach is still supported through the vertical interaction previously discussed.

Data privacy and sovereignty. As data flows across heterogeneous domains in the continuum, ensuring data privacy and sovereignty becomes paramount. Regulations such as GDPR demand that data remain under strict control, particularly when it crosses geographic or administrative boundaries. To this end, resources negotiated with the REAR protocol embed information on the compliancy with certain regulations. Therefore, upon purchasing resources from other entities, clients know whether the processing, and most importantly the corresponding data, is compliant with the requested regulations.

Cross-domain authentication and authorization. The continuum involves interactions between multiple administrative domains, in many cases coexisting in the shared resource space, making authentication and authorization complex. This all relates to the problem of multi-tenancy, which is even more exacerbated in the continuum as traditional centralized identity management systems are not suitable. To this end, FLUIDOS introduces a distributed authentication/authorization approach based on Distributed Ledger Technology (DLT) to validate the resources available in the continuum. Furthermore, the control plane of the device sharing computational resources can also configure the appropriate isolation primitives (e.g., resource quota, network and security policies, . . .), based on the underlying orchestration capabilities, to enforce the shared resource and secure the borders of the extended continuum.

Extension beyond Kubernetes. While Kubernetes is a dominant solution for container orchestration in cloud and edge environments, it faces limitations in the resource-constrained and heterogeneous devices prevalent in the continuum, such as IoT nodes. Such devices either do not satisfy the requirements to run even lightweight versions of Kubernetes or do not possess Linux compatibility. To include also far-edge devices in the continuum, FLUIDOS integrates with KubeEdge. In fact, KubeEdge allows to run containerized applications also on resource-constrained devices, natively supporting MQTT and HTTP-based communications.

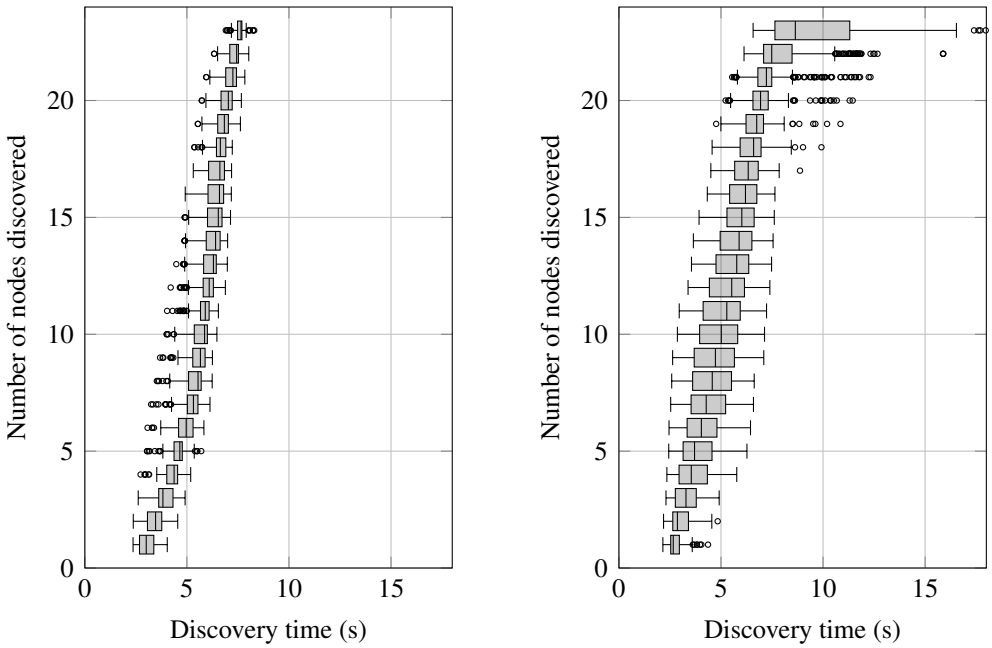
2.8 Experimental evaluation

FLUIDOS streamlines the discovery of available resources in the continuum space and the consequent exploitation. In the following, we first evaluate the time required to identify suitable candidate nodes in the continuum (Section 2.8.1), moving then to the evaluation of the REAR protocol, focusing on the time required to list all the available resources in the continuum (Section 2.8.2). Finally, a potential usage scenario for leveraging the newly acquired resources is presented (Section 2.8.3).

2.8.1 Peer discovery time

The peer discovery time is defined as the time required for a FLUIDOS node to identify all available nodes in the continuum. As described in the previous section (Section 2.7), the discovery process is managed by the *Network Manager*, which supports two discovery modes: multicast-based and Distributed-Hash-Table-based (DHT-based) discovery. The multicast-based allows for Local Area Networks (LANs) discovery, enabling the discovery of nodes connected to the same subnet. This scenario is typical in environments such as industrial premises (e.g., robots) or networks of drones connected to a common 5G antenna. In contrast, the DHT-based extends the discovery capabilities to Wide Area Networks (WANs), allowing FLUIDOS to support geographically distributed infrastructures. In the following, we will focus on the multicast-based discovery process.

The discovery time data is collected by measuring the time required for a single node to identify peers in the FLUIDOS continuum. Figure 2.6 illustrates the relationship between discovery time and the number of participating nodes under two conditions: an ideal communication channel (i.e., no packet loss) and a channel with 10% packet loss (since multicast is implemented using UDP). Starting from a fleet of 24 initial nodes, discovery time is measured as the time required by a single node to discover the first N nodes, where N varies between 1 and 23. The order of discovery is not considered relevant. Without packet loss (Figure 2.6a), a linear correlation emerges, with the shortest discovery time recorded as 3 seconds for a two-node system (i.e., discovering and discovered nodes) and approximately 8 seconds to discover all 23 nodes. Overall, discovery time remains consistent. With packet loss (Figure 2.6b), a shorter discovery time is initially observed due to the reduction in the number of packets to process. However, as more nodes need to be



(a) Discovery time for 20 nodes with no packet loss.

(b) Discovery time for 20 nodes with 10% packet loss.

Fig. 2.6 Network manager discovery times increasing the number of nodes in the continuum.

discovered, the process is increasingly affected by retransmission delays. Despite this, performance degradation remains minimal, with outliers appearing primarily in larger infrastructures. Additionally, FLUIDOS allows nodes to act as intermediate brokers, consolidating resources and reducing the apparent cardinality of the infrastructure.

2.8.2 REAR resource discovery

We here report the empirical evaluation of REAR by means of the testbed created for FLUIDOS[16]. In the testbed, we collect metrics from the three different stages of the REAR protocol, namely, *List Flavors*, *Reserve Flavor*, and *Purchase Flavor* to provide a breakdown of the overhead, varying the number of providers offering flavors of type K8Slice. The *Resource Acquisition* phase represents instead the reference time required to extend the pool of locally available resources in Kubernetes using the Liqo framework.

[16] https://github.com/fluidos-project/node

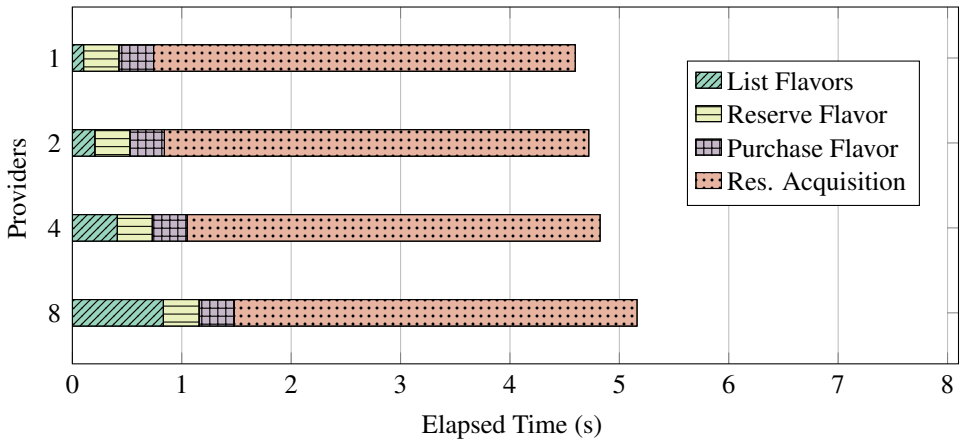
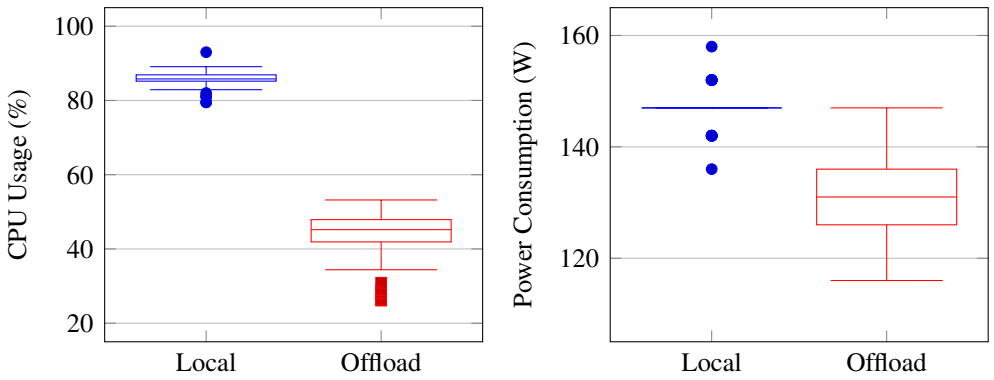


Fig. 2.7 REAR stages' timing vs. time to ready purchased resources.

Figure 2.7 details the elapsed time for a generic consumer before the purchased resources are available and ready to use. In the consumer, the list of REAR-enabled providers is manually configured for the specific test, but it can still be discovered through the *Network Manager*. As we can see, the *List Flavor* message is influenced by the number of available providers, leading to higher latencies when the consumer has an extensive list of available providers. In the worst-case scenario, the consumer must iterate the entire providers' list before identifying a suitable resource, resulting in a linear time complexity $\theta(n)$ w.r.t. the provider list size n. However, the *Reserve Flavor* and *Purchase Flavor* messages maintain stability through the different scenarios. It is worth noticing that despite any fluctuations, the REAR protocol's overhead remains minimal (approximately 30% of the total time in the worst case), and can be further reduced by aggregating resources through third-party brokering and aggregation services to create the "unique pool of resources" enabled by the continuum.

2.8.3 Application offloading

Once FLUIDOS nodes become aware of the surrounding environment in the continuum, enhanced policies can be enforced to select the best candidate to peer with (using FLUIDOS terminology), based on the expected goal. In the following, we bring a possible usage scenario, in which battery-constrained robots might dynamically offload part of the computation to nearby edge nodes or other charging robots.



(a) Robot CPU Usage

(b) Power Consumption

Fig. 2.8 Robot dynamics when only local processing is permitted, and when the offloading is enabled.

If the network in between can sustain the communication, the objective is to alleviate the burden on the onboard processing unit.

Data on the CPU and battery is collected using Prometheus. The battery data, in particular, is obtained by reading the output of the robot's Battery Management System (BMS), which measures the power consumption of the entire system, including sensors and actuators (such as the wheel motors). Figure 2.8 depicts the resource usage for a single robot in case only local onboard processing is possible, and when the FLUIDOS-enabled continuum allows for dynamic discovery of nearby available nodes where to perform the offloading. Specifically, Figure 2.8a represents the CPU usage measured on the robot, whereas Figure 2.8b focuses on the resulting power consumption derived from the computation. As we can see when the offloading is made available the pressure on the CPU is almost halved by moving (in this specific case) the computation needed for motion planning. Still, part of the robot logic required to interact with sensors and actuators cannot be moved and must be executed locally. Although not as drastic, the reduction in CPU usage also results in a reduction in power consumption for the entire robot (approximately 15%), increasing the duration of the battery, and, consequently, the overall operation time. This is because the BMS measures the power usage of the entire system, not just the computational components. As a result, the total power consumption decreased by less than 50%, since only the computing load was reduced, while other components continued to draw power as before.

2.9 Conclusions

This chapter introduced the computing continuum as a solution to the limitations of traditional siloed infrastructures, highlighting the need for seamless integration across heterogeneous computing layers. The proposed FLUIDOS framework, built upon Kubernetes and Liqo, enables deployment, communication, and resource availability transparency, forming the foundation for a decentralized meta-operating system. Through its intent-based resource negotiation and dynamic orchestration capabilities, FLUIDOS enhances the efficiency and flexibility of distributed computing, allowing workloads to be placed optimally based on application-specific requirements. The experimental evaluation demonstrates that this approach not only improves resource utilization but also reduces operational overhead, paving the way for a more adaptive and scalable infrastructure. However, while this work establishes a solid foundation, future research should focus on enhancing security mechanisms, optimizing real-time resource discovery, and integrating AI-driven scheduling strategies. Furthermore, improving interoperability with other cloud and edge frameworks will be essential to fully realize the vision of a unified and dynamic computing continuum.

Chapter 3

Cost-aware Allocation in the Computing Continuum

In the last decade, we experienced a paradigm shift in web application development patterns, moving from huge monolithic frameworks to the agile microservice approach. The strict decoupling of application logic into small, dedicated components enabled substantial benefits for IT companies both in terms of quality of experience provided to the end-user (*QoE*), and cost savings for *DevOps* practices. Although providing enhanced scalability and resiliency upon unexpected disruptive events, such application decoupling also resulted in increased complexity for traditional Cloud orchestration tools like Kubernetes [63] and Hadoop [64, 65]. Specifically, data center scheduling algorithms must ensure at any time to match microservice specifications in terms of SLOs, reserving computing and networking resources for their execution. Customers will then be charged based on the requested amount of resources (CPU, memory, disk, bandwidth, etc.) and the related guarantees in terms of service availability.

The complexity of the scheduling process lies *(i)* in the heterogeneity of microservice resource requirements, and *(ii)* in the additional limitation derived from modern data center architectures. In fact, until recently, major Cloud providers scaled up their computing facilities by building "mega-DCs" with hundreds of thousands of servers and interconnecting them into a wide-area backbone. However, a different scaling strategy has quickly become standard, shifting from "mega-DCs" to using a collection of smaller DCs located within close proximity. This shift is driven by

two pressures: *(i)* the difficulty of siting and provisioning large facilities; and *(ii)* the desire for fault tolerance to survive an outage in a single location [66, 67].

Additionally, recent trends towards Edge, Fog, and Liquid [14] computing solutions favored, even more, the geo-distribution of computing facilities [68, 69] in the attempt to guarantee the most appropriate hosting infrastructure for latency-sensitive applications. With the advent of Edge computing, telecommunication networks, IoT systems, and Smart City/Grid technologies have significantly enhanced their operational efficiency and resilience. These advancements may leverage thousands of distributed computing resources, markedly improving the Quality of Experience (QoE) and system robustness [21].

3.1 Main contributions

In [18], we argue that such Cloud solutions are far from behaving effectively when trivially adapted to the Edge scenario. Indeed, these solutions have been designed for centralized data centers, with guarantees of computing and network resources, and are not designed to identify suitable microservice placement considering their communication patterns. Therefore, they fail to scale on geographically distributed edge-like infrastructures seamlessly, specifically when dealing with nodes that are geographically spread over high-latency WANs [8–10].

Furthermore, the extreme dynamicity of microservice resource usage patterns drastically increases the complexity of the scheduling process (e.g., the workload may vary depending on the number of users connecting to the web application). Nevertheless, it is still possible to roughly differentiate them based on their *expected* execution time: production clusters deploy a huge variety of long-running applications (*LRAs*), long-lived microservices that continue execution for days to months. LRAs are commonly used for stream processing [70–72], web services [73] and machine learning tasks [74–76], and recent work estimated that a substantial portion of production cluster – ranging from 10% up to 50% – is entirely dedicated to LRAs workloads [77, 78]. In comparison, conventional offline batch processing workloads (e.g., Spark and MapReduce jobs) run short-lived tasks that typically finish within minutes or shorter. Long-running applications can typically withstand longer scheduling times, but they require optimal placement, whereas short-running applications are latency-sensitive. Scheduling algorithms must then be able to handle both LRAs

and SRAs, trading off between the different requirements. The scheduling process of such heterogeneous workloads, accounting not only their computing requirements (i.e., CPU, RAM, GPU share, and more) but also the networking requirements (i.e., communication bandwidth), while minimizing the deployment cost is — to the best of our knowledge — still unexplored.

In this work, we argue that adapting Cloud scheduling solutions to the Edge case is not effective and leads to suboptimality in practice. To this end, we present Phare, a decentralized scheduling algorithm that places microservices on geographically distributed infrastructures. Such distributed computing facilities constitute what we refer to as a *federation* (or *continuum*). Each constituent part of the federation (i.e., a cluster) offers (a subset of) its computing resources to the other members of the federation, allowing each individual and possibly autonomous entity to purchase resources when needed, creating a continuum of heterogeneous computing resources [14]. The primary objective of Phare is to optimize the execution of microservices by meeting their computing and communication needs while minimizing deployment costs. To accomplish this, we design a heuristic-based algorithm to solve the NP-Hard scheduling problem, and we evaluate the performance of Phare against Firmament [79], the Kubernetes state-of-the-art scheduling algorithm, on simulated federated infrastructures with hundreds of clusters. Our approach achieves almost a $10\times$ reduction in terms of deployment costs compared to Firmament while always guaranteeing a lower scheduling latency.

3.2 Related work

The problem of scheduling in Cloud computing has been deeply addressed in the last two decades, while only a few, more recent, solutions address the additional challenges that arise in Edge computing. The most adopted solutions for container orchestration, such as Kubernetes [63] and YARN [65], provide generic scheduling algorithms, responsible for placing jobs on the available machines, and have been designed to address a large portion of common use cases while balancing complexity, scheduling latency, and optimality. While such algorithms may effectively solve the scheduling problem in a traditional Cloud environment, characterized by homo-

geneous resources, adapting them to distributed Edge infrastructures may not be trivial.

In literature, jobs are typically classified based on the *expected* execution time: long jobs (LRA, Long Running Application) tend to be latency-insensitive and require near-optimal placement, as they are expected to run for days or even months, whereas short jobs (SRA, Short Running Application) are latency-sensitive, and typically finish within minutes or less. Consequently, especially in production environments, scheduling algorithms must deal with both SRAs and LRAs, providing a trade-off among the above requirements.

The problem of scheduling SRAs has been widely addressed in the literature, leveraging task reordering techniques to prevent head of line blocking [80, 81], and introducing also task bandwidth requirements to cope with the most network demanding tasks [82–84]. Still, inaccurate estimates of job completion time can be difficult to mitigate due to external factors such as data size, network congestion, and resource contention which make expected completion time highly variable.

While most relevant and recent works on Edge Computing focus on SRA scheduling, the problem of scheduling LRA, such as micro-service-based applications, is still overlooked to the best of our knowledge. For this kind of problem, the focus moves from completion time to deployment optimality in terms of the final deployment cost and the efficient usage of both computing and networking resources. The problem has been however widely addressed in the context of data centers since public Cloud computing has emerged as the most promising solution to host companies' IT services. A simple and flexible family of algorithms handles the problem one job per time, i.e., each unscheduled job is first retrieved from a queue and then assigned to a computation unit regardless of the other jobs that are still in the queue [80, 85]. This approach has the limitation of committing early to suboptimal decisions that can prevent the placement of subsequent jobs. To overcome such limitations, some solutions jointly process batches of tasks. For instance, Stratus [86] proposes an algorithm that targets the IaaS (Infrastructure as a Service) scenario; specifically, it aims to maximize the use of the purchased resources by co-allocating tasks onto the same VMs. Quincy [87] introduces the concept of *flow scheduling*, where the problem of job scheduling is converted to an equivalent *min-cost max-flow* problem. Such an approach is further improved by Firmament [79], which achieves the same high-quality deployments but at a much faster scheduling time. Firmament is cur-

rently adopted in widespread Kubernetes clusters and can efficiently minimize the overall application deployment cost while horizontally scaling up to thousands of servers.

Although very promising, all the solutions above have been designed to address a Cloud-like environment and do not account for the additional challenges of an edge infrastructure. In particular, inter-job communication may feature bandwidth requirements that are not trivial to satisfy: a series of new constraints can make such models ineffective, and, nonetheless, the communication requirements may lead to additional inter-cluster network costs based on the final job placement. As we will show in Section 3.5, it can be hard to cope with such additional problems by simply extending/adapting well-established cloud scheduling algorithms.

Motivated by the heterogeneity of resources and, therefore, of constraints that may affect the job placement at the edge of the network, a set of recent works addresses the problem in terms of inter-job dependencies, proposing the so-called *rule-based scheduling* [65, 88, 63]. Domain experts provide a qualitative representation of the *interferences* between jobs, which can be in terms of reciprocal affinity and anti-affinity. Then, the scheduling decision takes into account such information and places jobs accordingly. However, these approaches only take into account qualitative information, failing to capture and optimize quantitative effects on the cluster performance. Medea [77] tries to overcome such limitations by providing a highly expressive model to describe the job requirements; such an algorithm ensures low latency placements and enables cluster owners to specify enhanced placement constraints for long-running containers. Although the improved expressiveness guarantees better scheduling modeling, it still relies on experts to summarize the sophisticated interference.

As a further optimization, network-aware resource management strategies integrate data center topology information and/or application characteristics. [89–92] focus on Integer (Non) Linear Programming (ILP/INLP) models to find the optimal allocation scheme based on an optimization objective. Although able to identify optimal placements, these solutions cannot find a feasible solution within an acceptable time, thus limiting their applicability in production environments. The computational complexity can be reduced by either decomposing the optimization problem into parallel tractable INLP subproblems [93] or limiting the search space to a subset of compute nodes, based on the concept of open subscriber group mode [94] to balance

quality of allocation and convergence time. The high-dimensional search space can be further reduced considering only a subset of the possible requirements, allocating MapReduce tasks [95] or data-parallel distributing deep learning jobs [96] based solely on the network topology, selecting the most promising SmartNIC-Accelerated Server based on the compute demand (i.e., CPU and memory) of microservice-based applications [97], or allocating network intensive tasks on geographically distributed edge-to-cloud infrastructures [98, 99]. Still, few approaches effectively account for computing and networking resources, while minimizing the application deployment cost within a reasonable time.

Finally, in the last couple of years, researchers proposed various approaches to address the various challenges that arise in the Edge computing scenario, such as joint scheduling of computing and networking resources [100, 101], distributed scheduling in multi-provider environments [102, 103], support for mobility [104] and energy efficiency [105]. However, proposed solutions are still in an embryonal stage and far from guaranteeing the same scalability properties as cloud-oriented solutions (e.g., as Firmament does). In our work, we provide an enhanced scheduling model that *(i)* overcomes the limitation of cloud-based approaches by providing both qualitative and quantitative measures of inter-job interactions, and *(ii)* enables a highly scalable algorithm that can quickly schedule complex applications on thousands of nodes.

3.3 System model

We consider a distributed edge infrastructure where resources are grouped into clusters. Potentially, each cluster $v \in \mathcal{N}$ is owned by a different edge provider and participates in what we call cluster federation (see Figure 3.1). Clusters are heterogeneous and may provide different resource capabilities (e.g., centralized data centers, network access base stations, central offices, but also isolated user devices). In this work, we consider capabilities in terms of *computing resources* (e.g., the total amount of CPU and memory available in the cluster), and *communication resources* (i.e., amount of network bandwidth used to communicate with other clusters). Since different types of computing resources experience similarities in terms of provisioning and cost evaluation, we define Γ as the set of all computing resources (e.g., CPU and RAM), and treat every $r \in \Gamma$ jointly, as the subsequent steps need to be evaluated for each one of them. On the other hand, we differentiate

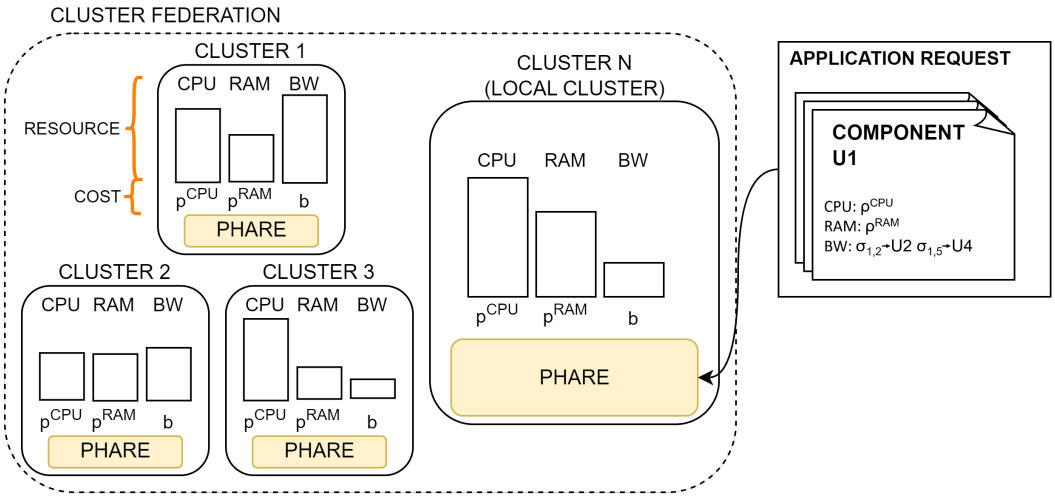


Fig. 3.1 High level overview of the Phare architecture.

the notation for communication resources, as we need to treat them separately in our model. In particular, on cluster $v \in \mathcal{N}$, we denote with $C_v^r \in \mathbb{R}_+$ the budget of computing resource of type r, and with $B_v \in \mathbb{R}_+$ the budget of communication resources.

Requests for deploying applications are issued to edge providers. Each application $i \in \mathcal{I}$ consists of a set of components $\mathcal{M}_i \subseteq \mathcal{M}$, where $\mathcal{M}$ is the set of all possible components. Each component $j \in \mathcal{M}_i$ features resource demands both in terms of computing $\rho_j^r \in \mathbb{R}_+$ (required amount of computing r-resource) and communication with other components $\sigma_{j,k} \in \mathbb{R}_+$ (bandwidth required by j to communicate with component k from the same application). Currently, we assume constant application demands; however, this static allocation can be easily extended, envisioning periodic re-scheduling to include fluctuating demands based on varying customer loads

Edge providers jointly deploy applications across their clusters, thus forming a federated edge infrastructure. Upon receiving the request for deploying an application, the concerned edge provider decides which of the application components should be executed locally (i.e., on its own cluster) and which of them will instead be offloaded to foreign clusters across the federation.

Each type of resource r that is available on a certain cluster features a given price per unit. To preserve generality, we assume that resources may be exposed with different prices to different partners of the federation. We denote with $p_{v,v'}^r \in \mathbb{R}_+$ and

$b_{v,v'} \in \mathbb{R}_+$ respectively the unitary price of computing resource r and communication resources on cluster v as seen by the provider of cluster v'. We denote by

$$x^i_{j,v} \in \{0,1\}, \text{ for } i \in \mathscr{I}, j \in \mathscr{M}_i, v \in \mathscr{N}, \tag{3.1}$$

the decision variable that indicates if component j from application i has been scheduled on cluster v for deployment. When deploying a certain component $j \in \mathscr{M}_i$ on cluster v, edge provider v' experiences a cost given by multiplying the amount of each demanded resource for the unitary price seen on the hosting cluster:

$$\mathscr{C}_{v'}(j,v) = \sum_{r \in \Gamma} \rho^r_j p^r_{v,v'} + \sum_{k \in \mathscr{M}_i} \sigma_{j,k} b_{v,v'} \mathbb{1}_{\{x^i_{k,v} \neq 1\}}. \tag{3.2}$$

Note that the cost $\sigma_{j,k} b_{v,v'}$ due to the communication between components j and k is accounted only if j and k are not deployed on the same cluster.

When scheduling components of application i on available clusters, the Edge Provider seeks cost minimization of the overall deployment, and its decision is subject to the resource constraints of the federated edge infrastructure. We formulate such optimization problem for Edge Provider[1] v' as follows:

$$\min \sum_{j \in \mathscr{M}_i} \sum_{v \in \mathscr{N}} x^i_{j,v} \mathscr{C}(j,v) \tag{3.3a}$$

$$\text{s.t.} \sum_{j \in \mathscr{M}_i} x^i_{j,v} \rho^r_j \leq C^r_v \qquad \forall v \in \mathscr{N}, \forall r \in \Gamma \tag{3.3b}$$

$$\sum_{j \in \mathscr{M}_i} \sum_{k \in \mathscr{M}_i} x^i_{j,v} \sigma_{j,k} \mathbb{1}_{\{x^i_{k,v} \neq 1\}} \leq B_v \qquad \forall v \in \mathscr{N} \tag{3.3c}$$

$$\sum_{v \in \mathscr{N}} x_{j,v} = 1 \qquad \forall j \in \mathscr{M}_i \tag{3.3d}$$

where constraint (3.3b) ensures that the computing budget of every cluster is not exceeded by deployed components, (3.3c) enforces the communication budget over networking demands between components that are deployed on different clusters, while (3.3d) ensures that all components are deployed.

[1] Since our algorithm operates in a decentralized fashion, we formulate the problem from the point of view of a certain edge provider v' and omit the under script $\cdot_{v'}$ for simplicity.

Note that Problem (3.3) is a variant of the $|\Gamma|$-dimensional multi-Knapsack problem with *bin packing* [106], that is, all items must be assigned minimizing a cost function.

Lemma 3.3.1. *Problem* (3.3) *is NP-Hard.*

Proof. To demonstrate the complexity of Problem (3.3), we show that it can be reduced from the Partition Problem [107], which is known to be NP-hard. Given a set of positive integers $a_1, a_2, \ldots, a_n$, the Partition Problem consists of dividing them into two subsets such that the sum of the integers in each subset is equal. We create a simplified instance of Problem (3.3) as follows: consider only two identical clusters v' and v'', each featuring the same budget $C_{v'} = C_{v''} = \frac{1}{2}\sum_{j=1}^{n} a_j$ of a single resource type; assume the unitary price of such resource is 1, i.e., $p_{v'} = p_{v''} = 1$; consider an application with n components, where component j has a 1-dimensional computing demand $\rho_j = a_j$ and no demands in terms of networking. Therefore, the deployment cost of component j numerically coincides with its demand, i.e., $\mathscr{C}(j, v') = \mathscr{C}(j, v'') = a_j$. Note that, if there exists a partition of integers $a_1, a_2, \ldots, a_n$ into two equal-sum subsets $\{S_1, S_2\}$, then there exists a solution to Problem (3.3) (i.e., assign the components corresponding to S_1 to one cluster and those corresponding to S_2 to the other). This solution is also optimal since all the unitary prices are the same. Conversely, solving Problem (3.3) leads to a valid solution to the Partition Problem. Hence solving Problem (3.3) is at least as hard as solving the Partition Problem. □

3.4 PHARE algorithm

This section describes a heuristic we designed to solve the NP-hard Problem (3.3a). We first provide some intuitions of what are the main challenges when scheduling components in distributed constrained infrastructure, and of the main concepts behind the algorithm logic. Then, we describe the algorithm and detail its steps.

3.4.1 Main Challenge

Decentralized allocation policies distribute decision-making among multiple agents, which improves scalability and resilience compared to centralized allocation. However, coordination and achieving global optimization can be challenging; the quality

of the allocation will be discussed in the evaluation section, but it is important to note that coordination among agents plays a crucial role in the scalability of the solution. When an agent receives a request to deploy an application, it can independently perform the allocation process without exchanging information with other agents, which drastically reduces the need for synchronization. However, for informed decisions to be made, clusters must share their status with other members of the federation. A detailed description of the real-world implications of this will be discussed in Section 3.5.7 after the evaluation.

When deciding to schedule a particular component on a given cluster, a key role is played both by how big the component is (i.e., how many cluster resources it demands) and by how much it communicates with other components of the same application. A component that requires a lot of computational resources will be harder to schedule (it has less feasible matches) compared to small components, but this is also true for small components that feature intensive mutual communication (e.g., if the chosen host cluster has not enough bandwidth, the communication with any component placed outside will not occur properly).

Since edge infrastructures are highly scattered and constrained, we argue that it is particularly challenging to jointly satisfy the communication and computational requirements of all application components. Intuitively, the more components are scheduled on available clusters, the harder it becomes to schedule the remaining ones. Therefore, to quickly converge to a feasible placement, the algorithm should prioritize "harder" components, i.e., the ones featuring more stringent constraints (both in terms of computing and communication). With this intuition, we design our algorithm with the idea of guessing a convenient order for placing components, which would (*i*) minimize the chances of unfeasible deployments (quick convergence), and (*ii*) seek cost minimization.

3.4.2 Algorithm Overview

Our heuristic performs the steps in Algorithm 1.

When a request for deploying a new application $i \in \mathscr{I}$ is received, we first evaluate every component $j \in \mathscr{M}_i$ of application i and assign an *importance* metric

Algorithm 1 PHARE scheduling for application i

Require: $\mathscr{M}_i,\ \rho_j^r\ \forall j \in \mathscr{M}_i,\ \sigma_{j,k}\ \forall j,k \in \mathscr{M}_i$

1: **for** $j \in \mathscr{M}_i$ **do**
2: **for** $r \in \Gamma$ **do**
3: Compute importance z_j^r of component j through (3.5)
4: $\mathscr{N}' \leftarrow$ extract from $\mathscr{N}$ the nodes that can host component j
5: **for** $v \in \mathscr{N}'$ **do**
6: Estimate computing affinity $\Phi_{j,v}^r$ through (3.7)
7: Compute coefficient $a_{j,v}^r$ through (3.9)
8: Use a^r to estimate comm. affinity $\Psi_{j,v}^r$ through (3.8)
9: **end for**
10: **end for**
11: Combine per-resource importance into z_j through (3.6)
12: Take $\Phi_{j,v} \leftarrow \min_{r\in\Gamma}(\Phi_{j,v}^r)$, and $\Psi_{j,v} \leftarrow \min_{r\in\Gamma}(\Psi_{j,v}^r)$ (3.6)
13: **end for**
14: $J_S \leftarrow$ sort $\mathscr{M}_i$ by importance z_j descending
15: **for** $j \in J_S$ **do**
16: Schedule j to cluster $v^* \leftarrow \arg\min_{v\in\mathscr{N}} (C(j,v)/(\Phi_{j,v}\Psi_{j,v}))$
17: **end for**

z_j to each of them (Algorithm 1, line 3)[2]. The higher the importance, the more the component is considered "hard to schedule", hence it will get a higher priority in the scheduling process. Details of how we estimate the importance of each component are provided in Section 3.4.3

After evaluating the importance of a component j, an affinity score is computed for each pair (j,v) of component j and feasible cluster $v \in \mathscr{N}'$. The affinity provides an indication of how convenient it is to assign component j to cluster v, with respect to a trade-off between convergence speed and optimality of the final scheduling decision. In particular, two separate affinities $\Phi_{j,v} \in [0,1]$ (*computing affinity* – Section 3.4.4) and $\Psi_{j,v} \in [0,1]$ (*communication affinity* – Section 3.4.5) are estimated and combined (Algorithm 1, lines 6 - 8). We detail how we estimate computing and communication affinities in Sections 3.4.4 and 3.4.5 respectively.

We then compute costs $\mathscr{C}(j,v)$ for every component j and feasible cluster v, i.e., the marginal cost that would be required if j is scheduled on v. Such raw costs are adjusted using the affinity values computed at the previous step, thus obtaining the

[2]In practice, we compute separate values of importance z_j^r for each computing resource r, and then combine them back in Algorithm 1, line 11.

so-called *perceived cost* $\mathscr{C}(j,v)/(\Phi_{j,v}\Psi_{j,v})$: the less the affinity between component j and cluster v, the higher the perceived cost of scheduling j on v.

Finally, the algorithm iterates over components sorted by their importance z_j (descending), i.e., prioritizing those that feature stricter requirements, and assigns each component to the cluster v^* providing the less perceived cost (Algorithm 1, line 16), evaluated as:

$$v^* = \arg\min_{v \in \mathscr{N}'} (C(j,v)/\Phi_{j,v}\Psi_{j,v}). \tag{3.4}$$

In the remainder of this section, we complement the algorithm description by providing details for the missing pieces, namely, how we estimate components *importance* z_j, *computing* and *communication affinities* between components and clusters. Finally, we describe how we deal with multiple computing resources.

3.4.3 Sorting Components by Importance

For each computing resource $r \in \Gamma$, we evaluate the importance of a component j mainly based on its demand ρ_j^r. This value is combined with the demands ρ_k^r of each "neighbor" component k, i.e., all those components that feature some communication constraint with j. By prioritizing components with "heavy" neighbors we increase the probability that such neighbors are scheduled on the same clusters.

Before describing how we compute the importance z_j^r of a component j with respect to resource r, we provide the following definition of *communication factor*.

Definition 1. *(Communication factor $\theta_{j,k}$). Given an application $i \in \mathscr{I}$ and two of its components $j,k \in \mathscr{M}_i$, we define $\theta_{j,k} = \sigma_{j,k}/\max_{j',k' \in \mathscr{M}_i}(\sigma_{j',k'})$ the communication factor between components j and k of application i.*

The communication factor $\theta_{j,k}$ is an indicator of how intense is the communication demand between components j and k. We use this value to weight the contribution of each neighbor of j when estimating the importance z_j^r, as defined below.

Definition 2. *(Importance z_j^r). Given component $j \in \mathscr{M}_i$ of an application $i \in \mathscr{I}$, we define the importance of component j with respect to resource r as*

$$z_j^r = \rho_j^r + \sum_{k \in \mathscr{M}_i \setminus \{j\}} \theta_{j,k} \rho_k^r. \tag{3.5}$$

Algorithm 1 uses z_j^r to sort the application components so that those featuring more stringent deployment constraints are scheduled first. Specifically, given the multi-dimensionality of computing resources, the different values of resource importance are then combined as follows:

$$z_j = \sum_{r \in \Gamma} \frac{z_j^r}{\max_{r \in \Gamma}(\rho_j^r)}. \tag{3.6}$$

Intuitively, components that feature *(i)* high computing demands, *(ii)* neighbors with high computing demands, and *(iii)* high communication demands will be characterized by high importance values. Aside from determining the scheduling order, value z_j^r is also used to compute the affinity between component j and the available clusters, as described in the next section.

3.4.4 Affinity between components and clusters: Computing

The first affinity factor we estimate only takes into account the computing resources of the target cluster, without considering its communication capabilities. To estimate the affinity between component j and cluster v, we first evaluate the quantity $y_j^r - \delta_v^r$, where δ_v^r is the residual computing resource r on cluster v, while y_j^r is the amount of overall computing resource of type r required by j and all its neighbors, i.e., $y_j^r = \rho_j^r + \sum_{k \in \mathscr{M}_i \setminus \{j\}} \rho_k^r \mathbb{1}_{\{\sigma_{j,k} > 0\}}$. We provide an intuitive definition of quantity $y_j^r - \delta_v^r$:

Definition 3. *(Resource scarcity $y_j^r - \delta_v^r$). Let us assume component j is considered for deployment on cluster v. We define the difference between the amount of r-type resource required by j and all its neighbors and those still available on cluster v as resource scarcity $y_j^r - \delta_v^r$.*

Note that when resource scarcity is less than zero, the cluster has enough type-r resources to accommodate j and its whole neighborhood.

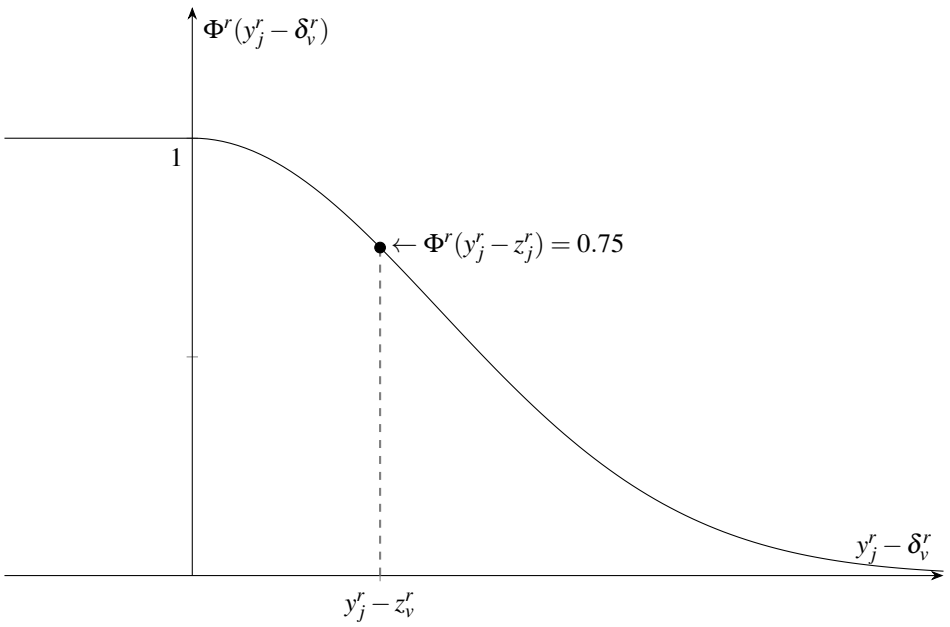


Fig. 3.2 Correlation between the computing affinity Φ^r and the resource scarcity $y_j^r - \delta_v^r$.

The computing affinity is then evaluated based on the resource scarcity $y_j^r - \delta_v^r$ as follows:

$$\Phi_{j,v}^r = \begin{cases} e^{-c\frac{y_j^r - \delta_v^r}{y_j^r - z_j^r}} & \text{if } y_j^r - \delta_v^r > 0, \\ 1 & \text{if } y_j^r - \delta_v^r \leq 0. \end{cases} \tag{3.7}$$

where the coefficient c is used to adjust how fast the affinity decreases with respect to the lack of resources in the cluster.

To understand the rationale behind Equation (3.7) it is helpful to visualize the relationship between $\Phi_{j,v}^r$ and the quantity $y_j^r - \delta_v^r$ (Figure 3.2). We provide an intuition below.

When $y_j^r - \delta_v^r \leq 0$, i.e., cluster v has enough resources to host j and all its neighbors, then the affinity is set to 1 (maximum affinity value). If the resources on v are not enough for hosting j and its whole neighborhood, the affinity starts to drop slowly, until the quantity $y_j^r - \delta_v^r$ reaches a critical value that leads to $\Phi_{j,v}^r = 0.75$; we set coefficient c so that this happens when $y_j^r - \delta_v^r = y_j^r - z_j^r$, i.e., when the residual resources on v are numerically equal to the importance z_j^r of component j.[3] The importance value here is used to estimate the portion of the neighborhood that is more

[3]This is achieved using $c = 0.287682$.

significant for j: if the cluster has enough resources for hosting j and a significant portion of its neighborhood, then the affinity $\Phi^r_{j,v}$ will be higher than 0.75. Finally, after the critical value where $y^r_j - \delta^r_v = y^r_j - z^r_j$ is reached, the affinity $\Phi^r_{j,v}$ starts to drop quickly, with values eventually approaching 0.

3.4.5 Affinity between components and clusters: Communication

To also take into account the networking capabilities of the target cluster, the computing affinity $\Phi^r_{j,v}$ is used in combination with a *communication affinity*. It is important to note that a task j will consume the communication capabilities (e.g., bandwidth) of the host cluster v only if its neighbors have been placed on some external clusters other than v, since otherwise, j would not need v's bandwidth to communicate with them. For this reason, when designing the communication affinity, we seek a mechanism that reduces the affinity of component j and cluster v the more it is difficult to accommodate the communication demands of j, but that has a lower impact if cluster v is large enough for potentially hosting a significant portion of j's neighborhood.

To calculate the communication affinity we first evaluate the quantity $Y_j/\Delta_v - 1$, where Δ_v is the residual communication capacity on cluster v, while Y is the overall communication demands for component j towards all its neighbors, i.e., $Y_j = \sum_{k \in \mathscr{M}_i \setminus \{j\}} \sigma_{j,k}$. Note that the quantity $Y_j/\Delta_v - 1$ is equal to zero when $Y_j = \Delta_v$.

The communication affinity is evaluated as follows

$$\Psi^r_{j,v} = \begin{cases} e^{-a^r(Y_j/\Delta_v - 1)} & \text{if } Y_j/\Delta_v - 1 > 0, \\ 1 & \text{if } Y_j/\Delta_v - 1 \leq 0, \end{cases} \tag{3.8}$$

where coefficient a^r is used to adjust the weight of the communication affinity so that it has a lower impact if the target cluster v has enough r-resources to host a significant portion of j's neighborhood: the higher coefficient a^r is, the more the communication affinity will affect the final solution (see below for details on how we compute coefficient a^r).[4]

[4]Note that $a^r_{j,v}$ is different for each computing resource r; hence, we estimate multiple communication affinities $\Psi^r_{j,v}$, each associated with a certain computing resource r. Section 3.4.6 describes how we deal with multiple computing resources.

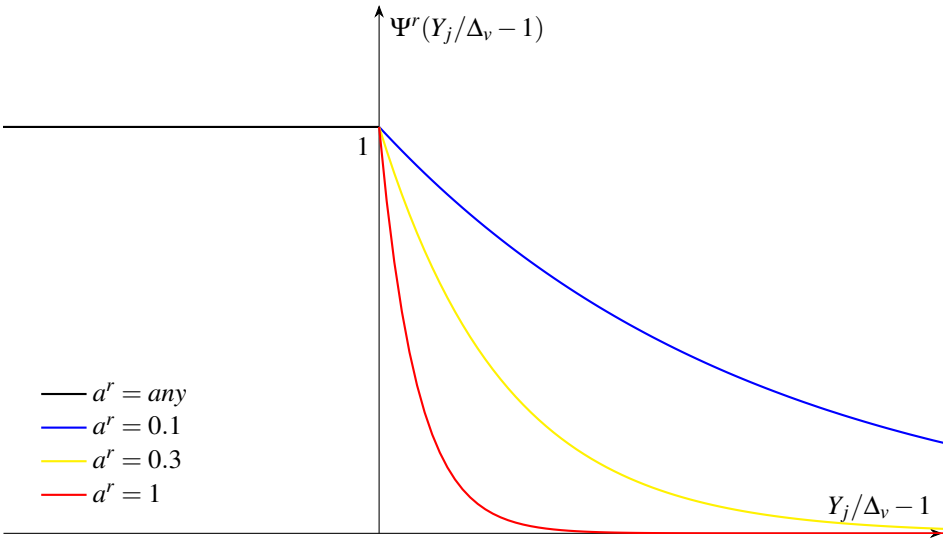


Fig. 3.3 Bandwidth affinity for different values of the coefficient a^r.

The relationship between $\Psi^r_{j,v}$ and the quantity $Y_j/\Delta_v - 1$ is visualized in Figure 3.3. When cluster v has enough communication resources to accommodate all the communication demands of component j, the communication affinity is set to its maximum value 1. If the resources are not enough (i.e., the residual bandwidth is less compared to what j needs to communicate with the other components of the application), $\Psi^r_{j,v}$ drops exponentially with a decreasing factor that is based on the coefficient a^r: for higher values of a^r, the value of $\Psi^r_{j,v}$ drops more quickly.

Coefficient $a^r_{j,v}$. We compute the coefficient $a^r_{j,v}$ so that $\Psi^r_{j,v}$ decreases more slowly the more cluster v is likely to host some neighbors of j. The rationale is that if cluster v has enough computing resources to host a subset of j's neighborhood, then it is unfair to decrease the affinity between v and j based on v's communication capabilities (as j will probably not need them). We empirically compute it as

$$a^r_{j,v} = S\left(4\frac{y^r_j - \delta^r_v}{y^r_j - z^r_j} - 2\right), \tag{3.9}$$

where S is the sigmoid function $S(x) = 1/(e^{-x}+1)$. The relationship between coefficient $a^r_{j,v}$ and the quantity $y^r_j - \delta^r_v$ is visualized in Figure 3.4. If cluster v has enough r-resource compared to the demand of j and its neighborhood, then $a^r_{j,v} \simeq 0$, i.e., the communication affinity will have a negligible impact. If residual resources

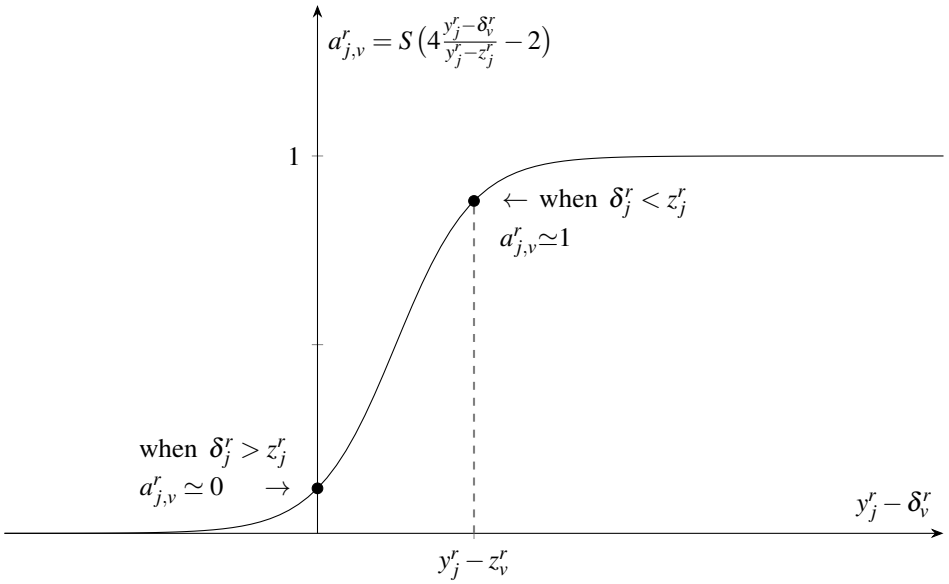


Fig. 3.4 Impact of a^r (determined based on the quantity $y^r_j - \delta^r_v$) with increasing resource scarcity.

are not enough for hosting j and a significant portion of its neighborhood (estimated with z^r_j), then $a^r_{j,v} \simeq 1$ and the communication affinity will have maximum impact.

3.4.6 Dealing with multiple computing resources

Until now, we generalized the concept of computing resources since they experience similarities both in terms of provisioning and cost modeling. Nevertheless, realistic application deployments often feature two or even more conjoined computational constraints; we now describe how multiple resource constraints can be combined within our heuristic.

The values of computing and communication affinity defined respectively at Eq. (3.7) and Eq. (3.8) indicate the confidence in scheduling a component on a given cluster. Multiple computational constraints lead to multiple $\Phi^r_{j,v}$ and $\Psi^r_{j,v}$ per component j, i.e., one for each type of resource $r \in \Gamma$. We generalize the overall computing and communication affinities between component j and cluster v as

$$\Phi_{j,v} = \min_{r \in \Gamma}(\Phi^r_{j,v}), \text{ and } \Psi_{j,v} = \min_{r \in \Gamma}(\Psi^r_{j,v}), \tag{3.10}$$

extracting the minimum affinity value among the existing resources, hence considering the most conservative scenario.

3.5 Experimental results

In this section, we demonstrate the experimental validation of Phare. First, we aim to comprehend the impact of each mechanism composing Phare on job scheduling time and bandwidth usage. Second, we evaluate the scalability properties concerning infrastructure size (Section 3.5.4) and the number of deployed applications (Section 3.5.5). We consider success rate, scheduling time, and the solution cost as key metrics for the subsequent evaluation. Ultimately, our assessment centers on the impact of Phare placement on network congestion within the infrastructure. This is achieved through a thorough analysis of the bandwidth consumption of scheduling solutions (Section 3.5.6). We compare our results against Firmament, the state-of-the-art for microservice placement in cloud infrastructures.

3.5.1 Implementation of the Scheduling Framework

We implemented a prototype version of Phare using the Golang language.[5] We designed the framework to be easily extensible to integrate and test additional scheduling algorithms in the same conditions.

The scheduling framework operates on a simulated environment in which both the infrastructures and the component-based applications are represented as data structures stored in memory. These can be either imported, to replicate specific scenarios, or be randomly generated for testing purposes. Particularly, the random generation is performed through configuration files that define boundaries for each type of resource both for application demands and for infrastructure availability.

The implementation of the scheduler interface for Phare relies on recursion to replicate the algorithm described in Section 3.4. Specifically, the recursive implementation extends the core algorithm, allowing it to probe multiple search paths, until a feasible solution is found, or a predefined timeout is triggered. Due to the

[5]The code is available at https://github.com/liqotech/scheduling.

design of Phare, the recursion tends to prioritize those initial placements that are more likely to lead to a feasible allocation, while jointly minimizing costs.

Using this framework, we also implement Firmament [79], a state-of-the-art scheduler and one of the few ones available in Kubernetes [108]. We aim to demonstrate that well-established Cloud solutions are far from behaving effectively when trivially adapted to the Edge scenario. Firmament exploits flow network representation of the scheduling problem to identify suitable placements by means of Flowlessly[6], an efficient minimum-cost-maximum-flow decision problem solver. Our implementation of Firmament first translates the internal representation of both the infrastructure and the applications to the corresponding flow network (according to the specification provided in [79, 109, 110]), then calls the Flowlessly C++ library to solve the associated minimization problem.

Performance enhancement and additional features. Phare has been designed to seek early the most promising steps in the recursive process, but still the worst-case complexity can be estimated as $\mathcal{O}(N*J)$, where N is the number of clusters and J the number of components of the application to be scheduled. Such a worst-case scenario requires the simultaneous occurrence of numerous factors including huge application size and massive, almost-saturated, infrastructures. Although such a scenario is theoretically possible, providers usually prevent the saturation of their infrastructure for resilience reasons; still, the worst-case complexity can be reduced to $\mathcal{O}(M*J)$, with M being only some of the N feasible clusters, as empirical evaluations have shown that the scheduling solution is always found within the first $M = 10$ clusters, or not found at all. Consequently, we improve the sorting algorithm used to rank the most promising clusters accordingly: in particular, we use a modified version of Heap Sort that runs the Selection Sort only for the first M clusters. This reduces its complexity from $\mathcal{O}(N \log N)$ to $\mathcal{O}(N)$ as $M << N$.

Additional implemented features target the deployment on real systems: *(i)* define thresholds both for computational and network resource usage to prevent saturation, *(ii)* constraint the placement of a subset of the application components onto specific clusters to replicate given execution requirements, and *(iii)* define shared network links between clusters.

[6] https://github.com/ICGog/Flowlessly

Table 3.1 Infrastructure setup.

	Cluster type A	**Cluster type B**
Number of vCPU:	300 – 500 cores	4 – 32 cores
Cost of vCPU:	0.15 – 0.4 $\$/core$	0.15 – 0.4 $\$/core$
Available RAM:	256 – 1024 GB	8 – 64 GB
Cost of RAM:	0.01 – 0.06 $\$/GB$	0.01 – 0.06 $\$/GB$
Bandwidth:	1 – 10 Gb/s	0.1 – 2 Gb/s
Cost bandwidth:	0.05 – 0.2 $\$/Gb$	0.05 – 0.2 $\$/Gb$

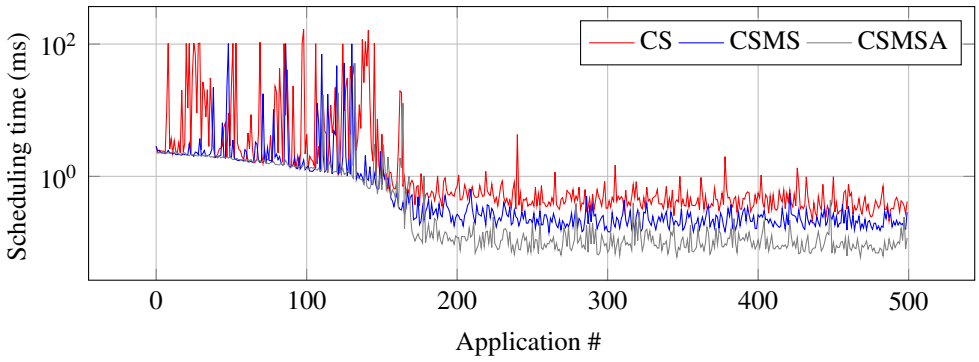
3.5.2 Experiment setup

In our tests, we simulate random edge infrastructure topologies of different sizes, with the number of clusters ranging between 50 and 1000. Each cluster in the infrastructure features a predetermined random amount of CPU cores and available memory. These clusters are linked together with virtual connections, i.e., they logically form a full mesh topology. Each connection is characterized by the available network bandwidth for inter-cluster communication. Moreover, each resource features a given cost expressed in $\$/unit$, properly sized to match major Cloud Provider resource costs (Table 3.1 summarizes the main values concerning the infrastructure configuration).[7]

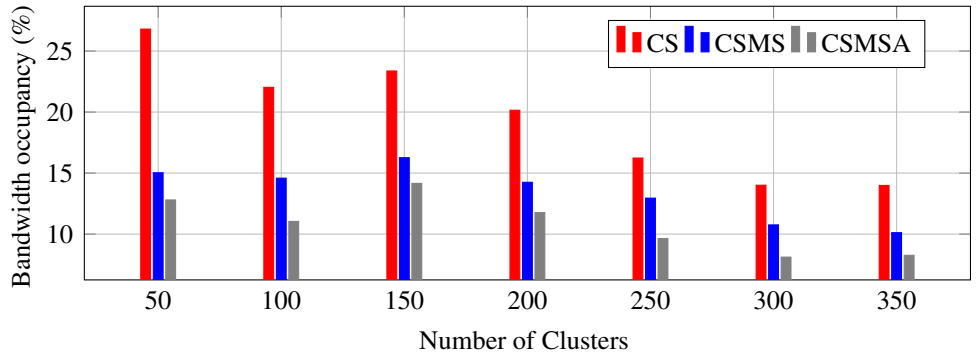
We use a sample 10-tier microservices application called Online Boutique[8] for our simulated workload. The workload reflects common patterns and challenges in distributed systems, thereby providing a credible and realistic benchmark for the use case applications in edge computing environments. To define our workload accurately, we first monitored the CPU, RAM, and bandwidth usage of the microservices and recorded the resource demands (our findings are detailed in Table 3.2). To account for the unpredictable randomness of the infrastructures and simulated workload, for each of our tests, we run multiple simulations (referred to as *simulation samples*) with varying random configurations. This was necessary to obtain statistically significant data.

[7] https://cloud.google.com/compute/all-pricing
[8] https://github.com/GoogleCloudPlatform/microservices-demo

(a) Scheduling time reduction.



(b) Breakdown of the scheduling time for the 50-cluster infrastructure.



(c) Average bandwidth occupancy of the links connecting the clusters.

Fig. 3.5 Evaluation of the heuristic components in PHARE.

Table 3.2 Workload setup.

Application size:	10 microservices
Microservice CPU requests:	0.2 – 1 vCPU
Microservice RAM demand:	0.1 – 0.5 GB
Microservice connectivity ratio:	35%
Microservice bandwidth requirements:	150 – 1500 kB/s

3.5.3 Evaluation of the heuristic components

In this section, we assess the effectiveness of the intuitions behind the proposed algorithm in determining proper placement for microservice-based applications. In practice, we evaluate the benefits that each main building block pieces of Phare bring to the scheduling tasks. We deploy 500 applications (modeled based on Table 3.2) to random infrastructures, with the number of clusters ranging between 50 and 350 (each modeled based on Table 3.1, column B).

For this set of experiments, we use a Cost-based Scheduling (CS) algorithm as our baseline. This algorithm ranks clusters based on their unitary resource costs and places the i-th microservice on the cluster with the lowest cost. We assess the effectiveness of sorting microservice based on component importance z (Definition 2) comparing CS with an extended version named Cost-based Scheduling with Microservice Sort (CSMS) that implements the sorting mechanism described in Section 3.4.3. Additionally, we evaluate the impact of introducing our affinity mechanism (described in Sections 3.4.4 and 3.4.5) by means of the Cost-based Scheduling with Microservice Sort and Affinities (CSMSA). Notice that CSMSA is equivalent to our final proposed heuristic Phare.

Figure 3.5a shows the scheduling time reduction of the proposed enhancements against the baseline. Specifically, sorting based on importance z enables the scheduling algorithm to first place the most demanding microservices (i.e., the ones with the most stringent requirements), moving to the less demanding ones afterward. Such an approach leads to a significant reduction in the average scheduling time (between 65% and 78%), also shortening the time required to identify unfeasible placements. The affinity mechanism further improves the scheduling time: according to resource availability, it effectively weights inter-component dependencies and leads to better

mapping of application components onto infrastructure clusters. Indeed, CSMSA features a scheduling time reduction between 76% and 88% in our experiments.

Additionally, Figure 3.5b shows in which condition the proposed enhancements contribute the most to the reduction of the scheduling time. It details the measured scheduling time for each of the 500 applications in the 50-cluster infrastructure. It is important to note that the 50-cluster infrastructure cannot accommodate all the applications that have been submitted (only around 150 applications will be scheduled successfully, while the rest will not be scheduled due to insufficient resources). The plot shows that in situations where there is a shortage of resources, both CSMS and CSMSA introduce huge improvements on the baseline, visibly reducing the scheduling time by effectively identifying unfeasible placements quickly; moreover when there are enough resources to accommodate all the applications (i.e., less than 150 applications), the proposed scheduling mechanisms leads to faster scheduling times on average (spikes of 100 ms are less frequent when using CSMS, and even almost disappear with CSMSA).

Finally, the proposed mechanisms improve not only in terms of scheduling time but also in the quality of the final placement. We evaluate this in terms of bandwidth occupancy of the links connecting the clusters: Figure 3.5c shows that the full algorithm (CSMSA) consistently outperforms other configurations and never exceeds 15% of bandwidth usage.

3.5.4 Scalability on infrastructure size

We now compare our algorithm against Firmament [79]. First, we evaluate the behavior of the two algorithms when scaling horizontally on the infrastructure size. Specifically, we run Phare and Firmament with infrastructures of multiple sizes, in order to study how the algorithms behave *(i)* when few (possibly saturated) clusters are available, i.e., limited scheduling options, and *(ii)* when many clusters are available (up to 1000 for our study case), with a large number of scheduling options to be evaluated.

Figure 3.6a shows the fraction of successfully scheduled applications out of the complete set. An application is accounted as scheduled only if all its microservices have been successfully placed. Small infrastructures do not have enough resources to accommodate all the applications, hence infrastructures with less than 150 clusters

experience a success rate lower than 1. On the other hand, larger infrastructures can host all the applications, hence a success rate is always equal to 1.

Results report a similar success rate for Phare and Firmament, i.e., both approaches consolidate almost the same amount of applications for each topology configuration. However, we experience a huge difference between the two algorithms in terms of both scheduling time and deployment cost. Figure 3.6b represents the average time needed to fully schedule each of the 10K applications (the time needed to determine any unfeasible placement is accounted for as well). Phare largely outperforms Firmament, featuring less than 70 ms average scheduling time in large infrastructures, and even sub-millisecond scheduling time when operating on constrained topologies with few clusters. Conversely, Firmament experiences very high scheduling delays, especially on small infrastructures. This is because of its inability to quickly identify unfeasible application deployments, which is rare in cloud environments: by design, it accounts for scheduling only a subset of microservices per application at a time when the infrastructure is resource-constrained; this leads to multiple calls to the underlying solving algorithm.

Interestingly, the small scheduling time required by Phare w.r.t. Firmament does not have a negative impact on the experienced scheduling cost. Figure 3.6c depicts the cumulative deployment cost of all the applications; not-scheduled applications contribute with a cost of 0 to the deployment cost. We can identify two trends throughout the test: *(i)* when considering small infrastructures with few federated clusters (less than 150), introducing additional clusters leads to a huge growth in the total deployment cost, as a considerable amount of new applications can be accommodated (as seen in Figure 3.6a); *(ii)* infrastructure with more than 150 clusters have enough resources to host all the applications, hence the total deployment cost slightly decreases as the number of clusters — and the placement options — grow. In practice, the huge deployment cost gap between Phare and Firmament (up to $10\times$) is related to the cost of inter-microservice communication. Specifically, Firmament accounts for network bandwidth dependencies between application components but fails to correctly weight their cost effectively w.r.t. computing costs in a highly scattered topology. On the other hand, the network-aware placement of Phare can drastically reduce the bandwidth occupancy among different clusters and the associated networking costs. As a reference, the two algorithms experience similar costs for computing resources in this setting (result not shown). This result highlights

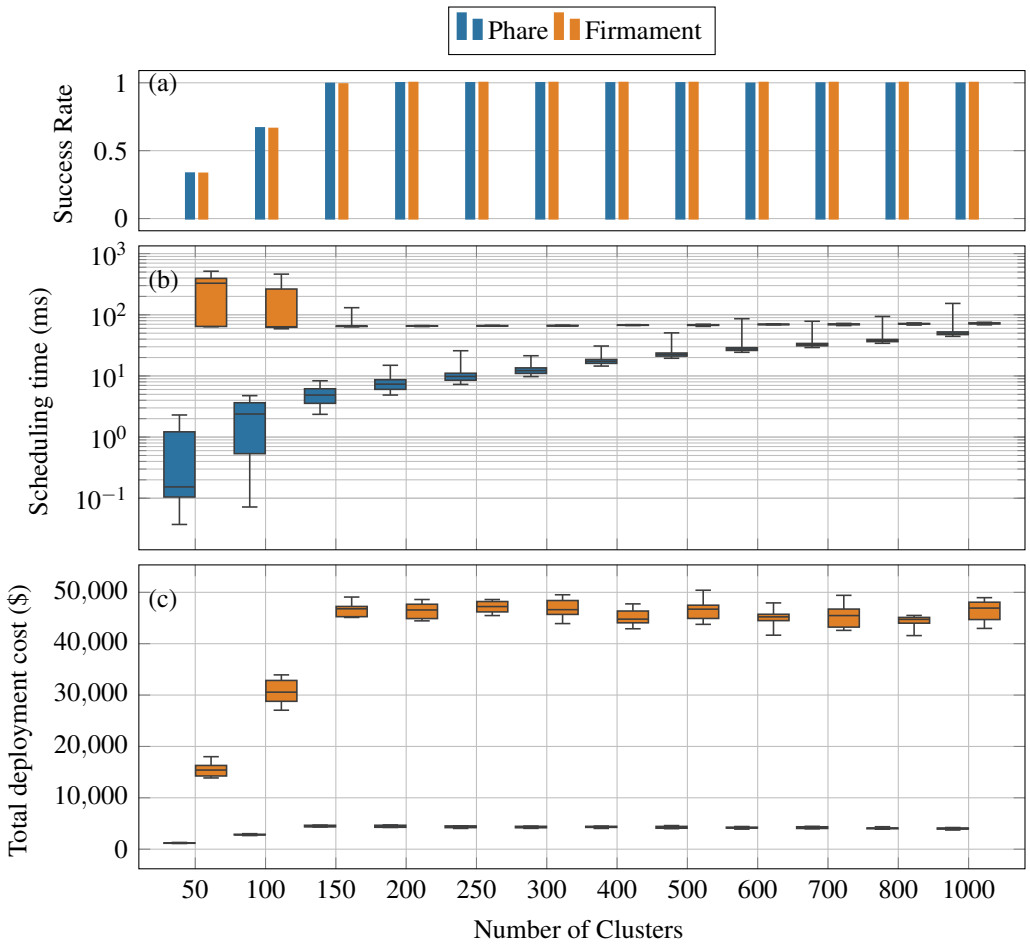


Fig. 3.6 Scheduling success rate, scheduling time, and experienced costs for Phare and Firmament with infrastructures of variable sizes.

the difficulties of adapting a Cloud-based algorithm to the Edge scenario, where network resources are far as uniform and optimally sized as they are in data centers.

3.5.5 Scalability on number of applications

In the previous set of tests, we identified the 100-cluster infrastructure as one of the most challenging, due to the limited amount of available resources to fulfill the demands of all the applications. Analyzing closely such a scenario it is possible to understand the performance of the algorithms both when the infrastructure is almost saturated — and the possible placements are limited — but also when there are no

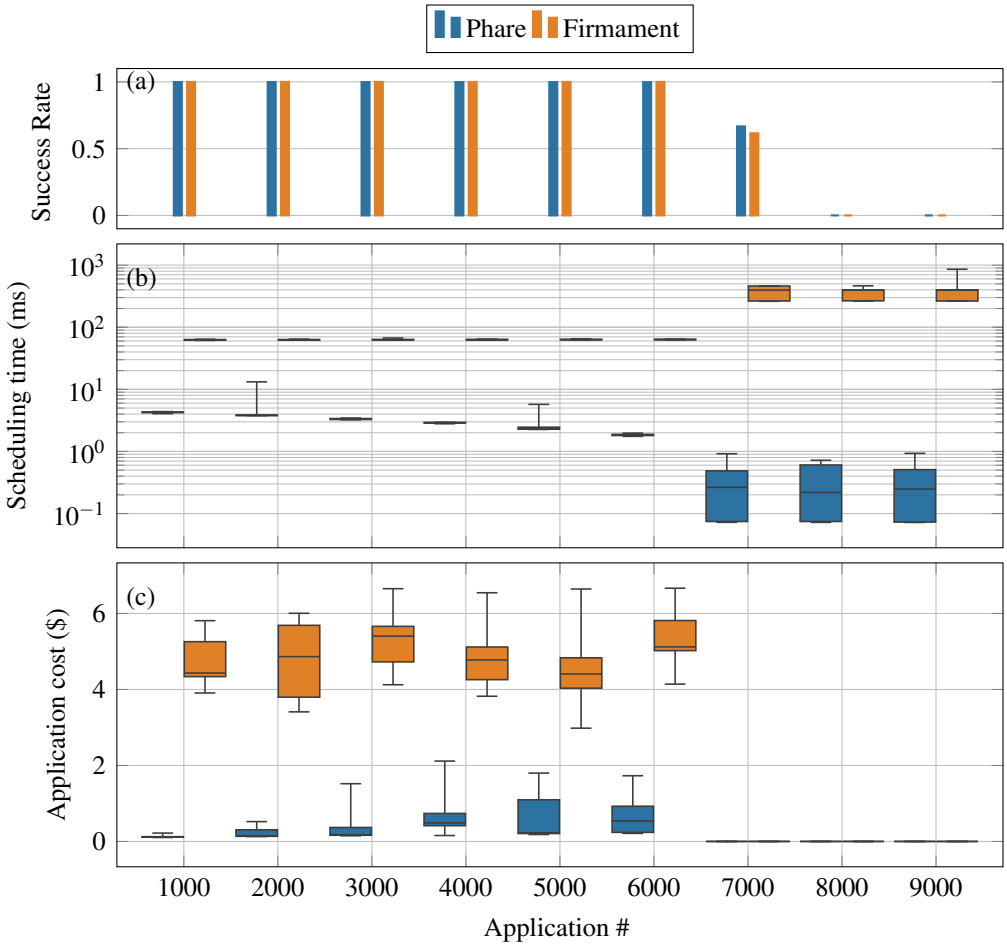
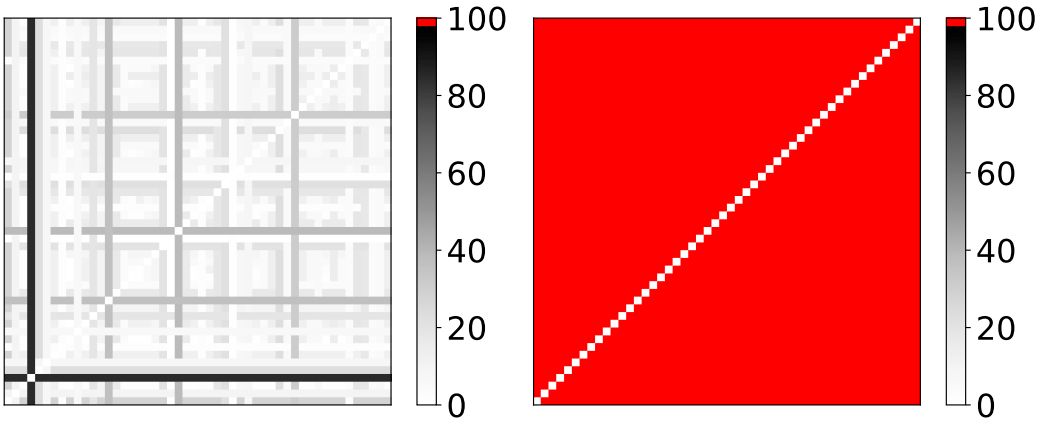


Fig. 3.7 Scheduling success rate, scheduling time, and solution costs for Phare and Firmament with infrastructures of 100 clusters.

more available resources in the infrastructure and the algorithm must quickly detect an unfeasible application placement.

Figure 3.7a breaks down the scheduling success rate for the complete set of applications. As for the previous case, one application is accounted for as scheduled only if all its microservices have been successfully placed. The infrastructure does not have enough resources to accommodate all the applications, in fact, only the first 6K applications are always properly scheduled across all the simulation runs. Phare achieves a slightly higher success rate in the 7K-set of applications, scheduling a few more applications with respect to Firmament thanks to more accurate management of the available resources.



(a) Phare. (b) Firmament.

Fig. 3.8 Inter-cluster congestion matrix describing the percentage of network bandwidth usage between two clusters.

As for the previous test, the two algorithms heavily differ in measured scheduling time: Figure 3.7b reports the observed scheduling time distribution for the i_{th} application. Phare outperforms Firmament being able not only to converge to a suitable scheduling solution in almost-saturated infrastructures but also to quickly detect unfeasible solutions when the amount of available resources is not enough to host the remaining applications. Conversely, Firmament experiences some performance drop with saturated infrastructures because of the same design issue mentioned in the previous section.

Finally, the two algorithms experience again similar costs for what concerns computing resources (not shown), but provide a huge gap when including the cost of networking resources (Figure 3.7c). Indeed, Firmament behaves poorly when network resources are not uniform and networking costs become relevant, which is definitely the case in multi-cloud environments or, in general, for highly distributed infrastructures. In this case, we observe that Phare reduces deployment costs by more than 5 times overall compared to Firmament.

3.5.6 Bandwidth consumption

We now evaluate the benefits of the communication-aware placement of Phare, thus breaking down the bandwidth consumption of the links between clusters.

For this specific set of tests, we use a 50-cluster infrastructure (dimensioned according to Table 3.1, column A). Figure 3.8 depicts the measured network pressure on the link between clusters; specifically, they represent the congestion matrix $N * N$, where N is the number of clusters of the federation and each cell (i, j) represents the percentage of network bandwidth usage between cluster i and j. Moreover, each cell value is represented on a gray-scale color code, except the ones in red that exceed 100% network usage (i.e. the target placement requires more bandwidth than the available amount). In these extreme settings, Firmament even fails to find placements that are feasible from a networking perspective. On the other hand, Phare is able to intelligently distribute the applications across the federation, properly aggregating within the same clusters those microservices that feature mutual dependencies.

3.5.7 Real-world implementation considerations

While our evaluation primarily focuses on the algorithm's performance within simulated environments, we acknowledge the importance of applicability and challenges in real-world settings. Besides scalability, there exist several critical areas to consider, which we briefly analyze below.

Integration with existing systems. Seamless integration with existing infrastructure and cloud/edge computing platforms is crucial for the adoption of any new technology. To this end, we anticipate the potential for integrating Phare in Kubernetes by replacing the default scheduling algorithm with our solution.

Forming a federation. In a federation, computing resources are shared between clusters, allowing each entity to access additional resources if needed. This resource sharing can be achieved through the multi-cloud functionalities provided by Liqo, which is already being used as an enabling technology in the European project FLUIDOS to create distributed continuum infrastructures. We envision a two-stage allocation process for microservice-based applications: (*i*) The FLUIDOS meta-orchestrator takes care of reshaping the continuum boundaries based on user-defined intents to identify the most suitable providers and, (*ii*) Phare then assumes the provisioned infrastructure to be static for the allocation problem. Currently, these two stages are completely independent; however, we will further investigate the possibility of triggering the FLUIDOS meta-orchestrator, based on the availability of resources seen and managed by Phare..

Practical performance metrics. Phare requires updated information on the clusters participating in the federation, including CPU, memory, and bandwidth usage. A real-world implementation should thus carefully balance the resolution of such data and the additional overhead required to process it. Specifically, whereas a peer-to-peer interaction to retrieve cluster usage metrics might be suitable for small federations, it does not scale with the number of clusters. Therefore, a central metrics aggregation point might be a suitable solution for a real implementation.

Privacy and security. Privacy and security are vital for ensuring that workloads are executed correctly in a distributed infrastructure. There are two major issues related to job allocation that must be addressed: First, the device hosting the container execution must provide a secure environment. This can be achieved by using the functionalities offered by the container runtime, Kubernetes, and Liqo. Second, the allocation process must take into account customer constraints. It is crucial to ensure that devices are authenticated through trusted parties to prevent allocation in insecure sites.

By addressing these considerations, we aim to bridge the gap between theoretical research and practical application, ensuring that our algorithm not only excels in simulated tests but also meets the demands of real-world deployment.

3.6 Conclusions

Compared to the Cloud scenario, scattered and constrained clusters in Edge infrastructure pose non-trivial challenges for computing and bandwidth resource scheduling, which can be hardly addressed by adapting traditional data center techniques.

In this work, we proposed a new approach to multi-cloud application scheduling, called Phare, that takes into account the communication and computing requirements of the entire application graph and leverages the capabilities of the underlying infrastructure to optimize resource usage. Phare is based on a heuristic to speed up the placement of the application while minimizing the overall application deployment cost. We compared Phare against Firmament, a state-of-the-art scheduler largely employed in cloud infrastructures, performing a series of tests over infrastructures of various sizes. Our results show that, despite featuring a similar scheduling success rate, Phare largely outperforms Firmament in terms of scheduling time, being able to

quickly identify suitable placements even for large infrastructures. Modeling real major provider resource costs, we show that networking costs can be a major factor in highly distributed infrastructures, and assess Phare benefits over Firmament with up to 10× deployment cost reduction scheduling 10K applications on 1K clusters. Furthermore, our tests on network congestion show that Phare can effectively reduce the bandwidth usage between clusters, resulting in more efficient resource usage and better overall performance.

Overall, our results demonstrate the effectiveness of Phare in optimizing microservice application scheduling on multi-cloud highly-distributed infrastructures and suggest that it can be a valuable tool for organizations that rely on scattered edge resources to run their applications.

Chapter 4

Energy-aware orchestration in the Computing Continuum

The widespread adoption of cloud computing technologies enabled a notable consolidation of computational resources, which resulted in unprecedented service agility and massive cost and energy savings [111, 112]. In recent years, the IT landscape has evolved further, with myriads of sensors and Internet of Things (IoT) devices being installed in homes, industrial premises, and public spaces [113]. The edge computing paradigm has been proposed as an approach to reduce computational latency (as processing happens closer to data sources) and bandwidth usage while improving privacy and reliability.

The *computing continuum* has been introduced as a viable solution to integrate cloud, edge, and IoT sensors while providing guarantees in Quality of Experience (QoE) even for computationally-intensive latency-sensitive applications [114–117]. However, as of today, most edge computing installations consist of servers deployed on premises that process data, which is further elaborated in remote data centers. This top-down approach, in line with the original cloud computing paradigm, simply extends the cloud data center approach to a local point of presence, failing to integrate with the edge substrate and to leverage the unused processing capacity.

In the following, we first present a preliminary evaluation of the energy consumption benefits provided by the computing continuum (Section 4.1). This is followed by a detailed description of the proposed distributed energy-aware allocation framework tailored for the continuum (Section 4.2).

4.1 Energy assessment of the Computing Continuum

In this first section, we validate the proposed continuum use case, emphasizing the possible benefits for the computing infrastructure in terms of overall power consumption.

This work, based upon novel technologies that enable an initial implementation of the computing continuum [14], extensively evaluates the possible benefits of such a paradigm shift in our university production and educational environment. These initial results can be easily generalized to cover a real enterprise department, in which each layer (i.e., Cloud, Fog, Edge, etc.) can leverage the resource continuum to actively execute the desired applications. However, such an approach requires an extension of both the concept of cloud and edge. First, the cloud is no longer perceived as the centralized computing power, located in the core of the network, but it rather generalizes to any set of servers that run cloud orchestration frameworks (i.e., including also *private* cloud). Second, the edge is further extended to include also end-user devices (e.g., laptops, mobile phones, etc.). Therefore, from the end-user perspective, applications can either be executed locally on end-user devices or in the continuum, based on the application QoE requirements and the specific goal of the infrastructure (e.g., reduce latency, power consumption, etc.).

4.1.1 Main contributions

This innovative approach to edge-to-cloud management and orchestration raises many additional challenges to cope with the dynamicity and heterogeneity of computing devices. Still, it has the potential to bring the same benefits of the cloud also to the edge of the network, e.g., with respect to power consumption. In fact, this work [19] presents a preliminary assessment of the potential energy savings when the *fluid* technology is used. This is achieved by analyzing a real use case in the computing continuum, namely a University lab that can benefit from the increased flexibility in running user applications on all the available devices. To achieve this goal, two additional minor contributions can be also envisioned: (*i*) we introduce a new definition of performance requirement, to describe the application QoE within the heterogeneous devices of the continuum, and (*ii*) we define a possible methodology that enables the shift from current desktop applications to the fluid containerized environment.

4.1.2 Related works

The computing continuum, as an emerging concept, has been intensely discussed in recent literature as a promising solution to allow further evolutions on many "smart" systems (e.g., Smart-cities, Smart-grid, and Smart-industry) [114, 118, 119]. In fact, the Cloud-to-Edge resource continuum guarantees low-latency communications with sensors and actuators, and high-performance computing (HPC) for further data analysis and forecasting. Function-as-a-Service (FaaS) showed some promising results, leveraging the heterogeneous resource continuum: function calls can be executed in the different sections of the resource continuum depending on the individual requirements. Although this work focuses on lightweight virtualization (i.e., containers), FaaS experiences similar requirements, because, as it happens with containers, computing resources must be granted to function call execution. Pilot-Edge [117] introduced a FaaS interface for application-level tasks on a resource continuum with heterogeneous devices. Delta [115] further extended the problem representation, introducing dynamically evolving estimates of function execution times to determine the most appropriate location. Still, both solutions focused on maximizing the performance for function call execution, neglecting possible implications in the infrastructure power consumption, which, instead, has been intensively investigated in [120–122]. However, they primarily focused on reducing the power consumption for battery-constrained end-user devices, not accounting for the additional energy requirements on servers. In our case, instead, we overcome such a narrow evaluation and encompass all the different devices of the continuum in the evaluation of the energy-aware application placement.

The Cloud-to-Edge continuum is a very dense and complex scenario. Resources can be considered unlimited at the core level (i.e., Cloud), whilst they become scarce at the Edge. Thus, simulation represents a way to achieve an early-stage evaluation before moving to real-world (and more expensive and complex) testbeds. Abreu, David Perez, *et al.* [123] proposed a comparative analysis of three of the most promising simulators for the Cloud-to-Fog continuum. Specifically, iFogSim [124], CloudSimSDN [125] (both extensions of the well-known CloudSim [126]) and YAFS [127] guarantee enhanced flexibility in the definition of both the infrastructure and the workloads, while providing empirical results also in terms of infrastructure power consumption. Still, none of the above simulators can provide an experimental evaluation to correctly model the heterogeneity of devices that may be part of the

computing continuum, as the only perceived difference between Edge and Cloud is strictly related to the amount of available computing power, and not to the performance of a specific device. Our evaluation is - to the best of our knowledge - the first to properly account for and simulate the heterogeneity of computing resources in the continuum while providing insights on the infrastructure power consumption.

4.1.3 Device characterization

University campuses may include a wide variety of devices, such as servers, laptops, desktop computers, and, lastly, low-end devices like Raspberry PIs. They differ not only in terms of the number of available computing resources (i.e., number of CPU cores) but also in terms of performance per core, as different CPU architectures may provide very different computing power (e.g., one Raspberry ARM Cortex core is significantly less powerful than a high-end Intel i7 core). Therefore, traditional performance metrics, usually expressed in terms of number of CPU cores reserved, need to be replaced with a CPU-independent mechanism, which can guarantee that the same amount of work is done despite any difference in CPU architectures.

We used the Passmark score[1] to compare the performance among different CPU models, which represents a free alternative to the costly SPEC[2] suite. At the same time, it avoids the known limitations of MIPS benchmarks, which focus on the number of instructions per second and do not capture the complex set of conditions affecting the CPU performance (e.g., core turbo boost, thermal throttling, etc.). In fact, the Passmark score mimics the behavior of several common applications, including complex mathematical calculations involving compression, encryption, and physics simulations, to effectively evaluate the performance of a given platform. It is widely adopted by the computing community and has shown consistent results even across different OS [128, 129]. For our evaluation, the Passmark tool has been containerized using Docker for two reasons: (*i*) Docker is widely used as virtualization technology in cloud solutions, allowing us to replicate the execution environment for a generic workload, and (*ii*) it offers the possibility to restrain the Passmark execution on a subset of computing resources (e.g., on N CPU cores), hence allowing us to map the "traditional" representation of computing resources into the new Passmark metric. Specifically, the Passmark container has been executed by

[1] https://www.passmark.com/.
[2] https://www.spec.org/benchmarks.html.

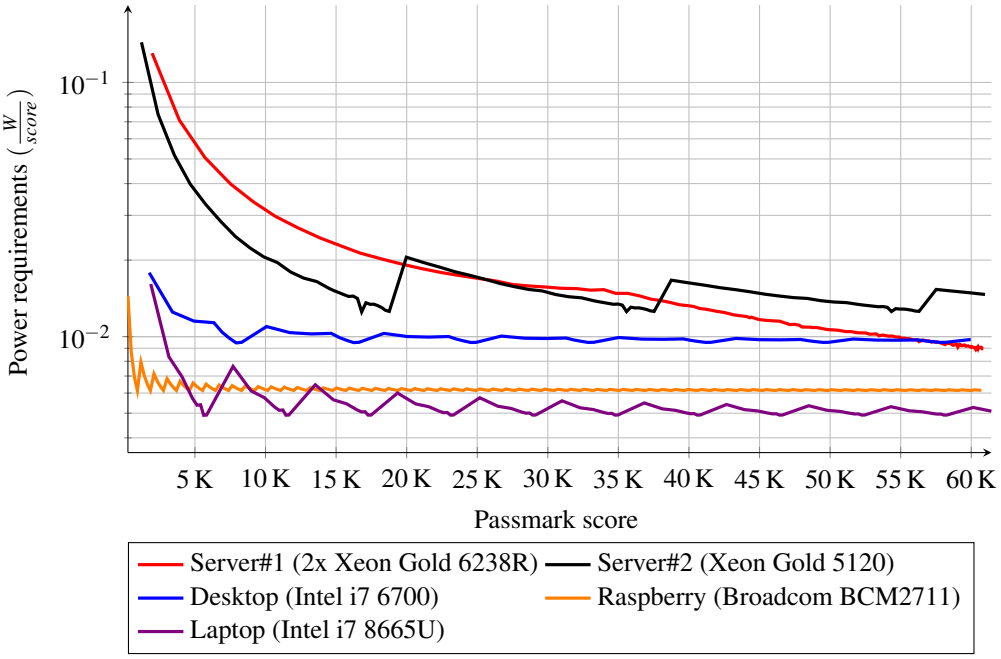


Fig. 4.1 Power requirements of different devices, computed as the ratio between the consumed energy and the measured performance.

linearly increasing the number of CPU cores assigned and collecting the related value of Passmark score. The proposed performance metric enables the characterization of the desired workload in terms of CPU resources (e.g., N vCPU) required on any given platform to deliver the same workload (e.g., M Passmark score).

In addition to performance, different CPU architectures heavily differ in power consumption, as some chips are designed for power efficiency (e.g., Raspberry PIs), whereas others privilege processing performance (e.g., servers). As a consequence, this work needs first to establish a common ground to compare performance and power consumption across different CPUs. Power consumption metrics are then evaluated to correlate device power requirements with delivered performance. Specifically, these metrics are collected using a smart plug, connected directly to the wall outlet, monitoring the total power consumption of the device, while progressively increasing the number of CPU cores assigned to the Passmark application.

Results in Fig. 4.1 correlate the performance of different devices (hence, different CPUs) in terms of energy required to deliver a single Passmark (the lower, the better). Due to the different levels of performance achievable by each CPU, a new CPU was added to the plot when the previous ones become 100% loaded, hence resulting in

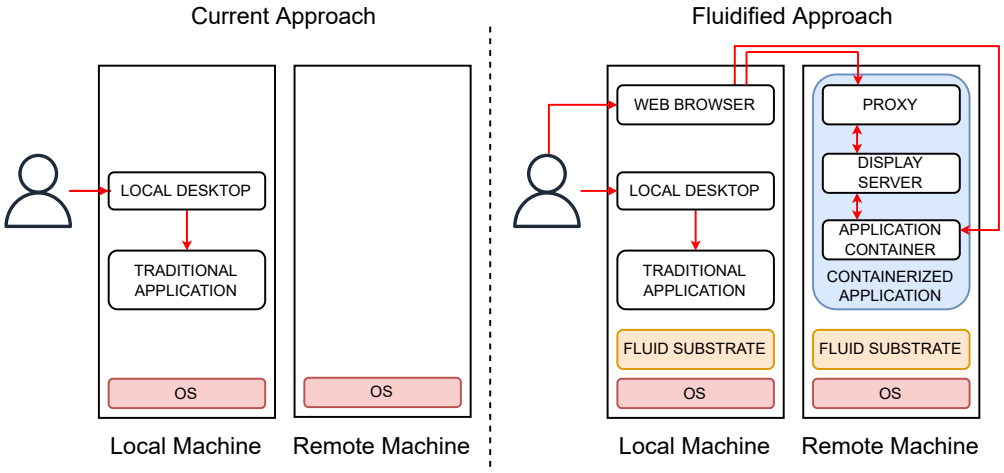


Fig. 4.2 Components involved when running a user application with the traditional approach vs using a fluidified approach.

the 'saw tooth' shape of the figure. As shown in the figure, it is not always possible to identify the most efficient device, as such efficiency may be extremely variable, depending on the applied workload.

4.1.4 Application characterization and containerization

This work relies upon the novel concept of fluid workload distribution, which fundamentally changes the way applications are perceived and executed, in particular when referring to most of the current desktop-oriented applications. This new paradigm defines a computing continuum of resources, in which applications are no longer executed solely on the particular device where they have been requested by the user. Instead, applications can be started on the most convenient device within the infrastructure, as it happens with the cloud computing paradigm. This paradigm shift requires not only a proper infrastructure, able to provide benefits in the given scenario, but it could potentially involve a substantial re-design of the application to allow seamless interaction for the end user.

Compared to the current monolithic approach (Figure 4.2), in which the end user directly interacts either with the application through the *local desktop* or with the remote server, such as running remote virtual desktops (i.e., Virtual Desktop Infrastructure – VDI) through a generic orchestrator, the fluidified approach heavily relies on containerization to run the same workloads, while maintaining the same

interaction with the end-user. Specifically, our fluid applications are composed of three layers:

1. *Application container*: the traditional, monolithic process now running in a containerized environment.
2. *Display server*: it captures all the graphical input/output of the application and makes them available to the remote user, without any modification to the running application. This is currently implemented as a VNC server.
3. *Web proxy*: it enables the interactions between a client and the display server through the HTTP/HTTPS protocols, hence through a web browser, which simplifies the user experience. Technically, it transforms WebSockets (RFC 6455), and the subsequent TCP bi-directional traffic, into the VNC protocol spoken by the Display server.

Although this layered approach enables a seamless shift of current applications towards the fluid approach, it could add some performance drawbacks, caused by the graphical processing performed both on the end-user and application side. A more recent trend in application development foresees the strict decoupling of application logic and content visualization (i.e., in a cloud-native client-server fashion). The fluidified approach would massively benefit from such a design choice, removing the unnecessary overhead caused by the proxy and display server layers and, additionally, with the possibility to share the same back-end replica between multiple front-ends.

Recently, our university has adopted a similar solution to deliver the Programming Fundamentals exam to the enrolled students.[3] Multiple containerized replicas of the PyCharm IDE are executed on private servers, and students were able to interact with the IDE using only a *web interface*, directly connected to the remote instance of the application through a noVNC server. This work considers a further evolution of such an approach, leveraging the computing continuum to distribute the applications across the entire infrastructure, according to specific execution constraints (i.e., power consumption).

In addition to the *web interface*, such a distributed system requires additional computing resources also on the server side for the control plane logic to ensure the

[3] https://medium.com/the-liqo-blog/liqo-in-production-at-turin-polytechnic-20ed71dca475.

correct execution. Specifically, the *orchestrator* logic can be shared among multiple devices (i.e, the control plane can be executed on a small subset of machines), whereas an additional layer of *fluid substrate* must be introduced to guarantee the dynamic infrastructure re-configuration.

4.1.5 Experimental evaluation

This section presents a preliminary assessment of the potential energy savings of the cloudified approach when used in a realistic use case. Specifically, devices are modeled, based on the performance and power consumption metrics described in Section 4.1.3, to better assess the heterogeneity of computing resources.

The purpose of this evaluation is to investigate the possible benefits of the cloudified approach, enabled by the consolidation of the computing resources in the continuum. In this respect, we implemented a scheduling algorithm based on a brute-force exhaustive search, which guarantees the capability to find the best job placement with respect to the overall power consumption.

Experiment Setup

The computing continuum is evaluated against the "traditional" workload execution, referred to as *baseline*, in which applications are executed directly on the end-user terminals or, in case of lack of computing resources, on the dedicated set of servers that are statically configured on purpose. While in the case of the local execution, the energy consumption is purely due to the reference application (plus the power required to run the physical machine), measurements with the remote execution always account for the web interface to connect to the remote application and the additional requirements of the containerized workload (this also applies in case of "traditional" workload execution on remote servers). Instead, the OS server-side overhead is always present, as well as the requirements of a generic orchestrator (i.e., Kubernetes in our case), whereas the additional fluid substrate is present only in the continuum approach.

The requirements of the fluidified approach (detailed in Section 4.1.4) are summarized in Figure 4.3 in terms of processing power. Particularly, [B] identifies processing components present in the baseline ([B*] applies only if static server

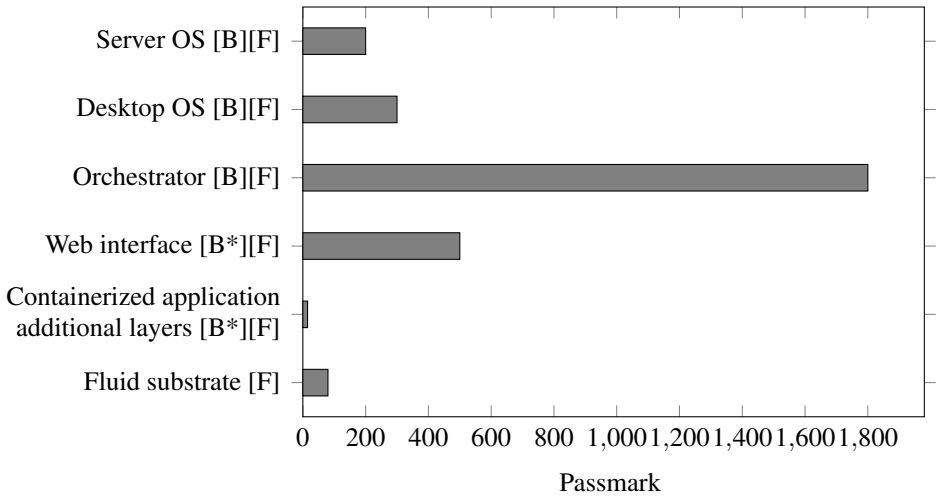


Fig. 4.3 Passmark requirements of the different components of the fluid solution.

remotization is present), while [F] highlights components that must be considered in the fluid approach. The web interface overhead is evaluated by measuring the Google Chrome CPU requirements to connect to the remote instance. Specifically, the resource usage of the client front-end never exceeds 500 Passmark, corresponding to the 7% of the CPU in our desktop (as a reference, the desktop OS settles at around 300 Passmark, ≈4% CPU). Instead, the overhead for the orchestrator can be evaluated by averaging the computing requirements on a given period and results to be about 1800 Passmark (≈1% of CPU usage of the server). Finally, the fluid substrate includes the requirements for the Liqo control plane logic,[4] the selected lightweight framework to create and manage the continuum infrastructure, which requires ≈80 Passmark.

Fluid workload distribution

We modeled an infrastructure based on a realistic university environment, which is composed of 15 user terminals (respectively, 5 desktop computers, 5 laptops, and 5 Raspberry PIs, the same depicted in Figure 4.1), to replicate the possible interactions of the end-users with the infrastructure, and 2 servers (server#1 in the same figure), to provide additional computing power and to cope with more demanding applications. We simulate 15 users, each one submitting the same workloads to the infrastructure (simulating end-user requests to execute applications), which are modeled based

[4] https://liqo.io.

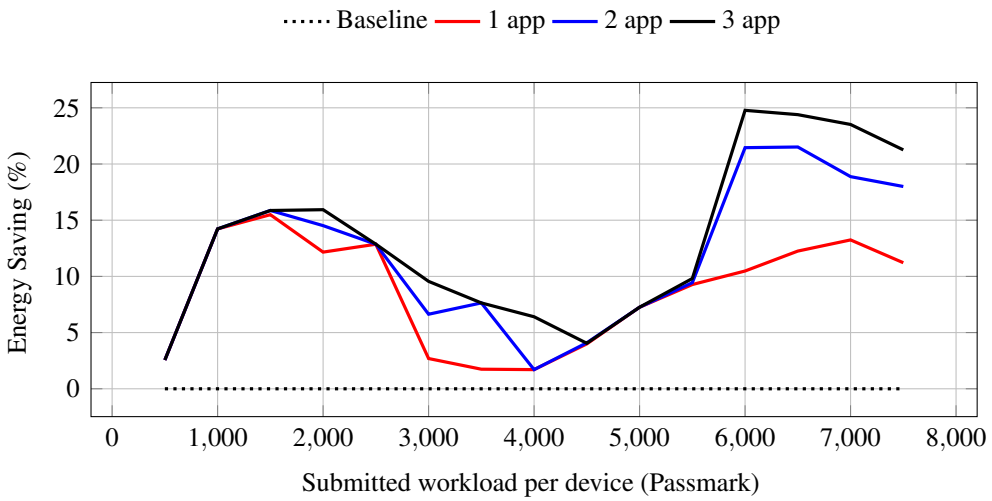


Fig. 4.4 Power consumption saving with respect to the baseline increasing the submitted workload.

on the Passmark described in the previous section. Specifically, we increased the workload from 500 up to 7500 Passmark.[5]

Results in Figure 4.4 compare the overall power consumption of the computing continuum against the baseline. Specifically, the continuum is evaluated in three different configurations (i.e., 1-2-3 app/user) representing the number of applications deployed by each user (hence, requested on each device), while the value on the x axis represents the cumulative workload deployed for every device (e.g., $x = 2000$ accounts for 1 application of 2000 Passmark, 2 applications of 1000 each or 3 applications of 666 each). The proposed simulation raises two considerations: (*i*) The computing continuum is always able to guarantee a non-negligible reduction in power consumption (i.e., approximately between 5% and 25% if we consider the most realistic case of 3 applications per user) (*ii*) The reduction of power consumption depends also on the processing requirements of the deployed applications. In fact, having multiple relatively small workloads can drastically increase the number of potential locations for their optimal placement.

Figure 4.5 depicts how the submitted workloads are spread across the available devices due to the combination of the fluid workload distribution and the heterogeneity of our hardware. The percentage of the utilization on the different devices in the infrastructure strictly depends on the cumulative demands of the deployed

[5]As a reference, 7500 Passmark corresponds to 7 CPU cores in our desktop.

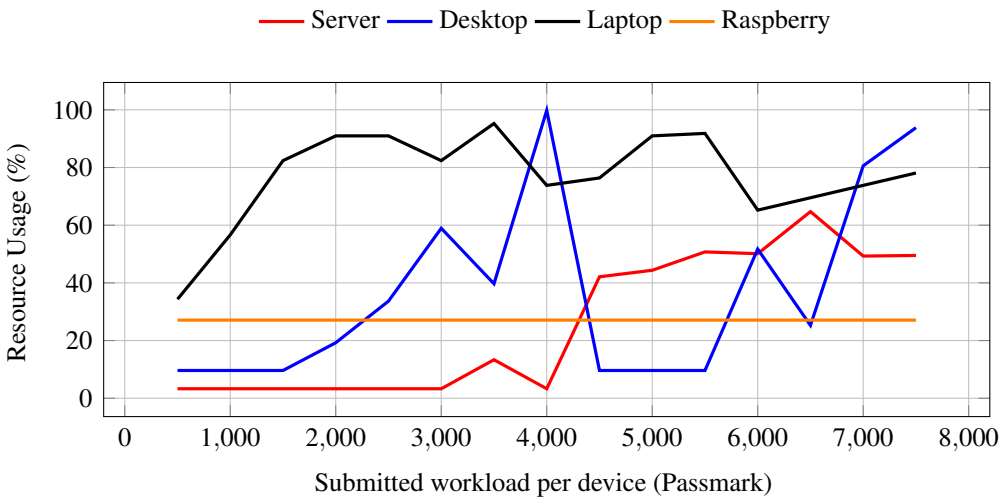


Fig. 4.5 Device resource usage increasing the submitted workload.

applications and is extremely variable, demonstrating the advantages of the rather aggressive optimization opportunities granted by the fluid approach. Overall, when the computing demand is rather low, desktops and laptops appear to guarantee the most efficient placement, whereas in the case of high computing demands servers can provide better scheduling opportunities. As a result, further optimizations can be implemented to make the most out of the continuum by dynamically turning on/off the devices, depending on the applied workload.

4.1.6 Conclusions

With the introduction of the concept of *computing continuum*, highly geographically-dispersed computing resources can be logically aggregated in the so-called resource continuum, with the possibility to define resource-sharing policies between cloud, fog, and edge and guarantees in QoE for computationally-intensive latency-sensitive applications. As an emerging concept, the computing continuum is still widely unexplored and the focus has only recently shifted from the application QoE to the possible benefits for the distributed infrastructure.

This work envisions a possible use case for the computing continuum, in which also end-user devices can share the unused processing capacities and the generated workloads may be executed on the different layers of the continuum, minimizing on the overall power consumption of the infrastructure. First, we propose a CPU-

independent mechanism for performance characterization, which can describe the application QoE within the heterogeneous devices. Performance metrics are then correlated with power requirements to assess device efficiency; results suggest an extreme variability, depending on the applied workload. Then, we discuss a possible methodology to shift from current desktop applications to a fluid containerized environment, including a detailed analysis of the related requirements. Finally, extensive simulations validate the proposed continuum use case, focusing on the overall power consumption of the computing infrastructure.

Our findings suggest that the capability of the continuum to leverage computing resources on all available devices enables a non-negligible reduction of power consumption, despite the additional requirements of the continuum orchestration components. Indeed, small-medium-sized applications allow for better consolidation and, consequently, are best-suited for the continuum with a 13.3% reduction of power consumption on average, and 25% peaks. In addition, as the percentage of the utilization on the different devices of the infrastructure is extremely dynamic and strictly related to the cumulative demands of the deployed applications, it is possible to further optimize the overall power consumption, leveraging the dynamic powering of the unused devices, which will be investigated in our future works.

4.2 Distributed energy-aware orchestration

In recent years, containerization has emerged as a lightweight and versatile method for packaging applications, enabling consistent operation across diverse infrastructures [1]. This paradigm has driven the cloud-native revolution, shifting the focus from individual servers to entire data centers. Moreover, edge and fog computing [3–5], which prioritize geographical proximity, low latency, and improved privacy, have adapted these principles for smaller, distributed data centers at the network's edge, enhancing service flexibility.

Within the European project FLUIDOS, we emphasized that effective resource utilization across a continuum requires exposing resources as a unified pool. Heterogeneous and geographically distributed computing resources enable advanced allocation policies by incorporating device location as an additional dimension. This spatial awareness enhances allocation policies related to the energy consumption of the infrastructure [130–133]: (*i*) workload characteristics can be leveraged to identify the most energy-efficient devices within the infrastructure, and (*ii*) workloads can be shifted in time and space to align with the availability of renewable energy sources, fostering sustainable computing.

4.2.1 Main contributions

Such goals can be achieved both by centralized and distributed allocation policies. From a functionality perspective, the former can guarantee the best allocation scheme for the submitted workloads but, at the same time, requires a constantly updated knowledge of the infrastructure. The latter, instead, reduces or, in some cases, eliminates the non-trivial need to share the infrastructure nodes utilization status leading to an (sub)optimal allocation. Therefore, we identify the protection of sensitive node utilization metrics as one of the main concerns when choosing the most suitable approach.

To this end, this work proposes a distributed allocation framework, which relies on a *distributed consensus algorithm* to allocate tasks using private utility functions. The algorithm operates in two phases: (*i*) a task *dispatching* during which each node decides how green it is to execute that task on its available resources using the local private utility function and (*ii*) a *consensus* during which the nodes communicate to

each other the output value of the function, deciding for an optimal task allocation. The framework allows each node within the infrastructure to employ a utility function, indicating its willingness to accept (and allocate) a particular task. Later in this work, we propose a utility function for energy-awareness, still, each node can implement a custom one, reducing the need to disclose information and enhancing its privacy with the infrastructure. Eventually, for each task, the framework will converge on the unique optimal task allocation.

4.2.2 Related Work

Recent advancements in distributed task allocation have significantly improved mobile edge computing, multi-robot systems, and distributed computing environments. In the realm of multi-robot systems [134] presented a distributed task allocation and scheduling algorithm for missions with tight coupling between tasks, emphasizing the importance of addressing temporal and precedence constraints through a distributed metaheuristic algorithm. Generalizing the scenario, [135] proposed a load-balancing technique that randomly probes two nodes in edge-clouds and selects the one with the less load to place the tasks to provide a QoE guarantee. Finally, focusing on the distributed algorithm, [136] presented an allocation policy based on the CBBA consensus algorithm that, differently from other distributed algorithms like RAFT [137] and Paxos [138, 139], have a convergence guarantee. Still, while being able to provide privacy guarantees, none of the above proposals included energy consumption in the problem formulation.

Including energy into the allocation problem, [120–122] tackled the problem of task offloading in mobile computing. However, they primarily focused on offloading techniques to reduce power consumption for battery-constrained end-user devices, neglecting additional energy requirements on servers. [133] further extended the problem, including an online joint offloading and resource allocation framework to prevent edge devices from exceeding a specified energy budget. Cloud-to-edge infrastructures are typically shared among multiple users thus requiring both to correctly model user mobility [140], and balance between energy-awareness while providing delay-guarantees in task execution [141]. Authors in [142] extended the problem formulation with a time-division multiple-access system for minimizing the weighted sum of mobile energy consumption under the constraint of computation latency. Finally, the energy-aware workload re-distribution has also been

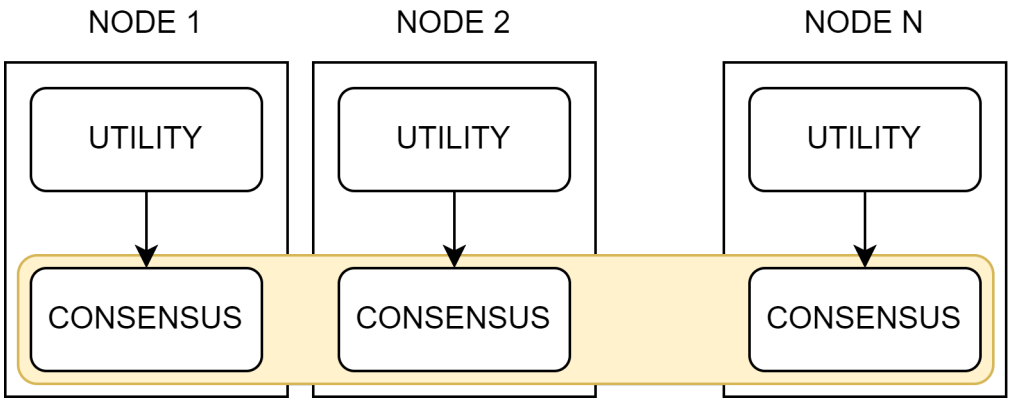


Fig. 4.6 Each node maintains a utility function to determine its suitability for executing a specific task, and the underlying framework analyzes the outputs from these utility functions to converge on an optimal task allocation.

addressed, including heterogeneous edge devices in the problem formulation [131], and evaluating the applicability on vehicular networks [130].

In the present study, we have examined several works related to task allocation in edge-to-cloud infrastructures. However, all of the works presented thus far require varying degrees of knowledge on infrastructure status, compromising the privacy of certain nodes unwilling to share sensitive information. We, therefore, assert that the joint allocation of tasks in distributed edge-to-cloud infrastructures, which considers both energy and privacy requirements, remains largely unexplored.

4.2.3 System architecture

The edge-to-cloud approach follows a decentralized and peer-based model similar to the Internet. This approach allows diverse participants, including large cloud providers, smaller enterprises linked to specific territories, and even small offices or homeowners, to independently and dynamically determine with whom they want to share resources by choosing who to peer with. After joining this edge-to-cloud infrastructure, each participant in this continuum may pursue individual goals (e.g., maximize resource usage, or profits) or share some common objective (e.g., minimize the overall infrastructure power consumption). It is worth mentioning that if the nodes in the continuum pursue a common goal, some level of information disclosure is required.

On top of this continuum infrastructure, applications must be efficiently allocated (i.e., satisfying all the execution requirements), and nodes should compete to identify the most suitable location to host the execution. In this regard, each node is equipped with its utility function that maps how much a given node is willing to host one application or a part thereof in the case of microservice-based applications. Customizable utility functions can include metrics such as resource usage and power consumption; however, the output value must fall within a predefined range to ensure fairness in allocation. Nonetheless, multiple nodes can share the same utility function to pursue global objectives or opt for a custom one in the case of local objectives.

We then use a distributed consensus-based RAFT-like algorithm to reach a consensus among the participants in the continuum to select the node with the highest utility value. The RAFT protocol allows the creation of a compute node mesh (see Figure 4.6) to exchange the output of the utility functions consistently. This can be achieved through an automatic leader election process that selects the node that is fully responsible for managing log replication on the other servers of the cluster. [6] Our distributed protocol operates in two phases: *task dispatching* and *consensus*. First, each node receives the allocation request of an application. To secure the hosting of the job or part of it (e.g., a subset of constituent microservices), the node shares the output value of its utility function. Specifically, each node checks the value of the current highest utility with the other cluster members, overriding it if higher. At the end of the consensus phase, the information of the node with the highest utility is replicated among all the nodes in the continuum to perform the actual allocation.

4.2.4 Problem Formulation

We now describe the resource allocation protocol designed to guarantee the optimal allocation. Such a mechanism is designed to clear the resource allocation problem when competing jobs must be concurrently run on the hosting physical infrastructure, participating in what we call a federation $\mathscr{F}$ while minimizing the overall power consumption of the computing infrastructure.

[6] In the RAFT terminology, a cluster comprises the set of nodes involved in the distributed consensus.

We assume a collection of $\mathscr{N}$ physical nodes, potentially hosting one or more jobs, and we index such nodes with $n \in \mathscr{N}$, where $\mathscr{N} = \{1, \ldots, N\}$, with N being the total number of nodes participating in the federation $\mathscr{F}$. Each node is then equipped with computing resources (i.e., CPU), denoted as c_n. It is worth mentioning that most Cloud-based solutions typically describe computing resources in terms of CPU and memory resources; however, since the highest correlation occurs mostly between CPU usage and power consumption, memory requirements will be omitted in the problem formulation without losing generality.

Furthermore, physical nodes are also described using the transfer function $P(x)$, which correlates the CPU usage x with the energy required to sustain such load (the exact formulation of $P(x)$ will be detailed Section 4.2.5).

We assume that each hosting node n has a (private) utility function U_n, and we are seeking an allocation solution that maximizes the sum of the utilities of all nodes. Such utility function is a policy of our resource allocation mechanisms and can be tuned to be engineered for various application and infrastructure goals. In the scope of this work, we design the utility function for a generic node n to be as follows:

$$U_n(x) = \frac{1}{P_n(x) - P_n(x_0)} \tag{4.1}$$

where $U_n(x)$ is a function of the CPU usage x. Specifically, upon receiving the request to allocate a task that would increase the resource usage of the node to a value of x, the value of utility is the reciprocal of the difference between the power consumption of the node n at the expected usage x, and the power consumption computed for the prior resource usage x_0 (i.e., the step increase in power consumption). Intuitively, the lower the increase in power consumption to host the requested job, the higher the utility value.

Our primary objective is to determine an allocation of a set of jobs, referred to as $j \in \mathscr{J}$, where $\mathscr{J} = \{1, \ldots, J\}$ onto the nodes in $\mathscr{N}$ of the federation ensuring a conflict-free assignment, i.e., satisfying the computing requirements of the job without exceeding the available computing capability c_n of each node n. Indeed, every job j has particular resource needs that must be met during allocation to ensure optimal execution across the nodes' resources. To this end, following the microservice-based approach, we represent each job j as a set of loosely coupled components $\mathscr{M}_j$, and we index such components for each job j with $m_j \in \mathscr{M}_j$, where

$\mathscr{M}_j = \{1, \ldots, M_j\}$. We assume that each component m_j of the job j has a specific computing resource demand, and we denote it with r_{m_j}. Given such notation, a *conflict-free assignment* is an assignment in which each component m_j of the job request j is mapped to one and only one of the hosting nodes n (note that multiple valid mappings of a job over the topology are possible).

Based on the above notation, we model the (NP-hard) *constrained graph matching resource allocation* problem upon the arrival of the job j with the following:

$$\max_{x} \quad \sum_{n \in \mathscr{N}} \sum_{j \in \mathscr{J}} \sum_{m \in \mathscr{M}_j} U_n(y_j) \tag{4.2}$$

$$\text{s.t.} \quad \sum_{j \in \mathscr{J}} \sum_{m \in \mathscr{M}_j} y_j r_{m_j} \leq c_n \qquad \forall n \in \mathscr{N} \tag{4.3}$$

$$\sum_{n \in \mathscr{N}} \sum_{m \in \mathscr{M}_j} y_j = M_j \qquad \forall j \in \mathscr{J} \tag{4.4}$$

$$\sum_{m \in \mathscr{M}_j} y_j \leq 1 \qquad \forall n \in \mathscr{N}, \forall j \in \mathscr{J} \tag{4.5}$$

where the allocation variable $y_j \in \{0,1\}^{N \times M_j}$ is intended to assign job j (i.e., the set of its components) on the node n. Equation (4.2) seeks to maximize the node utilities as a function of the allocation variable y_j under the constraint (Equation (4.3)), which enforces that the sum of resources assigned to the node n does not exceed the physical capacity c_n for the same node. Finally, the conflict-free assignment is enforced with (Equation (4.4)) and (Equation (4.5)), respectively stating that all the components of the job j are allocated (i.e., we don't consider a partial allocation as valid) and the same component is not assigned to multiple nodes.

4.2.5 System Model

Beyond performance, CPU architectures vary significantly in terms of power consumption, with some chips being optimized for energy efficiency (for instance, Raspberry Pis) and others prioritizing processing power (like those used in servers). Therefore, it is essential for this study to first establish a baseline for comparing both performance and power consumption across various CPUs.

We started by conducting real measurements to evaluate the power consumption of the devices and then extrapolated the results to obtain an accurate mathematical model. To assess power consumption, the study uses a smart plug connected to a wall outlet and monitors the total power usage of the device as the number of CPU cores allocated gradually increases (following the work in [19]). Using such experimental results, we can define a mathematical model to represent devices' power consumption as a function of the CPU usage x as follows:

$$P(x) = \begin{cases} \alpha x + \beta, & \text{if } 0 < x \leq \rho \\ \delta x + (\alpha\rho + \beta - \delta\rho), & \text{if } \rho < x \leq 2\rho \end{cases} \tag{4.6}$$

where ρ represents the number of physical cores of the system. In detail, devices experience different power consumption patterns, depending on whether or not the current CPU usage x exceeds the number of physical cores (i.e., logical cores are also required to sustain the load). Specifically, with a CPU usage $= 0$ all devices experience an idle power consumption β, which is typically more prominent in the case of servers. Until all the physical cores are reserved, the power consumption typically follows a linear increase with a slope value of α. Then, the power consumption keeps increasing linearly but with a lower gradient value ($\delta < \alpha$). The values of α, β, δ are device-specific, but the values for devices belonging to the same class (e.g., server, workstation, Raspberry PI) fall into predefined class ranges.

Next, in the consensus phase, the optimal job energy-aware allocation described in Equation (4.2) can be achieved by weighting the components m_j of each job j based on the individual computing requirements r_{m_j} and prioritizing the ones with the lowest demands for the bidding process. As a result, this approach drastically reduces resource fragmentation, allowing the most energy-efficient devices to maximize their efficiency.

4.2.6 Experimental evaluation

Using a simulation-driven approach, this section details the experimental evaluation of the proposed solution, further validated within the software framework provided by the FLUIDOS project.

Table 4.1 Infrastructure setup.

	SERVER	DESKTOP	RASPBERRY
CPU Model	(Intel Xeon Gold 5120)	(Intel Core i7-6700)	(ARM Cortex-A72)
Count	5	5	5
# CPUs	28	8	4
α	1-7	10-20	0.5-1
β	150-180	10-20	2.5-4.5
δ	0.5-α	0.2-6	NA

Setup

The simulated infrastructure is composed of 15 devices in total, 5 Desktops, 5 Servers, and 5 Raspberry PIs. Table 4.1 summarizes the main characteristics of the devices and the ranges for the α, β, and δ values used to represent devices' power consumption based on the model described in Equation (4.6). Figure 4.7 depicts the resulting correlation between the CPU usage and the expected power consumption of the different classes of devices in the infrastructure.

As discussed in the introduction, we aim to achieve the best allocation for infrastructure power consumption while preserving the privacy of devices unwilling to share any internal metrics. To this end, throughout the simulation, we tested different configurations, namely DA-n (i.e., distributed-allocation-n), representing scenarios in which all the nodes participate in the process using the utility function described in Equation (4.1) (i.e., DA-0), and scenarios in which some of the nodes are not willing to share sensitive information and, instead, rely on custom utility functions (i.e., DA-[2,4,6,8] in which the number represents the number of devices not participating in the energy-aware process, thus introducing noise in the process). Specifically, such nodes implement a random utility function to represent the maximum possible unpredictability.

Results are then compared with the well-known Kubernetes scheduling algorithm (shortened in k8s in the following figures), which is not energy-aware and accounts only for available computing resources in the nodes, and a brute force allocation policy (shortened in BF), which has the full knowledge of the infrastructure and can thus perform the best allocation concerning power consumption.

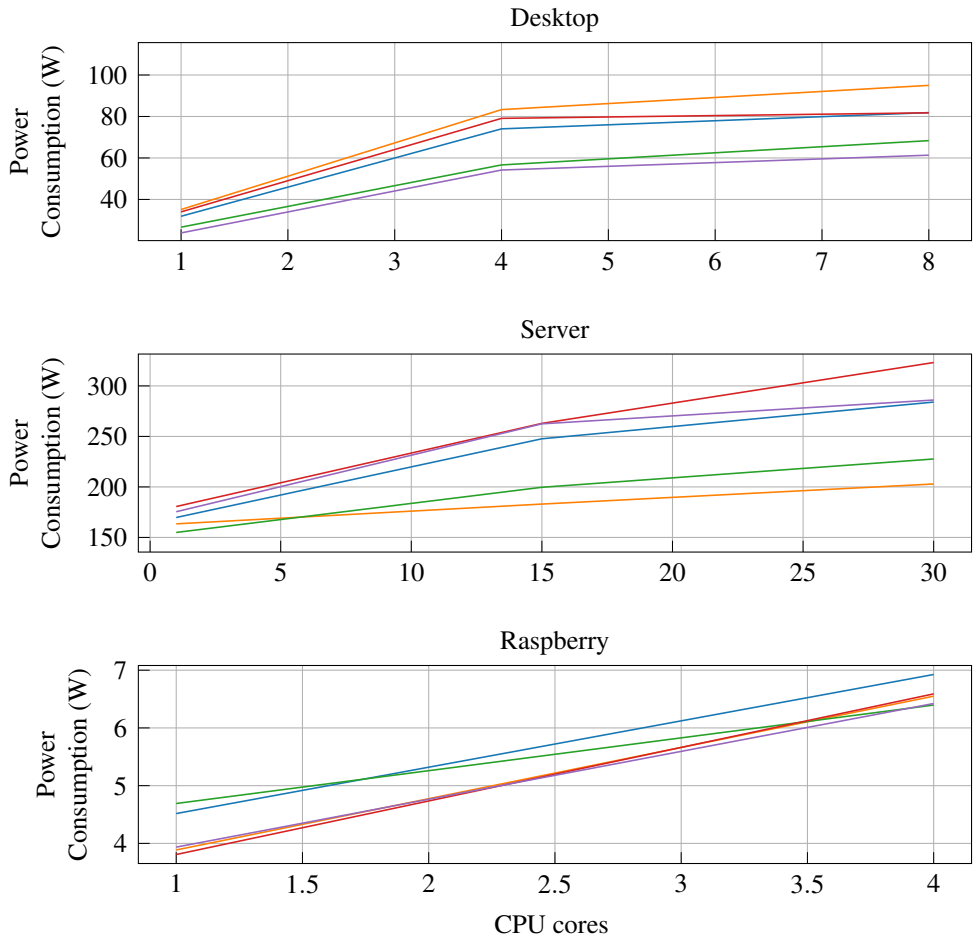


Fig. 4.7 Correlation between CPU usage and power consumption of the different classes of devices included in the simulated infrastructure.

The submitted workloads consist of 500 tasks, arriving at a rate of one task per second, with CPU needs varying within a specified range as shown in Figure 4.8. As we can see, the majority of tasks require approximately 2 CPU cores, but a conspicuous set of jobs has a much higher resource demand (up to 15 CPU cores). [7] Instead of each task's CPU demand being uniformly distributed across its components, it is randomly divided in a non-uniform manner across 1 to 6 components (i.e., microservices). This means that for a task with six components, the CPU requirement for each component isn't simply one-sixth of the total demand, but rather, the demands of all components together equal the overall task requirement. Each task is defined by its execution time, and resources are freed up immediately upon completion. To

[7]The dataset has been characterized starting from the Alibaba's public trace available at https://github.com/alibaba/clusterdata.

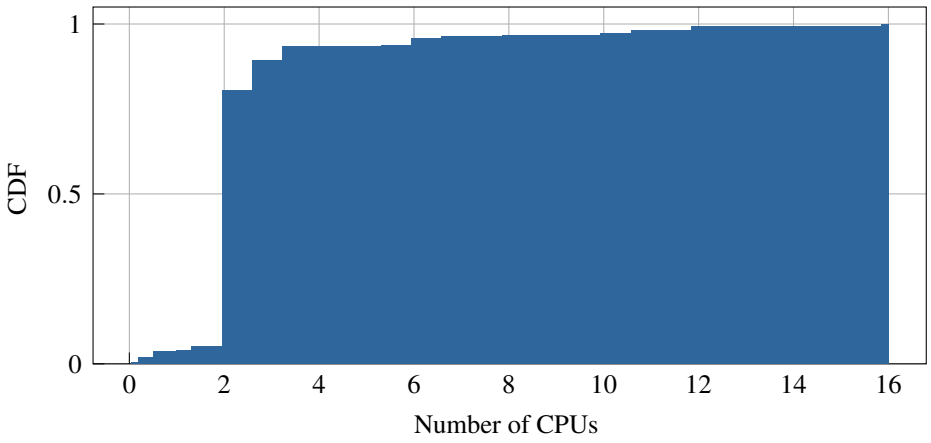


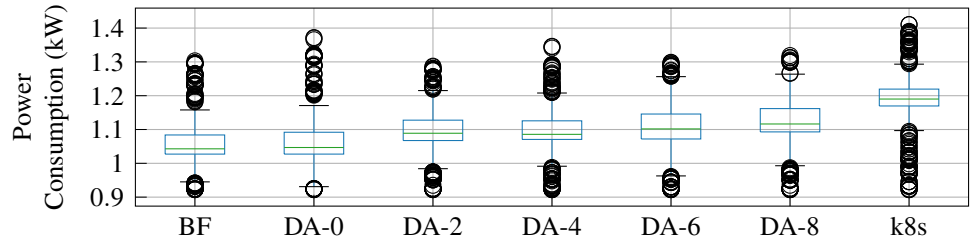
Fig. 4.8 Distribution of the CPU demand of the submitted workloads.

evaluate how the system handles different demand levels, we conducted simulations using varied execution times, ranging from 1 second to 140 seconds.

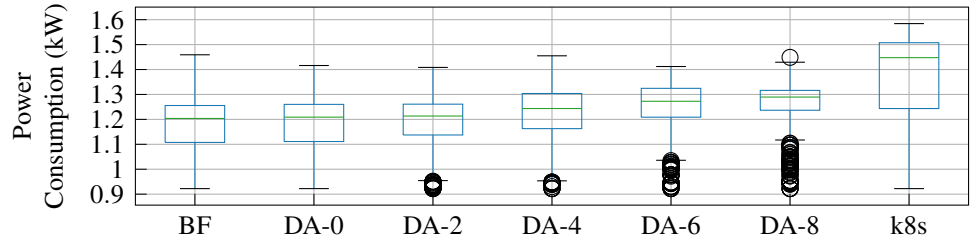
Energy-aware allocation

Figure 4.9 depicts the power consumption of the different configurations previously detailed, along with the brute force and Kubernetes-like allocation policies as a reference. Specifically, Figures 4.9a to 4.9c represent the result of the allocation varying the duration of the submitted jobs, respectively, with 60, 100, and 140 seconds. Intuitively, increasing the task duration while keeping the same arrival rate results in different pressures for the computing infrastructure.

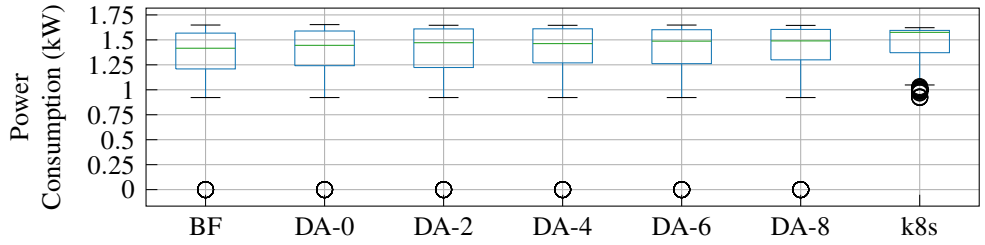
The results show that if all nodes use the energy-aware utility function presented in this work (DA-0), it is possible to obtain almost the same results as the optimal brute force approach for the overall power consumption of the infrastructure. The difference with respect to the optimal allocation is always $<<$ 5%, demonstrating that we can always achieve a near-optimal allocation. Instead, if we increase the number of devices not actively participating in the energy-aware allocation, we notice that the gap with the optimal allocation keeps increasing. Still, even with eight devices not participating in the energy-aware allocation (i.e., more than 50% of the devices of the infrastructure), we can always obtain better results than the Kubernetes-like allocation policy. Overall, we can say that, on average, an energy un-aware allocation policy like k8s always requires between 5% and 20% more energy than our energy-aware allocation policy.



(a) Duration range 1-60s.



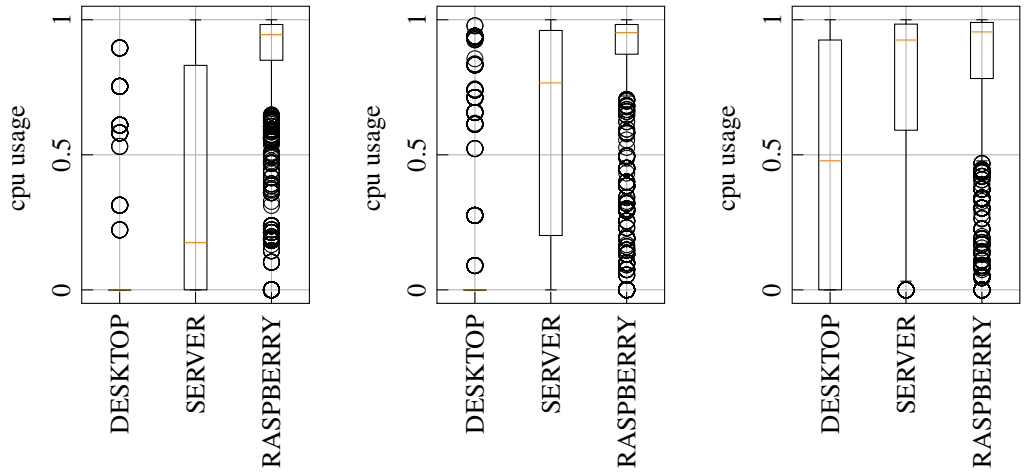
(b) Duration range 1-100s.



(c) Duration 1-140s.

Fig. 4.9 Power consumption of the infrastructure varying the duration of the submitted jobs (i.e., the load on the infrastructure) for the different configurations.

It is worth noting that the possible benefits derived from energy-aware allocation strictly depend on the amount of workload submitted to the infrastructure. With highly saturated infrastructures, as the scenario depicted in Figure 4.9c, the possible solutions for the allocation problem are extremely limited, resulting in almost comparable results for all the configurations. In fact, if the infrastructure is overloaded with job requests, the allocation problem becomes a simplified version of the knapsack problem (i.e., fit as many jobs as possible with the few available resources). Still, both the brute force approach and our allocation policy could allocate all the jobs in less time than k8s, resulting in the outliers in Figure 4.9c (in those time instants, the consumption is $= 0W$).



(a) Duration range 1-60s. (b) Duration range 1-100s. (c) Duration range 1-140s.

Fig. 4.10 CPU consumption of the infrastructure varying the duration of the submitted jobs (i.e., the load on the infrastructure) for the different classes of devices included in the simulation.

Such results can motivated by the fact that the energy-aware allocation policy can preemptively select only the most efficient nodes for the job execution while leaving the less efficient ones only in case of saturation of resources. In fact, Figure 4.10 details the CPU utilization of the different classes of devices in the infrastructure for the case of DA-0 configuration. As we can see, with low saturated infrastructures as in Figure 4.10a, our proposal relies almost entirely on servers and Raspberry PIs to host the submitted tasks (being the most energy-efficient devices of the lot). Instead, if we increase the load submitted to the infrastructure as in Figures 4.10b and 4.10c, the system has to select less energy-efficient nodes to host the workload.

As a final comment on the results, although our approach proved some interesting results on this specific setup, the same considerations and benefits can be achieved on any infrastructure with heterogeneous devices, in which it is important to consider not only the processing capability of the different devices but also the energy required to obtain such computation.

4.2.7 Conclusion

This study presents a distributed framework to optimize energy consumption in heterogeneous computing environments for cloud-to-edge computing environments.

Leveraging custom utility functions, the framework demonstrates a promising approach to optimizing task allocation while minimizing energy consumption across the network, bypassing the need to share private resource information. Compared to traditional centralized allocation policies and Kubernetes-like scheduling algorithms, our method offers significant improvements in energy efficiency without compromising the privacy of the compute nodes participating in the federation.

In the forthcoming works, the temporal shifting of workloads shall be evaluated alongside the present geographical shifting to enhance the already encouraging outcomes further. This shall be accomplished by incorporating information on the proportion of energy utilized by various devices (green versus brown energy) to encourage the development of sustainable cloud and edge computing infrastructure. In addition, a trade-off between QoE and energy consumption should be evaluated to provide execution guarantees for high-priority applications.

4.3 Final Remarks

The exploration of energy efficiency within the Computing Continuum has highlighted the necessity of bridging theoretical energy assessments with practical orchestration strategies. First, a detailed energy assessment revealed the intricate relationship between device performance and energy consumption. By characterizing both devices and applications, we laid the foundation for understanding the inherent energy requirements of the continuum. These insights were pivotal in identifying opportunities for optimization, particularly in dynamic workload allocation and energy-aware computing.

Building on these findings, we later introduced a distributed energy-aware orchestration framework that leverages these assessments to optimize task placement dynamically. This framework integrates energy consumption metrics into its decision-making process, enabling the continuum to balance performance with energy efficiency. By adopting decentralized strategies, the framework minimizes resource contention and ensures a scalable approach to managing the energy impact of computational tasks.

Together, these sections demonstrate how theoretical analysis and practical implementation can converge to address the critical issue of energy sustainability in

distributed systems. By combining rigorous energy characterization with intelligent orchestration mechanisms, the Computing Continuum can achieve not only operational efficiency but also alignment with broader environmental and sustainability goals. This holistic approach reinforces the role of the continuum as a dynamic, adaptive, and energy-conscious infrastructure, paving the way for future innovations in energy-aware computing.

Chapter 5

Resilient Orchestration in the Computing Continuum

The proliferation of renewable energy sources (RES) into power systems poses new challenges to grid operators, who have to ensure the proper and reliable operation of the electrical grid [143]. To this end, efficient and accurate state estimation (SE) is fundamental to achieve near-real-time monitoring, especially at the distribution level (i.e., the last mile of the electrical grid) where this function plays a central role in the implementation of smart grid features [144]. Research on distribution system state estimation (DSSE) began near 1990 [145] and recently, many projects have introduced the applications of phasor measurement units (PMUs) in the distribution networks [146].

Managing the bi-directional energy flows introduced by the emergence of small-medium-sized distributed energy resources is a complex task. Due to the increasing dynamics and uncertainly changing behavior of the distribution grid entities, such as dual load-generator actors, advanced monitoring and real-time decision-making are becoming crucial elements to ensure the stability and reliability of the grid. The Information and Communication Technology (ICT) infrastructure plays a vital role in enabling real-time large-scale data collection from advanced metering devices and offering high processing capabilities. The modern distribution grid must be considered as a complex cyber-physical system where the grid components and the ICT infrastructure constitute a unified fault-tolerant system that ensures operational continuity [147].

Synchrophasors supplied by Phasor Measurement Units (PMUs) have gained attention as one of the key measurements that can enable enhanced monitoring and control of distribution grids [148, 149]. Widely deployed in the last decades on the transmission grid, PMUs and their corresponding data collectors, known as Phasor Data Concentrators (PDCs), have provided unprecedented situational awareness to Transmission System Operators (TSOs) [150]. The enhanced reporting and sampling rates of phasor, frequency, and Rate Of Change Of Frequency (ROCOF) estimates from voltage and current signals, coupled with precise time synchronization across vast areas, represent a significant technological advancement capable of addressing the observability limitations within the distribution system. Although PMUs can improve distribution network monitoring, a crucial aspect of future power systems is their resiliency also in the case of natural disasters (e.g., extreme weather events, earthquakes, etc.) or cyberattacks.

Applications that leverage the advantage of synchronized phasor measurements in active distribution grids include wide-area visualization, advanced monitoring, control and protection, advanced microgrid operation, distributed energy resources integration and control, and high-accuracy fault detection and location purposes [151]. This diverse range of applications presents significant challenges for the ICT infrastructure, requiring solutions for ensuring measurement accuracy, availability, resilience, low latency, optimal reporting rates, as well as precise time synchronization. Also, the 2021 IEEE guidelines on PMUs installation [152] highlight the importance of "data quality" to support real-time mission-critical applications at the utility scale, where it is stressed that data availability is as crucial as data accuracy. Furthermore, it highlights how the planning of PMUs installation shall carefully evaluate the IT network capability to assess which services can be guaranteed.

5.1 Main contributions

In this context, emerging computing paradigms have been considered to tackle these challenges: *(i)* the *cloud computing paradigm*, that offers the possibility to acquire scalable, high-performance, elastic computing resources thus allowing for the storage, management and real-time analysis of the significant volume of data generated by PMUs, facilitating seamless data sharing across operators [153]. *(ii)* the *fog/edge computing paradigm* emerged in the distribution grid scenarios to satisfy the strict

latency requirements of applications (for control purposes, typically on the scale of tens of milliseconds), to reduce the burden of PMU data transportation and to enhance the overall system resiliency through local processing capabilities [154]. To efficiently manage applications in the different computing layers (cloud, fog, and edge), containerization and orchestration techniques have been recently implemented in synchrophasor measurement systems [21]. Among the various advantages, simplified declarative configuration, automation, and and self-healing capabilities in case of faults are particularly valuable. However, despite the emergence of common interfaces for applications orchestration over infrastructures spanning across the different layers of computing, industry-standard approaches handle each infrastructure as a multitude of (connected) isolated silos instead of a unique virtual space. This work proposes and validates a synchrophasor measurement infrastructure that encompasses the adoption of a novel architectural computing paradigm, called liquid computing [14], which builds upon and extends the well-established cloud and edge computing approaches towards a computing continuum.

The liquid computing combines the features of cloud and fog/edge solutions while bringing two main advantages to synchrophasor measurement systems: *(i) automate* the management of application life-cycle *(ii) improve* the resiliency of the ICT infrastructure in failure scenarios. The creation and maintenance of the liquid continuum have been evaluated considering the computing cost and communication overhead. This study aims to identify the optimal balance between the benefits of liquid computing and the associated costs for the synchrophasor measurement infrastructure.

5.2 Related Work

PMUs are sophisticated devices designed to provide synchronized estimates of phasors, frequency, and ROCOF derived from voltage and current signals. These devices achieve synchronization with absolute timing by utilizing Global Positioning System (GPS) signals and have reporting at rates of 10, 25, 50, 100, or 200 frames per second in 50 Hz power systems [155]. Synchrophasor data collected from multiple PMUs are aggregated by PDCs, which ensure the time-synchronized integration of data streams and deliver PMU data to monitoring and control applications. Distribution grids generally contain hundreds more nodes than equivalent high-voltage systems.

Given the expected high penetration of PMU deployments in the distribution grid and the associated high reporting rates, the volume of data is projected to increase significantly, requiring scalable and resilient data processing and management infrastructures. For this reason, cloud-based architectures for synchrophasor collection and analysis have been implemented in the last decade. A first implementation has been proposed by Maheshwari *et al.* [156] that presented a PMU-based state estimation application on a cloud architecture. Luo *et al.* [157] demonstrated how data-intensive and compute-intensive power applications can leverage the advantages offered by the cloud computing paradigm. Recently, industry real-world use cases of synchrophasor measurement infrastructure leveraging cloud computing have been implemented. The cloud deployment of a wide-area monitoring and data-sharing platform is presented in [158]: ISO-New England and New York Power Authority shared real-time data collected from PMU/PDC measurement infrastructure through the so-called "GridCloud" platform based on a popular public cloud. Other cloud computing real-world PMU use cases are presented in [153], which highlights the enhancement in resilience offered using cloud computing and implementing particular architectures that mitigate network disruptions and service outages. Certainly, current cloud-based power systems provide robust solutions for high-demand applications; however, they face several significant challenges. While traditional cloud architectures offer high-speed processing and ample storage capacity, they are often inefficient for real-time services. They are typically bound to cloud premises with limited or no control over geographically dispersed computing resources at the edge of the network. Decision-making and power system operations are more sensitive to network latency and have to be highly resilient against faults.

To tackle this challenge, the edge computing paradigm has been applied to smart grid monitoring and control applications. Edge computing poses the process capabilities near the terminals, alleviating the service latency of real-time decision-making algorithms. In addition, edge computing can help to enhance the resilience of the system, providing local edge intelligence when communication with control centers is disrupted. In this context, A. Meloni *et al.* [159] proposed and validated a Cloud-IoT-based architectural solution for a PMU-based state estimation application that combines cloud capabilities and edge-computing advantages. The authors proposed the introduction of a "Virtual Object" at the edge which implements local policies in order to decide whether to send or not PMU data remotely but has limited processing capabilities. In a wider context, Chen *et al.* [160] presented a

layered architecture to describe the deployment of distributed metering applications to the edge. Data measured from meters and IoT devices placed on the power grid are analyzed, processed, and stored at the edge of the measurement infrastructure. In [161], a 3-phase Smart Meter Architecture was implemented to self-configure, collect electric measurements from the power grid, and run power grid algorithms on edge devices. While these works utilize the local processing power of edge computing, they neither assess the system's resiliency nor implement any self-healing capabilities.

In parallel with the development of edge and cloud computing paradigms, containerization and orchestration technologies have emerged to enhance the automation and resilience of power grid services. Monitoring and control applications are restructured into containerized microservices, enabling them to be efficiently orchestrated across multiple physical devices. Leveraging the potential capabilities of orchestration and containerization technology in a smart grid environment, Li *et al.* [162] introduced an Energy Management System (EMS) orchestrated architecture built on a Kubernetes cluster, to solve the reliability problem of EMS software under limited resources. Pau *et al.* [154] presented a service-oriented Distribution Management System (DMS) based on Low-Cost PMU measurements leveraging containerization and Kubernetes-based orchestration technologies. In this context, at the end of the paper, they hinted at a possible DMS implementation that also exploits edge computing to achieve scalability, relaxing the communication towards the upper-level control center and decoupling different portions of the distribution system, while still allowing the coordination of services running locally. Recently, Stoupis *et al.* [163] proposed an edge architecture for distribution grid applications, e.g. fault detection, isolation, and restoration application based on Docker and K3s technologies fully leveraging the capabilities of the edge paradigm combined with orchestration technology. They highlighted the concept of hierarchical grid intelligence as the future of the grid's architectures. The application of orchestration and edge computing technologies in the context of synchrophasor measurement infrastructure is described in the work of Galantino *et al.* [21]. Automatic configuration of PMU-PDC communications during network reconfigurations, automatic redirection of PMU traffic in the event of PDC service redeployment to a different location, and reduced data loss during PDC failures are the key benefits achieved. Additionally, overall system resilience is improved due to minimized downtime of ICT services.

The works presented implement distribution grid applications leveraging either the cloud computing paradigm, collecting data and processing them in a centralized cloud location, or the edge computing paradigm, allocating processing resources close to data sources. They do not consider leveraging the two layers as a unique virtual environment. Moreover, the presented works do not evaluate the resilience of the proposed architectures, which is a critical factor for ensuring the operational reliability of smart grids. While orchestration techniques can enhance application resilience, cross-layer orchestration across both edge and cloud layers has not yet been widely adopted, limiting the full resilience and automation benefits that orchestration can offer. In this work, we propose a synchrophasor measurement infrastructure based on a novel architectural computing paradigm called liquid computing, which unifies edge and cloud layers into a single virtual and resilient environment. Through this approach, we assess and demonstrate improvements in resilience and service automation.

5.3 Requirements

The introduction of distributed energy resources requires essential changes in distribution planning, control, operation, and protection methods. Geographically distributed sensors providing synchronized measurements and real-time decision-making are needed to address the challenges of the distribution system. The PMU's high reporting rate and precise synchronization over wide areas bring numerous advantages for a large number of applications. Each application has specific requirements that have to be satisfied by the ICT infrastructure. The application requirements can be classified as:

- *Availability*: availability is defined as the ratio of measurement expected by an application at a given reporting rate versus the measurement that arrives at the application.
- *Resiliency*: strictly related to availability, resiliency is defined as the capability to react and recover from failures minimizing the system downtime.
- *Latency*: strictly related to availability, latency is the time difference between the arrival time of PMU data and the time of PMU data being generated at

the source. If latency is above a specific threshold the application will discard the data, impacting availability. Depending on the type of application, latency requirements can vary from hundreds to thousands of milliseconds.

- *Accuracy*: measurement accuracy is defined by the PMU device and is evaluated considering for steady-state synchrophasor measurement the Total Vector Error (TVE) of 1 %, the Frequency Error (FE) of 0,005 Hz, and ROCOF error (RFE) of 0,4 Hz/s (only for P class measurements) [155]; The PMU device packages the data and transmits it to its destination, if the data changes due to network availability and latency, the communication protocol will account for the variation.

- *Time synchronization*: strictly related to the accuracy, the time synchronization error should be sufficiently low to keep the TVE, FE, and RFE within the specified limits, e.g. to measure the phase angles of voltage and current, the clock must be accurate within 25 microseconds [155].

- *Reporting rate*: applications may require different rates of PMU measurement being reported, for 50 Hz systems typical reporting rates are 10, 25, 50, 100, 200, for 60 Hz systems are 10, 12, 15, 30, 60, 120, 240.

- *Scalability*: selected applications may require multiple measurement points per feeder, the measurement infrastructure architecture should account for the acquisition of data from hundreds to thousands of PMU devices, requirements are qualitatively expressed considering a low, medium, and high-density architecture needed.

In general, advanced distribution monitoring and control applications have more relaxed requirements in terms of latency and availability with respect to advanced distribution protection applications. Minimum availability of 95%, maximum latency of 1000 ms, and minimum reporting rate of 25 Hz (for 50 Hz systems) are required by applications of wide-area monitoring which leverage the time-synchronization advantage introduced by geographically scattered PMUs. [1] These applications require at least one measurement per feeder, depending on observability criteria, and

[1] As a reference, distribution protection, fault detection and stability management applications require a much higher availability of data (more than 99%, up to 99.9%), lower latency (between 150ms and 300ms), but approximately the same reporting rate.

data processing is expected to be at the control center site, resulting in a medium-density architecture. Volt-Var control applications, which address the complex voltage and reactive power control in modern distribution systems, have similar requirements (95% availability, 2000 ms latency, and 10 Hz reporting rate) as long as real-time distribution system operation applications. The latter uses PMUs to monitor voltage and current data to operate the distribution system reliably identifying and mitigating impacts caused by DER's variable behaviour. The ICT architecture for monitoring and control applications should be highly resilient and tolerant of communication interruption and software/hardware failures. Higher latency values are tolerated by analysis applications that are typically deployed in near real-time and at control centers' premises.

Finally, requirements are even more relaxed for applications targeting the planning of the distribution system (or combined transmission-distribution systems), DER and loads forecasting, improved load-shedding schemes, commercial loss analysis, or power grid resilience analysis. Particularly noteworthy is the case of microgrids, small-scale power grids that can connect to the main power grid or operate independently from it, thanks to the significant integration of DER. Advanced microgrid applications use synchronized high-reporting rate measurements to enable the microgrid's reliable operation and control in grid-connected and islanded mode. Local processing capabilities are required within the microgrid, resiliency should be guaranteed inside the microgrid and in case of communication breakdowns with control centers.

The aforementioned applications based on synchrophasor measurements aren't isolated and must be deployed on a shared ICT infrastructure. Therefore, architectural decisions in designing the ICT architecture should take into consideration the full spectrum of requirements, ensuring a more integrated approach. The system requires distributed and centralized processing capabilities. Substations should be equipped with edge computing capabilities to perform initial data processing (e.g., filtering, aggregation, fault detection) close to PMU data sources. This reduces the need for high-bandwidth transmission to centralized systems and ensures faster response times for critical applications. Applications that do not have strict latency requirements can be deployed on centralized servers (at control center premises) or even on cloud platforms for advanced analytics, long-term storage, and control center operation. The ICT architecture should be designed to scale both horizontally (adding more PMUs) and vertically (e.g. increasing computational power and storage

capacity at the edge or integrating cloud-based services that provide flexibility to scale resources on demand). A key feature is that the management of the applications should be fully automated and easily configurable, allowing for seamless operation and adaptability. The communication layer of the ICT architecture should be carefully designed supporting real-time data transmission with low latency and high availability. Over large geographic areas, a reliable Wide Area Network (WAN) is needed and at a substation level communication at high-speed, low-latency Local Area Network (LAN) is expected. Ensuring the resilience of the ICT architecture is critical for sustaining the reliability of grid operations. Protection, control, and monitoring systems must be capable of managing failures without significant degradation in availability. More importantly, these systems should possess the ability to autonomously recover with minimal human intervention. This necessitates the design of an ICT architecture with inherent self-healing capabilities, ensuring no adverse impact on system performance. In addition, the capability to continue functioning even in case of network disruption with a control center is crucial for selected applications. In this context, it is imperative to assess the resilience of these systems considering the impact of multiple ICT failures.

5.4 Liquid Computing

To achieve observability in power grid distribution systems, monitoring devices are required at the power grid edge. The simplest approach would involve collecting data streams in a single, centralized management cluster. However, while theoretically straightforward, this approach is impractical, as the central cluster would be overwhelmed by the need to collect and process potentially hundreds or thousands of data streams. Additionally, the management of such dispersed infrastructure requires manual and potentially error-prone intervention from the Distribution System Operators (DSOs). A more feasible solution would involve the creation of a single “big cluster” using state-of-the-art technologies like Kubernetes to simplify the management and combine all the compute nodes in the network. As a result, this approach provides the support for relocating part of the computation closer to the data sources, thus reducing the computational burden on the central cluster.

However, despite the emergence of common interfaces for applications orchestration over infrastructures spanning across the different layers of computing (i.e.,

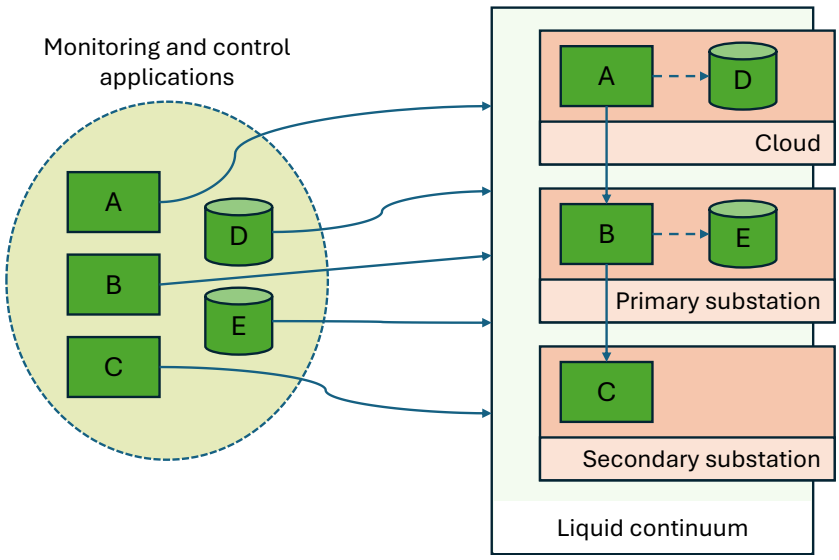


Fig. 5.1 The Liquid continuum breaks down the siloed-based approach in the computing scenario by creating a logical infrastructure that aggregates the computing resources of the various layers. Applications can then be deployed anywhere in the continuum depending on execution requirements.

cloud, fog, and edge), industry-standard approaches handle each infrastructure as a multitude of isolated silos instead of a unique virtual space. In fact, none of the technologies available on the market natively support the "big-cluster" architecture previously mentioned as they are typically designed for cloud environments (i.e., with guarantees in network connectivity). Each compute facility, hereinafter referred to as a cluster, is thus a separate entity, independently managed and does not integrate with the other clusters of the grid (i.e., closer to the end users). This leads to: (*i*) unmanageable complexity when controlling such highly-dispersed infrastructure, and (*ii*) a sub-optimal fragmented view of the overall available resources, preventing the seamless deployment of fully distributed applications. No resource compensation is ever possible, hence preventing applications from transparently moving from an overloaded or temporarily unavailable cluster, e.g., due to unexpected spikes of requests, to another one, underused and potentially offering better performance. At the same time, the deployment of complex applications composed of multiple microservices, each one with specific requirements (e.g., low latency, access to specialized hardware, ...), as well as the enforcement of proper geographical distribution and high-availability policies, requires the interaction with different infrastructures.

Accounting for these demands, this work relies on a novel architectural paradigm, called liquid computing as described in the seminar paper[14], which builds upon and extends the well-established cloud and edge computing approaches towards an endless computing continuum. Overall, the resulting computing domain abstracts away the specificity of each cluster, presenting to the final users (e.g., a system administrator) a unique and borderless pool of available resources, the so-called big cluster. Thanks to this abstraction (see Figure 5.1), applications are no longer constrained in a specific silo, but are free to fly possibly in the entire infrastructure, selecting the most appropriate location depending on its requirements (e.g., data proximity), and the available resources, while retaining full compatibility with traditional orchestrating frameworks (e.g., Kubernetes). With liquid computing, all communications between applications are mediated by the virtual network fabric, which guarantees seamless communications independently from the location of each microservice.

In the context of smart-grid monitoring, liquid computing can bring a twofold contribution: (*i*) *automate* the management of the application life-cycle, providing the state-of-the-art control strategies to enforce the desired state for the execution, and (*ii*) *improve* the resiliency of the entire IT infrastructure by relying on nearby compute nodes in the event of unforeseen network partitioning events, or hardware/-software failures in one of the clusters. Obviously, the creation and maintenance of the liquid continuum has a cost, e.g., in terms of computing (additional software services running on the nodes) and communication overhead (exchanging messages to synchronize data in the different entities). This work has the ambition to determine the best trade-off between the advantages brought by liquid computing and the corresponding costs.

5.5 Architecture

The electric power control and monitoring network, as shown in Figure 5.2, shares some similarities with a logical hierarchical tree topology. The Operative Center (OC) oversees a given number of primary stations (i.e., substations that deal with high voltages in the range from 4 kV to 35 kV phase-to-phase), collecting measurements and enforcing control strategies to bring the entire system to the desired state. Primary stations can either be considered as the leaf of the infrastructure (i.e., no

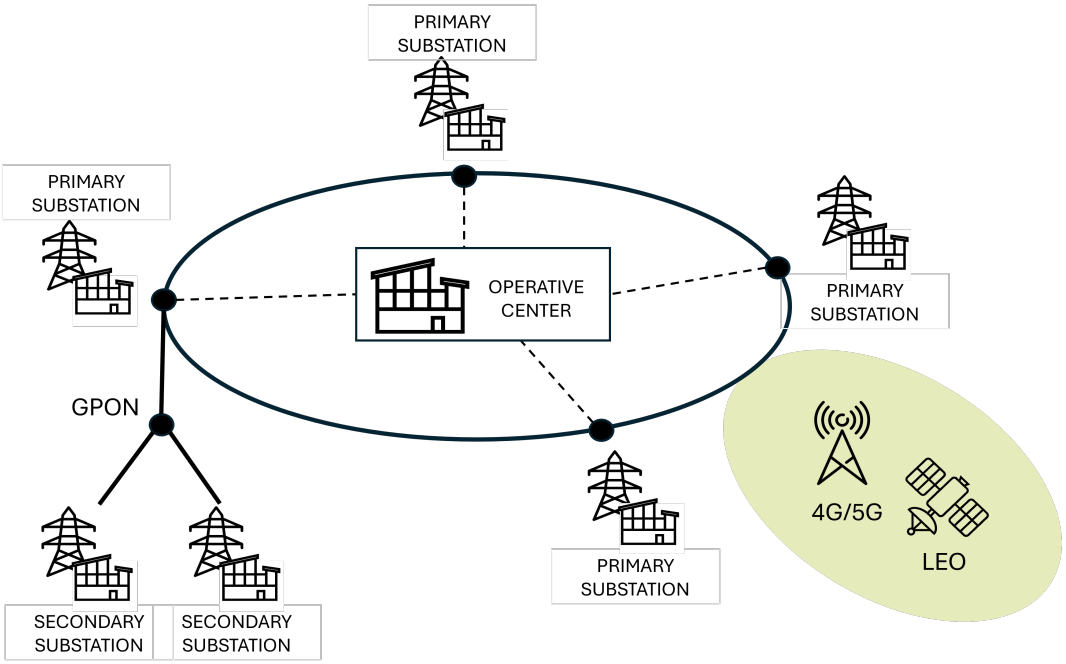


Fig. 5.2 Logical hierarchical structure of the electrical grid, with the Operative Center (OC) overseeing the entire infrastructure of primary and secondary substations.

outgoing connections towards other stations) or can, in turn, oversee other primary or secondary substations (i.e., last mile to the domestic customers). Although this work focuses on monitoring applications, we can assume that applications serving different purposes in the electrical grid might coexist in the OC.

Although quite simple, this hierarchical architecture cannot be directly replicated using the default version of Liqo. In fact, Liqo does not allow for multi-level architectures as the resulting topology may lead to circular behaviors that would impact the convergence of workload allocation. For instance, in a scenario in which cluster A is connected to cluster B, which, in turn, is connected backward to cluster A, when cluster A receives the request to deploy an application it can decide to delegate the allocation to cluster B. Now, if cluster B autonomously decides to offload the application back to cluster A the entire process would never reach convergence and the application would never be executed.

However, the hierarchical topology can still be implemented by flattening the entire infrastructure, relying then on labels to differentiate the multiple substations. The resulting infrastructure is based on the concepts of *groups* and *levels*, i.e., each cluster can be categorized based on logical group and hierarchical level. Groups

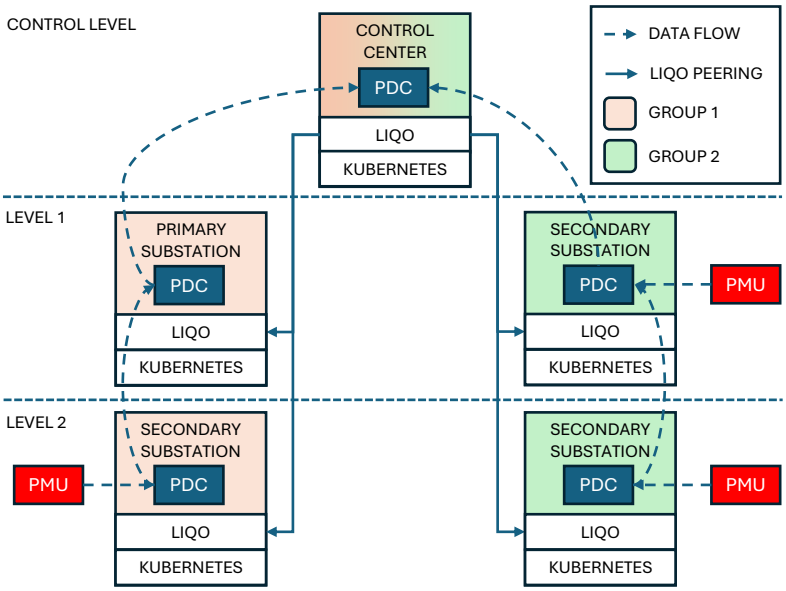


Fig. 5.3 Proposed architecture for the monitoring of the electrical grid, differentiating substations based on groups and levels.

represents different domains for the monitoring system, i.e., all the substations that oversee the same geographical area and whose metrics have some sort of correlation in the processing seen by the OC. We further assume that monitoring services, when possible, can be executed on any cluster belonging to the same group. Levels, instead, prevent the loss of information derived by the flattening process previously described by representing the depth of the substation in the hierarchical architecture. Specifically, Figure 5.3 details the resulting architecture implemented using Liqo. The leaf clusters are segmented into independent groups by assigning each node within the cluster a label that uniquely identifies its respective group. These groups are not mutually exclusive regarding the ownership of a node, provided there is no logical contradiction among the identifying labels.

5.5.1 Implementation

This implementation is depicted in Figure 5.3, where the Operative Center (OC) occupies the central position of the topology, establishing unidirectional peering with

offloading capabilities towards every other entity (i.e., substations) in the topology, whether it is a primary station or a secondary station.

This allows the OC to manage all application deployments, making it the central access and management point for the entire infrastructure. The remaining clusters are divided into groups, typically consisting of a primary substation and its associated secondary substations. These groups represent a logical domain of applications with their own high-availability distributed database system for data and configuration persistence and, therefore, do not have interconnections among them. Within a group, all clusters form a full mesh of unidirectional peerings simply to enable communication in case applications are deployed onto different clusters. As a result of this configuration, applications can be deployed on the OC, which will in turn delegate the execution to the most appropriate cluster within the infrastructure based on the geographical constraints of the same application (e.g., the execution is bound to the set of clusters belonging to the "metropolitan area" group).

The cluster representing the Operative Center is a critical point as it hosts the central logic of the network, but the effects of a failure or disconnection of this node are negligible when compared to environmental constraints. Specifically, in case of physical failure of the OC, the peripheral substations can continue to operate transparently, i.e., the applications therein deployed are not affected by the failure. In fact, Liqo simply extends the visibility of resources from the OC, without affecting the control capabilities of the peripheral clusters that still maintain control over the local resources. However, since the OC represents the control point of the infrastructure, the obsevability is deeply affected until the failure is resolved. Nonetheless, we can argue that this failure scenario is negligible because without the central node's logic, the network would not be observable by default, and, moreover, such failure may affect as well other applications outside the scope of monitoring. Therefore, in the following, we assume the OC to be implemented using cloud-based solutions that can guarantee a level of availability $> 99.99\%$.

In contrast, all other failures are recoverable from the perspective of the OC. Failures within a cluster are generally recoverable in a short time as the applications are automatically and quickly recreated into a healthy cluster of the same group in case of disconnection or internal pod failure. Disconnections of single clusters, multiple clusters, or parts of the network containing data production sources (PMUs) are automatically recoverable only with the restoration of the connection itself as the

PMU is physical hardware tied to its node and cannot be moved to others. However, as discussed before, the disconnected clusters maintain their functionality also when disconnected.

5.6 Experimental evaluation

In the following, we demonstrate the properties of the liquid computing architecture when applied to the monitoring of the power grid. The proposed architecture is first evaluated based on the requirements within our proprietary testbed facility, replicating the scenario of an isolated section of the ICT infrastructure, focusing on the *local* properties enforced in each substation (Section 5.6.1). Then, through experiments we evaluate the capability of the infrastructure to support application requirements in terms of *latency*, *availability*, and *resiliency* at a *global* scale (Section 5.6.2).

5.6.1 Computing requirements

Given the recent trend in the industry, our architecture is heavily based on Kubernetes,[2] which includes implementations also targeting low-resource devices, hence not requiring datacenter-grade servers. In addition to its native features (e.g., automatic service restart/re-spawn in case of failure, multi-master capabilities, etc.), it includes a large software ecosystem that can provide well-tested solutions for many common problems, such as data redundancy. Specifically, *K3s*[3] is the Kubernetes distribution chosen for edge sites, which features a very limited resource consumption (CPU, memory, disk), performance close to vanilla Kubernetes [164] and a very simple setup.

Monitoring services (i.e., PMUsim and OpenPDC) have been containerized to be executed in a cloudified environment, hence only Linux-based operating systems have been considered for our evaluation. To keep the results consistent, Ubuntu 20.04 has been used as the base OS for all measurements. We considered both x86 and ARM architectures, 64-bits, which will be hereinafter referred to respectively as x64 and arm64. Further information are available in Table 5.1.

[2] https://kubernetes.io
[3] https://k3s.io/

Table 5.1 Relevant specifications of the machine used to carry out the tests.

Architecture	x86 (64-bit)	arm (64-bit)
Machine	VM	Raspberry Pi 4B
linux kernel	5.4.0-48-generic	5.4.0-1042-raspi
CPU model	Intel Xeon (Cascadelake)	Cortex-A72
CPU cores	4	4
CPU frequency	2.2 GHz	1.5 GHz
Memory size	8 GB	4 GB

Evaluation method

CPU and memory consumption metrics have been collected using `sysstat`, gathering information using Linux standard counters with a `cron` job executed every minute. CPU usage represents the time in which the CPU is not idle and the system does not have an outstanding disk I/O request. Memory usage simply accounts for the non-free memory at a given time. Applications were executed in their standard operating conditions, e.g., PMU producing data at a constant rate, while the PDC receives, processes and stores the above data flow. Tests have been carried out for at least 4 hours to demonstrate that applications have a consistent behavior over time and to collect enough data to perform significant statistical analysis, identifying confidence intervals and outliers.

Our solution introduces an additional layer of abstraction for data persistence leveraging the enhanced storage features provided by Longhorn CSI,[4] which features advanced data management capabilities coupled with minimal resource requirements e.g., compared to other software such as Rook/Ceph. The Longhorn service can be replicated on multiple nodes within each site, each instance attached to a distinct data volume created on the node itself. Longhorn instances are coordinated to guarantee that any data is replicated on different physical volumes (hence nodes), providing the selected level of redundancy to survive the departure of N data volumes.

Reaction times tests have been carried out using helper `bash` scripts to poll system events and store their timestamp for subsequent analysis. Each single case will be explained more in-depth in the dedicated section.

[4] https://longhorn.io/

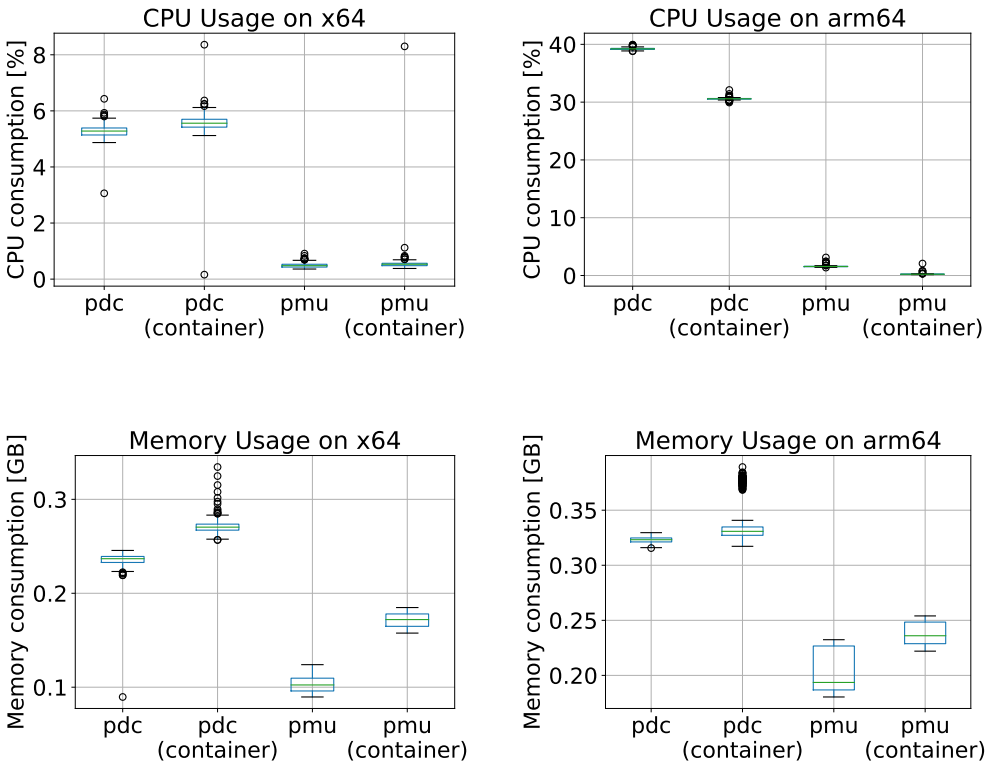


Fig. 5.4 CPU and memory usage comparison for service execution on bare metal and with containerization.

Containerization overhead

This test quantifies the additional resources required by containerization for the same application installed on bare metal. The PDC service has been containerized using an Alpine base image on arm64 and an Ubuntu base image on x64, which resulted in the most efficient CPU and memory usage in their respective environments. However, this experience suggests that the base image to be used when containerizing a service cannot be given for granted and should be properly assessed in a real production environment.

Figure 5.4 shows that the CPU overhead added by the container environment is negligible both in case of x64 and arm64 (actually, CPU consumption even improves in case of arm64), whereas the additional memory is in the order of a few megabytes (53MB for x64 and 40MB for arm64). This confirms that containerization overhead

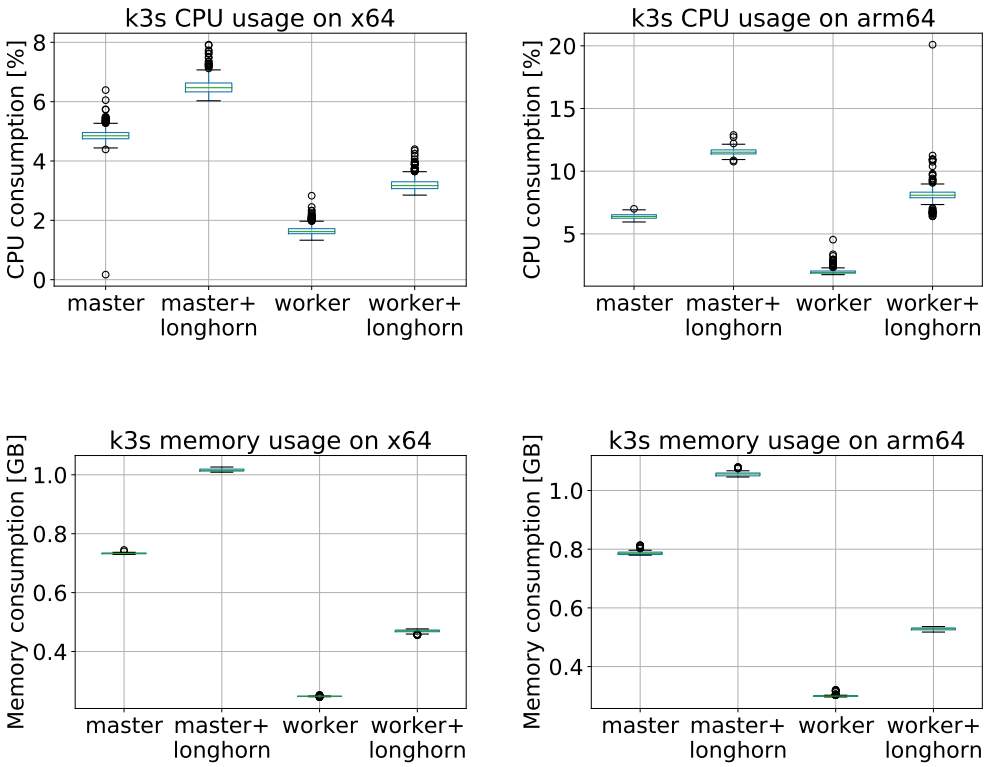


Fig. 5.5 Orchestrator CPU and memory requirements on K3s master and worker nodes, with and without Longhorn.

is almost negligible – regardless of the device architecture – even for edge devices with limited computational power.

Orchestration and distributed storage overhead

This test quantifies the resources required by K3s, i.e., the cost of the orchestration (without any additional workload), differentiating between worker and master nodes. This evaluation is needed to quantify the resource requirements for each cluster (i.e., substation) in the infrastructure. Figure 5.5 shows the resource requirements of the different setups. As expected, the master node results in a higher CPU and memory footprint with respect to worker nodes, because of the additional requirements of the control plane components; the use of Longhorn further increases the footprint in master nodes on arm64 devices (5% of CPU and 30% of memory). Although the orchestration overhead is no longer negligible as for containerization, K3s and

Longhorn can provide a good balance between resource usage – still tolerable even for a low-end device such as a Raspberry Pi – and the advantages brought in by an orchestrated system, which provides enhanced data and service resiliency.

Orchestrator reaction times

In Kubernetes the reaction time upon node/service failures can be dramatically reduced using *replicas*, i.e., N instances of the same service are executed simultaneously, guaranteeing service resiliency in the event of $n < N$ failures. Although appealing, replicas cannot provide benefits in our case. In fact, multiple PMUs instances would share the same physical measurement device, hence resulting in hardware contention. PDCs are not suitable either, as the default Kubernetes load balancing mechanism sends data to either one of the replicas. This results in data partitioned across all the PDC instances and the consequent necessity of data re-aggregation before further processing, which would not satisfy our requirement of leveraging existing applications. Therefore, only single replica services are evaluated.

Our test evaluates the orchestrator reaction time upon the occurrence of two possible failure events to assess *local* resiliency (the resiliency introduced by the liquid computing will be evaluated later): (*i*) container restart after unexpected execution failure (simulated by forcibly sending a *kill* signal within the container, hence killing the process delegated to the synchrophasor exchange and triggering the re-scheduling policy of the orchestrator); and (*ii*) container re-instantiation in a healthy node after a node either fails or becomes unreachable.

We used the *tcpdump* command to measure the time required by PMU and PDC to actively restore the synchrophasor exchange process, monitoring then the actual network packet exchange between the components. Results are then compared to the widespread `nginx` web server, which provides insights on the maximum performance of a cloud-native application. The entire process is repeated 10 times to obtain statistically relevant data.

Results, depicted in Figure 5.6, show that Nginx, designed to be fully cloud-native, experiences the fastest restart time, rarely exceeding $5s$. Instead, PMU and PDC are not designed to operate on a Cloud environment and the measured restart time ranges between $6-12s$ for the PMU, and between $18-25s$ for the PDC. The proposed results raise some additional considerations: (*i*) In our configuration, the

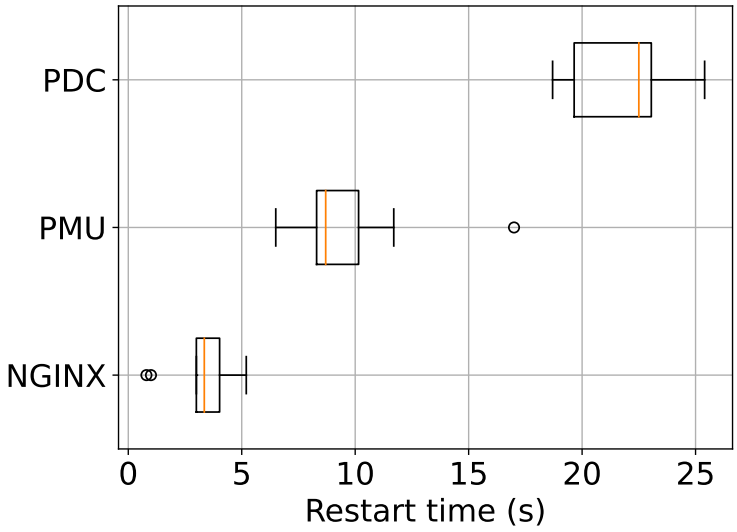


Fig. 5.6 Data flow restart time interval in case of nginx, PMUs and PDCs.

Kubernetes control plane checks every 5*s* the state of the specific service (e.g., healthy, unhealthy). Therefore, the contribution of the orchestrator on the final restart time cannot exceed 5*s* in the worst case and can be further reduced upon configuration. The remaining time is thus related to the service control logic to re-instantiate the communication and can be reduced only with proper code refactoring. (*ii*) As of today, the recovery upon monitoring service failure is not automated, and in many cases still requires manual intervention. This implies that monitoring services have different resiliency requirements, compared to control services, and can withstand longer service disruption (e.g., minutes), without compromising the smart-grid operability.

The node failure is emulated by pushing a set of firewall rules in `iptables` to isolate the target node from the rest of the infrastructure. The isolation process is repeated multiple times, in order to cover any possible failure, as the application placement within the infrastructure is (almost) completely delegated to the orchestrator and may vary over time. The K3s master is continuously polled to check when the target node is marked as unreachable and measure the time required to re-instantiate the containers running on it. Figure 5.7 shows the three critical reaction intervals: (*i*) time required for the master to recognize a failed node and set its status as NotReady/Unreachable, (*ii*) time to re-create all the containers hosted in the failed node and to restore the application data flow, and (*iii*) the total time to

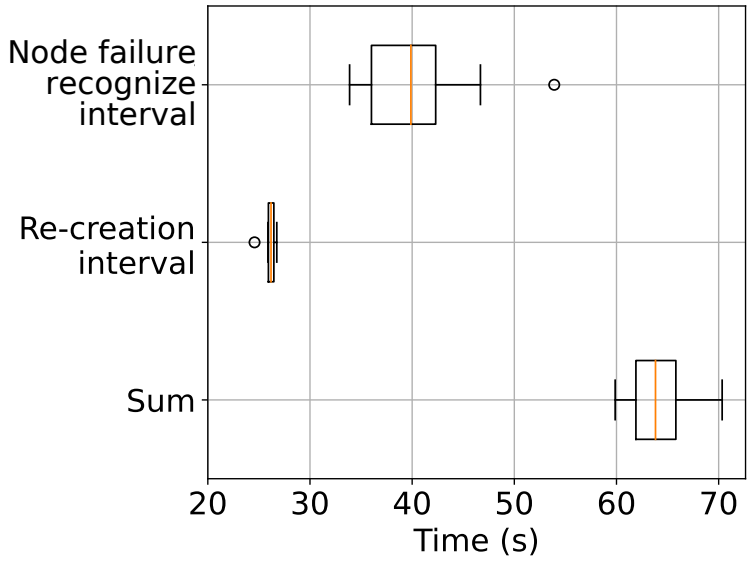


Fig. 5.7 Time required to recover services on a disconnected node.

have the service restored on the remaining running nodes. Specifically, the total re-creation interval is bounded to the slowest-restarting service (i.e., usually the PDC), and the time interval required to identify a node failure strictly depends on Kubernetes control logic and experience high variability, depending on the moment of the failure and the next node health check. Still, even in the worst case, the proposed infrastructure can recover to node failure event within 70*s*, well below the requirements.

5.6.2 The continuum resiliency property

The focus now shifts from the local cluster to the liquid infrastructure proposed in this work. The infrastructure replicates the logical topology depicted in Figure 5.3. Specifically, one central cluster oversees all the different substations (i.e., primary and secondary), which, in turn, are labeled to differentiate the respective role in the grid. Each substation is then implemented with a Virtual Machine (VM), each with 4 CPU cores, 8GB of RAM, and 25GB of storage, to replicate the limited computational capacity that can be deployed on the substations of the distribution layer. It is worth noticing that this evaluation focuses on devices with limited computing capabilities to replicate a conservative deployment scenario. In fact, considering the number of substations in the distribution layer it is economically

unfeasible to install dedicated servers on each substation. Still, increasing the computational capacity available in the substations would drastically improve the performance of the overall infrastructure. This evaluation thus serves as a baseline for future improvements.

Each VM hosts a Kubernetes cluster, specifically *K3s*[5] is the Kubernetes distribution chosen for edge sites, which features a very limited resource consumption (CPU, memory, disk) [21], performance close to vanilla Kubernetes [164] and a very simple setup. Applications are executed in their standard operating conditions, e.g., PMU producing data at a constant rate, while the PDC receives, processes, and stores the above data flow. In detail, we consider a three-layered chain of monitoring services, with the PMUs generating synchrophasor measures, a first-level PDC collecting the flows from one (or more) PMU, and a second-level PDC aggregating the streams from one (or more) first level PDCs. The functionalities of the PDC are based on the openPDC software system [6], which has been containerized to ensure compatibility with Kubernetes. In addition, the distributed DB Percona XtraDB [7] has been adopted to ensure data persistency across clusters for PDC configurations and phasor measurements.

Finally, the execution of PMUs is considered to be fixed, i.e., bound to the specific substation equipped with the proper monitoring sensors, whereas the PDCs are bound to a specific group in the infrastructure and can therefore be executed on any cluster within the group.

[Latency] Infrastructure latency

Given that data is typically transmitted using TCP protocols (i.e., IEEE C37.118 communication protocol for synchrophasors, or IEC 61850-90-5 standard), latency is measured as the round-trip time of a packet. This first evaluation focuses on the additional latency introduced by Kubernetes and Liqo in the proposed infrastructure by measuring the latency between the different clusters.

Firstly, to establish a baseline for the tests, the network latency is calculated by averaging the mean of values obtained from the virtual machines. Figure 5.8 shows that the measured network latency in our testbed is negligible (as the VMs

[5] https://k3s.io/
[6] https://github.com/GridProtectionAlliance/openPDC
[7] https://github.com/percona/percona-xtradb-cluster-operator

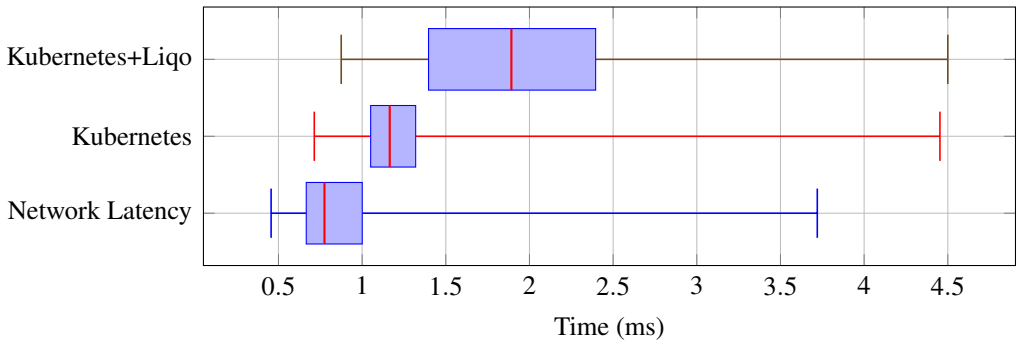


Fig. 5.8 Latency contribution of the liquid computing expressed in terms of network and infrastructure latency. Overall, the liquid computing infrastructure introduces 1.3ms of latency, well below the requirements of monitoring applications.

are all executed within the same data center) and is approximately 0.8ms. Then, the latency introduced by the infrastructure is evaluated. In fact, Liqo and Kubernetes both handle and forward the traffic flows, introducing potential sources of overhead. To this end, we evaluated the time difference between the moment in which a packet is received by the application, and the time the packet was generated from another application running on a different cluster (i.e., different machine). As a reference, we compare the latency introduced by liquid computing with the latency for the same communication in case all compute nodes belong to the same Kubernetes cluster (1.2ms as a reference value). Specifically, we estimated this total (end-to-end) latency of the Liquid infrastructure to be on average 1.9ms, which allows us to estimate the infrastructure latency as the difference between the total latency and the network latency. As a result, the infrastructure only introduces a fixed latency contribution of 1.1ms, which is two orders of magnitude smaller than the network latency experienced in most wide area network installations for the IT system and well below the requirements of the monitoring applications. Finally, it is also worth mentioning that the maximum latency measured for liquid computing is 4.5ms, but this is most likely due to our testbed facility experiencing temporary latency spikes as the same behavior can be observed by considering only the network latency.

[Availability] Data stream

The combination of Liqo and Kubernetes preserves the functionality of the monitoring infrastructure in case of disconnection of part of the clusters from the main communication network (a.k.a. *island mode*). In fact, assuming that a substation

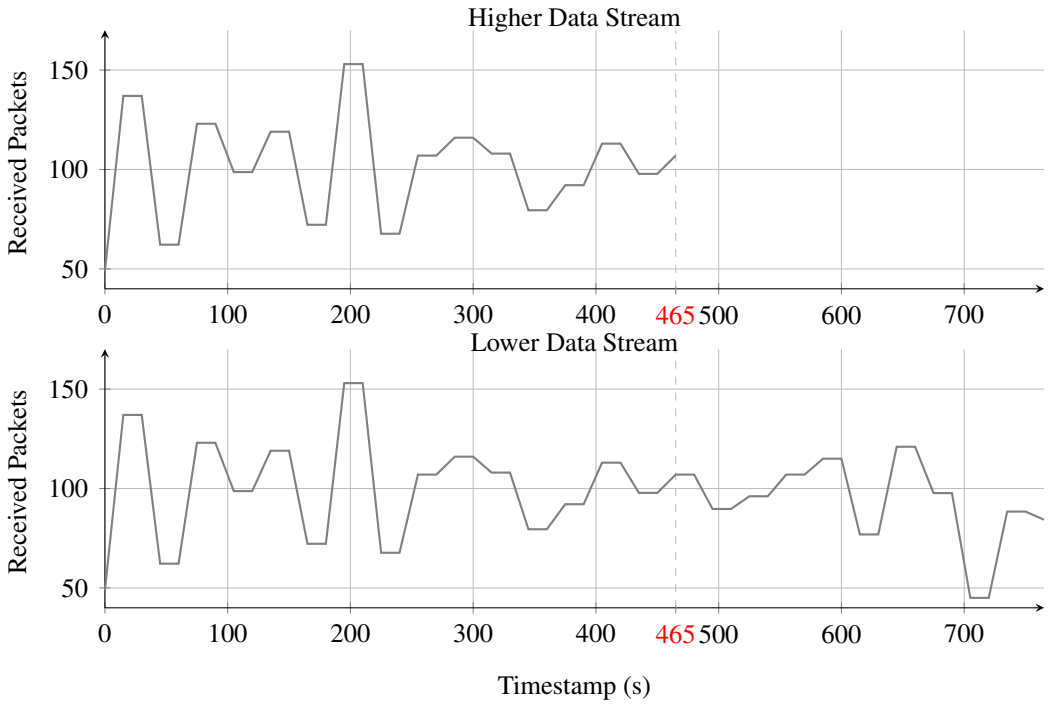


Fig. 5.9 Data Stream continuity in case of failure. In case of a disconnection from the main network grid, services running on a different cluster experience a downtime in the data flow (higher data stream), however, services running on the isolated site are not affected and continue to seamlessly operate (lower data stream).

and the controlled substations detach from the rest of the infrastructure as a consequence of a disruptive event, the execution of the applications therein deployed is not affected, and enhanced control logic can be implemented to instantiate essential applications in the isolated site.

The continuation of operations can be observed in Figure 5.9, which illustrates the data stream represented as the number of received packets seen by an instance of PDC lower and its directly superior PDC higher, shortly before and shortly after the disconnection of the cluster hosting the PDC lower and its data sources, which occurred at 465s. The PDC lower continues to receive data from the sources, operating in an isolated environment, while PDC higher stops receiving the data stream from the isolated source. As a result, the data is still collected from the lower level PDC, potentially used by applications scheduled locally or stored in a local database, and can be later forwarded to the higher level PDC as soon as the connection is restored.

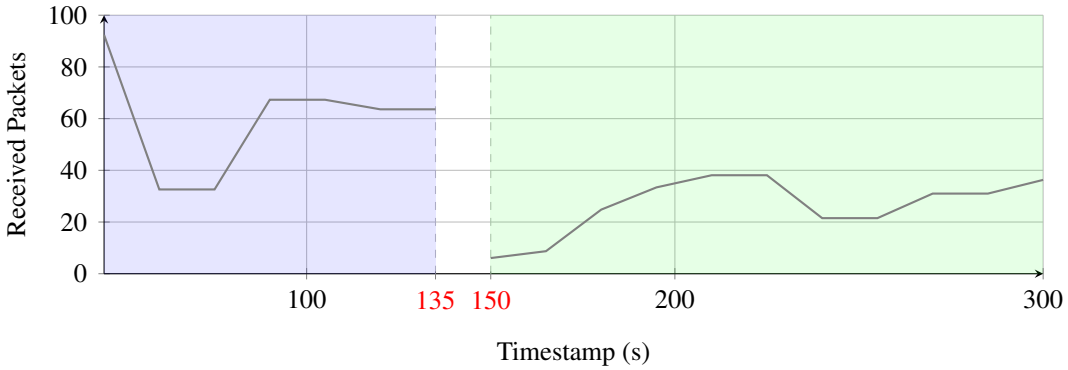
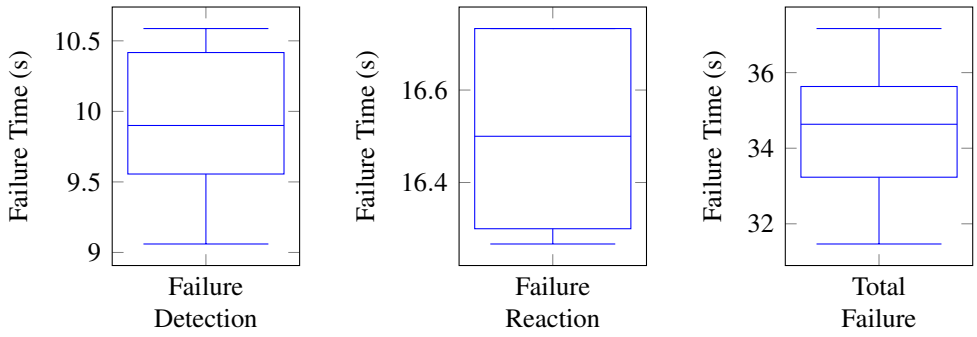


Fig. 5.10 Data Stream continuity in case of failure. In case of a disconnection from the main network grid, applications running on remote clusters can be instantiated locally to cope with the loss of connectivity. The figure combines the data stream received by the first PDC instance (blue), and the data stream received by the new PDC instance (green).

In addition, the isolated site still maintains control over the computing resources and can adopt (if needed) countermeasures to overcome the lost of connectivity. In the scenario depicted in Figure 5.10, the isolated site detects the connectivity failure with the upstream portion of the network, and automatically deploys a replica of the PDC instance locally. This can be achieved by enhanced control logic deployed in the clusters of the infrastructure with the task of checking the status of the communication. If issues are detected, then local instances of the remote application can be enforced locally. As a result, once the new replica is ready, the communication can be re-instantiated. Specifically, in Figure 5.10 we represented the data stream seen by the PDC before the failure at time 135s, and the data stream seen by the newly instantiated replica at time 150s. Additionally, we can also observe that the new replica receives quite fewer packets compared to the initial one, due to the fewer endpoints to scrape information from. It is worth mentioning that this self-healing process can be customized depending on the requirements of the application (e.g., maximum acceptable downtime, ...). In fact, high response time can reduce downtime, but at the same time, it might affect the overall system performance in case of intermittent communication channels.



(a) Reaction time to identify a cluster failure from the remaining (healthy) clusters in the infrastructure.

(b) Synchrophasor flow downtime in the event of a failure in the PDC application.

(c) Median time to recover after a disruptive network event.

Fig. 5.11 Reaction times with liquid computing.

[Resiliency] Reaction time

In the following, we evaluate the time required for the OC to detect a disconnection from any of the downstream substations. This time is, in fact, crucial to evaluate the responsiveness of the infrastructure to implement countermeasures in the event of network partitioning events. Specifically, unforeseen and disruptive network events are replicated in the testbed by randomly selecting one of the clusters, disabling the network interface of the machine (i.e., isolating the machine from the rest of the network) while constantly checking at the Kubernetes level in the OC when such failure is detected.

As we can see from Figure 5.11a a failure in one of the substations can be always detected within 11s (we label this time as *detection* time). This is due to the fact that clusters constantly report to each other their health status. The absence of such checks would notify the occurrence of a disruptive event. As a result, if the disconnected substation is hosting the PMU, then also the stream of data will be interrupted until the substation becomes reachable again. In this case, we assume no countermeasure is possible if the PMU and the corresponding measuring sensors are unreachable. Instead, if the substation is hosting the PDC, after approximately 10s on average it is possible to implement countermeasures to re-schedule some of the applications hosted in the failed cluster into a suitable location. It is worth mentioning that such mechanism is natively provided by Kubernetes and will be shown in following section.

[Resiliency] Stream reaction time

The following tests measure the downtime of a data stream originating from the PMU in the following scenarios: (*i*) Internal failure of a PDC, i.e., unexpected failure in the application, resulting in the rescheduling of the same application (we label this as *reaction* time). (*ii*) Fault/disconnection of the cluster hosting a PDC, i.e., unexpected failure of the cluster hosting the PDC, resulting in the application being rescheduled to another cluster (ideally, equal to *detection* + *reaction* time).

The downtime is calculated from the timestamp of the last data frame of the old stream to the timestamp of the first data frame of the new stream. The use of the data frame timestamp is crucial due to the PMU's real-time production of data frames at 40 millisecond intervals (based on EU frequency standard), ensuring precise downtime calculations and analysis.

PDC Failure The failure scenario of the PDC was simulated by raising a custom exception to the PDC application. As a result, the Kubernetes control plane immediately identify the error state and takes care of starting a new healthy replica either in the same cluster or in another cluster belonging to the same group. The test results, displayed in Figure 5.11b, illustrate the *reaction* time as the median duration required for the lower-level PDC application to resume normal operation. Specifically, this duration encompasses the time from detecting the PDC failure to its subsequent recovery, including the processes of restarting the a new instance of the application, retrieving configurations from the system database, and re-establishing connection to the data stream. Overall, this entirely automated process requires approximately 16s to converge to a healthy state. It is worth mentioning that this time is strictly related to the application. In fact, the code used to replicate the PDC functionalities had been adapted to operate in cloud and multi-cloud environments, but it is not designed to follow the best practices for cloud application development. This means that a fully cloud-native PDC application (which is not available at the time of writing this paper) would drastically reduce the measured downtime (as a reference, a fully cloud-native web server like Nginx in a similar scenario would require 1-2s to recover).

Cluster Failure The failure scenario was simulated by deliberately disabling the network interface of the cluster hosting the lower-level PDC. The deployment configuration of the lower-level PDC includes specific affinities to ensure that in the

event of rescheduling, it can only be placed on another cluster belonging to the same group. Compared to the results in Section 5.6.2 the focus shifts from the Kubernetes perspective to the application perspective. In fact, detecting a cluster failure is just an intermediate step in the recovery process and might not indicate whether the data flow is resumed as well.

Figure 5.11c illustrates the median duration required for the lower-level PDC application to resume normal operation. This duration includes the time required for the cluster to detect that the virtual node hosting the PDC is unreachable (9.75 seconds as shown in Figure 5.11a), the waiting time before it can be rescheduled to another node (5 seconds on average as a configurable parameter in Kubernetes), and the time necessary for the application to restart (16 seconds as shown in Figure 5.11b). Therefore, we can conclude that, upon disruptive events in one of the substations, the correct flow of data can be automatically resumed in approximately 30s.

5.7 Conclusions

This work has addressed the challenges associated with integrating distributed energy resources (DERs) into modern smart grids by proposing a novel architectural paradigm, Liquid Computing. This paradigm extends traditional cloud and edge computing frameworks, forming a unified and dynamic computing continuum. The proposed infrastructure effectively automates application lifecycle management and ensures high levels of resiliency by leveraging the dynamic relocation of workloads across distributed clusters. Experimental results show that the additional latency introduced by the Liquid Computing approach is negligible for monitoring applications, and the architecture supports seamless data continuity even under network partitioning scenarios. Additionally, the adoption of advanced containerization and orchestration technologies, such as Kubernetes and Liqo, underpins the feasibility of implementing this architecture within the constraints of current smart grid ICT systems.

Chapter 6

Concluding Remarks

The rapid evolution of distributed computing has underscored the need for systems capable of overcoming the limitations of fragmented infrastructures. This dissertation proposes the computing continuum as a holistic solution, unifying cloud, edge, and fog environments into a seamless and adaptive architecture. By addressing challenges in resource discovery, task allocation, energy efficiency, and resilience, this research advances the state of the art in distributed orchestration.

The introduction of FLUIDOS and the REAR protocol demonstrates how intent-based resource negotiation and dynamic orchestration can enable efficient and sustainable computing. The energy-aware and carbon-conscious allocation strategies presented here emphasize the importance of integrating environmental considerations into system design, offering pathways toward a greener computing paradigm. Furthermore, the fault-tolerant frameworks developed in this work highlight the potential for supporting critical applications in highly dynamic and resource-constrained scenarios, ensuring uninterrupted service delivery even in the face of disruptions.

Future research can build upon this work by exploring the integration of emerging technologies such as artificial intelligence and machine learning to further optimize resource allocation and system adaptability. Additionally, expanding the scope of FLUIDOS to include broader interoperability standards and cross-domain collaborations will enhance its applicability in diverse contexts.

References

[1] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[2] NIST (National Institute of Standards and Technology). The nist definition of cloud computing. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf, 2011.

[3] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges, 2015.

[4] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[5] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

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

[7] Luciano Baresi, Danilo Filgueira Mendonça, Martin Garriga, Sam Guinea, and Giovanni Quattrocchi. A unified model for the mobile-edge-cloud continuum. *ACM Transactions on Internet Technology (TOIT)*, 19(2):1–21, 2019.

[8] Mulugeta Ayalew Tamiru, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. Instability in geo-distributed kubernetes federation: Causes and mitigation. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2020.

[9] Lars Larsson, William Tärneberg, Cristian Klein, Erik Elmroth, and Maria Kihl. Impact of etcd deployment on kubernetes, istio, and application performance. *Software: Practice and experience*, 50(10):1986–2007, 2020.

[10] Lirim Osmani, Tero Kauppinen, Miika Komu, and Sasu Tarkoma. Multi-cloud connectivity for kubernetes in 5g networks. *IEEE Communications Magazine*, 59(10):42–47, 2021.

[11] Abdullah Yousafzai, Abdullah Gani, Rafidah Md Noor, Mehdi Sookhak, Hamid Talebian, Muhammad Shiraz, and Muhammad Khurram Khan. Cloud resource allocation schemes: review, taxonomy, and opportunities. *Knowledge and information systems*, 50:347–381, 2017.

[12] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco AS Netto, et al. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM computing surveys (CSUR)*, 51(5):1–38, 2018.

[13] Pieter-Jan Maenhaut, Bruno Volckaert, Veerle Ongenae, and Filip De Turck. Resource management in a containerized cloud: Status and challenges. *Journal of Network and Systems Management*, 28:197–246, 2020.

[14] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. Computing without borders: The way towards liquid computing. *IEEE Transactions on Cloud Computing*, 11(3):2820–2838, 2022.

[15] Stefano Galantino, Elisa Albanese, Nasir Asadov, Stefano Braghin, Francesco Cappa, Andrea Colli-Vignarelli, Amjad Yousef Majid, Eduard Marin, Jacopo Marino, Lorenzo Moro, et al. Building the cloud continuum with rear. In *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*, pages 67–72. IEEE, 2024.

[16] The FLUIDOS consortium. Fluidos public deliverable - d2.1 scenarios, requirements and reference architecture. https://fluidos.eu/public-deliverables/, 2023.

[17] The FLUIDOS consortium. Fluidos public deliverable - d2.2 scenarios, requirements and reference architecture. https://fluidos.eu/public-deliverables/, 2024.

[18] Gabriele Castellano, Stefano Galantino, Fulvio Risso, and Antonio Manzalini. Scheduling multi-component applications across federated edge clusters with phare. *IEEE Open Journal of the Communications Society*, 2024.

[19] Stefano Galantino, Fulvio Risso, Vlad C Coroamă, and Antonio Manzalini. Assessing the potential energy savings of a fluidified infrastructure. *Computer*, 56(6):26–34, 2023.

[20] Stefano Galantino, Andrea Pinto, Flavio Esposito, Antonio Manzalini, and Fulvio Risso. Balancing energy efficiency and infrastructure knowledge in cloud-to-edge task distribution systems. In *Proceedings of the 1st International Workshop on MetaOS for the Cloud-Edge-IoT Continuum*, pages 28–34, 2024.

[21] Stefano Galantino, Fulvio Risso, Andrea Cazzaniga, Fabrizio Garrone, Roberta Terruggia, and Riccardo Lazzari. An edge-based architecture for

phasor measurements in smart grids. In *2022 AEIT International Annual Conference (AEIT)*, pages 1–6. IEEE, 2022.

[22] James L Peterson and Abraham Silberschatz. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[23] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.

[24] Muhammad Zeeshan Khan, Saad Harous, Saleet Ul Hassan, Muhammad Usman Ghani Khan, Razi Iqbal, and Shahid Mumtaz. Deep unified model for face recognition based on convolution neural network and edge computing. *IEEE Access*, 7:72622–72633, 2019.

[25] Lars Nagel, Juan Jose Hierro, Eugenio Perea, Douwe Lycklama, Christoph Mertens, Anne-Sophie Taillandier, Maria Marques, Joshua Gelhaar, Angelo Marguglio, Ulrich Ahle, et al. Design principles for data spaces: Position paper. Technical report, E. ON Energy Research Center, 2021.

[26] Jacopo Marino, Leonardo Camiciotti, Francesco Cheinasso, Alessandro Olivero, and Fulvio Risso. Enabling compute and data sovereignty with infrastructure-level data spaces. In *Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum*, pages 77–85, 2023.

[27] Loris Belcastro, Fabrizio Marozzo, Alessio Orsino, Domenico Talia, and Paolo Trunfio. Edge-cloud continuum solutions for urban mobility prediction and planning. *IEEE Access*, 11:38864–38874, 2023.

[28] Xiaojie Zhang and Saptarshi Debroy. Resource management in mobile edge computing: a comprehensive survey. *ACM Computing Surveys*, 55(13s):1–37, 2023.

[29] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[30] Liqo. Liqo official website. https://liqo.io/, 2024.

[31] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[32] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: bringing the cloud to the mobile user. In *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, MCS '12, page 29–36, New York, NY, USA, 2012. Association for Computing Machinery.

[33] Tom Goethals, Filip De Turck, and Bruno Volckaert. Extending kubernetes clusters to low-resource edge devices using virtual kubelets. *IEEE Transactions on Cloud Computing*, 10(4):2623–2636, 2022.

[34] Lars Larsson, Harald Gustafsson, Cristian Klein, and Erik Elmroth. Decentralized kubernetes federation control plane. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 354–359, 2020.

[35] Aris Leivadeas and Matthias Falkner. A survey on intent-based networking. *IEEE Communications Surveys & Tutorials*, 25(1), 2022.

[36] Barbara Martini, Molka Gharbaoui, and Piero Castoldi. Intent-based network slicing for sdn vertical services with assurance: Context, design and preliminary experiments. *Future Generation Computer Systems*, 142, 2023.

[37] Ankur Chowdhary, Abdulhakim Sabur, Neha Vadnere, and Dijiang Huang. Intent-driven security policy management for software-defined systems. *IEEE Transactions on Network and Service Management*, 19(4), 2022.

[38] Tarandeep Kaur and Inderveer Chana. Energy Efficiency Techniques in Cloud Computing: A Survey and Taxonomy. *ACM Computing Surveys*, 48(2), October 2015.

[39] Michal Sedlacko, Andre Martinuzzi, and Karin Dobernig. A systems thinking view on cloud computing and energy consumption. In *ICT for Sustainability 2014 (ICT4S-14)*, pages 95–102. Atlantis Press, 2014.

[40] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of Historical Trends in the Electrical Efficiency of Computing. *IEEE Annals of the History of Computing*, 33(3), March 2011.

[41] Ana Radovanović, Ross Koningstein, Ian Schneider, Bokan Chen, Alexandre Duarte, Binz Roy, Diyue Xiao, Maya Haridasan, Patrick Hung, Nick Care, et al. Carbon-aware computing for datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, 2022.

[42] Young Geun Kim, Udit Gupta, Andrew McCrabb, Yonglak Son, Valeria Bertacco, David Brooks, and Carole-Jean Wu. Greenscale: Carbon-aware systems for edge computing. *arXiv preprint arXiv:2304.00404*, 2023.

[43] Thanathorn Sukprasert, Abel Souza, Noman Bashir, David E Irwin, and Prashant J Shenoy. Quantifying the benefits of carbon-aware temporal and spatial workload shifting in the cloud. *CoRR*, 2023.

[44] Noman Bashir, David Irwin, and Prashant Shenoy. On the Promise and Pitfalls of Optimizing Embodied Carbon. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23, New York, NY, USA, August 2023. ACM.

[45] Jaylen Wang, Udit Gupta, and Akshitha Sriraman. Peeling Back the Carbon Curtain: Carbon Optimization Challenges in Cloud Computing. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23, New York, NY, USA, August 2023. ACM.

[46] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. ACT: designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, New York, NY, USA, June 2022. ACM.

[47] Christoph Buck, Christian Olenberger, André Schweizer, Fabiane Völter, and Torsten Eymann. Never trust, always verify: a multivocal literature review on current knowledge and research gaps of zero-trust. *Computers & Security*, 110, 2021.

[48] World Wide Web Consortium (W3C). Decentralized Identifiers (DIDs) v1.0: Core architecture, data model, and representations, 2022. Accessed on April, 2024.

[49] W3C. Verifiable Credentials Data Model v1.1, 2022. Accessed on April, 2024.

[50] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. A survey on essential components of a self-sovereign identity. *Computer Science Review*, 30, 2018.

[51] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. Rsvp: A new resource reservation protocol. *IEEE network*, 7(5), 1993.

[52] Anup Kumar Talukdar, BR Badrinath, and Arup Acharya. Mrsvp: A resource reservation protocol for an integrated services network with mobile hosts. *Wireless Networks*, 7, 2001.

[53] Xin Wang and Henning Schulzrinne. Rnap: A resource negotiation and pricing protocol. *Transit*, 6(B7), 1999.

[54] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. Rfc3209: Rsvp-te: Extensions to rsvp for lsp tunnels, 2001.

[55] Karl Czajkowski, Ian Foster, Carl Kesselman, Volker Sander, and Steven Tuecke. Snap: A protocol for negotiating service level agreements and coordinating resource management in distributed systems. In *Job Scheduling Strategies for Parallel Processing: 8th International Workshop, JSSPP 2002 Edinburgh, Scotland, UK, July 24, 2002*. Springer, 2002.

[56] Srikumar Venugopal, Xingchen Chu, and Rajkumar Buyya. A negotiation mechanism for advance resource reservations using the alternate offers protocol. In *2008 16th Interntional Workshop on Quality of Service*. IEEE, 2008.

[57] Erik Elmroth and Johan Tordsson. A grid resource broker supporting advance reservations and benchmark-based resource selection. In *International Workshop on Applied Parallel Computing*. Springer, 2004.

[58] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials*, 19(3):1657–1681, 2017.

[59] Ramraj Dangi, Praveen Lalwani, Gaurav Choudhary, Ilsun You, and Giovanni Pau. Study and investigation on 5g technology: A systematic review. *Sensors*, 22(1):26, 2021.

[60] GSMA. Gsma operator platform telco edge requirements 2022, 2022. Accessed on March, 2024.

[61] GSMA. Gsma operator platform group – east-westbound interface apis, 2023. Accessed on March, 2024.

[62] IB Otto, S Steinbuß, A Teuscher, IS Lohmann, A Auer, S Bader, H Bastiaansen, H Bauer, IP Birnstil, and M Böhmer. International data spaces association—reference architecture model—version 3.0, 2019.

[63] Kubernetes. Kubernetes website. https://kubernetes.io/, 2024.

[64] Apache Hadoop. Apache hadoop website. https://hadoop.apache.org/, 2024.

[65] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[66] Vojislav Dukic, Ginni Khanna, Christos Gkantsidis, Thomas Karagiannis, Francesca Parmigiani, Ankit Singla, Mark Filer, Jeffrey L Cox, Anna Ptasznik, Nick Harland, et al. Beyond the mega-data center: networking multi-data center regions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 765–781, 2020.

[67] Kenneth Church, Albert G Greenberg, and James R Hamilton. On delivering embarrassingly distributed cloud services. In *HotNets*, pages 55–60, 2008.

[68] Iyswarya Narayanan, Aman Kansal, Anand Sivasubramaniam, Bhuvan Urgaonkar, and Sriram Govindan. Towards a leaner geo-distributed cloud infrastructure. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.

[69] Francesco Lucrezia, Guido Marchetto, Fulvio Risso, and Vinicio Vercellone. Introducing network-aware scheduling capabilities in openstack. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–5, 2015.

[70] Apache Flink. Apache flink website. https://flink.apache.org/, 2024.

[71] Apache Storm. Apache storm website. https://storm.apache.org/, 2024.

[72] Apache Kafka. Apache kafka website. https://kafka.apache.org/, 2024.

[73] Apache Solr. Apache solr website. https://solr.apache.org/, 2024.

[74] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[75] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.

[76] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[77] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[78] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. George: Learning to place long-lived containers in large clusters with operation constraints. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 258–272, 2021.

[79] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 99–115, 2016.

[80] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 497–509, 2016.

[81] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 135–148, 2018.

[82] Frédéric Giroire, Nicolas Huin, Andrea Tomassilli, and Stéphane Pérennes. When network matters: Data center scheduling with network tasks. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2278–2286. IEEE, 2019.

[83] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 977–987. IEEE, 2017.

[84] Jiaying Meng, Haisheng Tan, Chao Xu, Wanli Cao, Liuyan Liu, and Bojie Li. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2287–2295. IEEE, 2019.

[85] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, 2016.

[86] Andrew Chung, Jun Woo Park, and Gregory R Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 121–134, 2018.

[87] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[88] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[89] Luís Alexandre Rocha and Fábio Luciano Verdi. A network-aware optimization for vm placement. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 619–625. IEEE, 2015.

[90] Marwa A Abdelaal, Gamal A Ebrahim, and Wagdy R Anis. Network-aware resource management strategy in cloud computing environments. In *2016 11th International Conference on Computer Engineering & Systems (ICCES)*, pages 26–31. IEEE, 2016.

[91] Federico Larumbe and Brunilde Sansò. Elastic, on-line and network aware virtual machine placement within a data center. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 28–36. IEEE, 2017.

[92] Leilei Wang, Xiaoheng Deng, Jinsong Gui, Xuechen Chen, and Shaohua Wan. Microservice-oriented service placement for mobile edge computing in sustainable internet of vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 2023.

[93] Zehong Lin, Suzhi Bi, and Ying-Jun Angela Zhang. Optimizing ai service placement and resource allocation in mobile edge intelligence systems. *IEEE Transactions on Wireless Communications*, 20(11):7257–7271, 2021.

[94] Lixing Chen, Cong Shen, Pan Zhou, and Jie Xu. Collaborative service placement for edge computing in dense small cell networks. *IEEE Transactions on Mobile Computing*, 20(2):377–390, 2019.

[95] Xin Li, Zhen Lian, Xiaolin Qin, and Wu Jie. Topology-aware resource allocation for iot services in clouds. *IEEE Access*, 6:77880–77889, 2018.

[96] Bon Ryu, Aijun An, Zana Rashidi, Junfeng Liu, and Yonggang Hu. Towards topology aware pre-emptive job scheduling with deep reinforcement learning. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, pages 83–92, 2020.

[97] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3:{Energy-Efficient} microservices on {SmartNIC-Accelerated} servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[98] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications*, 159:161–174, 2020.

[99] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Towards delay-aware container-based service function chaining in fog computing. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2020.

[100] Balázs Németh, Márk Szalay, János Dóka, Matthias Rost, Stefan Schmid, László Toka, and Balázs Sonkoly. Fast and efficient network service embedding method with adaptive offloading to the edge. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 178–183. IEEE, 2018.

[101] Vinu Sundararaj. Optimal task assignment in mobile cloud computing by queue based ant-bee algorithm. *Wireless Personal Communications*, 104(1):173–197, 2019.

[102] Gabriele Castellano, Flavio Esposito, and Fulvio Risso. A distributed orchestration algorithm for edge computing resources with guarantees. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2548–2556. IEEE, 2019.

[103] Onur Ascigil, Truong Khoa Phan, Argyrios G Tasiopoulos, Vasilis Sourlas, Ioannis Psaras, and George Pavlou. On uncoordinated service placement in edge-clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 41–48. IEEE, 2017.

[104] Ibrahim Alghamdi, Christos Anagnostopoulos, and Dimitrios P Pezaros. Delay-tolerant sequential decision making for task offloading in mobile edge computing environments. *Information*, 10(10):312, 2019.

[105] Hossein Badri, Tayebeh Bahreini, Daniel Grosu, and Kai Yang. Energy-aware application placement in mobile edge computing: A stochastic optimization approach. *IEEE Transactions on Parallel and Distributed Systems*, 31(4):909–922, 2019.

[106] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[107] Richard E Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181–203, 1998.

[108] Deepak Vij and Shivram Shrivastava. Poseidon-firmament scheduler – flow network graph based scheduler. https://github.com/kubernetes-sigs/poseidon. Accessed: 2021-08-23.

[109] Ionel Corneliu Gog. *Flexible and efficient computation in large data centres*. PhD thesis, University of Cambridge, 2018.

[110] Malte Schwarzkopf. *Operating system support for warehouse-scale computing*. PhD thesis, University of Cambridge, 2018.

[111] Stefanos Kaxiras and Margaret Martonosi. Computer architecture techniques for power-efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[112] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-efficient cloud computing. *The computer journal*, 53(7):1045–1051, 2010.

[113] Amir Vahid Dastjerdi and Rajkumar Buyya. Fog computing: Helping the internet of things realize its potential. *Computer*, 49(8):112–116, 2016.

[114] Pete Beckman, Jack Dongarra, Nicola Ferrier, Geoffrey Fox, Terry Moore, Dan Reed, and Micah Beck. Harnessing the computing continuum for programming our world. *Fog Computing: Theory and Practice*, pages 215–230, 2020.

[115] Rohan Kumar, Matt Baughman, Ryan Chard, Zhuozhao Li, Yadu Babuji, Ian Foster, and Kyle Chard. Coding the computing continuum: Fluid function execution in heterogeneous computing environments. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 66–75. IEEE, 2021.

[116] Daniel Rosendo, Pedro Silva, Matthieu Simonin, Alexandru Costan, and Gabriel Antoniu. E2clab: Exploring the computing continuum through repeatable, replicable and reproducible edge-to-cloud experiments. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 176–186. IEEE, 2020.

[117] Andre Luckow, Kartik Rattan, and Shantenu Jha. Pilot-edge: Distributed resource management along the edge-to-cloud continuum. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 874–878. IEEE, 2021.

[118] Daniel Balouek-Thomert, Eduard Gibert Renart, Ali Reza Zamani, Anthony Simonet, and Manish Parashar. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The International Journal of High Performance Computing Applications*, 33(6):1159–1174, 2019.

[119] Víctor Casamayor Pujol, Philipp Raith, and Schahram Dustdar. Towards a new paradigm for managing computing continuum applications. In *2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI)*, pages 180–188. IEEE, 2021.

[120] Shiwei Cao, Xiaofeng Tao, Yanzhao Hou, and Qimei Cui. An energy-optimal offloading algorithm of mobile computing based on hetnets. In *2015 International Conference on Connected Vehicles and Expo (ICCVE)*, pages 254–258. IEEE, 2015.

[121] Maofei Deng, Hui Tian, and Bo Fan. Fine-granularity based application offloading policy in cloud-enhanced small cell networks. In *2016 IEEE International Conference on Communications Workshops (ICC)*, pages 638–643. IEEE, 2016.

[122] Yun Zhao, Sheng Zhou, Tianchu Zhao, and Zhisheng Niu. Energy-efficient task offloading for multiuser mobile cloud computing. In *2015 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 1–5. IEEE, 2015.

[123] David Perez Abreu, Karima Velasquez, Marilia Curado, and Edmundo Monteiro. A comparative analysis of simulators for the cloud to fog continuum. *Simulation Modelling Practice and Theory*, 101:102029, 2020.

[124] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[125] Jungmin Son, Amir Vahid Dastjerdi, Rodrigo N Calheiros, Xiaohui Ji, Young Yoon, and Rajkumar Buyya. Cloudsimsdn: Modeling and simulation of software-defined cloud data centers. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 475–484. IEEE, 2015.

[126] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[127] Isaac Lera, Carlos Guerrero, and Carlos Juiz. Yafs: A simulator for iot scenarios in fog computing. *IEEE Access*, 7:91745–91758, 2019.

[128] Changshi Yang, Xianping Wang, Ying Luo, and George Stefanek. Overcome it lab challenge in covid-19 with windows-to-go. In *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 0791–0796. IEEE, 2020.

[129] Boris Ryabko and Anton Rakitskiy. An analytic method for estimating the computation capacity of computing devices. *Journal of Circuits, Systems and Computers*, 26(05):1750086, 2017.

[130] Amal A Alahmadi, Taisir EH El-Gorashi, and Jaafar MH Elmirghani. Energy efficient processing allocation in opportunistic cloud-fog-vehicular edge cloud architectures. *arXiv preprint arXiv:2006.14659*, 2020.

[131] Marios Avgeris, Dimitrios Spatharakis, Dimitrios Dechouniotis, Aris Leivadeas, Vasileios Karyotis, and Symeon Papavassiliou. Enerdge: Distributed energy-aware resource allocation at the edge. *Sensors*, 22(2):660, 2022.

[132] Eyhab Al-Masri, Alireza Souri, Habiba Mohamed, Wenjun Yang, James Olmsted, and Olivera Kotevska. Energy-efficient cooperative resource allocation and task scheduling for internet of things environments. *Internet of Things*, 23:100832, 2023.

[133] Hongbo Jiang, Xingxia Dai, Zhu Xiao, and Arun K Iyengar. Joint task offloading and resource allocation for energy-constrained mobile edge computing. *IEEE Transactions on Mobile Computing*, 2022.

[134] Barbara Arbanas Ferreira, Tamara Petrović, Matko Orsag, J Ramiro Martínez-de Dios, and Stjepan Bogdan. Distributed allocation and scheduling of tasks with cross-schedule dependencies for heterogeneous multi-robot teams. *arXiv preprint arXiv:2109.03089*, 2021.

[135] Li Lin, Peng Li, Jinbo Xiong, and Mingwei Lin. Distributed and application-aware task scheduling in edge-clouds. In *2018 14th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pages 165–170. IEEE, 2018.

[136] Joanna Turner, Qinggang Meng, Gerald Schaefer, and Andrea Soltoggio. Fast consensus for fully distributed multi-agent task allocation. In *Proceedings of the 33rd annual ACM symposium on applied computing*, pages 832–839, 2018.

[137] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[138] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[139] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[140] Wei Hua, Peng Liu, and Linyu Huang. Energy-efficient resource allocation for heterogeneous edge-cloud computing. *IEEE Internet of Things Journal*, 2023.

[141] Mian Guo, Lei Li, and Quansheng Guan. Energy-efficient and delay-guaranteed workload allocation in iot-edge-cloud computing systems. *IEEE Access*, 7:78685–78697, 2019.

[142] Changsheng You, Kaibin Huang, Hyukjin Chae, and Byoung-Hoon Kim. Energy-efficient resource allocation for mobile-edge computation offloading. *IEEE Transactions on Wireless Communications*, 16(3):1397–1411, 2016.

[143] João Abel Peças Lopes, André Guimarães Madureira, Manuel Matos, Ricardo Jorge Bessa, Vítor Monteiro, João Luiz Afonso, Sérgio F Santos, João PS Catalão, Carlos Henggeler Antunes, and Pedro Magalhães. The future of power systems: Challenges, trends, and upcoming paradigms. *Wiley Interdisciplinary Reviews: Energy and Environment*, 9(3):e368, 2020.

[144] Fiaz Ahmad, Akhtar Rasool, Emre Ozsoy, Raja Sekar, Asif Sabanovic, and Meltem Elitaş. Distribution system state estimation-a step towards smart grid. *Renewable and Sustainable Energy Reviews*, 81:2659–2671, 2018.

[145] D.M. Falcao, F.F. Wu, and L. Murphy. Parallel and distributed state estimation. *IEEE Transactions on Power Systems*, 10(2):724–730, 1995.

[146] Alexandra von Meier, David Culler, Alex McEachern, and Reza Arghandeh. Micro-synchrophasors for distribution systems. In *ISGT 2014*, pages 1–5, 2014.

[147] Xinghuo Yu and Yusheng Xue. Smart grids: A cyber–physical systems perspective. *Proceedings of the IEEE*, 104(5):1058–1070, 2016.

[148] Hamed Mohsenian-Rad, Emma M. Stewart, and Ed Cortez. Distribution synchrophasors: Pairing big data with analytics to create actionable information. *IEEE Power and Energy Magazine*, 16:26–34, 2018.

[149] Junqi Liu, Junjie Tang, Ferdinanda Ponci, Antonello Monti, Carlo Muscas, and Paolo Attilio Pegoraro. Trade-offs in pmu deployment for state estimation in active distribution grids. *IEEE Transactions on Smart Grid*, 3:915–924, 2012.

[150] Jaime De La Ree, Virgilio Centeno, James S. Thorp, and Arun G. Phadke. Synchronized phasor measurement applications in power systems. *IEEE Transactions on Smart Grid*, 1:20–27, 2010.

[151] Rich Hunt, Sean Kantra, Damir Novosel, Julio Romero Aguero, Daniel W Dietmeyer, and Tariq Rahman. Roadmap for distribution synchronized measurements. In *2022 International Conference on Smart Grid Synchronized Measurements and Analytics (SGSMA)*, pages 1–6. IEEE, 2022.

[152] IEEE. Ieee guide for synchronization, calibration, testing, and installation of phasor measurement units (pmus) for power system protection and control. *IEEE Std C37.242-2021 (Revision of IEEE Std C37.242-2013)*, pages 1–98, 2021.

[153] Song Zhang, Amritanshu Pandey, Xiaochuan Luo, Maggy Powell, Ranjan Banerji, Lei Fan, Abhineet Parchure, and Edgardo Luzcando. Practical adoption of cloud computing in power systems—drivers, challenges, guidance, and real-world use cases. *IEEE Transactions on Smart Grid*, 13(3):2390–2411, 2022.

[154] Marco Pau, Markus Mirz, Jan Dinkelbach, Padraic Mckeever, Ferdinanda Ponci, and Antonello Monti. A service oriented architecture for the digitalization and automation of distribution grids. *IEEE Access*, 10:37050–37063, 2022.

[155] IEEE. Ieee standard for synchrophasor data transfer for power systems. *IEEE Std C37.118.2-2011 (Revision of IEEE Std C37.118-2005)*, pages 1–53, 2011.

[156] Ketan Maheshwari, Marcus Lim, Lydia Wang, Ken Birman, and Robbert van Renesse. Toward a reliable, secure and fault tolerant smart grid state estimation in the cloud. In *2013 IEEE PES Innovative Smart Grid Technologies Conference (ISGT)*, pages 1–6. IEEE, 2013.

[157] Fengji Luo, Junhua Zhao, Z.Y. Dong, Yingying Chen, Yan Xu, Xin Zhang, and Kit Wong. Cloud-based information infrastructure for next-generation power grid: Conception, architecture, and applications. *IEEE Transactions on Smart Grid*, 7:1–1, 09 2015.

[158] Dave Anderson, Theo Gkountouvas, Ming Meng, Ken Birman, Anjan Bose, Carl Hauser, Eugene Litvinov, Xiaochuan Luo, and Qiang Zhang. Gridcloud: Infrastructure for cloud-based wide area monitoring of bulk electric power grids. *IEEE Transactions on Smart Grid*, 10(2):2170–2179, 2019.

[159] A. Meloni, P.A. Pegoraro, L. Atzori, A. Benigni, and S. Sulis. Cloud-based iot solution for state estimation in smart grids. *Comput. Netw.*, 130(C):156–165, January 2018.

[160] Songlin Chen, Hong Wen, Jinsong Wu, Wenxin Lei, Wenjing Hou, Wenjie Liu, Aidong Xu, and Yixin Jiang. Internet of things based smart grids supported by intelligent edge computing. *IEEE Access*, 7:74089–74102, 06 2019.

[161] Matteo Orlando, Abouzar Estebsari, Enrico Pons, Marco Pau, Stefano Quer, Massimo Poncino, Lorenzo Bottaccioli, and Edoardo Patti. A smart meter infrastructure for smart grid iot applications. *IEEE Internet of Things Journal*, PP:1–1, 12 2021.

[162] Zongsheng Li, Hua Wei, Zhongliang Lyu, and Chunjie Lian. Kubernetes-container-cluster-based architecture for an energy management system. *IEEE Access*, PP:1–1, 05 2021.

[163] James Stoupis, Rostan Rodrigues, Mohammad Razeghi-Jahromi, Amanuel Melese, and Joemoan I. Xavier. Hierarchical distribution grid intelligence: Using edge compute, communications, and iot technologies. *IEEE Power and Energy Magazine*, 21(5):38–47, 2023.

[164] Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *ZEUS*, pages 65–73, 2021.