PhD-FSTM-2024-101
The Faculty of Science, Technology and Medicine

DISSERTATION

Defense held on 25/11/2024 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Samuel, Vadim, Vincent RAC
Born on 13 November 1997 in Nancy, France

OPTIMIZATION AND ORCHESTRATION IN THE CLOUD-TO-EDGE COMPUTING CONTINUUM

Dissertation defense committee

Dr Radu STATE, Chairman
Professor, Université du Luxembourg

Dr Pascal BOUVRY, Vice-Chairman
Professor, Université du Luxembourg

Dr Mats BRORSSON, Supervisor
Research Scientist, Université du Luxembourg

Dr Rajarshi SANYAL
Proximus Luxembourg

Dr Vladimir VLASSOV
Professor, KTH Royal Institute of Technology

This research has been supported by the Luxembourg National Research Fund (FNR) under contract number 16327771 and has been supported by Proximus Luxembourg SA.

Abstract

In this thesis, we improve existing orchestration techniques to address the new challenges the Cloud-to-Edge Computing Continuum raises.

Edge computing is a paradigm that moves computation and storage outside of the data centers to the edge of the network. The Cloud-to-Edge Computing Continuum refers to the aggregation of computing resources from traditional data centers to the edge. Edge Computing extends the capabilities of cloud computing. Deploying servers close to end users reduces the delays and enables new use cases. However, these geographically distributed machines create new challenges; they can be addressed by improving existing cloud computing techniques. In this dissertation, we aim to simplify application deployment in the Cloud-to-Edge Computing Continuum. We present three main contributions that help move toward that goal.

In the first part of this dissertation, we describe an experimental methodology to study orchestration in the Cloud-to-Edge Computing Continuum. We evaluate the performance of a 5G core network deployed in the Computing Continuum to illustrate our methodology.

Then, we propose a new orchestration approach for reducing the costs of deploying applications in the Cloud-to-Edge Computing Continuum. This orchestrator chooses an optimal location for deploying applications in terms of costs and quality of service. It also offers a mechanism to update scheduling decisions when the environment changes. We evaluate this new orchestration approach with a realistic 5G use case: Vehicular Cooperative Perception.

Finally, we study the performances of container CPU limitation mechanisms. Setting limitations is essential to maximize the utilization of servers in the Computing Continuum, especially at the edge, where resources can be more limited. However, the different CPU limitation mechanisms available offer different performances depending on the application type. An inadequate setting could lead to negative impacts on the application's performance. Therefore, we propose a methodology for automatically selecting the best CPU limitation mechanism.

Acknowledgments

A PhD is a long journey, which would not have been possible without help.

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Mats Brorsson, for his unwavering support, guidance, and encouragement throughout my PhD journey.

I am also grateful to my thesis committee members, Prof. Radu State and Dr. Rajarshi Sanyal, for their constructive feedback and suggestions, which greatly improved the quality of my work over these last four years. Many thanks to Prof. Pascal Bouvry and Prof. Vladimir Vlassov for agreeing to review my work. Your time and expertise are greatly appreciated.

I would like to thank the financial support provided by Proximus Luxembourg and the Luxembourg National Research Fund (FNR), without which this research would not have been possible. I would like to extend my special thanks to Rajarshi and Julien from Proximus for their invaluable inputs and support.

I am deeply thankful to all my colleagues from the SEDAN research group. Your collective support, collaboration, and camaraderie have been invaluable throughout this journey. I deeply appreciate your help and friendship.

I am profoundly grateful to my family for their unwavering support and encouragement. To my mother, your love have been the foundation of my success. To my brother and my sister, thank you for always being there for me and for your constant encouragement.

I would also like to express my wholehearted gratitude to my late father. Your memory has been a source of strength and inspiration throughout this journey. I dedicate this thesis to you.

Finally, I am profoundly grateful to Anne-Claire, for your unconditional love, patience, and understanding. Your belief in me and your constant support have been my anchor throughout this journey. Thank you for being always there for me.

Thank you all for your support and belief in me.

Samuel Rac
Luxembourg, September 2024

Contents

List of Figures

List of Tables

Acronyms

AR Augmented Reality. 48, 51

CRI Container Runtime Interface. 23–25

CUPS Control user plane separation. xi, 33

eMBB enhanced Mobile Broad Band. 32, 47, 48

FaaS Function as a Service. 26, 28

FPGA Field-programmable gate arrays. 20, 54

gNB gNodeB. 47–50, 52

GPGPUs General-purpose computing on graphics processing units. 13, 20, 23, 44, 54, 114

IIoT Industrial IoT. 32, 48, 49, 52

K8S Kubernetes. 4, 16

MEC Multi-access Edge Computing. 3, 7, 35, 36

MIoT Massive IoT. 32, 48, 49

mMTC massive Machine-Type Communications. 32, 48

NF Network Function. 50, 53

NFs Network Functions. 41, 42, 46–48, 52–54

NFV Network Function Virtualization. 32, 33

OCI Open Container Initiative. 23, 37

PDU Protocol Data Unit. 47–49, 52, 53, 117

QoS Quality of Service. 1, 8, 16, 28, 81, 86, 87, 92

RAN Radio Access Network. 32, 34, 47, 54

SDN Software-defined networking. 31, 33

SLA Service Level Agreement. 37

SLO Service Level Objective. 29, 38, 86, 87

UE User Equipment. 17, 33, 34, 53, 117

UEs User Equipments. 47–50, 52, 54, 117

URLLC Ultra-Reliable Low-Latency Communications. 32, 47, 48

VNF Virtual Network Function. 55

WASM WebAssembly. 21, 22, 24, 25, 29

Chapter 1.

Introduction

1.1. The Edge Computing Context

Cloud computing has changed the way people develop and use software. With the arrival of widely available generative AI tools (e.g., ChatGPT, Copilot, Gemini), it continues to generate interest as the worldwide public cloud service revenue grows by 42% between 2022 and 2024, from 678.8 to 478.3 billion US dollars 20%, according to the forecast of Gartner [1]. Running the Large Language Models (LLMs) that support generative AI requires powerful and scalable computing capabilities. However, the future of cloud computing is not only in larger and denser data centers but also outside. Outside of the traditional data centers, closer to the end users, at the edge of the network.

The growing adoption of cloud computing can be attributed to its intrinsic features, including on-demand resource allocation, a pay-as-you-go pricing model, and scalability. These aspects provide both cost savings and operational advantages for businesses [2, 3]. However, public clouds are not the perfect tool for hosting every kind of application; there are limitations to what they can offer.

Public clouds offer limited Quality of Service (QoS) to latency-sensitive applications; it requires special efforts to have decent performances. For example, Cloud Gaming requires the gamers to be in the same region as the data centers, and these offers do not have the same latency as dedicated gaming equipment. Cloud gaming is a paradigm where games run in data centers instead of the gamers' devices. Gamer's inputs are sent to the cloud, and video is streamed back to the players' devices. It is important that the gamers and the data center are located in the same area to have low latency. Therefore, edge computing emerged as a new technology that can enable use cases that require very low latency (around 1 ms and below).

Edge Computing

Edge computing is a distributed computing paradigm that moves computation and data storage outside of the data centers, at the edge of the network, where data is produced. Edge computing solves the delays and limitations of cloud computing. When an application runs on a server located at the edge, the communication delays between the server and a user are more likely to be lower than those between a server in a data center.

The definition of the *edge* varies in the literature. From a cloud provider perspective, the point of presence that a user connects to is the edge. In a cyber-physical system (CPS), the edge is likely to be the system to which IoT devices are connected.

The main idea of edge computing is to enable the usage of computing resources that are closer to end users. Therefore, data can be processed locally, where it is created at the edge of the network, close to the users. Processing data at the edge enables lower network delays, reduced bandwidth usage, increased security, and lower energy consumption. Edge computing enables new use cases that were not possible before due to the high latency between the cloud and devices at the edge [4].

Edge computing proposes moving computing capacities to the edge of the network, closer to where data are produced and consumed. However, edge computing raises new challenges. At the edge, devices are more heterogeneous than in the data center, where everything is optimized to achieve economies of scale. Edge devices can be mobile, like a car, which complicates architecture with dynamic topologies. IoT devices produce a considerable amount of data that can be processed at the edge [5].

In this thesis, our aim is to simplify the adoption of edge computing. We propose enhancements to some aspects of orchestrating workloads that are enabled by edge computing.

Edge computing offers many benefits.

Edge computing is a paradigm that moves the computation outside of the data centers at the edge of the network where the data is generated. It is made for applications with strong network requirements, such as reliability and strict response time.

The main benefit of edge computing is **lower latency** [6, 7, 8, 9, 10]. Reduced network delays are the main consequence of a closer geographical location. For optimal latency experience, bringing computing resources close to the end user should be done in addition to performant networking, especially for wireless networking that has more physical limitations than wired networking with optical fibers. Applications need performant networking and edge computing to offer low latency. 5G/6G telecommunication networks and WiFi 6E are the main wireless networking technologies enabling ultra-low latency use cases. Wired networks (mainly optical fibers) already offer satisfying delays for edge computing.

Another benefit of local processing is the **reduction of bandwidth usage** [6, 7]. Data can be processed where it is generated. Therefore, it is possible to use data centers or centrally located servers only for data aggregation. Local processing can also help reduce network congestion for the same reasons.

Local processing security is a double-edged sword (pun intended). On the one hand, it improves security, as data are processed locally. Local processing reduces the attack surface. On the other hand, it can be harder to ensure security at the edge because of the larger number of heterogeneous devices. Also, if many edge nodes could be federated, a distributed environment would have a larger attack surface. Therefore, security is one of the main challenges for edge computing.

Privacy can also be improved with edge computing. Data can be processed locally where it is created. Therefore, sending only processed and/or anonymized data to central locations is possible. This processed data can be anonymized during the processing on edge devices. In this situation, data will remain private, as well as the entire property of the tenant. Federated learning is a machine learning technique that can improve user privacy. It is further described in the next section.

Compliance, it may be necessary to process data locally to comply with some regulations. For example, data privacy laws can stipulate that personal information cannot be transferred outside of the user's jurisdiction.

Edge computing commercial offers

Large public cloud providers have started offering edge computing services to complete the galaxy of services they already provide to their customers. There are three main categories of edge computing services currently on the market: i) managed hardware, ii) IoT solutions, and iii) manually installed servers at the edge (at user facilities).

AWS Wavelength [11] is a Multi-access Edge Computing (MEC) product; servers close to telco data centers. AWS makes embedded storage and computing capabilities available inside telecommunication providers' data centers. It enables ultra-low latency use cases for 5G devices. Google distributed cloud [12] is a general offer that ranges from managed hardware and software to on-premises offers (servers deployed to their customer's facilities). Proposes to compute and storage on edge location, mainly for AI inference workloads. IBM Edge Application Manager [13] is an IBM product for orchestrating edge applications. Akamai EdgeWorkers offers to run applications on their CDN facilities [14]. Intel Smart Edge is Intel offering software orchestration and hardware for edge computing (meant to be integrated with a 5G core network) [15].

Commercial offerings in edge computing also present various solutions for setting up local nodes to process data from IoT nodes. These nodes can be connected to IoT devices for local data processing and AI computations. AWS IoT Greengrass offers to run serverless functions and containers at the edge on local nodes [16]. Azure IoT Edge is a similar offer to Greengrass [17]. It can help reduce bandwidth costs and avoid transferring terabytes of raw data to the cloud. It can pre-process and aggregate data locally and then only send aggregated data to the cloud for analysis. Alibaba IoT Edge is another offer similar to Greengrass [18].

Amazon and Microsoft offer hardware as a service at the edge with AWS Outpost [19] and Azure Stack Edge [20]. This equipment can be installed on-premises and interact with other cloud services. However, inbound and outbound traffic is a paid service for these offers.

Large cloud computing providers mostly offer new hardware that can run their cloud solutions at the edge. However, these solutions are not edge-native and still have strong dependencies on cloud infrastructure. These proprietary solutions create vendor lock-in environments that might become problematic for future interconnection between systems in the Cloud-to-Edge Computing Continuum.

This section is not an exhaustive collection of all edge computing commercial offers. However, the creation and development of these commercial offers highlight a growing interest in edge computing from the major public cloud providers and telecom operators.

Edge computing orchestration frameworks

A whole ecosystem is emerging around edge computing. In addition to commercial offers for hardware, there are numerous frameworks, open-source projects, and research initiatives whose goal is to improve edge computing workload orchestration. In this section, our main interest is to describe existing orchestration tools that help manage edge services.

Kubernetes (K8S) is the *de facto* standard for workload orchestration in the cloud computing industry. It is available at a different level according to users' needs: user-managed cluster (hosted on VMs or bare-metal), managed cluster (the users only need to write the K8S manifests), and serverless (users only provide code or a container, Kubernetes can be used by cloud providers for managing serverless workloads, but it is not the only option available).

However, Kubernetes is not ready for orchestration at the edge out of the box. Kubernetes was designed for large data centers where hardware is similar (e.g., mostly racked servers with x86 CPUs, but it is changing now. ARM-based CPUs are getting more attention than in previous years), and networking is homogeneous (wired network between all machines, higher reliability and bandwidth and lower latency than wireless technologies. Networking is very performant within the same public cloud area).

Although not ready for edge orchestration, it is possible to extend Kubernetes to support it. Kubernetes has two main limitations for the edge: i) orchestration has to consider additional information such as networking (delays, available bandwidth, limited access to the control plane), users' location, node characteristics (e.g., AI accelerator available, battery-powered device), ii) lightweight cluster management to run on the smaller nodes. Also, it is worth mentioning that Kubernetes is not the only tool available for orchestration at the edge.

Different Kubernetes flavors are tailored for orchestration at the edge. KubeEdge is an open-source system that extends Kubernetes to support edge and cloud interactions [21]. KubeEdge control plane can support offline mode, which is helpful if an edge node is temporarily disconnected from the cloud. It also supports IoT protocols such as HTTP and MQTT. k0s is a single binary Kubernetes cluster [22], a lightweight alternative to K8S. k3s is also lightweight (also single binary) Kubernetes, made for ARM devices [23]. MicroK8s canonical's lightweight K8S distribution made for the edge and development purposes [24]. These three lightweight Kubernetes distributions can easily be executed at the edge.

This last paragraph presents various alternatives to Kubernetes for orchestration at the edge. EVE is an initiative from the Linux Foundation to build an Operating System to orchestrate devices across the Edge-to-Cloud Computing Continuum [25]. Ekuiper is an edge IoT processing framework that can help link the edge and the cloud. It allows

data processing at the edge with a very small footprint (low compute and memory requirements) [26]. Fogflow is an edge computing orchestration framework [27]. NearbyComputing is an edge orchestration platform for the edge [28]. Nebula Orchestrator is a container orchestrator for IoT and distributed systems. Nebula is built to run on edge nodes or CDN facilities [29]. Nomad is an alternative to Kubernetes for orchestrating and scheduling workloads in cloud computing [30]. Nomad also offers useful features for edge computing: it is easier to use it to federate multiple regions (which can be at the edge or in the cloud), and it is better for scaling (it can handle more nodes). It is also suitable for orchestrating non-containerized applications. According to Hashicorp, the company developing Nomad, their product is easier to use than Kubernetes, which makes it a better candidate for the smallest applications. Onap is an orchestration platform for telco networks [31]. Open Horizon is the Linux Foundation's initiative to orchestrate machine learning workloads across the Computing Continuum [32]. It is a high-level orchestration that can federate and orchestrate many Kubernetes clusters across the Computing Continuum. Openshift at the edge, Redhat orchestration framework for edge computing [33]. StarlingX is a service for deploying and managing Cloud infrastructures across the Computing Continuum [34]. StarlingX is a pilot project from the OpenStack Foundation. OpenStack is one of the most used open-source cloud software. In [35], the authors explain how it can be extended to support edge computing and its integration with 5G.

Edge computing problematics

Limited resources are the main problem of edge computing. As described previously, commercial offerings for edge computing are still in their infancy. Therefore, there is a lack of resources to enable edge use cases everywhere.

First, computing and storage **resources are limited**. Edge resources are proposed as dedicated servers (that can be placed on-site or at telco facilities) or as micro data centers. Micro data centers are a small-scale replica of what can be found in public cloud facilities. Then, available resources are limited due to the smaller size of the site. Consequently, use cases that require heavy computing or to store large data sets could be limited (without additional resources we can find in the Computing Continuum).

Also, there are **geographical disparities** between the edge locations. Some areas might have many available servers, while others have no local micro data center at all. These disparities are similar to what happens during the deployment of telco's new generation of networks. For example, 5G was primarily implemented in dense urban areas, but it is still missing in some rural areas. We think that edge computing adoption is following the same pattern. Some areas already have access to it, while others are going to wait years for it.

The **cost** might also be problematic; limited resources at the edge might imply higher costs. Micro data centers are smaller than their public cloud equivalents. Therefore, it is not possible to have the same energy due to the scale; it is possible to save money in data centers due to the huge numbers of servers and to achieve economies of scale. However,

there is a trade-off to be found between the higher price of computing resources and saved bandwidth economies. Such trade-offs are explored in chapter 4.

Scalability and orchestration are two other challenges. A good orchestration system is required to keep up with the increasing number of devices. The industry mostly offers semi-manual placement methods to its customers; they need to choose the area to deploy their applications. However, it is not possible to "guess" where end users will be, so this approach is limited. However, research offers many scheduling techniques to address this issue, either for developing new systems or extending existing ones.

Interoperability also needs to be addressed to offer edge services to the broadest geographic area possible. The different platforms and providers need to be interoperable to offer wide area coverage and enable services in all locations. A large number of protocols are available in this heterogeneous environment. Establishing robust standards is crucial to facilitate seamless communication between the diverse equipment at the edge.

The Computing Continuum and a unified orchestration methodology can help solve most of these limitations, as described in the following section.

1.2. The Edge-to-Cloud Computing Continuum

Cloud computing allows organizations to use computing resources on demand without investing in costly hardware. The National Institute of Standards and Technology (NIST) standardized the cloud computing paradigm in 2011 [36]. We refer to the *Edge-to-Cloud Computing Continuum* as an extension of the traditional cloud computing paradigm, an extension in the sense of the resources managed, a paradigm that includes the resources from edge and fog computing in addition to the cloud resources. The Edge-to-Cloud Computing Continuum is a distributed Computing Continuum; it includes communication (networking), computing, and storage resources from traditional data centers to the edge of the network [6, 37].

Fog computing is an intermediate computing layer in the Edge-to-Cloud Computing Continuum, located between the edge and the cloud [38, 6, 7, 39]. The main goal of fog computing is to extend cloud capabilities to the edge in order to reduce response time and outbound traffic from the data centers, using servers that are closer to the end users than the data centers. However, it differs from the Cloud-to-Edge Computing Continuum; fog extends cloud capabilities while the Computing Continuum offers seamless integration of all resources from the cloud to the edge. Therefore, fog computing is a part of the Computing Continuum. Fog computing was developed to answer the increasing demand for processing and storage of IoT devices' generated data[40]. The purpose of fog computing is not to replace cloud computing but to complement it [41].

Fog computing shares many problems with edge computing; fog resources are heterogeneous devices located outside of traditional data centers. Also, a fog orchestrator must deal with a very large number of geographically distributed nodes. Fog users mostly use wireless networks to access content or applications. Therefore, strong

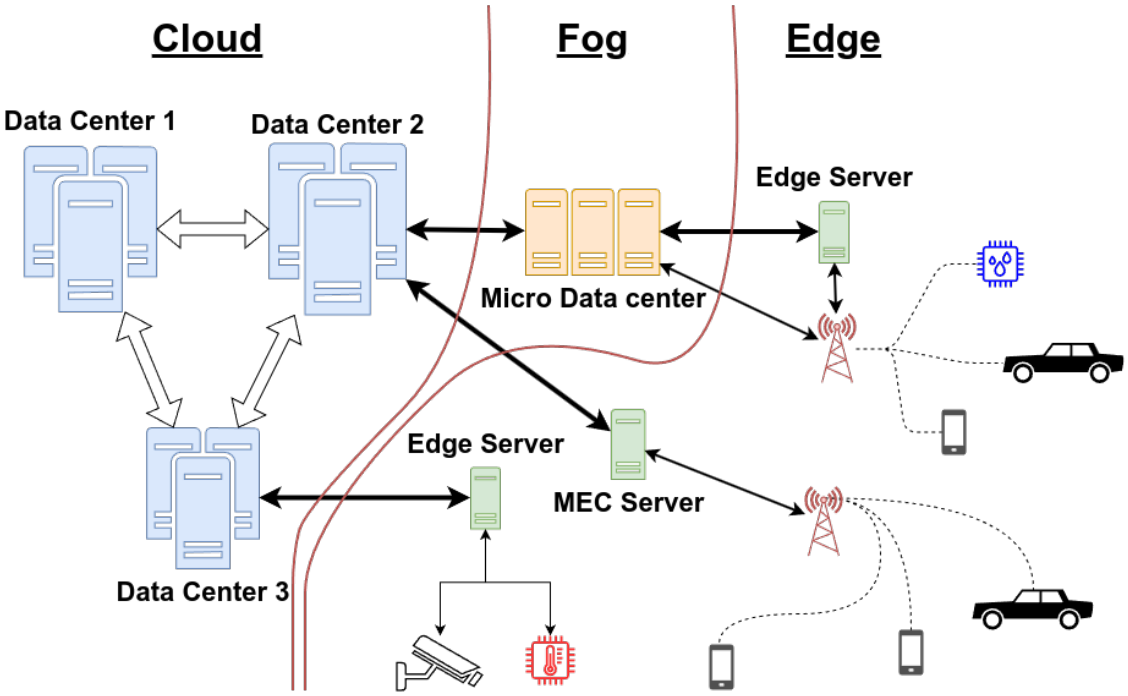


Figure 1.1.: The Edge-to-Cloud Computing architecture.

compatibility is required between the compute, storage, and network layers.

Figure 1.1 represents The Edge-to-Cloud Computing layered architecture. Edge, Fog, and Cloud are the main layers of this Computing Continuum.

Edge and fog are often used interchangeably (in the scientific literature) to refer to moving computation outside of the data centers. Even though they both exist to reduce the amount of data sent to the cloud (or traditional data centers), there are some differences. Figure 1.1 illustrates these differences. The edge represents the part that is the closest to the end users (e.g., close to a 5G antenna). The fog part encompasses all the resources between the edge and the cloud (traditional data centers). The aggregation of Edge, Fog, and Cloud forms the Edge-to-Cloud Computing Continuum. We can also refer to this Computing Continuum as the Computing Continuum.

For clarity and conciseness, we prefer to use the terminology *Computing Continuum* instead of the *Edge-to-Cloud Computing Continnum* in the following dissertation. Indeed, these two terms refer to the same concept.

According to Figure 1.1, software can be deployed at several levels: at a data center in the cloud, at a data center of another cloud provider, at a micro data center somewhere in between cloud and edge, at a MEC node (MEC is further described in the next chapter), at an edge or an IoT device.

The Computing Continuum complements the edge Computing paradigm; it creates another abstraction layer that includes various computing resources. Building an orchestration system at the Computing Continuum level would offer a solution to

address the lack of scalability and resources at the edge. Traditional data centers can host applications if there are no available resources at the edge. Based on application requirements, a unified (over the whole Computing Continuum) orchestration system can deploy applications anywhere in the edge, fog, and cloud layers. It is possible to reduce the usage of edge resources (that are more limited and expensive than cloud resources; it is possible to perform economy of scale in traditional data centers) if a nearby data center satisfies all the requirements of an application.

Therefore, the Computing Continuum enables the same use cases as edge computing; the Computing Continuum aggregates the compute resources at the edge, and then edge applications can be deployed in the Computing Continuum. The Computing Continuum provides a high-level view of available resources over many geographical locations, which opens a way for addressing scalability and orchestration of edge resources. A global view of the situation makes orchestration decisions easier; it provides access to more information. Using this global view can help adopt edge computing, making edge resources easily accessible. With a performant and unified orchestration system, deploying applications at the edge (or anywhere in the continuum) should be as easy as it is in traditional public clouds.

The continuum includes computing, networking, and storage resources, not only compute capabilities [6, 39]. However, this thesis mainly focuses on computing resources, even though networking plays an essential role in our placement methodologies. Our methodologies search for the best way to orchestrate and allocate computing resources (with networking as a constraint); our goal is not to orchestrate or allocate networking or storage resources.

Benefits of unification of resources in the Computing Continuum

Offering seamless access to many resources of the Computing Continuum with unified orchestration has many benefits. These benefits are similar to edge computing benefits because the Computing Continuum includes the resources at the edge in addition to those in the cloud. Here are the main benefits of unified orchestration for the Computing Continuum: lower latency [6, 7, 8, 9, 10, 39], reduced bandwidth usage [6, 7], better support for mobility, improving context awareness [6, 8], reduced energy consumption (for IoT and other small battery-powered devices) [7, 10], enhancing the QoS [8], better integration with telecommunication technologies like 5G [9, 10], and improved resource utilization with a global approach to orchestrate resources.

Computing Continuum use cases

To better understand the benefits of the Computing Continuum, we think it is important to know the essential use cases it enables with edge computing. Therefore, this section presents many use cases but is not exhaustive. Indeed, the primary use case for the Computing Continuum is latency-sensitive applications. Also, we have the real-time applications category, which is close to the latency-sensitive ones. Latency-sensitive

applications are applications with strict networking delay constraints. In addict to delays, real-time applications have reliability constraints. Data-intensive applications are another category of use cases. These applications exchange a lot of data with their users. They have a significant impact on bandwidth usage, and they can lead to network congestion. Even though we have identified these categories of use cases, it is not easy to categorize actual use cases. Actual use cases belong to one or many of these categories.

Video processing and data processing is a mature use case of edge computing (and therefore for the Computing Continuum) [37, 10]. Computing data and video at the edge helps to save network costs and reduce latency. It is perfect for answering a vast need for computation to process a huge volume of data and provide real-time or near-real-time performances. A data processing application can be software that aggregates data from IoT devices and sends the aggregated data to the cloud. A video processing application can be, for example, a surveillance camera in a security system that can use edge resources to enable immediate response to potential threats. For example, body-worn cameras (BWCs) are embedded devices that law enforcement officers can wear to help them in their duties. BWC streams can be locally processed at the edge to help detect hazardous behaviors. Images can be processed at the edge and not be sent to central locations to preserve people's privacy.

Federated learning is an important use case that can benefit from edge computing [42, 43]. Federated learning can run at the edge; it improves privacy and security and reduces bandwidth usage and the computation needs for machine learning training. Traditional machine learning was developed to provide near-optimal solutions to complex problems. Tons of data in centralized locations are required to train machine learning models. In fact, centralized machine learning training has many limitations: i) high computational load, so training is long, ii) data coming from heterogeneous devices need to be pre-processed (normalized), iii) sending vast volumes of data to centralized locations raise security and privacy concerns.

Decentralized machine learning offers two strategies to reduce the computation load and training time: model parallelism and model aggregation. Model parallelism consists of training different parts of a model (e.g., different neural network layers) on different machines. The model aggregation idea is to train a model on different devices, giving each device a subset of the data and aggregating all of the parameters. In other words, training a full model on each device with only a subset of the data, and then aggregating the results. Model parallelism allows shorter training times but still requires uploading all the data from the devices to central locations. On the contrary, a model aggregation approach allows us to keep the data on the devices and to upload only the results of each individual training.

Federated learning is a distributed machine-learning technique that ensures the privacy of data while training models. The idea is that each node in federated learning can access a model that is trained with its data and the data from the other nodes. Each

node first trains its model using its data, then, the parameters of the local models are frequently pulled from the devices, aggregated, and finally, new parameters can be sent to all devices. This method allows models to be trained with large datasets (data from all devices) while ensuring the privacy of each device. The data never leaves the devices; only the parameter updates are sent. Federated learning also reduces bandwidth usage; only the parameter updates are sent instead of the full datasets.

Edge computing is particularly helpful for implementing Federated Learning. Devices at the edge can train models using local data and send parameters to a cloud location. For example, an autonomous vehicle model can be fine-tuned at the edge. In this case, video streams from the vehicles are the training datasets. They can be sent to an edge server using the 5G network to train a model. Training is a compute-intensive task; it is unsuitable for a car with limited energy and computing capabilities. Therefore, videos can be sent to a nearby edge node for local training. Then, trained models can go back to vehicles where they can easily execute the inference of these models. Federated learning for autonomous vehicles is very helpful; it increases the size of the datasets while preserving the user's privacy. These datasets can include a wider variety of landscapes and weather conditions. Also, traditional vehicle embedding cameras can be a part of the local training process without negatively impacting the drivers' privacy, increasing the global performance of autonomous vehicles.

Autonomous vehicles are an important use case for the Edge-to-Cloud Computing Continuum. Even though fully autonomous self-driving vehicles are far from commercialization, it is not unrealistic to consider these use cases in the short term. Current edge computing use cases can be integrated into non-autonomous cars first. They can improve the actual driving experience by providing extra information to improve safety and convenience. Then, while already adopted and running in a real environment, these applications can be the foundation for future generations of vehicles.

Improving vehicle perception is the main benefit of the Computing Continuum for autonomous vehicles [44]. Edge computing enables cooperative perception for a whole group of vehicles. Vehicles with embedded GPUs can share the objects they perceive, while vehicles only equipped with cameras can share video streams instead. Processing a huge amount of data from video streams in real-time requires edge computing. Also, processed video streams can include video streams from vehicles as well as streams for fixed cameras located close to the road (adding interesting view angles). Edge resources can then be used to run software aggregating this data and broadcast it to nearby vehicles. Such data can provide additional information about objects or obstacles located in the vehicles' blind spots. It provides immediate feedback to the drivers or driving software. In addition, aggregated data can provide more details about road anomalies or the locations of Vulnerable Road Users (VRUs). Finally, this data can be helpful for a city traffic management system.

Augmented Reality and Virtual Reality (AR/VR) are use cases that can be improved with edge computing. The actual limitations of AR/VR headsets (except for price) are lack of portability and energy consumption.

AR/VR headsets are worn equipment, so they have limited space. Limited space implies a trade-off between performance and portability. Headsets need to be battery-powered to be portable, which greatly impacts autonomy and performance. It is impossible to embed a powerful chip or GPU without an adequate power source.

To address this issue, offloading the applications outside the headset to the edge is possible. Edge allows computing-intensive applications to run without the risk of running out of battery. However, to maintain a similar quality of experience, the application requires high bandwidth and low latency. Otherwise, the image quality would be lower, and users might experience motion sickness due to the delays. In addition to the latest wireless technologies (e.g., 5 G, WiFi 6), edge computing can achieve good performance without sacrificing autonomy or portability. In the long term, the mass adoption of AR/VR, powered by edge computing, can enable new paradigms like the metaverse [45].

Edge computing and the Computing Continuum enable many other use cases than those presented in this section. We can cite some of the most notable of them: smart factories (e.g., predictive maintenance using AI inferance at the edge [6], Industrial IoT [10]), improved health care (e.g., improve medical imaging while keeping data private), smart agriculture (e.g., using IoT devices to improve efficiency [10], monitor crop using drones), Holographic Communication [46], Digital Twin (TD) [6]. For example, an edge device can aggregate data from sensors in a factory and use it to predict maintenance for some equipment before the break occurs or to reduce the maintenance frequency. Predictive maintenance helps reduce cost (avoiding unneeded maintenance), increase the equipment lifespan (identifying issues early), and enhance safety (quicker identification of failures that could lead to accidents).

1.3. Motivations for unified orchestration in the Computing Continuum

The Computing Continuum is a resource set that spreads from traditional data centers to servers at the network's edge. These resources need easy access to facilitate their adoption. Also, aggregating resources across the Computing Continuum would help better distribute workloads. Deploying applications at the edge or in a data center regarding the application requirements can avoid over-provisioning more limited edge resources. However, aggregating these resources using a unified approach is challenging. Ullah et al. have identified six problems to address for building an orchestration system: lack of standardization support for application description, SLA management, context-aware resource discovery, proactive runtime reconfiguration, decentralized architectures, and security management [39].

The Computing Continuum needs unified orchestration

We think that all Computing Continuum resources should be accessible easily. Deploying applications in the Computing Continuum and at the edge should be as easy as deploying applications in current data centers. Solutions that only integrate edge resources are unlikely to be sufficient for most users due to limited resources. There is a need for considering edge, fog, and cloud as a whole, not separate components [6, 37].

Edge computing adoption should be greatly facilitated by software that can run on every node without requiring additional work from developers, as well as easy access to resources in the Computing Continuum. Having the same software that can run on different hardware is a different research problem. Solutions like WebAssembly are described in chapter 2. The Computing Continuum needs a unified orchestration of resources to improve access to its resources.

The Computing Continuum needs unified orchestration to improve resource utilization. The Computing Continuum is a terminology that describes all the resources we can find from the traditional data centers to the edge as described in Figure 1.1. However, no tool permits to access all of the resources seamlessly. There is a need for an orchestration system for the Edge-to-Cloud Computing Continuum [37, 8]. To deploy software at the edge, application developers need to choose a service provider and specific regions. The development of location-aware mechanisms that can deploy services according to the actual location of the end-user is still in its infancy. Location awareness is necessary in order to achieve low latency; services and users must be close to each other. A unified orchestration system (centralized or decentralized) would enable a seamless experience for application developers; they could easily deploy software at the edge, close to their users without thinking about it. Therefore, deploying applications at the edge would be as easy as it is with current public clouds.

Traditional cloud computing techniques are impractical in the context of the Computing Continuum [6, 37, 9, 39, 47]. Cloud computing techniques need to be extended to support the specificities of the Computing Continuum. New parameters need to be addressed at the orchestration level: high heterogeneity of the nodes, the geographic distribution of the nodes, the continuum infrastructure, networking heterogeneities (e.g., variable latency and bandwidth between the nodes), automated or seamless integration of the various resources (from multiple providers and for different tenants/multi-tenants, multi-cloud), the energy footprint of the nodes.

They are important to consider when building a new unified orchestration system for the Cloud-to-Edge Computing Continuum. To allow the usage of edge computing at scale, we need a global approach that considers all the nodes regardless of their location.

The orchestration system should address the dynamic behavior of the resource cluster. Achieving constant quality of service when users are moving is challenging. User mobility generates problems unreliable connection problems. Unreliable connections or energy management (e.g., a node running out of battery) can also be problems for nodes at the edge. It can cause node churn, resources appearing and disappearing from the system over time [48, 49]. The orchestration system should address the node churn

and the user mobility.

Kubernetes is the *de facto* standard method to deploy an application consisting of multiple components (i.e., micro-services) in the cloud. To use this container orchestration system, the developers can write a descriptive model (named *manifests* in the K8S ecosystem; they are written in Yaml [1]) to specify the desired state of an application. Kubernetes works at making sure that the current state matches the desired state even during disturbances such as node failure and changes of requirements such as sudden surges in traffic.

Kubernetes works well for applications running in traditional data centers. CPUs and networking are similar in most nodes in a traditional data center. However, Kubernetes is not made for deploying applications across multi-layers of a hierarchical cloud, according to Figure 1.1. All layers outside the traditional cloud are mainly managed manually without any adaptation to the changing needs of an application. Consequently, Kubernetes and other orchestration systems need to evolve to support the multiple layers of the Computing Continuum.

Most of the commercial offers that provide access to edge computing do not offer seamless orchestration and deployment of applications throughout the whole Computing Continuum. Some public cloud providers offer to connect their edge resources to their data centers, but there is no seamless way to deploy applications over the whole Computing Continuum; defining the data center region or the edge location is necessary.

Computing Continuum unified orchestration challenges

Unified orchestration of the Computing Continuum can improve edge computing adoption. However, achieving effective unified orchestration over the whole Edge-to-Cloud Computing Continuum is a whole set of problems.

Unifying the Computing Continuum resources raises many research challenges to tackle: multi-cloud and tenants, orchestration, security and privacy, context awareness, low latency and location awareness to geographically distributed nodes, mobility, a vast number of nodes, predominant role of wireless access, strong presence of network intensive and real-time applications, scalability, node and networking heterogeneity, cost-effectiveness, regulatory compliance [39, 8, 50].

The first issue for unified orchestration in the Computing Continuum is **heterogeneity** [10]. There are two kinds of heterogeneity: computing and networking. In traditional data centers, networking is mostly homogeneous (same bandwidth and latency between the nodes). The networking conditions might vary significantly at the edge or in the Computing Continuum. That makes a tool like Kubernetes unsuitable for scheduling workloads in heterogeneous environments. Also, the Computing Continuum is an aggregation of various resources that are geographically distributed. Among them, we can find servers with different CPU architectures and various accelerators such as General-purpose computing on graphics processing units (GPGPUs), TPUs (Tensor Pro-

[1] https://yaml.org/

cessing Units, mainly used for machine learning workloads), ASICs (Application-specific integrated circuits), or FPGAs (Field-programmable gate array). Software needs to adapt to these different platforms to use most of their capabilities and have better performance and energy efficiency. Developing for this heterogeneous infrastructure landscape is difficult and may lead to cloud vendor lock-in, infrastructure over-provisioning, and inferior performance and/or power consumption characteristics.

Location awareness is another issue. In order to deliver ultra-low latency, application placement needs to be aware of the geographical location of the nodes and of the end users. Therefore, orchestration can make decisions based on available computing and networking resources. It is harder to manage a geographically distributed infrastructure.

Reacting to volatility and mobility in the Computing Continuum changes is challenging. In the Computing Continuum, end users can be mobile (e.g., a 5G device), and networking requirements need to be monitored and watched to keep a constant QoS. Available resources may also vary in the Computing Continuum. A node might become unavailable in case of network failure, or a new node might become available when a resource-intensive application finishes its computation. As a result, a unified orchestration approach for the Computing Continuum should be able to react to changes to maximize resource utilization and ensure the QoS. Predictive approaches might also be a good complement to reactive approaches, especially to minimize the number of application migrations and avoid wasting time and resources by reacting to a stimulus that could have been predicted. For example, anticipating patterns of utilization (e.g., time and location) of some applications to adjust the scheduling decisions.

A unified orchestration system for the Computing Continuum should be able to **scale** to address the increasing number of edge devices and the future adoption of edge computing use cases. Either centralized or distributed, such an orchestration system requires to be scalable. Also, **privacy and security** deserve special attention in the continuum [6]. Micro data centers improve privacy and security by lowering the attack surface. In contrast, a unified orchestration system will increase the attack surface; more accessible devices are more accessible either for users or malicious actors. A trade-off needs to be found between usability and security. Security is an essential challenge for the Computing Continuum. Distributed systems are known to be harder to defend: they have a larger attack surface and are more complex.

Also, the **scalability** of an orchestration approach needs to be investigated to cope with an increasing number of nodes and end users.

Finally, **interoperability** is a significant challenge for unified orchestration in the Computing Continuum. It is very unlikely that cloud (and edge) providers offer edge resources in every geographic area. Therefore, companies will need services from multiple providers to make their applications available to most people. Then, resource management standards will be very beneficial for deploying applications using a single interface. Moreover, standards would help create a unified orchestration approach to deploy applications using resources from multiple providers. Currently, edge services are dependent on the implementations of a few cloud computing companies. Overcoming

vendor locking might be crucial to the adoption of edge computing and the unification of resource orchestration in the Computing Continuum.

1.4. Research Questions

In this thesis, we propose investigating optimization and orchestration in the Cloud-to-Edge Computing Continuum. We are interested in effectively studying the Continuum with a reliable and reproducible method to reduce costs/energy in the whole Continuum and how to adapt to a changing environment (e.g., when users are moving or computing resources are freed).

This thesis aims to facilitate access to resources across the Edge-to-Cloud Computing Continuum. We want to improve and contribute to current proposals to make a unified orchestration possible for the Edge-to-Cloud Computing Continuum. There are two main problems to make this possible: i) making hardware more accessible through unified orchestration systems, and ii) having software that can run on all different hardware without extra difficulties for developers.

In this thesis, we explore how to make hardware more accessible. First, build an experimentation methodology to ensure future orchestration or scheduling approaches are usable in real-world conditions. Then, we explore some orchestration strategies that aim to reduce deployment costs or energy consumption. Lower costs are an incentive to maximize the usage of resources and avoid wasting them. Finally, we implemented these approaches by extending Kubernetes, the *de facto* industrial standard for container orchestration. Following that standard, our methodology can easily be used on any cluster.

The research presented in this thesis focuses specifically on two tasks related to orchestration in the Edge-to-Cloud Computing Continuum: creating a methodology for evaluating scheduling and orchestration strategies in the Cloud-to-Edge Computing Continuum and providing orchestration strategies.

Effective experimentation is necessary for developing new orchestration methodologies. Maia et al. highlight the need for experiments in a real environment; simulations often lead to low performances in real environments [6]. They also insist on the need for datasets that are as complete as possible for machine learning-based solutions experimented in simulated environments.

Building an evaluation platform for the Edge-to-Cloud Computing Continuum can be both expensive and time-consuming. It prompts our first research question:

Research Question 1 (RQ1)

How can orchestration in the Cloud-to-Edge Computing Continuum be efficiently evaluated?

Palomares et al. highlight many challenges to be addressed, among them: scalability,

cost-effectiveness, and resource utilization [8].

In this thesis, we explore different possibilities for improving the performance of orchestration in the Edge-to-Cloud Computing Continuum in an effort to answer the following research questions:

Research Question 2 (RQ2)

How to reduce costs and energy consumption while ensuring QoS?

Research Question 3 (RQ3)

How to adapt scheduling decisions to a changing environment?

Research Question 4 (RQ4)

How to orchestrate applications in a heterogeneous cluster?

An **heterogeneous cluster** (as mentioned in RQ4) is a set of computing resources of different kinds (e.g., different CPU architectures), all connected by networking links that can have various characteristics (e.g., different bandwidth and latency).

1.5. Contributions

We first propose a methodology for evaluating resource orchestration in the Cloud-to-Edge Computing Continuum. Moreschini et al. highlight the fact that K8S support is important [7]. In [9], the authors state that Kubernetes support is a critical feature for automation. This evaluation technology is based on a Kubernetes cluster. It helps build a reliable testbed without spending too much time and money. Testbeds can be set up in traditional cloud environments.

We also develop an orchestration methodology that helps reduce the costs of deploying applications in the Computing Continuum. It tries to answer the above-mentioned problems with a network-aware approach and a solution that reacts to the dynamicity of the Computing Continuum environment.

Our vision here is to create a seamless Edge-to-Cloud Computing experience. Unlike many approaches where data centers (traditional or located at the edge) are managed independently, we want to propose a global approach where any resources from the continuum can be accessed. We do not categorize nodes into edge or cloud types. Our orchestration methodology is based on the node's characteristics (e.g., number of available CPUs, networking delays with some locations). Our methodology is made with the Edge-to-Cloud continuum in mind, but it can fit any cluster, either in public, private, or hybrid cloud.

Finally, we investigate the performances of container CPU limitation mechanisms. Tuning and selecting CPU limits and limitation mechanisms can significantly impact the performance of parallel applications. Therefore, we study the performances of the two main limitation mechanisms and build a tool to automatically select the best mechanism to save computing resources and energy. Saving resources is particularly important at the edge, where resources can be more limited than in a traditional data center. We also make sure that our methodology also works on heterogeneous clusters.

1.6. Overview

This section presents an overview of this thesis.

Chapter 2 presents the background. This chapter is based on the following publication:

- Rac and Brorsson. "At the Edge of a Seamless Cloud Experience". 2021. [51].

Chapter 3 presents our experimental methodology. Building a Cloud-to-Edge Computing Continuum testbed is costly and time consuming. It requires different computing equipment to reflect the diversity we can find in the Continuum. In addition, networking with various properties is necessary to reflect the challenges they induce. To answer this problem, we propose a methodology for building testbeds and studying orchestration in the Edge-to-Cloud environment. To illustrate our methodology, we implement a 5G network (with a simulated radio part) to ensure the performance and quality of our methodology. This 5G network includes both the core side with the 5G Network Functions and the client side with the User Plane Function and the User Equipments (UEs). We implement a scenario with many UEs, moving from one 5G cell to another, using the 5G control plane with a data-intensive workload (high-resolution video streaming). We also study different topologies for the 5G core to assess how different placements of 5G NFs can impact the 5G core performances. This chapter is based on the following publication:

- Rac, Sanyal, and Brorsson. "A Cloud-Edge Continuum Experimental Methodology Applied to a 5G Core Study". 2023. [52].

Chapter 4 presents our work to optimize application placement in the Cloud-to-Edge Computing Continuum. In that part, we use a monetary cost, but that work can be extended to other costs and metrics, such as energy consumption. We present a scheduling methodology that can react to dynamic situations, e.g., when UEs are moving (we need to move the application to ensure low latency) and when new nodes are available (ending a job frees some space for new applications). This chapter is based on the following publications:

- Rac and Brorsson. "Cost-Effective Scheduling for Kubernetes in the Edge-to-Cloud Continuum". 2023. [53].

- Rac and Brorsson. "Cost-aware Service Placement and Scheduling in the Edge-Cloud Continuum". Mar. 2024. [54].

Chapter 5 presents our study of container CPU limitation mechanisms. It first describes the performance difference we observe between the two main CPU limitation mechanisms. Then, it presents our tool for automatically setting CPU limitations. This chapter is based on the following publications (still under review):

- Rac and Brorsson. "Understanding CPU Limitation Mechanisms in Containerized Parallel Applications". 2024. [55].

Chapter 6 finally concludes this dissertation by summarizing the main results and discussing future research directions.

Chapter 2.

Background

This chapter presents the necessary background to understand better the work presented in this thesis.

This chapter is based on the following publication:

- Rac and Brorsson. "At the Edge of a Seamless Cloud Experience". 2021. [51].

This chapter first presents computing and cloud-native technologies that are the foundations for edge-native technologies. Then, it introduces networking technologies that are important for building the Computing Continuum. Networking is key for connecting nodes in the Computing Continuum between various locations. Finally, it presents the orchestration and scheduling techniques in the Computing Continuum.

2.1. Computing

This section presents the main tools and techniques that can be used for deploying and managing applications in the Computing Continuum. Most of these tools were initially built for cloud computing. However, they are also very helpful for the Computing Continuum, even if they need some improvement to support the Computing Continuum specificities. This section first presents available hardware in the Computing Continuum. It shows the virtualization technologies that can help deploy applications in the Computing Continuum. Finally, it explains the main orchestration tools for building the Computing Continuum.

The Edge-native concept was first introduced by Satyanarayanan et al. in 2019 [56]; it is to edge computing what cloud-native is to cloud computing: a set of technologies with edge computing support at their core.

2.1.1. Hardware

One characteristic of the Edge-to-Cloud Computing Continuum is heterogeneity. This section explains the three main categories of hardware: CPUs, Accelerators, IoT devices, and micro-controllers

CPU architectures

All kinds of CPUs can be found in the Computing Continuum; therefore, it is important to deploy software that can support all of them. If only one kind of edge node is available in an area (e.g., ARM architecture), it is important to be able to support it. Virtualization helps support these various hardware. For example, building a container image for each CPU architecture is possible. Then, Kubernetes can select the one corresponding to each node. Another solution would be to use WebAssembly modules on every device without needing to compile for each architecture. The most common CPU architecture is x86, which can be found in Intel and AMD CPUs. However, ARM-based CPUs are increasingly used due to their better energy efficiency. Finally, RISC-V, a new open-source architecture, is getting increasing interest from both industrial and academic words [57, 58]. The main benefits of RISC-V chips are its cost-effectiveness (due to open source standards), its power and energy efficiency, and its security [59].

Accelerators

Edge computing has to manage a growing demand in data processing while reducing latency. CPU alone can not meet this increasing demand, especially for use cases that involve image processing or AI inference. Hardware accelerators respond to this challenge by designing circuits for specific use cases. In their field of applications, hardware accelerators are really better than CPUs and can consume less energy. It is also possible to virtualize accelerators to make the most of their capabilities [60]. Virtualization brings flexibility in utilization. GPGPUs, FPGA, and ASIC are the main accelerators for edge computing. They can be used together with CPU to form heterogeneous platforms [61].

GPGPU General Purpose Graphics Processing Units (GPGPUs) are circuits designed to execute parallel applications faster. They were initially designed to process images and are used in a multitude of applications like Artificial Intelligence or Machine Learning. They can be found everywhere: in data centers, personal computers, or embedded devices. Frameworks like OpenCL or CUDA can be used to program GPGPUs.

ASIC Application-specific integrated circuits (ASIC) are circuits designed for a specific purpose. They propose high performance and security for small sizes and lower power consumption. [62]. ASICs are the best alternative for a specific task, but the flip side is their lack of flexibility. They are totally incapable of doing anything different than what they were designed to do.

FPGA Field-programmable gate arrays (FPGA) are circuits designed to be reconfigured after manufacturing. Reconfiguration can be static, i.e., before program execution, or dynamic, i.e., during runtime. Reconfiguration can be partial or global depending on the circuit [61]. Although they are designed to achieve specific tasks like ASICs, they are more versatile and can be reconfigured for many other uses. FPGAs provide low latency

and power consumption, making them suitable for edge or embedded devices. Public cloud providers propose access to FPGA; analogous access could be made available for edge computing. FPGAs are secure devices from an external perspective, but giving access to reconfigurable hardware in a multi-tenant environment could raise security issues. FPGAs' reconfigurability can be very useful for a scheduler at the edge. An FPGA can be dynamically reconfigured to execute a new task if a computing unit is missing or unavailable. Hardware description language (HDL), like VHDL, can be used to configure FPGAs.

IoT devices

IoT devices are important components of the Computing Continuum. These little devices collect most of the data, which is later processed in the continuum. IoT hardware mainly consists of small microcontrollers with limited power. These tiny devices are well suited for embedded with a small size and low energy consumption. However, they are not our primary interest for this thesis. Embedded devices are usually designed with one particular goal in mind. Therefore, they are not made to run various applications like a traditional server. It is, however, to run WASM on some microcontrollers. It then raises new opportunities for efficiently deploying the applications on these devices. They are important components of the Computing Continuum; they are the data collector for many use cases.

Facilities

The hardware described above can be located in various facilities. The size of these hardware components can range from a single-board computer to a full-scale data center. Micro data centers, despite their smaller size, are capable of providing local computation. Additionally, cloudlets or servers can be found directly at the user's application facilities. Even within a factory, small servers can be found.

Cloudlet Cloudlet is a terminology that describes a server or a micro data center between the edge and the cloud or among the fog resources. It is part of the Computing Continuum and can enable some edge computing use cases. The usage of this terminology is less common now, but we present it here as it is an important concept for the understanding of chapter 3.

Cloudlet can be a server, a computer, or a little cluster located at the edge of the network, typically at one hop of the devices. With mobile devices and cloud, they achieve a 3-tier continuum architecture [63]. Cloudlets are designed to provide services to mobile devices [64]. The key idea of Cloudlet is to enable mobile devices to offload computation in their vicinity. Energy consumption is one of the significant parameters for offloading on these devices. Cloudlets must be close to mobile devices to satisfy the Quality of Service (QoS). Cloudlets should also be able to work without the Internet. Working in a local area would also avoid congestion in the core network.

2.1.2. Virtualization

Virtualization is an essential technology for maximizing resource utilization in the Computing Continuum, as it does for cloud computing. With sandboxed software execution, it offers the necessary security for sharing hardware between multiple tenants, reducing under-provisioned servers. Virtualization is a layer of abstraction that allows the virtual division of hardware resources like servers into multiple shares. Each share can be allocated for dedicated usage.

Virtual Machines (VMs) are the first level of virtualization. Many VMs can run on one physical host, and each VM runs a full Operating System (OS). Therefore, the VM virtualization process has a significant CPU and memory overhead; the host runs many Operating Systems at the same time.

Containers are the second level of virtualization. They are lightweight VMs that share libraries with the host system. Sharing libraries instead of running a full Operating System significantly reduces resource usage compared to VMs. However, containers' runtime is strongly dependent on the host's OS (they use the same libraries). For example, Linux containers cannot run on a Windows computer without the Windows subsystem for Linux (WSL) installed. Containers are also dependent on the CPU architecture of their host. A container supports only one architecture. Developers targeting many CPU architectures need to build many container images, one for each supported architecture. However, there are many different containers, all following the same standards.

WebAssembly (WASM) is a third virtualization level that can be used in the Computing Continuum. It is a technology that can be used everywhere, not only in web environments. It was first proposed for the web but now goes outside of the browser. It implements the WASI standards to interact with machines that are similar to POSIX [65]. WebAssembly is a binary file that can be executed on almost every device, from traditional x86 CPUs to tiny microcontrollers. WebAssembly is a compilation target for major high-level programming languages such as Rust, Go, and C/C++. WASM can run on every hardware that supports at least one WebAssembly runtime and on every major web browser (e.g., firefox, chrome, safari, edge). The key advantages of WebAssembly (WASM) technology lie in its ability to compile software once and run it everywhere (it is very important for the Computing Continuum where devices are mostly heterogeneous), its speed, and its inherent security. With sandboxed execution as its default setting, it ensures a high level of security, a crucial aspect in today's digital landscape.

Figure 2.1 compares the different virtualization techniques to a server without virtualized components.

WebAssembly is also a technology to consider in an edge computing environment. It brings lightweight and fast execution, like containers, but in a more secure way by providing sandboxing and architectural independence. Security is fundamental in a multi-tenant context where providers execute untrusted codes on their platforms.

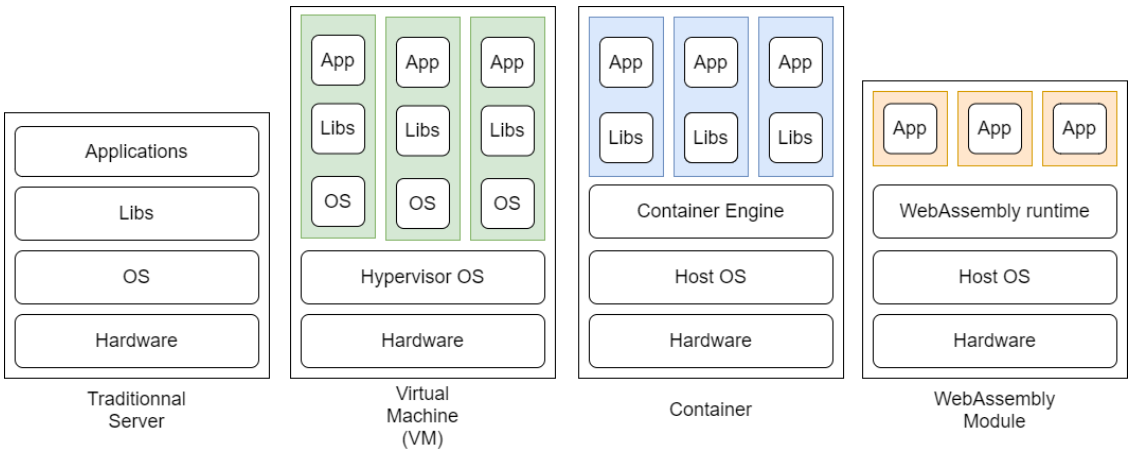


Figure 2.1.: Comparison of virtualization techniques

Containers

Containers are different from Virtual Machines (VMs). They are built on top of an Operating System (OS) and share libraries with their OS. Sharing libraries makes them much lighter than VMs, which use shared resources. It is easier to run more of some, and it is very low-cost (in terms of memory and CPU) compared to a VM.

Containers are rapidly becoming the standard deployment technology in cloud computing. They allow better resource management and make application development/deployment easier. Container technology can also be considered for edge computing.

According to Hong and Varghese, [66], containers propose good performance for edge computing. However, they are not ready to deal with heterogeneity. Even if containers support GPGPUs, they do not support specific hardware like data processing units (DPU), tensor processing units (TPU), or field-programmable gate arrays (FPGA). Those hardware accelerators are essential to achieve ultra-low latency on specific applications at the edge. However, building containers for multiple targets requires additional steps. Containers strongly depend on their host; e.g., a Linux image cannot be directly executed on a Windows host; it needs access to the Windows Subsystem for Linux (WSL).

Kata containers [67] are really lightweight VMs that can be managed like a container (e.g., in Kubernetes). Using them permits workload isolation and better security. Combining VMs and containers could be a solution to manage hardware accelerators and security.

Container standards are helpful to build container orchestrater. These standards enable orchestrators such as Kubernetes to use different container runtimes. There are two major standards for containers: Open Container Initiative (OCI) that defines standards for container images, runtime, and distribution [68] and Container Runtime Interface (CRI) which is an API to define how Kubernetes can use different runtimes [69].

OCI standardizes container images, runtimes, and distribution (container registry).

Runc is the default container runtime for Kubernetes. The CRI defines how Kubernetes interacts with container runtimes. The default container runtime interface is Containerd (previously Docker engine [70]. CRI-O is a lightweight alternative to Containerd [71].

These container standards help define edge-native applications and use traditional orchestrators made ready for the edge. Following these standards should make application deployment easier.

Live migration between edge devices is an important mechanism. Some devices at the edge can be mobile, and their applications have to follow them to always be as close as possible. Xu et al. [72] present a Docker container live migration tool. They explain that container migration is more complicated than VM migration. Docker container migration needs to migrate image, runtime, and context.

WebAssembly

WebAssembly (also called WASM) is a new technology design for web applications that rapidly gains interest outside the web. The idea behind WASM is to create a new assembly standard that proposes a unique binary that can run on every processor with good performance. WASM also comes with a memory-safe structure and lightweight sandboxed execution that allows a high-security level and an efficient execution. It is compatible with most high-level programming languages (e.g., Rust, C/C++, golang).

WebAssembly could have a significant contribution to edge computing. WASM can help developers maintain a single code-base and address most devices with a single binary. Considering that edge devices will have different architectures, WebAssembly could become a game-changer. WebAssembly applications could seamlessly be deployed on every edge device without worrying about the platform or architecture.

The major web browsers (Chrome, Edge, Firefox, and Safari) all support WebAssembly. WASM can run outside browsers using one of the many available runtimes.

WebAssembly Standard Interface (WASI) aims to propose standards to facilitate communication between WebAssembly conceptual machine and its host system [65]. WASI has an objective similar to POSIX.

The flexibility and performance of WebAssembly are strongly linked to its runtime. WebAssembly can be interpreted; it supports more platforms but has lesser performance. Some runtimes support Just In Time Compilation (JIT), performances are better for heavier tasks, but binaries are larger. Then, Ahead-Of-Time (AOT) compilation provides the best performances. Execution performance really depends on the runtime because they do not have the same level of optimization. Runtimes evolve quickly and often; therefore, it is not so easy to evaluate WASM performance.

In [73], Kakati and Brorsson describe the performances of Webassembly Modules outside of the web browsers. The different execution and compilation options make WASM suitable for many use cases. Just as the standardization efforts (WASI) to bring as many POSIX features available make us think it could be a good candidate for orchestration at the edge. Also, WASM can run on most available hardware: most CPU architectures (x86, ARM, and even RISC-V), microcontrollers, and other IoT devices. In

addition to portability (which is very helpful in the context of the Cloud-to-Edge Computing Continuum, WebAssembly also has security features that provide sandboxing and isolation.

Running WebAssembly everywhere WebAssembly binary can be executed on almost every CPU architecture (e.g., x86, ARM, RISC-V) and platform (e.g., laptop, smartphone, Single Board Computers like a Raspberry Pi, or microcontroller unit (MCU) like Arduino) with runtimes like wasm3[1]. This wide support means that the same software can run on almost every device in the Computing Continuum, even with limited resources.

Figure 2.2 shows the different platforms and architectures that can be targeted.

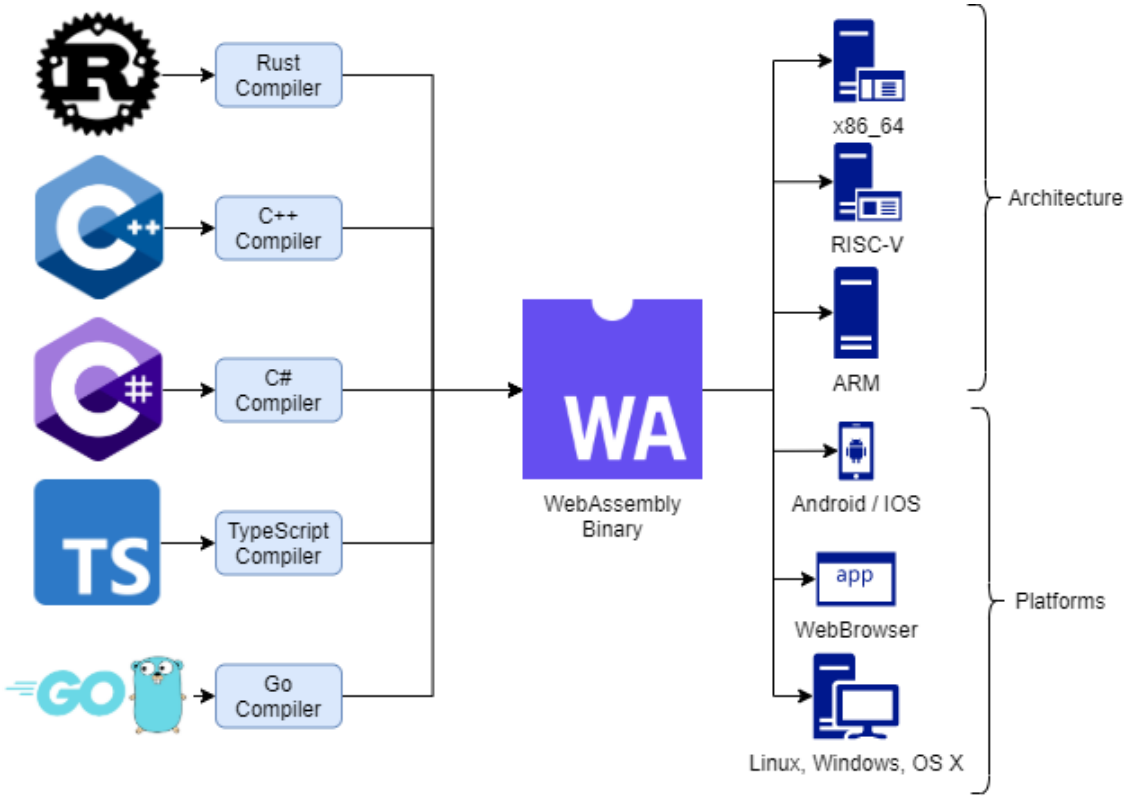


Figure 2.2.: WASM

Kubernetes can also orchestrate WebAssembly applications just like any other container. The CRI Containerd can manage WebAssembly application [74]. This means that Web Assembly applications can easily be deployed on Kubernetes clusters, even with nodes of different architectures; the same binary can run everywhere. However, hardware support for accelerators (e.g., Tensor Processing Unit) still needs to be added to WASI and Kubernetes. It would be an appreciated addition to AI inference use cases.

[1]https://github.com/wasm3/wasm3

2.1.3. Orchestration tools

In this section, we present the main orchestrators and their associated monitoring tools. Then, we present the serverless (FaaS) and auto-scaling paradigms.

Kubernetes

Kubernetes is the *de facto* industry standard for container orchestration. It manages pods, which are groups of one or more containers deployed together on a node. A pod is the atomic deployable unit in Kubernetes. Kubernetes offers a variety of useful features, such as:

i) auto-healing: Kubernetes can automatically restart failed or crashed pods, or pods move them to another node if their current node crashes.

ii) rolling updates: This feature allows pods to be updated one by one to avoid service disruption. A new pod is started with the updated version, and Kubernetes ensures the new pod is healthy before terminating the previous instance.

iii) Scalability: Kubernetes supports horizontal scaling, which involves deploying more replicas of a pod. This can be done manually or automatically based on CPU utilization. Kubernetes also supports vertical scaling as an experimental feature, which involves allocating more resources (e.g., CPU, memory) to a pod.

iv) Service Discovery and Load Balancing: Kubernetes can expose containers using DNS names or IP addresses and distribute network traffic to ensure stable deployments.

v) Namespace Isolation: Kubernetes supports namespaces to divide cluster resources between multiple users, which is useful in large organizations.

Also, Kubernetes supports multiple CPU architectures, including x86, ARM, and others. This allows for a diverse set of hardware environments, making it flexible for various use cases in the Computing Continuum. However, Kubernetes has built-in support for GPUs but none for other accelerators. GPU support is essential for workloads that require heavy computational power, such as machine learning, scientific computing, and video processing. This allows specific pods to be scheduled on nodes equipped with GPUs, optimizing resource utilization for these intensive tasks. Another significant benefit of Kubernetes is its extensibility. It is designed to be easily improved and customized to meet specific needs. This feature is particularly important for adapting Kubernetes for edge computing environments.

Kubernetes is not the only software that can be used to manage container life cycles. In chapter 1, we present some of these alternatives. However, Kubernetes is the *de facto* industrial standard. This is why most of the research presented in this thesis relies on Kubernetes.

Kubernetes scheduling framework

The Kubernetes scheduling framework is built to be entendable [75]. This framework has a pluggable architecture that is easily extendable. It is possible to implement a new plugging without re-implementing the whole scheduler. The scheduling framework

allocates pods (i.e., a set of at least one container) on nodes on the cluster. The extensibility of this component is important to make it support the Computing Continuum specificities.

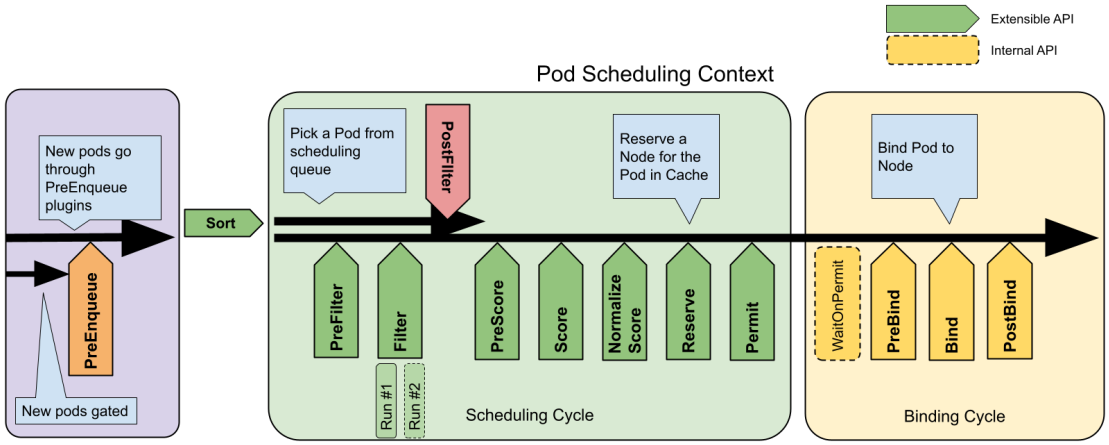


Figure 2.3.: Scheduling framework extension points [75]

Figure 2.3 show the Kubernetes scheduling framework extension points. The framework architecture has two main parts: the scheduling and the binding cycle. The scheduling cycle selects nodes to host the new pods, and the binding cycle applies the scheduling decision. In this thesis, the main interest is in the scheduling cycle. Extending some of these plugins to implement new scheduling policies designed for the Edge-to-Cloud Computing Continuum is possible.

The goal of the scheduling cycle is to assign a node to each new pod. First, the nodes are filtered: all nodes that cannot host the pod (e.g., insufficient resources) are removed from the list of allocatable nodes. Then, the remaining nodes are ranked. The default ranking method is to give better scores to the least allocated nodes. Then, the scores are normalized, and the node with the highest rank is selected to host the pod. If two (or more) nodes have a similar score, one of the nodes is randomly selected. Using different ranking plugins simultaneously is possible, and then an average score will be computed. Finally, the selected node is reserved for the pod, which will be deployed on the selected node after the binding cycle.

Tracing and monitoring

It is important to monitor workloads for optimal orchestration. Accurate data is, therefore, very helpful. Data collection can be automated; open telemetry[76] defines standard APIs for monitoring applications. Having standards avoids vendor lock-in from one cloud provider. It standardizes the way applications can export traces, logs, and other numerical metrics. In addition to boosting application performance (e.g., detecting and fixing problems and identifying performance issues), it facilitates

application debugging. From a research perspective, monitoring tools are helpful for collecting experimental results. There are open-source tools for collecting the metrics defined by open telemetry. Prometheus [77] is a tool for collecting numerical metrics about applications or cluster states. Jaeger [78] is a tool for collecting distributed traces and logs from applications. Although these tools are designed for Kubernetes integration, they are also helpful for VM or serverless workloads.

Serverless

Serverless, or Function as a Service (FaaS) is an abstraction that allows running code without managing any servers or VMs [79]. According to [80], serverless technology is increasingly used. Developers can upload code (either in a container or just source code) to define a function; that function will be executed each time an end-user uses it. So, the FaaS provider charges the developers only when the application is triggered. This abstraction removes the decision-making for scaling; a new function is created on demand each time it is triggered. For example, a data processing application can run a pre-processing function each time it receives new data; a new function is instantiated for each new request.

Serverless removes the tedious task of managing servers or complex orchestration systems from the developers. Developers can specify Quality of Service requirements during the function definition. QoS requirements can include latency or bandwidth requirements. Serverless can help deploy workloads in the Computing Continuum. It removes the need to define resource requirements for an application developer.

Resource definition is even more complicated in the Computing Continuum due to the vast number of possible locations where applications can be deployed. Also, applications have to be placed regarding users' location. With serverless, developers only have to pay attention to their code. Resources and infrastructure are dynamically provided according to the application's needs. To better utilize the edge nodes' capabilities, it is possible to extend these requirements with new parameters, such as location awareness. Serverless is, therefore, very helpful; developers specify only the application requirements without the need to select the right location. Serverless is helpful to implement seamless offload from edge devices to data centers if needed [10].

At first glance, serverless is more cost-effective than dedicated resources; users only pay for what they use. However, serverless is less suitable for long-running tasks. The cost per hour of FaaS is higher than the cost of dedicated resources. Serverless is an event-driven paradigm; an external event triggers the application execution. Serverless is cheaper when there are many short tasks because resources are not always used. Resources are charged only for some events when using serverless. When execution times become longer, serverless becomes more expensive. Dedicated servers are better candidates for long-run resource-intensive execution. This is why serverless is well suited for data processing pipelines. A new function can be instantiated and charged only when new data arrives. Then, the function is deleted. There are no idle resources to pay for when using serverless.

Sledge is an open-source framework for ServerLess at the EDGE [81]. It is designed to run on a single-host server; with WASM support, it can run on most servers, from the smallest, like Raspberry Pi, to the more powerful ones. Sledge bypasses the kernel scheduler to achieve ultra-low latency and takes advantage of WASM. This helps to have a rapid startup time. Sledge uses a custom compiler and runtimes *aWams*. WASM also provides benefits in terms of security with process isolation and sandboxing. With WASM, executing the same binary on all platforms running Sledge is possible. WebAssembly could be a key enabler for serverless at the edge. WASM permits developers to stay focused on their application without worrying about available architectures. Then, every application could be deployed at the edge.

Nuclio [82], OpenFaaS [83], and Knative [84] are other frameworks for serverless at the edge. Unlike Sledge, they are not based on WASM but on container technology. They can run on multiple platforms and be scheduled by Kubernetes.

Autoscaling

Auto-scaling is a method to adjust the amount of resources allocated dynamically. This method can be applied at different levels, such as deploying more physical machines or application instances. It first helps to scale up applications: add or remove resources to adapt to changing workloads (e.g., more or fewer clients connected to an application server). Auto-scaling also helps to reach SLOs, offering expected performances to the users. Then, it helps to reduce the costs of idle resources. It allocates only the necessary resources. Therefore, it reduces overprovisioning.

There are two main forms of scaling: vertical scaling and horizontal scaling. Vertical scaling allocates more resources from a server to an application. Horizontal scaling allocates more servers to host application replicas. Figure 2.4 illustrates the difference between horizontal and vertical scaling.

Major cloud providers offer auto-scaling as a service for their resources. Kubernetes has a built-in auto-scaler [85]. It is a reactive auto-scaler. The Horizontal Pod Autoscaler (HPA) deploys more or less pod replicas. The Vertical Pod Autoscaler (VPA) adjusts the resources allocated to a pod. It is only available in *beta*; it is not fully operational like the HPA. The *Cool Down Time* parameter sets the duration between two scaling operations up or down. It prevents the auto-scaler from making new decisions before observing the effects of the previous actions. The cluster scaling adjusts the number of nodes in the Kubernetes cluster. It offers API compatible with most cloud providers. The default mode of the VPA and the HPA is to monitor resource usage (CPU and memory) to make auto-scaling decisions. However, it is also possible to use custom metrics.

Auto-scaling behavior can be proactive or reactive [86]. Proactive scaling techniques try to anticipate the load variations to adapt the allocated resources accordingly. Reactive scaling reacts to load variations based on predefined rules. For example, reactive scaling techniques can increase allocated resources when CPU usage is above a 90% threshold.

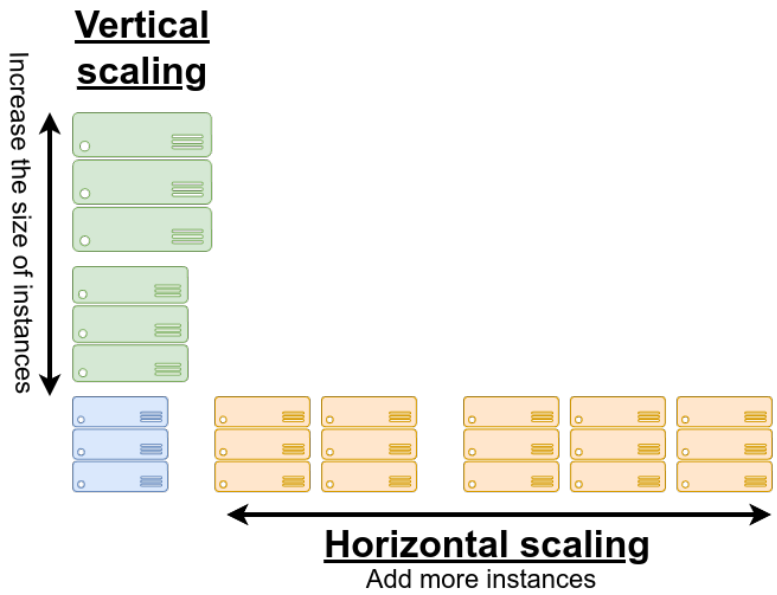


Figure 2.4.: Horizontal and vertical scaling illustration

2.2. Networking

There are two parts to reduce user delays: computing and communication delays. In this part, we explore networking delays. Networking is an important part, even if it is not the main interest of this thesis.

Network-aware orchestration approaches are important for orchestrating the heterogeneous resources of the Computing Continuum. Networking is a keystone in the construction of an Edge-to-Cloud Computing Continuum. A reliable network architecture is required for running distributed applications. The development of paradigms like microservices architectures also shows dependence on the network. In addition, networking devices can have an active role in edge computing. In-network computing is a paradigm in which applications are distributed through the network infrastructure [60]. For example, a switch or router could become a small edge node and propose computing power to run small applications. This section describes the network virtualization technologies we use in the Computing Continuum. Then, we provide details about the 5G network software. It is important to understand its features, components, and architecture to understand better chapter 3. This chapter demonstrates our testbed environment using a 5G core as a workload; it is an example of a distributed application that can be deployed at multiple locations in the Computing Continuum. Edge computing enables some 5G use cases. One goal of edge computing is to provide low latency and location-aware services. Therefore, edge computing also benefits the Telco (Telecom Operator) industry by enabling use-cases with 5G [9] [10]. Edge computing servers and the 5G network are two necessary components for many use cases.

2.2.1. Networking virtualization

This section presents the definitions of the main networking technologies used in the 5G core. The 5G core software is detailed in the next section.

Virtualization is not only for computing resources but also for networking resources (equipment, NFs, and even bandwidth with slicing)

Networking virtualization is crucial for understanding this and thesis the 5G technologies. Virtualization technologies result in the main software benefits of 5G over 4G (LTE). As described in the previous section, virtualization technologies are capital for cloud computing to improve resource utilization and simplify application management.

Networking resources can be virtualized, too; these techniques are not limited to computing hardware.

Virtual Network Function

Virtualization is increasingly important in networking. With Network Function Virtualisation (NFV), network function (NF) can be managed like a VM; it does not need to run on specific, dedicated hardware. According to [87], NF can be containerized, and VNF should become Containerized Network Functions (CNFs). Containerization allows even more flexibility and enables Kubernetes-based networking.

SDN

Software-defined networking (SDN) represents a set of techniques to bring virtualization to networking hardware. Instead of having a device dedicated to one task, it is possible to reconfigure it on the fly to run different functions (e.g., router, firewall). SDN techniques allow separating the Control Plane and the User Plane [88]. This considerably simplifies the configuration of networks.

SDN will make networking easier to deploy for telco operators. There will be less need for specific and expensive hardware thanks to virtualization. Also, resources will be mutualized for better performance.

Software-defined networking (SDN) is a technology that can provide seamless networking configuration. Configuring networking takes time; devices are configured manually (e.g., routers and firewalls). SDN permits the automation of networking configuration and management. The idea of SDN is similar to Operating System, adding abstraction on top of hardware to simplify uses. SDN has a separation between the control plane and the user plane to increase network management flexibility.

Edge devices can be mobile, have intermittent connections, or have limited bandwidth. Therefore, it is important to easily manage networking to adapt to this dynamic topology.

SDN flexibility allows having a service-centric architecture [89]. Networking can be dynamically adapted to the task's needs. The SDN controller has a high-level view of the network that helps deploy an application and its networking. If the application needs to scale up or down, networking will dynamically follow its needs.

SDN flexibility is important to achieve mobility of the end-user devices. For example, routing will be dynamically adapted if a mobile device goes from one access point to another. If this device has offloaded a task to the cloud or a cloudlet, the results will be delivered directly to its new position; this will avoid unnecessary traffic.

Network slicing

Network slicing is a technique for building virtual networks on top of physical networks. This technique relies on the Network Function Virtualization technology [90]. With slicing, building a private network for different tenants is easier. With network slicing, it is possible to define slices that are part of the bandwidth reserved for specific applications. For example, we can define applications with low latency or high bandwidth requirements. Using a network slice ensures that enough resources are available for the resources; resources are reserved and dedicated to this application to ensure the application's QoS. A slice can have dedicated NF in addition to reserved resources.

Network slicing allows for a more dynamic and flexible infrastructure. Network slices can be designed for a particular purpose, e.g., reserve resources to achieve ultra-low latency on one slice and manage many users on another. Slices can guarantee some levels of Quality of Service (QoS).

Network slicing is an important feature of the 5G network that can benefit edge computing. Slices are virtual network partitions; they are virtual resources that can be reserved. For example, it is possible to create a slice that reserves some bandwidth for one kind of device. It is possible to set up slices for specific use cases (MIoT, IIoT). A slice is a set of virtual networking resources that can be reserved for a use case. Each slice can have different service level agreements, depending on the use case. In [37], Tusa and Clayman extend the notion of slice to *end-to-end* slice. With this definition, a slice can be either computing, storage, or networking elements. Any of these elements can be deployed in the Computing Continuum.

Radio Component Virtualization

Virtualization can even be used in the Radio Access Network (RAN), with virtualized RAN (vRAN). Open RAN is non-proprietory software for RAN; these initiatives were created to avoid vendor lock-in and develop multi-tenancy cooperation. The O-RAN Alliance is one of the major contributors to Open RAN initiatives. Virtualization of RAN goes one step further with cloud RAN (cRAN). The idea is to distribute RAN functions on cloud infrastructure using cloud-native tools and practices. cRAN brings cloud flexibility to the telco universe.

2.2.2. 5G network software

The 5^{th} generation of the cellular network (5G) brings higher data rates (enhanced Mobile Broad Band (eMBB)), lower latency (Ultra-Reliable Low-Latency Communications (URLLC)), and greater connectivity supporting a large number of simultaneous connections (massive Machine-Type Communications (mMTC)), mMTC is particularly important for massive IoT use-cases (i.e., intermittent connection of many small battery-powered devices). These new features enable a new range of data-intensive or latency-sensitive applications. Together with edge computing and, more generally, with resources of the Computing Continuum, it is a key component for any application. This thesis is about the software side of 5G technologies. The hardware improvements

(e.g., new antennas, modulation techniques, beamforming) are not described in this thesis as it is unnecessary for developing our orchestration methodologies. These new hardware technologies are crucial for delivering the expected performance. However, a high-level view of this capability (e.g., delays, jitter, or bandwidth between two points in the network) is enough for our needs. Even if we do not describe the hardware's new features, there is much to say about the software. Also, in the scope of this thesis, we only describe standalone 5G (5G SA). Non-standalone 5G relies on a 4G network core and does not offer the 5G improvements we want to study.

5G has a new software architecture compared to 4G; 5G has a micro-service based architecture, while 4G has a monolithic architecture. Network Function Virtualization (NFV) and SDN are the features that enable a micro-service based architecture. Control user plane separation (CUPS) is a major feature the micro-service based architecture enables. Figure 2.5 illustrate how CUPS can improve data throughput and lower networking delays. With CUPS, user and control plane data can have separate dedicated routes. Therefore, application data can be routed directly to the application servers without being sent back to telco facilities. This shorter trip reduces delays and reduces network congestion, two features necessary to enable data-intensive and latency-sensitive use cases. It becomes very interesting when a server is located at the edge; application data can reach this server directly. NFV make it possible to use cloud computing techniques like containerization to improve how NFs are deployed and managed.

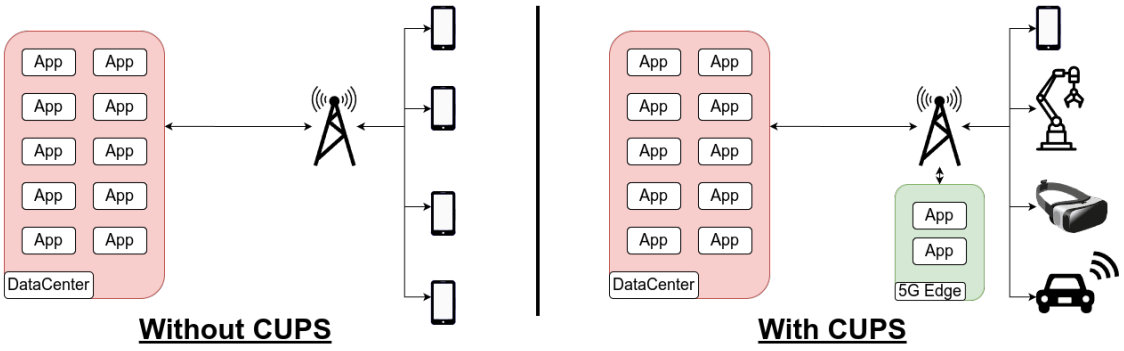


Figure 2.5.: Control user plane separation illustration

Figure 2.5 illustrate the Control user plane separation (CUPS). On the left side, user traffic must go to Telco facilities before reaching applications in data centers. On the right side, user traffic can reach applications directly without going through Telco facilities. Using CUPS reduces the networking delays.

Table 2.1 presents the main network functions of the 5G core. This table highlights the principal network functions involved in the UE registration and the PDU session establishment procedures. These are important procedures for chapter 3. The first procedure corresponds to the process of connecting a UE to the network, for example, when it turns on. The latter procedure establishes a communication channel for the device. Appendix A presents the detailed traces of these procedures, as recorded during

Table 2.1.: Main 5G Network Functions

NF	Name
AMF	**Access and Mobility management Function**
AUSF	Authentication Server Function
N3IWF	Non-3GPP Interworking Function
NRF	Network Resource Function
NSSF	**Network Slice Selection Function**
PCF	Policy Control Function
SMF	**Session Management Function**
UDM	**Unified Data Management**
UDR	Unified Data Repository
UPF	**User Plane Function**

our experiments. We use the open-source 5G core network (5GC) (based on the 3GPP Release 15 definition) [2], 5G UE and RAN (gNodeB) simulator[3] for these experiments.

Figure 2.6 presents the 5G core software architecture. Most of the network functions are deployed in a data center at a central location in telco facilities. 5G NFs are micro-services inter-connected using the HTTPS protocol. The User Plane Function (UPF) is a router that handles user equipment traffic. The user's traffic goes to a Data Network (DN), which can be an edge node (or MEC) or the Internet. GNodeB (GNB) provides connectivity to the 5G devices and handles the radio traffic; it is the equivalent of an eNobdeB for 4G. *DB* is not a standard 5G component defined by the 3GPP; it is a database that stores 5G core information such as subscriptions or slice configurations. To ensure the quality of service, 5G NF can have many instances, each dedicated to a slice. Figure 2.7 show an example of a configuration with three slices (S1, S2, S3), and four User Equipments. It is important to note that each NF is a microservice and can be containerized. Therefore, the Computing Continuum techniques and resources can be used to improve the 5G core performances and orchestration. Also, 5G network information (e.g., UE cell location, handover/user mobility) can be used to improve application placement in the Cloud-to-Edge Computing Continuum.

2.2.3. Other networks

The 5G networks are not the only network available for edge computing. Bluetooth (IEEE 802.15.1) and ZigBee (IEEE 802.15.4) [91] are energy-efficient technologies well suited for IoT [92].

Wi-Fi 6E (i.e., 802.11ax [93]) offers performances of throughput and latency approaching 5 G (i.e., around 1ms of latency and many Gbps of throughput). Wi-Fi has the benefit of being much cheaper to deploy than a private 5G network. However, the coverage for

[2]https://github.com/free5gc/free5gc
[3]https://github.com/aligungr/UERANSIM

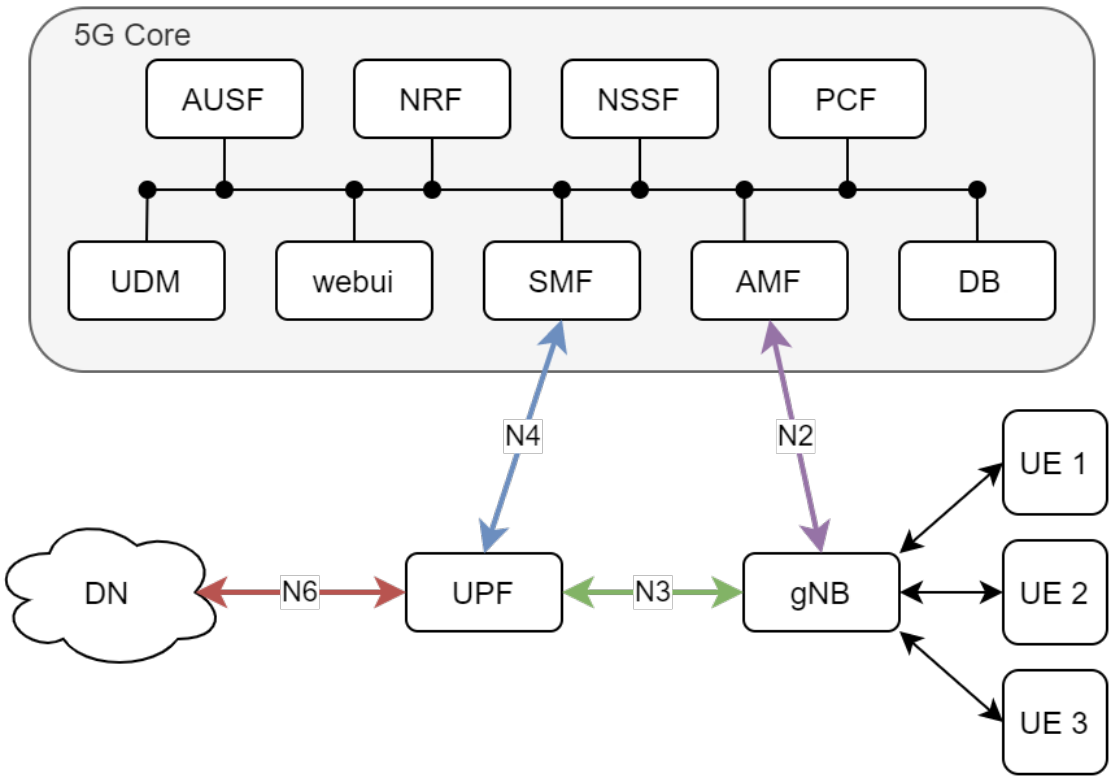


Figure 2.6.: 5G core software architecture

Wi-Fi is much smaller.

The 3GPP is also already working on 6G specifications, the successor to 5G. The main improvements include a higher throughput and reliability, and a lower latency [94]. It should also include improved satellite communication support and more network intelligence.

Low Power Wide Area Networks such as Sigfox [95] and LoRa (LoRaWAN)[96] should also be considered. Fog nodes can be IoT gateways to devices using these networks to exchange small amounts of data. However, these small amounts of data might be more significant due to the large number of devices.

2.2.4. Multi-access Edge Computing

Multi-access Edge Computing (MEC) is a paradigm where servers are deployed at the edge of mobile networks (e.g., close to 5G antennas) or sometimes in telecom provider's facilities [6, 9, 97]. ETSI has standardized this paradigm and its integration with 5G networks [98]. The purpose of MEC is to enable some 5G use cases. To achieve high throughput and ultra-low latency, the physical proximity of the servers matters. Therefore, 5G networks and edge computing are two necessary components to enable use cases with demanding throughput and latency requirements. MEC has a key role in

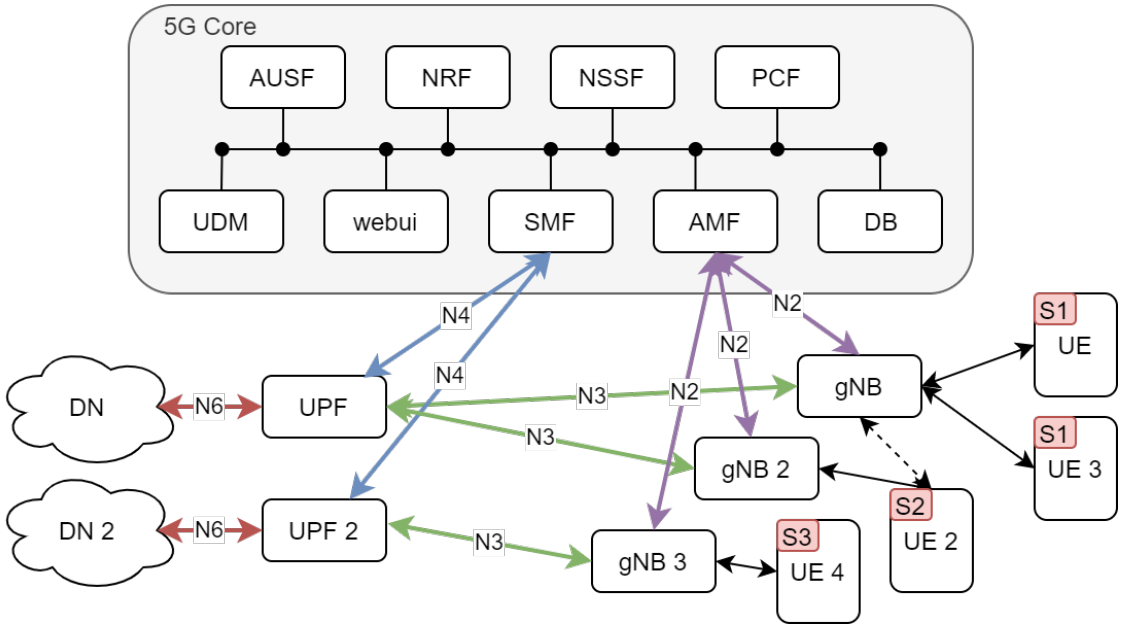


Figure 2.7.: 5G core software architecture with slices

edge computing. It is a link between Cloud and Telco environments.

MEC helps achieve ultra-low latency, high bandwidth, and Real-Time performance at the edge. These technologies enable many applications, such as gaming, Virtual Reality (VR), Augmented Reality (AR), the Internet of Things (IoT), or Vehicle-to-everything (V2X).

MEC is also helpfull for application offloading and data aggregation [99]. Tasks can be offloaded from small devices to more powerful ones at MEC facilities. Offload might concern whole applications or only a few tasks (e.g., power-intensive tasks). There are three kinds of offloading: partial offloading (at the edge), full offloading (at the edge), partial/full offloading (at the edge and at the cloud) [7]. Energy is another essential criterion for offloading. A battery-powered device can offload a task to save battery.

2.3. Orchestration and Scheduling

Orchestration includes a set of tools and techniques for handling and managing applications over the Computing Continuum during their life cycle. The main orchestration techniques for cloud computing are described at the beginning of this chapter. This section describes the main specificities of orchestration for the Edge-to-Cloud Computing Continuum.

Deploying edge-native applications over the continuum brings new challenges. For orchestration of the Computing Continuum, we consider a dynamic set of resources. Auto-discovery mechanisms can update the list of managed resources. New nodes can appear and join the cluster, and others can leave it. Intermittent connections can affect

the availability of nodes and, therefore, their place in the cluster. Some of this cluster's nodes can also be mobile, e.g., inside a vehicle. Therefore, we need to update the set of orchestrated resources periodically.

Standards are needed to federate resources from edge to cloud and form a Computing Continuum. Standards are helpful for building schedulers and orchestrators. Standards can define the way we package applications and how we deploy them. Container standards like the one defined by the Open Container Initiative are a good starting point. They are compatible with Kubernetes, which is the *de facto* industrial standard for container orchestration. Vergara et al. also highlight the fact that the lack of formalization makes the federation between cloud and edge harder [41].

In this thesis, we start with existing cloud computing techniques and tools and try to extend them to address the Computing Continuum challenges. Tools like Kubernetes can be extended to manage heterogeneous nodes, which sometimes appear or disappear. Scheduling algorithms can be extended to manage volatility (of nodes and of users), connection uncertainties, geographic distribution, heterogeneous resources, and the additional parameters (e.g., delays, energy) that need to be part of the decision.

In this section, we first introduce the formulation of the scheduling problem and its complexity. Then, we present the decision parameters and optimization objectives of scheduling in the Computing Continuum. Finally, we present the main orchestration approaches.

2.3.1. Scheduling problem formulation

There are many ways to formalize the resource allocation problem, but the main idea remains the same. On one side, we have applications. Applications are made of many interconnected services. Each service has computing resource requirements (e.g., CPU, memory) and network requirements (i.e., communication needs with other services or end-users). On the other hand, we have servers with various characteristics (e.g., CPU, memory). The goal of resource allocation is to map services on servers. It is possible to choose one or many optimization objectives. The usual approach to optimize more than one metric is to include the other objectives as constraints. For example, the allocation that minimizes networking delays or monetary costs. Vergara et al. present a formal description of the scheduling problem [41]. Here are the most common optimization objectives in the literature.

- Reducing network delays [6] [41]
- Minimizing energy consumption [6] [86] [41]
- Lowering monetary cost [6] [86]
- Maximizing resource Usage [6]
- Guaranty of SLA [6] [86]

- Increasing throughput [6] [41]

The application placement problem is not easy to solve. In literature, we find the problem formulations that are $\mathcal{NP}$-hard or $\mathcal{NP}$-complete. Eidenbenz et al. propose a formulation for a task allocation problem [100]. They demonstrate that it is $\mathcal{NP}$-hard. Kwok et al. describe an $\mathcal{NP}$-complete formulation of the DAG scheduling problem [101]. Cohen et al. model a deployment and migration problem [102]. They prove it is $\mathcal{NP}$-hard. The application placement complexity explains why we use approximation. Scheduling decisions need to be quick when implementing a Kubernetes scheduler.

2.3.2. Main orchestration approaches

In order to efficiently place applications in the Computing Continuum, we can use various resolution strategies. Then, we can use these techniques to build orchestrators. There are, however, additional constraints to consider, like the control topology or the type of scheduling. Finally, scalability is important to consider when choosing a strategy. Scheduling in the Computing Continuum should be quick to ensure SLOs and offer low latency. The number of nodes in the Computing Continuum can be significant. The topology of a compute cluster can be dynamic or static (nodes can appear or disappear) [41].

Resolution strategies

To resolve the above-defined problems, we can use various resolution strategies. Vergara et al. classify these strategies into the following categories: exact solution, approximation, heuristic, meta-heuristic, and other [41]. Exact solutions are not commonly used for such complex problems, but they can be helpful in smaller instances or very specific cases. Heuristic methods, like greedy first, are the most commonly used methods. These methods give satisfying trade-offs at a much lower complexity. They provide good approximation at the cost of finding meaningful heuristics. Meta-heuristic methods are also widely used. Techniques like Tabu search or simulated annealing are often used. Bio-inspired methods, including genetic algorithms or ant colony optimization algorithms, have proven to be beneficial as well. Finally, other techniques can be used. Machine learning, and more specifically, Reinforcement Learning or Deep Reinforcement Learning, has become very common. Game theory also offers some solutions. Other optimization techniques can also be used, such as Bayesian optimization, Markov Decision Process, Lyapunov optimization, or constrained optimization (e.g., mixed integer linear programming or nonlinear programming).

Centralized, decentralized, and distributed control topology

An orchestrator can have various control topologies based on the resolution strategies for solving the allocation problem. An orchestrator control can be centralized, decentralized, or distributed [86]. The centralized approach is the most common one. It is easier to

design and implement. However, the centralized approach has the following limitations: lack of scalability, it is a single point of failure, and it can be a central target for cyber-attacks [39]. A distributed approach helps to tackle scalability. Decentralized approaches fix most issues of centralized ones, but they are much more complex to elaborate.

Reactive and proactive scheduling

Scheduling consists of allocating services to servers. There are two main approaches to scheduling. Proactive and reactive scheduling are two approaches to deal with dynamic resource allocation. The proactive scheduling principle involves forecasting future computing resource needs and allocating resources based on this prediction. The main benefit of a proactive approach is better resource allocation. However, this approach relies heavily on accurate forecasting. Scheduling decisions are far from optimal if estimations are not accurate. Reactive scheduling adapts to the current resource needs and adjusts resource allocation accordingly. The main benefit of a reactive approach is its flexibility. It can quickly react to unexpected situations. However, the key issue of the reactive approach is the reaction delays between the time an action is needed and the time the action is executed. A reactive system might be prone to oscillation if it takes too much time to respond. Also, real-time processing constraints often lead to suboptimal solutions.

Chapter 3.

Experimental methodology for orchestration in the Cloud-to-Edge Computing Continuum

This chapter presents an experimental methodology for orchestration in the Cloud-to-Edge Computing Continuum. Running orchestration experiments in the Computing Continuum is challenging. It needs all specificities of this environment (e.g., node heterogeneity, networking delays) to produce accurate results. Current orchestration approaches need to be adapted for the Cloud-to-Edge Computing Continuum. However, building an entire experimental testbed is expensive because it requires the geographical distribution of the nodes. It is also time-consuming. The experimental methodology presented in this chapter tackles this issue. It offers a way to run orchestration experiments in the Cloud-to-Edge Computing Continuum at a lower cost in a traditional cloud infrastructure. Anyone who wants to study orchestration or the impact of deploying an application over the Computing Continuum on performance can use this experimental methodology. Any containerized application can be deployed using that methodology. It is not a methodology for building a simulator but realistic testbeds.

In addition to the experimental methodology, this chapter studies the impact on the performance of deploying a 5G core network in the Computing Continuum with various geographical locations. It presents the variations of performances of the 5G core network according to different system architectures. When using network slicing, allocating Network Functions (NFs) at different geographic areas is possible. Therefore, it is possible to study the impacts on performances of different geographical placements of the 5G NFs. This performance study is important for Telco providers to understand better where to place NFs over the Computing Continuum. Also, the experimental setup can be easily reproduced to study different configurations or 5G procedures.

Finally, this methodology is the basis on which relies chapter 4. Chapter 4 presents new orchestration approaches that are evaluated using the methodology presented in this chapter.

This chapter first presents the experimental methodology for orchestration in the Computing Continuum. Then, it shows an application of the methodology to study the performances of the 5G core network.

This chapter is based on the following publication:

- Rac, Sanyal, and Brorsson. “A Cloud-Edge Continuum Experimental Methodology Applied to a 5G Core Study”. 2023. [52].

3.1. Introduction

Unified orchestration of the resources in the Computing Continuum should improve the adoption of *edge-native* technologies. Unified orchestration ensures that applications have the resources available to run; if nothing is available at the edge, traditional data centers can be used instead. This process should be seamless for the application developers. Chapter 1 details the benefits and challenges of unified orchestration with the Computing Continuum.

However, the geographical distribution of the nodes in the Computing Continuum makes the elaboration of unified orchestration challenging. Therefore, experimental methodology for replicating the Computing Continuum infrastructure and challenges is valuable for developing new orchestration systems. Building a real infrastructure with distributed nodes can be costly and time-consuming. Hence, this chapter presents a methodology for experimenting with orchestration and scheduling approaches using real applications while considering the constraints of the Edge-to-Cloud Computing Continuum.

To validate our experimental methodology's performance, we study the 5G core network. It is a complex distributed application that can run in the Computing Continuum. We choose to experiment with placing the 5G Network Functions (NFs) at different geographical locations in the Computing Continuum. This experiment has two main objectives: i) ensuring that the setup built using our methodology behaves as expected, and ii) providing helpful insights to Telco providers to configure their 5G infrastructures.

In this chapter, we make the following contributions:

Contributions

1. Propose an experimental methodology for testing unified orchestration and analyze the effects of running applications over the Computing Continuum.
2. Validate that methodology with a 5G core study.
3. Provide insights to Telco providers about the impact of the placement of 5G Network Functions (NFs) in the Computing Continuum and provide information for building a similar setup to study and analyze other 5G procedures and use-cases.

3.2. Experimental methodology to study orchestration in the Computing Continuum

This section presents an experimental methodology for studying orchestration in the Computing Continuum. The Computing Continuum has additional constraints compared to traditional data centers. Nodes are geographically distributed, which impacts

the way we orchestrate applications. Therefore, we need a specific methodology to study the Computing Continuum environment.

This experimental methodology is designed to be easily reproducible. Any set of servers can be used to implement it. Once nodes with the right characteristics are selected, anyone can set up a testbed using our guidelines and open-source tools. Testbeds built with this methodology can easily be deployed in public clouds. More details about the cloud-native technologies we use can be found in chapter 2 section 2.1. The tools and scripts needed for this are publicly available on github [103].

Figure 3.1 illustrates the testbeds that can be created using our experimental methodology. In this figure, the targeted infrastructure is the system we want to study. It is the set of servers deployed at various geographical locations. The servers have different characteristics and compute performances. In this example, the targeted infrastructure has three nodes of different sizes at various locations. The white boxes on the figure are the services of an application deployed over the Computing Continuum. The emulated infrastructure is a testbed built using our experimental methodology. The emulated infrastructure mimics the characteristics of the targeted infrastructure. The nodes of the emulated infrastructure are real servers that can run the same applications as the targeted infrastructure. Nodes are labeled according to their characteristics to facilitate the definition of orchestration policy. Networking delays vary according to the geographical location of the nodes. They are longer between Edge and Datacenter nodes than between Edge and Cloudlet. Networking characteristics between nodes are configurable on the emulated architecture.

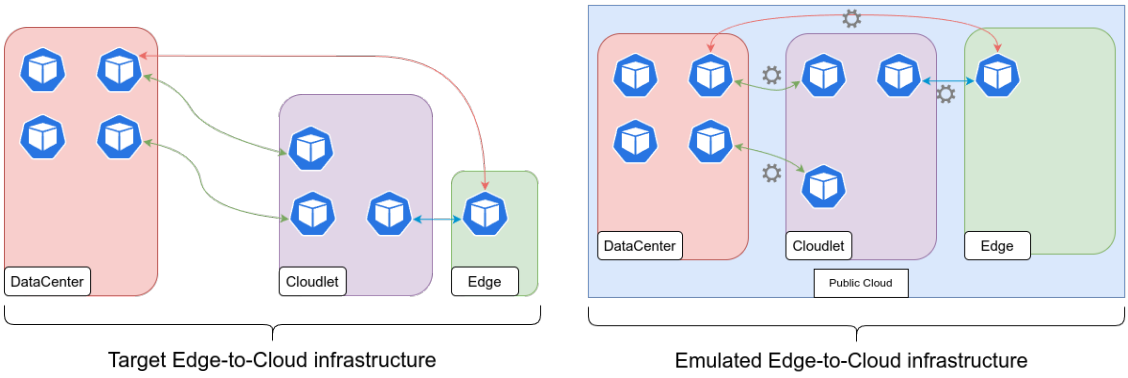


Figure 3.1.: Edge-to-Cloud environment can be emulated on the public cloud.

Orchestrator and workload definition

Our methodology for studying orchestration in the Cloud-to-Edge Computing Continuum is built upon Kubernetes. Kubernetes is the *de facto* industrial standard for container orchestration. New orchestration approaches developed on our testbed can be easily used on production clusters. There is no need to implement the orchestrator

again because we are using the same software for production and experiments. Also, as it is widely used, it is easier to compare new approaches to existing ones in the state of the art.

Another benefit of using Kubernetes is that it can run any containerized application. This is an important feature because we want our methodology to be able to deploy real applications. Kubernetes supports most CPU architectures (e.g., x86, ARM) and also GPGPUs.

Compute and storage

Running real applications is an essential part of this experimental methodology. Testbeds assembled using our methodology use real hardware. Any server can be used to build testbeds, from private to public cloud. Therefore, most CPU architectures and server sizes are available. Then, in our testbeds, any kind of node representative of the Edge-to-Cloud Computing Continuum is possible. It is possible to use small nodes with low energy requirements with an edge device's characteristics. It can also be a powerful node like the ones in traditional data centers. Accelerators like GPGPUs can also be part of the compute resource. Nodes in testbeds are labeled using their characteristics (e.g., CPU architecture, available hardware accelerator, geographical location). These labels are then useful for defining custom orchestration policy.

Networking

Configurable networking is the key to this methodology. With the proper network configuration, any topology can be created, and the effects of geographic distribution can be emulated. This methodology explains the fine-grain configuration of network throughput, latency, and other parameters.

We propose to use *traffic control*(`tc`) to parameter networking. This utility program can reconfigure the Linux kernel packet scheduler. Therefore, it can delay packets and create artificial latency. Additional delays help simulate the interaction with a distant node. Theoretically, the more distance there is, the longer the delays will be. Then, communication between an edge node and a node in a data center should have more delays than communication between two nodes in the same data center. It can also reduce the maximum bandwidth or simulate packet loss (following a given distribution). We can run this program in each pod as a *sidecar container*[1] to configure each pod networking. Finally, we develop scripts to configure each pod to set up any global topology.

[1] sidecar containers are containers running in the same pod as a primary application. They offer additional services like monitoring or logging.

Monitoring

Monitoring helps collect experimental data. We want to embed monitoring as a part of our methodology to facilitate experimenting. Logs, metrics, and traces about software running on the testbed can be automatically collected and stored.

In addition to applications, we can monitor the cluster state with these tools. For example, monitoring node utilization or network congestion. It is important not only for experimental purposes but also for orchestration. An up-to-date overview of the cluster is helpful for making scheduling decisions.

Monitoring is an integral part of our experimental methodology. Monitoring tools are automatically deployed when creating a testbed with our scripts. In addition to monitoring tools presented in chapter 2 section 2.1.3, we propose to deploy *sidecar container* to access additional data. For example, a *sidecar container* can access additional networking data such as current throughput or complete network traces.

Orchestration policy definition

Orchestration policy defines on which node a pod should run. We can define orchestration policies using Kubernetes features. The Kubernetes scheduler is the component that decides which node will host each pod. The default policy of the Kubernetes scheduler is to select the least allocated node in the cluster. Adding rule-based policies on top of the default scheduler or implementing custom policies is possible.

Rule-based policies affect pods according to pod and node labels and a pre-defined set of rules. *Taints*, *Tolerations*, and *Pod affinity* Kubernetes tools for defining rule-based policies. They favor or avoid placing pods on some kinds of nodes or collated with some kinds of pods. The kind of a node or pod depends on its labels Taints and tolerations can prevent pods from running on some nodes. For example, it prevents applications from running on the control plane nodes. Affinities help or prevent co-hosting pods.

Advanced policy involves implementing a new Kubernetes scheduler (at least some parts of it). Chapter 2 section 2.1.3 describes the plugin architecture of the Kubernetes scheduler. Implementing a new plugin creates a new orchestration policy. These advanced policies can use any extra metrics available in the cluster.

Testbed configuration and deployment

A testbed is defined by a set of *manifest files*, *scripts*, and *configuration files*.

The *configuration file* stores information about network parameters (e.g., latency or bandwidth) configuring each instance of `tc` (traffic control) running as a sidecar. It also stores cluster information (e.g., master node IP, pod subnetworks) and node parameters (e.g., taints, tolerations, labels). It defines the cluster's desired state and the manifests needed to deploy the application. A *script* initializes the cluster state based on the configuration file.

The *Kubernetes manifests* (YAML files) store information about the applications to deploy on the testbed, as well as a description of the orchestration policy. Application

affinities and *tolerations* are defined in manifests and the selected scheduler. Additional scheduler and monitoring tools deployments are also defined in these manifests.

Finally, a sequence of *scripts* set up the cluster and the experiment's life cycle. For example, they deploy application pods, start pcap (Wireshark) trace recording, stop the experiments (deleting the pods), and collect the results.

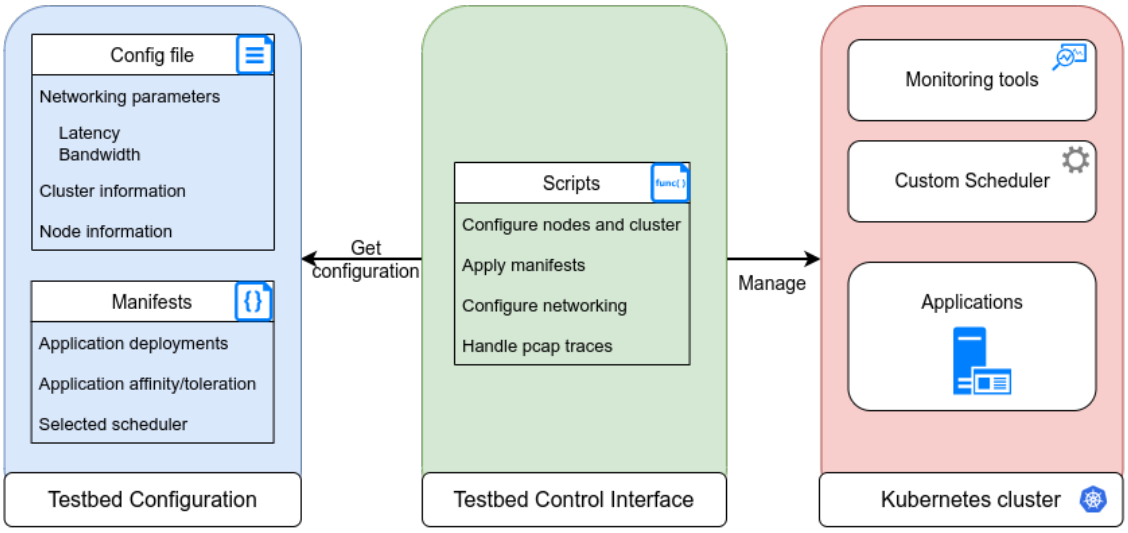


Figure 3.2.: Experimental methodology overview

Figure 3.2 summarizes our experimental methodology.

3.3. 5G system study

To demonstrate the effectiveness of our experimental methodology, we study a 5G system. A 5G system is a typical application that can be deployed in the Computing Continuum. 5G users are, by definition, at the edge.

In addition to testing our methodology, we want to study the 5G core and see how different system architectures (i.e., 5G Network Functions running at various locations in the Computing Continuum) affect the performances of a set of key 5G procedures we select. 5G is a distributed application, unlike the previous generations of cellular networks. It is then possible to deploy each of its services (i.e., Network Functions) at various geographical locations, including the edge, traditional data centers, and computing resources between the two. Chapter 2 section 2.2 describes the 5G network and the main network technologies.

This section details the 5G system we study with our experimental methodology. This section first presents the main 5G Network Functions (NFs) involved in the procedures we study. Then, we introduce the system architectures we reproduce with our methodology. Finally, we describe the three 5G use cases we use as a workload for our 5G core.

3.3.1. Principal 5G Network Functions

A complete description of a 5G System (Radio Access Network, devices, and core) is out of the scope of this thesis. However, some familiarity with the core components is necessary to understand the study. The 5G core consists of several Network Functions (NFs). The gNodeB (gNB) represents the Radio Access Network (RAN) to which User Equipments (UEs) (e.g., a mobile phone) is connected over cellular radio. Most of the details about 5G NFs are not important for this study, but we detail three of them: AMF, SMF, and UPF. These are essential to understanding how the system architectures are defined.

AMF (Access and Mobility management Functions) handles incoming connections and session requests of UEs and manages mobility (handover between two 5G cells).

UPF (User Plane Function) handles user data traffic. The UPF is directly connected to a Data Network (i.e., the Internet or an Application Server).

SMF (Session Management Function) establishes PDU sessions (Protocol Data Unit) for the UEs. A PDU session is a data tunnel that links a UE to a data network (DN) through a UPF.

3.3.2. System architectures

We define three system architectures to study the 5G core. The first architecture is our baseline; it is the most common architecture for deploying a 5G core. All 5G Network Functions are running in traditional data centers, far from the edge. The second architecture is designed to reduce the latency between edge nodes (hosting applications) and the end users. The last architecture is built to reduce the duration of the Protocol Data Unit (PDU) session establishment procedure. The last two architectures propose deploying the AMF, UPF, and SMF at various locations in the Computing Continuum.

We build these architectures using three kinds of nodes: *Data center*, *Edge*, and *Cloudlet*. We define the kind of a node based on its geographical location. Cloudlet nodes represent servers located between the edge and the traditional data centers. Cloudlet nodes can be micro data centers. These three kinds of nodes are all part of the Computing Continuum.

Figures 3.3, 3.4, and 3.5 represent the three architectures we study. For each architecture, the RAN elements (i.e., gNB and UEs) are deployed on separate nodes. These nodes are dedicated to RAN elements; they cannot host any of the 5G NFs. We call N2 to N6 the network links between the nodes following the 3GPP naming definition [104].

The **Baseline** architecture, represented in Figure 3.3, is the most common architecture for the 5G core. All 5G NFs runs in traditional data centers. However, this architecture does not effectively support the use cases of the eMBB and URLLC. Cloud gaming or AR/VR applications require ultra-low latency and high bandwidth. The UPF needs to run at the edge to achieve ultra-low latency or high throughput.

The **LatOpt** system architecture, shown in Figure 3.4, is designed to reduce latency and increase throughput between UEs and applications. The UPF is deployed at the edge,

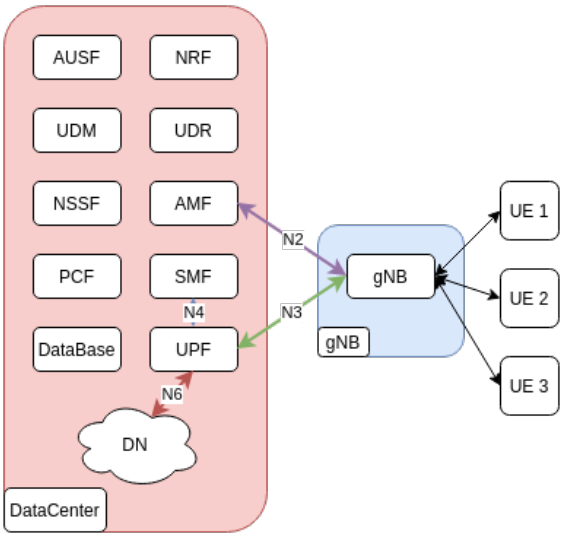


Figure 3.3.: **Baseline** architecture.

close to gNBs and UEs. The other 5G NFs runs in traditional data centers. Deploying the UPF at this location should significantly lower latency and improve throughput on the link N3. This architecture should enable the use cases of the eMBB and URLLC.

The **AccessOpt** system architecture, illustrated in Figure 3.5, has similar benefits as the LatOpt architecture. In addition, AccessOpt should reduce the time to complete the PDU session establishment procedure. This architecture is designed to be a simplified implementation of a multi-layered 5G architecture. AMF and SMF are deployed on cloudlet nodes to reduce the delays with the UEs and handle UE connection, session management, and mobility procedures quicker. Extra performances and quicker execution help handle use cases with many connections or mobility requests.

3.3.3. 5G use cases

To experiment with our 5G setup, we introduce three 5G use cases: Augmented Reality (AR), Industrial IoT (IIoT), and Massive IoT (MIoT). We choose them to test the main 5G features (eMBB, URLLC, and mMTC). Using the above-described system architectures should produce different performances for each use case. Studying these use cases is helpful for i) building a 5G infrastructure with network slices at the edge and ii) developing new orchestration methodologies for deploying NFs in the Computing Continuum.

Siriwardhana et al. describe the AR and IIoT use cases in [105]. We adapt the workload to the capabilities of our testbed. Our study uses real production-grade 5G core and NFs, not simulated components.

In the **Augmented Reality use case**, the User Equipments recieve high-quality video with low latency. We monitor the end-to-end latency to evaluate the different system architectures. The LatOpt architecture should improve this metric by reducing the

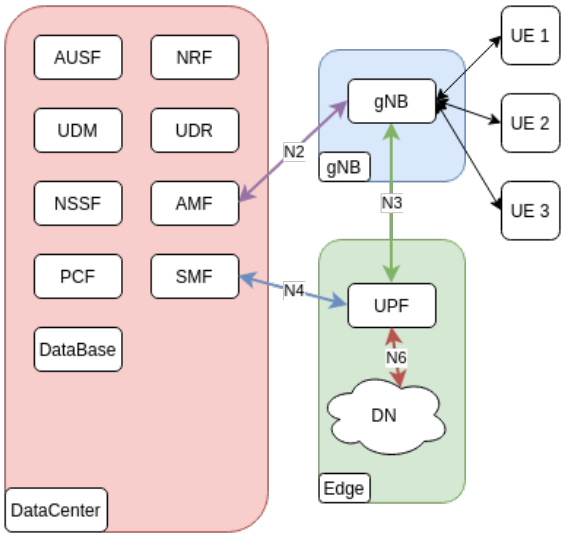


Figure 3.4.: **LatOpt** architecture: optimized for end-user latency and bandwidth.

distance between the UPF and the gNB.

In the **Industrial IoT use case**, the User Equipments are sensors in a smart factory. In this industrial use case, we consider that the UEs are mostly statics; they stay in the same network cell most of the time. However, they create a new connection each time they turn on. These devices are periodically turning on and off in this scenario. This routine consists of turning on, collecting data, establishing a connection with the nearby antenna, establishing a data session with the UPF, sending the data to a server for local processing, and turning it off. We monitor the end-to-end latency to evaluate the performance of this use case. This metric includes the time to establish a PDU session, send the data, and get a response from the server. The AccessOpt architecture should lower the E2E latency for this use case. Deploying the AMF and the SMF on a Cloudlet node should reduce the PDU session establishment time, while a UPF closer to the gNB should reduce networking delays for the data session.

In the **Massive IoT use case**, the User Equipments are small battery-powered IoT devices that periodically connect to the cellular network. These devices turn on and off periodically to send data and spend most of the time in energy-saving mode. The 5G core needs to quickly handle the procedures generated by the devices turning on and off or moving to another 5G cell. We monitor the time to complete the registration and the PDU session establishment procedures to evaluate the performances of the different system architectures. The AccessOpt architecture, with the SMF and the AMF deployed on Cloudlet nodes, should improve the completion time of these procedures.

Table 3.1 summarizes the use cases we study. It presents which part of the 5G infrastructure is tested and the metrics we use to evaluate performance.

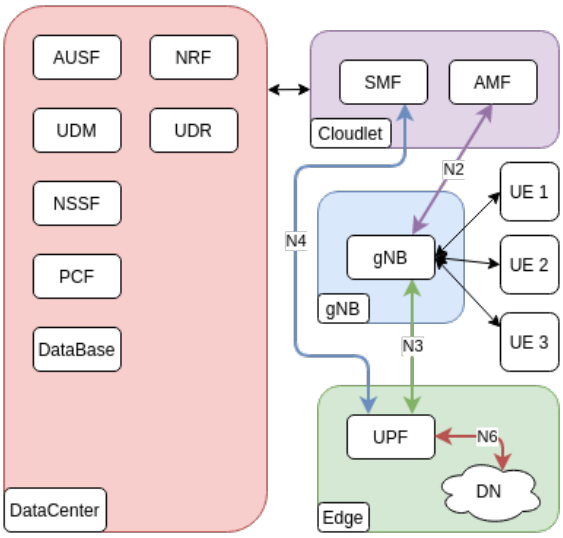


Figure 3.5.: **AccessOpt** architecture: optimized for session throughput.

3.4. Evaluation

This section evaluates our experimental methodology with a 5G workload. We first present our experimental setup and parameters. Then, we present the results of our 5G study.

3.4.1. Experimental setup

We run all the experiments in a public cloud environment. We use a self-managed Kubernetes cluster with one master node and seven worker nodes. All of these machines have 2 CPUs and 4 GB of RAM. On this cluster, we run the open-source 5G core free5G [106]. Every 5G Network Function (NF) runs inside its pod. The User Equipments (UEs) and gNodeBs are emulated using an open-source RAN simulator [107].

To validate our approach and ensure our results, we analyze the pcap traces of these experiments. These records ensure that this 5G core open-source implementation behaves as expected. We compare the recorded procedures to the one defined by the 3GPP [108]. We present more details about these procedures in Appendix A.

3.4.2. Experimental parameters

System architecture parameters

Table 3.2 presents the delays we configure for the various system architectures defined in section 3.3.2. We use our experimental methodology to set up these artificial latencies between the nodes of our testbed cluster. The artificial latencies presented in this table are additional delays we add to the existing ones in the data center infrastructure.

Table 3.1.: Use cases and their characteristics.

Use cases	**Favoured System architecture**	**Type of workload**	**KPIs**
AR (Smart Factory)	Baseline or LatOpt	High data rate on the User Plane (UP)	E2E latency
IIoT (Smart Factory)	Baseline or AccessOpt	PDU session establishment process + Low data rate the UP	E2E latency
MIoT (Massive IoT)	Baseline or AccessOpt	Registration + PDU session establishment process + Low data rate on the UP	Time to register + establish a PDU session

Table 3.2.: Delays in the system architectures.

System Architecture	**N2 (ms)**	**N3 (ms)**	**N4 (ms)**	**N6 (ms)**	**DC-Cloudlet (ms)**
Baseline	12.5	12.5	0	0	0
LatOpt	12.5	1	12.5	0	0
AccessOpt	3.5	1	3.5	0	9

Workload parameters

Table 3.3 summarizes the use cases' workload parameters of the different experiments. The workload of the IIoT and MIoT use cases should mainly be managed by the control plane (respectively, on SMF and AMF). In contrast, the User Plane (UPF) should support the AR use case workload.

3.4.3. Results

In this section, we analyze and compare the performances of an open-source 5G core when using different system architectures. These results provide insights into which architecture performs best for each use case. Figures 3.6, 3.7, and 3.8 show the mean KPI values for each use case according to the chosen architecture.

Figure 3.6 presents the results for the AR use case experiment. That experiment compares the end-to-end latency of the application when using the baseline and the LatOpt architectures. We observe that the average end-to-end latency is four times lower when using the LatOpt architecture. With the LatOpt architecture, the UPF is

Table 3.3.: Workload parameters.

Use Case	Workload size	#UEs
AR	460 Mbit of video stream sent to the UE	3
IIoT	PDU session establishment requests + 640 kB of data	20
MIoT	UE registration requests + PDU session establishment requests + 100 kB of data	50

closer to the gNB (the delays on the link N3 are lower), which explains the lower end-to-end latency. This first experiment shows that the testbed built with our methodology successfully reproduced a well-known result.

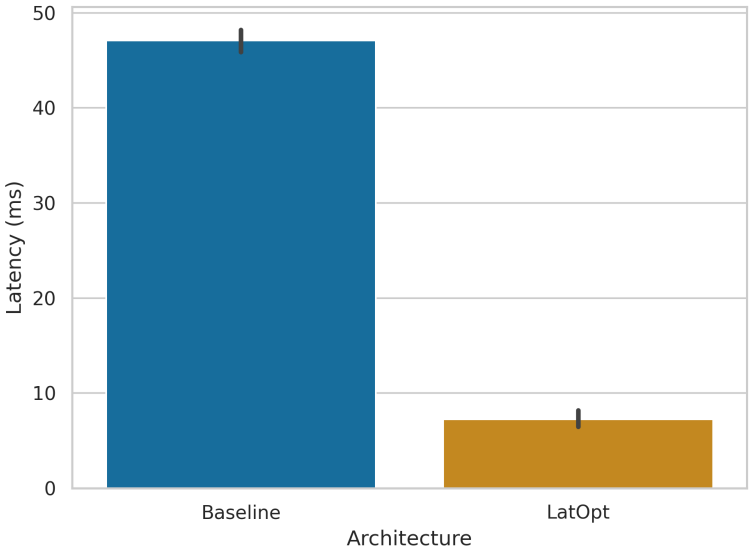


Figure 3.6.: Average end-to-end latency: AR use case.

Figure 3.7 shows the average end-to-end latency, data session latency, and duration of the PDU session establishment procedure for the IIoT use case. The end-to-end latency is four times lower for the AccessOpt architecture than the baseline. The AccessOpt architecture reduces the time to complete the PDU session establishment procedure and the data traffic duration (i.e., networking delays and processing time). Lower latency on the N3 link explains the reduced data session latency. The AMF and SMF are the main NFs involved during the PDU session establishment procedure. Lower the communication delays between these two NFs and the UEs shorter the procedure duration. During the procedure, almost half of the requests from AMF or SMF go to data center nodes, and the other half go to the edge. Having AMF and SMF on a Cloudlet reduces the duration of the whole procedure. The AccessOpt architecture reduces the end-to-end latency for the IIoT use case.

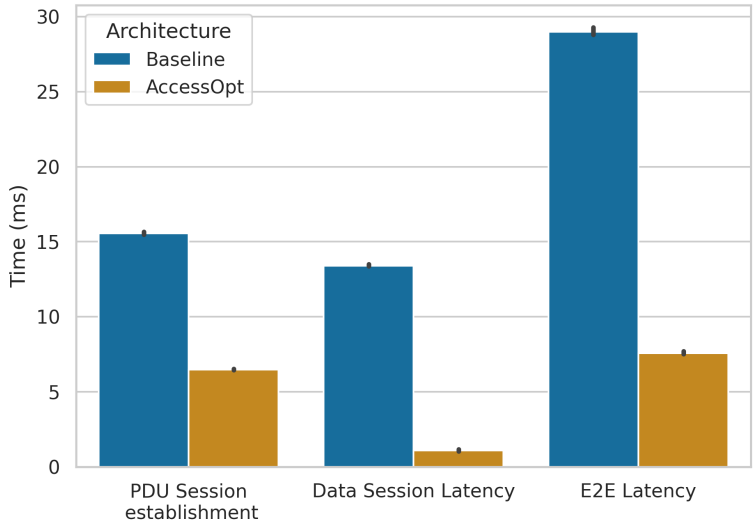


Figure 3.7.: Average end-to-end latency: IIoT use case.

Figure 3.8 presents the average duration of the UE registration and PDU session establishment procedures. The total time to complete these two procedures is 13 times faster on the Baseline compared to the AccessOpt architecture. Also, the UE registration procedure is 14 times faster on the Baseline architecture than on AccessOpt. However, the PDU session establishment procedure is ten times faster on the AccessOpt architecture. The UE registration procedure is significantly longer than the PDU session establishment. The session establishment represents only 0.06% of the total time for AccessOpt and 8% for Baseline. Therefore, the PDU session establishment procedure duration has a limited impact on AccessOpt architecture's total performance. The UE registration procedure strongly impacts global performance. The AMF exchanges requests mainly with the NFs running on data center nodes during the UE registration procedure. Therefore, the delays for each request are longer when the AMF is running on a Cloudlet node. Placing the AMF closer to the edge improves the performance of the PDU session establishment procedure. However, it negatively impacts the performance of the UE registration procedure. Using our experimental methodology helps detect issues with architecture. For example, a procedure becomes longer to complete if one of the NF is placed at the edge without other Network Functions.

Our experimental methodology helps to choose the best architecture for each 5G use case. Placing the UPF at the edge reduces the latency on the link N3 in every configuration tested. The optimal position of the AMF depends on the use case's procedures. AMF improves KPIs for the PDU session establishment procedure when placed at the edge (or nearby), while results are better for the UE registration procedure when it stays in the data center.

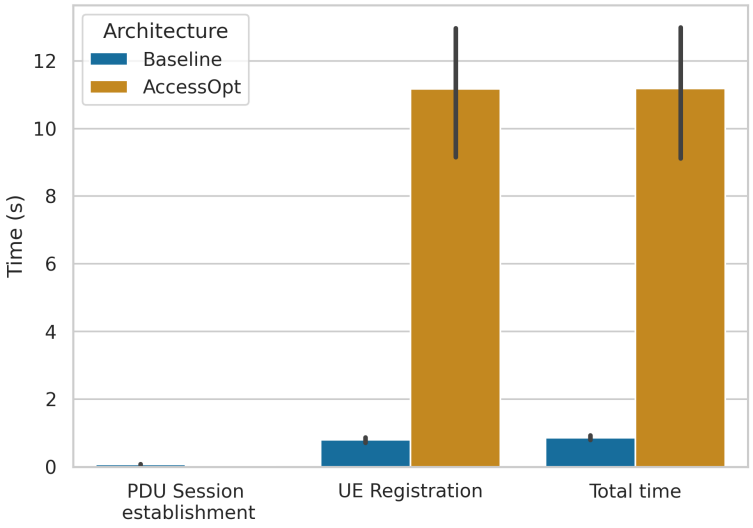


Figure 3.8.: Average duration of different procedures in the MIoT use case.

3.5. Limitations

This section presents the main limitations of our experimental methodology and 5G study.

We do not demonstrate the usage of accelerators with our 5G study. However, they are supported by our methodology. Kubernetes can easily handle GPGPUs, and it is not a problem to adapt scheduling for selecting nodes that have them if necessary. It can also support other accelerators like FPGA [109]. However, our experiments with the 5G system do not demonstrate these capabilities.

Our 5G study has two main limitations: we do not experiment with real 5G RAN, and the size of the cluster and the workload are limited. Using a real 5G RAN might change the values of the KPIs we record. However, they do not change our conclusion about the placement of the Network Functions. Additional testing on a larger cluster and with more User Equipments can improve the strength of our results. However, our experimental methodology will remain the same. It is open-sourced and easily reproducible. Therefore, it is possible to reproduce these experiments on any scale.

3.6. Related work

This section presents the related work of this chapter. Related works include other testbed studies, 5G architectures, and other 5G testbeds.

Goshi et al. describe a testbed that highlights Inter-NF dependencies [110]. Kube5G is a cloud-native 5G testbed designed to handle the whole 5G stack [111]. COPA is an orchestration framework for networking running above the Kubernetes layer [112]. These

three testbeds (and the one mentioned in their studies) do not aim to investigate how the various topologies of the Computing Continuum impact application performance. These studies do not offer the possibility of configuring networking between the cluster's nodes to create different architectures.

[113, 114, 115, 116] describe multi-layered 5G architectures similar to the *AccessOpt* architecture presented in section 3.3.2. It is a common 5G architecture with network slices deployed at different geographic locations. Sarrigiannis et al. describe a two-tier architecture (Cloud and Edge) for virtual NF placement with a VNF orchestrator [117]. They are not using the benefits offered by Kubernetes for their VNF orchestrator. Ejaz et al. present a three-tier architecture (Cloud IoT, Edge IoT, and Local Edge IoT) to improve reliability for mission-critical processes, based on *iFogSim* simulator [118]. This study helped us to define our system architectures. However, the iFogSim simulator does not allow deploying a real containerized application.

Edgenet, as described by Şenel et al. [119], uses a Kubernetes cluster. However, it is unsuitable as a testbed for 5G core or other edge-based applications as networking cannot be configured, and there is no access to the Edge nodes. Enoslib [120] is another suggestion to facilitate experimentation with distributed systems. It is a general tool to facilitate reproducible experimentation and is thus orthogonal to our methodology, which could be used as the backend in an Enoslib experiment.

To the best of our knowledge, there is no methodology for studying the performances of applications running over the Cloud-to-Edge Computing Continuum. Simulations do not offer to run the actual software, so we cannot access real performances. Testbeds offer no way to replicate the various architectures of the Computing Continuum or the effects of its geographically distributed nature on performance.

3.7. Conclusion

This chapter presents a new experimental methodology for studying the orchestration of applications in the Edge-to-Cloud Computing Continuum. We ensure the efficiency of this methodology by conducting a 5G core study. That study adds valuable input to understanding 5G systems and provides guidance for building new infrastructure. A Telco operator can reproduce and extend our experiments to study new infrastructure at a lower cost without deploying expensive hardware over the Computing Continuum.

In the next chapter, we use this experimental methodology to design and investigate new unified orchestration approaches for the Computing Continuum.

Chapter 4.

Network-aware orchestration in the Computing Continuum for cost minimization

This chapter presents a unified orchestration approach for the Cloud-to-Edge Computing Continuum. It is a network-aware approach that aims to reduce the costs of deploying applications in the Computing Continuum.

This chapter is based on the following publications:

- Rac and Brorsson. "Cost-Effective Scheduling for Kubernetes in the Edge-to-Cloud Continuum". 2023. [53].
- Rac and Brorsson. "Cost-aware Service Placement and Scheduling in the Edge-Cloud Continuum". Mar. 2024. [54].

This chapter is organized as follows. Section 4.1 presents the main concepts and definitions of this chapter and an overview of our unified orchestration approach for the Computing Continuum. Our orchestration methodology is exposed in section 4.2. In section 4.3, we present our experimental results. Our experiments are based on a vehicular cooperative perception workload running on a Kubernetes cluster. Then, we present the limitations of our approach and the relevant related work in section 4.4 and section 4.5. Finally, we present our conclusions and future work directions in section 4.6.

4.1. Introduction

Edge computing can help reduce the costs of deploying an application. Cloud providers offer their resources as *Everything as a Service (XaaS)* [121]. Therefore, the customers pay only for the resources they use. For example, customers pay an hourly rate for the server they use. In addition, cloud providers also charge for the traffic that goes outside of their data centers. Deploying network-intensive applications at the edge may save the outgoing network costs; data stays local, and traffic between the edge and the data center is not charged.

To minimize the cost of deploying applications in the Computing Continuum, we built a system that automatically finds a trade-off between computing and networking costs.

We call our approach network-aware because we minimize the total costs (i.e., compute and communication) based on an estimation of network traffic. Deploying applications at the edge should be as simple as it is for data centers to facilitate the adoption of edge computing. Our orchestration approach should help make application deployment at the edge easier. Application developers do not need to specify where applications should run using this system. The system will choose a location to minimize the costs and dynamically update that choice if the situation changes.

We propose two schedulers[1] to lower the costs of deploying distributed applications in the Cloud-to-Edge continuum. We consider applications that consist of several services that need to interoperate. To make application deployment easy and as close as possible to industrial standards, we implement our methodology as a Kubernetes scheduler plugin [75] based on network requirements and costs. Any containerized application can be deployed and make use of our schedulers. We want to leverage local data processing to reduce the costs of running applications to save bandwidth costs.

When end-users (e.g., a phone, a vehicle, or a camera connected to an application hosted in the Computing Continuum) are moving, the delay to connect or communicate with a service of the application may become longer. This means that the optimal location of a service may vary over time. We need to be able to move services automatically when users move; otherwise, the Quality of Service (QoS) will deteriorate. Therefore, we also propose a *rescheduler* to monitor the application and trigger service migration when needed. Our rescheduler is a background process that periodically checks costs and KPIs to identify better nodes on which a service can be deployed. If the rescheduler detects an improvement, it will automatically migrate the service pod[2]. Also, when a cheaper resource becomes available, the rescheduler can detect it and move the associated service to reduce the costs (provided the QoS is not negatively affected).

We make the following key contributions in this chapter:

Contributions

1. Design and implementation of two cost-aware Kubernetes scheduling plugins. One based on latency between a service and an end-user (or another service), and the second based on communication patterns.
2. Design and implementation of a network-aware rescheduler to keep application placement decisions optimal when end users are moving.
3. A performance evaluation of our scheduling plugins and the rescheduler methodology using a realistic 5G use-case of cooperative perception for autonomous vehicles.

From our experiments, we have verified that our custom schedulers can achieve, in

[1] We use the Kubernetes terminology here as they are strictly not schedulers but rather *placement algorithms*.
[2] A *pod* is the smallest schedulable unit in Kubernetes, containing at least one container

most cases, about 10% to 25% lower costs than the default Kubernetes scheduler. It is worth noting that this is not a simulation study. We have made real scheduler plugins demonstrated in a real Kubernetes cluster.

4.1.1. Definitions

This section defines the main terminology of this chapter. In this, we use a simplified representation of the Computing Continuum. We use only *cloud* and *edge* nodes to simplify our explanations and modelization. However, it is possible to use any type of node in practice. Only the characteristics of the nodes matter for our orchestration approach.

We define the *Edge-to-Cloud Computing Continuum* as the aggregation of servers located at the edge of the network and those in traditional data centers (cloud). The edge nodes are servers located at the edge of the network, outside of the data centers, e.g., near 5G base stations. Figure 4.1 represents such an Edge-to-Cloud-Computing Continuum. The blue servers are the cloud nodes, and the green ones are the edge nodes.

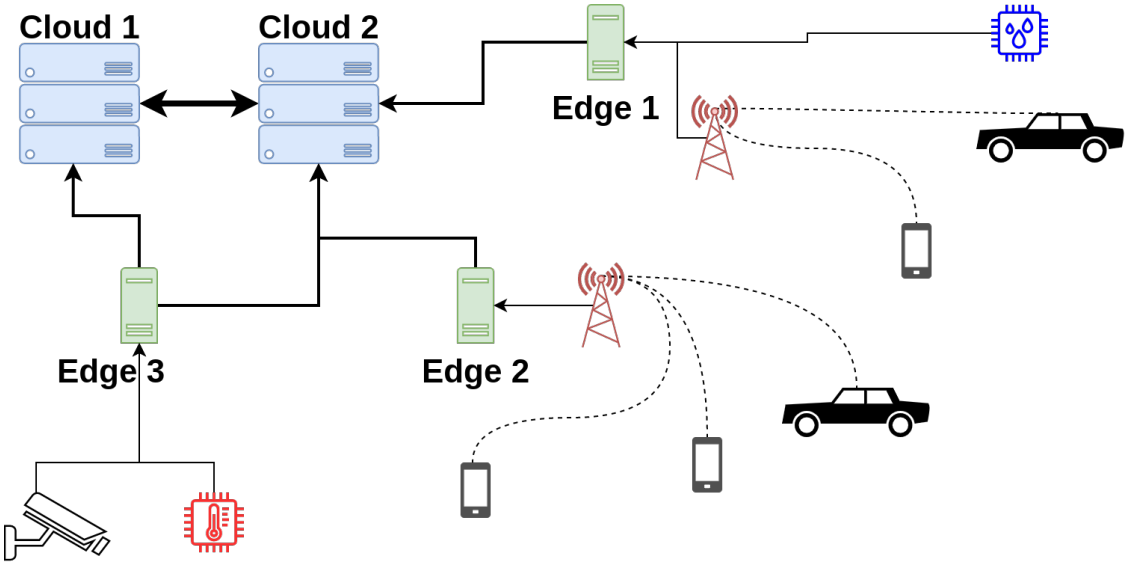


Figure 4.1.: The Edge-to-Cloud Computing Continuum - Computing resources are in blue and green.

Different kinds of nodes could be added to this Computing Continuum. The scheduling decisions are based on node characteristics such as available resources, bandwidth, latency, or costs. We use static node types like edge and data center nodes to understand the Computing Continuum behaviors better.

End-users (e.g., vehicles, phones, cameras, or smart sensors) are connected to services hosted in the Computing Continuum. The end-users may be connected directly to cloud centers or to an edge node. Some end-users can be mobile, and in this case, their closest node changes over time.

We are considering distributed *applications*, made of *services*, to be deployed on this Cloud-to-Edge Computing Continuum. A service can run on an edge node as well as on a cloud node. A *scheduler* maps the services to the nodes. The *rescheduler* is a background process that can update a service allocation. It can move a service to another node in the cluster.

There are two kinds of costs in our edge to data center Computing Continuum: instance cost (price paid by hours and number of CPU used), and communication costs (per GB). It is possible to define additional costs like storage (per GB), but these are outside of the scope of this study.

Communication costs represent the price to pay to transfer data from the edge to a data center. We are not counting the transfer price between an edge node and its end-user (e.g., through 5G). This parameter can be optimized by choosing an internet provider or a telco operator, but it is outside of the scope of this study. Also, if two data center nodes are part of one data center, we are not counting communication costs. Nevertheless, communication costs apply for transfers between two edge nodes.

4.1.2. Unified orchestration methodology overview

We think that Kubernetes is one of the best candidates for orchestrating applications over the whole Cloud-to-Edge Computing Continuum. Kubernetes is the *de facto* industrial standard for container orchestration. It is not only limited to Docker containers [122], but it can manage any artifact following the Open Container Initiative (OCI) specification [123]. Using standard containers is important to ensure compatibility with most of the different hardware (e.g., with different CPU architectures) we can find in this Computing Continuum. Also, it is easier for the industry to adopt this technology if they can continue using the same containers they already have. Chapter 2 section 2.1 provides more details about Kubernetes and container technology.

However, Kubernetes is not yet ready to orchestrate resources over the whole Cloud-to-Edge continuum. The default scheduling approach of Kubernetes is to spread the containers over the cloud, choosing the least allocated server. This is good practice in a traditional data center as it avoids server overload. However, this is not applicable to edge nodes. It is important to consider the geographical location of the servers in order to achieve low latency to access the services. Therefore, it is not possible to efficiently place services in the whole Computing Continuum without considering the networking resources.

We propose to use and extend Kubernetes to orchestrate applications deployed in the Cloud-to-Edge Computing Continuum. With a network-aware scheduling approach, we think it is a good candidate for orchestrating heterogeneous resources. Using the Kubernetes self-healing mechanism also helps improve the reliability of the system. If an edge node fails or if it has networking issues, its pods can be redeployed on another available node.

In this heterogeneous Kubernetes cluster, we define a cost policy based on two values: CPU and network usage. Instance price is based on the time and the number of CPUs

used. Networking costs are charged based on the amount of data transferred. If data stays in the same geographical location, networking costs do not apply. They only apply when data is transferred from one region to another, e.g., from one edge location to a central data center. This cost policy is in line with what public cloud providers use.

We use the Kubernetes scheduler framework [75] to implement our scheduling methodology for the Edge-to-Cloud Computing Continuum. The Kubernetes scheduling framework is easy to extend. Every scheduling stage is defined as a plugin; we can write new plugins to replace the default ones.

There are four main scheduling stages in the *kube-scheduler*: *filter*, *score*, *normalize score*, and *reserve*. The scheduling pipeline first selects an unallocated pod (i.e., a set of containers, the atomic schedulable unit in Kubernetes) from the scheduling queue. Then, un-schedulable nodes (e.g., reserved nodes, control plane only, out of resources) are filtered and removed from the possible nodes. After the filtering stage, the remaining nodes are scored, and the scores are normalized to between 0 and 100. Normalization is necessary to get an average score when using many scoring plugins. The node with the highest score is selected and reserved for the pod. If two nodes get the same score, one of them is randomly selected.

We propose to extend Kubernetes to offer a unified orchestration to the Computing Continuum. Our Kubernetes extension is a twofold solution: a service *initial placement* method and a *rescheduler* (RS). The service initial placement assigns a node to every service. The rescheduler watches the existing services and migrates those that are not in an optimal location. When the optimal solution changes over time, a rescheduling process is necessary to adapt the service locations. For example, when end-users move, the optimal solution changes over time.

We propose two methods for the service initial placement: One based on latency and the other based on communication patterns.

The latency-based initial placement method assigns unscheduled services to nodes. To select which node will host a service, we rank the nodes using both their costs and their latencies to other nodes. In the scope of this study, the cost is the money paid to a cloud provider to run a service. Other quantities, such as energy (to power the servers and the network equipment), can also be considered and minimized using this methodology, which we will do in future work. Checking the delays with other nodes helps to reduce the end-to-end latency of an application and ensure the quality of service.

The communication-based initial placement method also assigns unscheduled services to nodes. However, it chooses to host services using a different approach. Instead of using latency, this scheduler uses the communication patterns between services (inter-service traffic) and their end-users. The scheduler uses the communication patterns of previously deployed services to predict the behavior of the new services. It assumes that new services with the same code will behave similarly.

The rescheduler is a background process that re-evaluates the initial placement decision of the services and moves the services to different nodes if a better solution is found. The optimal solution changes over time; users can be moving, or new resources

may become available. Using the rescheduler ensures keeping the costs low by taking new placement decisions; moving the services to a cheaper node if it is possible.

We include a monitoring system in our placement methodology to support the scheduler and the rescheduler. The monitoring system collects metrics about the cluster state and the applications running in the cluster. Also, it stores the metrics and serves them to the scheduling components. The monitoring system can serve raw data or aggregated data, e.g., using basic statistical functions such as an average or a median. The implementation of the monitoring system relies on open-source components. It is described in section 4.2.4.

Figure 4.2 presents a high-level view of our system. We run a custom scheduler to deploy applications in the Edge-to-Cloud Computing Continuum. There are two versions of this custom scheduler: one implementing the latency-based approach and the other implementing the communication-based approach. We use the custom scheduler to orchestrate workloads, but the default scheduler is still usable for deploying the control plane and the monitoring functions. In the cluster, we reserve nodes for the control plane (in a traditional data center) and nodes for deploying workloads (at the edge or in traditional data centers). The custom scheduler and the rescheduler are running on the control plane nodes of the cluster. These nodes are located in a traditional data center. Applications can run on nodes located either at the edge of the network or in traditional data centers.

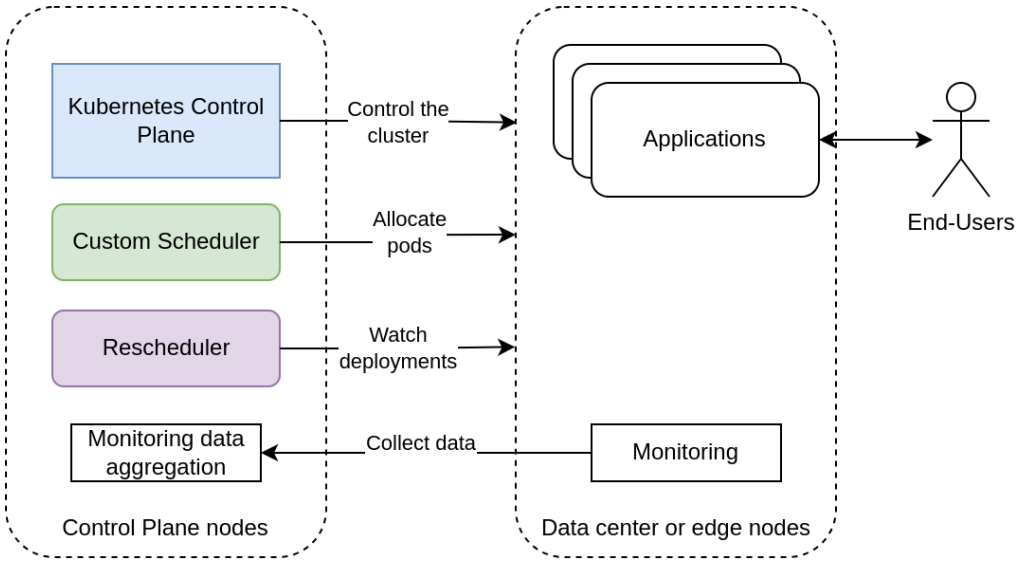


Figure 4.2.: Architecture overview of the system. Our scheduling components are running on control plane nodes. Applications can run anywhere in the Cloud-to-Edge Computing Continuum.

4.2. Orchestration methodology

In this section, we are presenting an overview of the optimization problem we are solving. Then, we present the algorithms for i) the scheduler scoring plugin, and ii) the rescheduler. Finally, we expose how we implement this scheduling methodology on a Kubernetes cluster.

4.2.1. Optimization problem overview

This chapter presents an orchestration methodology that minimizes the cost of deploying applications in the Cloud-to-Edge Computing Continuum. We are considering two different kinds of costs: i) computing costs and ii) networking costs.

Computing cost is the price to pay to use a server from a cloud provider. It is usually charged by hour or month and depends on the characteristics of the machine. In this chapter, we assume that every node in the Computing Continuum uses the same pricing policy: an hourly rate that depends only on the characteristics of the instance. Note that we are not labeling the edge nodes as special nodes; all the nodes are the same for the scheduler. Only the instance characteristics (e.g., processors, memory, geographical location) and its networking capabilities (e.g., latencies with other machines, available bandwidth) matter for the scheduling decision.

Networking cost is the price to pay to send data over the network. This price depends on the quantity of data sent and its destination. Data traffic is not charged for machines in the same data center. However, if data goes outside of the data center to the edge, then traffic is charged. Also, no additional network cost is charged if edge data is processed locally. However, if data traffic is sent to a data center, then network traffic is charged.

Our objective is to find a trade-off between paying computing resources and networking costs. Depending on the workload characteristics, deploying the application in a data center or at the edge can be cheaper. For instance, deploying a network-intensive workload at the edge where data is produced is cheaper than in a data center since data stays local.

Equation 4.1 presents the optimization problem we are solving; it is the minimization of the total costs (i.e., the computing and the networking costs).

Equation 4.2 details the computing costs, and equation 4.3 details the networking costs. Equations 4.4, 4.5, and 4.6 are the constraints for, respectively, the maximum allocation per server, CPU limit, and memory limit.

$$min(\mathcal{C}_{comp} + \mathcal{C}_{netw}) \tag{4.1}$$

$$\mathcal{C}_{comp} = \sum_{n=1}^{N} \sum_{m=1}^{M} cpuPrice(s_n) \times cpuReq(\sigma_m) \times I(s_n, \sigma_m) \tag{4.2}$$

$$\mathcal{C}_{netw} = \sum_{m=1}^{M} \sum_{m'=1}^{M} \nu(\sigma_m, \sigma_{m'}) \times commPrice(\sigma_m, \sigma_{m'})) \tag{4.3}$$

s.t.

$$\forall n \in [1, N], \sum_{m=1}^{M} I(s_n, \sigma_m) \leq 1 \tag{4.4}$$

$$\forall n \in [1, N], \sum_{m=1}^{M} cpuReq(\sigma_m) I(s_n, \sigma_m) \leq cpuCap(s_n) \tag{4.5}$$

$$\forall n \in [1, N], \sum_{m=1}^{M} memReq(\sigma_m) I(s_n, \sigma_m) \leq memCap(s_n) \tag{4.6}$$

With:

Server: s_n, $n \in [1, N]$

Service: σ_m, $m \in [1, M]$

Amount of data sent from the services $\sigma_m$to $\sigma_{m'}$: $\nu(\sigma_m, \sigma_{m'})$

$I(s_n, \sigma_m) = \begin{cases} 1 \text{ if } \sigma_m \text{ is allocated on } s_n \\ 0 \text{ otherwise} \end{cases}$

We are using heuristic optimization to find an approximate solution to the above-mentioned problem. Figure 4.3 presents a high-level view of the scheduling workflow of our solution. The first part details the initial placement stage, where the scheduler assigns an unallocated pod to a node. Then, the rescheduler periodically checks in the background to see if a better node can be found for this pod.

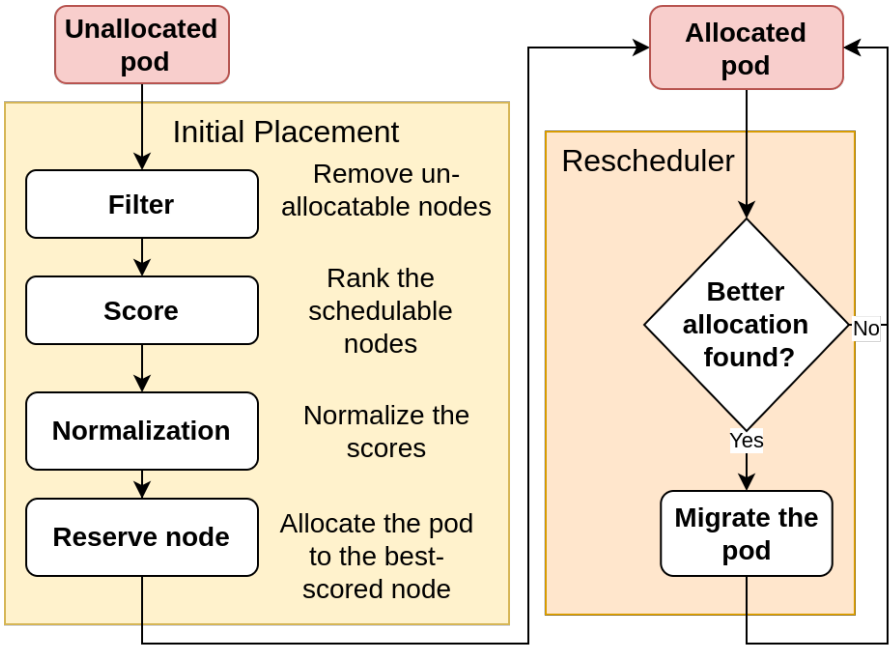


Figure 4.3.: Scheduling workflow: Initial Placement (yellow), Rescheduler (orange)

4.2.2. Service initial placement

The objective of service initial placement is to associate the unallocated pods from the scheduling queue with a node in the Edge-to-Cloud Computing Continuum. The first step is to build a list of nodes where the pod can run. We remove from that list the nodes that cannot host the pod (e.g., those reserved for the control plane, or those with insufficient resources). Then, the nodes are scored, and the best one is selected to host the pod.

We propose two Service Initial Placement algorithms that share the objective of minimizing the total costs.

Latency-based Initial Placement

Algorithm 1 describes the node scoring method of the Latency-based Initial Placement (LIP) approach. The LIP scheduler runs the corresponding scoring method for each schedulable node. The scoring method sorts the nodes using the networking delays and their price (i.e., the hourly price paid to use a server). We choose a node close to the connected services to improve the end-to-end latency and lower the networking costs (saving bandwidth).

Algorithm 1 Latency-based Initial Placement (LIP) Scheduler: Scoring

Require: $\alpha > 0$, $\beta > 0$, Pod, Node
  $n_s \leftarrow 0$ ▷ Node score
  for service in Pod.dependencies **do**
    $\lambda \leftarrow GetLatency(Node, service)$
    if $\lambda < \alpha$ **then**
      $n_s \leftarrow n_s + \beta$
    end if
  end for
  $n_s \leftarrow n_s + \frac{1}{Node.Price}$
  return n_s

Nodes that are close to connected services get a higher score. To reduce the distance between a pod and the connected services, we defined a list of these services for each pod: *pod.dependencies*. Then, we get a list of the nodes where these services are deployed. We do not add the server to the list if the service is not deployed yet. Once the server list is built, we evaluate the latencies between these servers and the evaluated node. If the latency is lower than α, we add β to the final score (α is a latency distance in ms, and β is a score modifier.). The more connected services in a radius of α, the higher the score.

To lower the deployment costs, we add the inverse of the node price to the final score. The lower the node price, the higher the score.

The values of α and β should be chosen regarding the cluster characteristics. α should be chosen relative to the values of the latencies between the nodes. β should be chosen

relative to the values of the inverse of the node price.

In every case, this algorithm selects a node for the pod to deploy. If two nodes get the same score, one is randomly selected by the scheduler.

Communication-based Initial Placement

The Communication-based Initial Placement (CIP) approach uses the knowledge gained from the communication patterns of already deployed pods to estimate the networking costs accurately. The Communication-based Initial Placement scheduler can choose the cheapest node using this information. I.e., with the lowest sum of instance and networking costs.

The main component of the CIP approach is the *traffic exporter*. It estimates the networking costs and makes that information available to the scheduler. Section 4.2.4 presents the implementation details of this component.

The traffic exporter needs to access previous communication patterns to provide accurate estimations. The first services are placed using only the instance prices. Then, once there is enough data for the traffic exporter to build an estimation, the scheduler places the following pods using instance and networking prices. The first services placed only with instance prices can be migrated later by the rescheduler if there is a better place for them.

Initial placement methods comparison

The CIP approach presents many benefits over the LIP approach. CIP does not require a manual setting of its parameters, while we need to set LIP parameters: connected services, α, and β, which are not always trivial to choose. Also, using an estimation of the networking costs (in the CIP approach) leads to better performance.

However, the CIP approach needs to collect some data before it can be effective. The CIP scheduler starts collecting data when the first pod is deployed. In practice, this startup time is not an issue, according to experimental results presented in section 4.3.4.

Future work may, however, investigate a hybrid approach: using the LIP scheduler while the CIP is collecting the initial data and then using the CIP scheduler. The LIP scheduler could be used to deploy the first instance of each pod, and the CIP scheduler could be used to deploy the following instances.

4.2.3. Service rescheduling

The rescheduler is a background process that periodically checks if a better node is available to host a pod. We build the service rescheduler for two main reasons: i) the best location for a service varies over time (e.g., node availability changes depending on the current load, end-users are moving), ii) we can use data collected when the service is running to improve the scheduling decision. If the rescheduler finds a better node for a pod, it will migrate the pod to the better node.

Algorithm 2 describes the rescheduling process. This function is called periodically. The algorithm iterates over every workload pod in the cluster. The first step of this algorithm is to estimate the current cost of the evaluated pod. This evaluation includes the computing and networking costs. Then, all of the other schedulable nodes are evaluated similarly; we keep the node with the best score. If it is the same as the original one, there is nothing to do; if it is a different node, the rescheduler will migrate the pod toward this node.

The estimation of networking costs is crucial to this algorithm. Computing costs are easy to evaluate; they are static and known. However, networking costs depend on each service behavior and are not known *a priori*. Using the monitoring setup described in 4.2.4, we can consult how much data was sent to which destination. Using this information, we can estimate future data usage and networking costs.

Algorithm 2 Rescheduler: Background routine

for pod in WorkloadPods **do**
  $pod_{c\_cost} \leftarrow pod_{CPU} \times$ pod.node.Price
  $pod_{n\_cost} \leftarrow GetNetwCostEstimation(pod.node)$
  $best_{score} \leftarrow pod_{c\_cost} + pod_{n\_cost}$
  $best_{node} \leftarrow$ pod.node
  for node in SchedulableNodes **do**
    $pod_{c_cost} \leftarrow pod_{CPU} \times$node.Price
    $pod_{n\_cost} \leftarrow GetNetwCostEstimation(node)$
    $node_{score} \leftarrow pod_{c\_cost} + pod_{n\_cost}$
    if $node_{score} < best_{score}$ **then**
      $best_{score} \leftarrow node_{score}$
      $best_{node} \leftarrow$ node
    end if
  end for
  if pod.node $\neq best_{node}$ **then**
    pod.MigrateTo($best_{node}$)
  end if
end for

4.2.4. Implementation on Kubernetes

This section describes how we implemented the above-described methodology to make it usable on a Kubernetes cluster with any containerized workload.

We use the default Kubernetes plugins for the Filter and Reserve Node phases. The default Filter plugin removes nodes with a *taint* that the pod does not tolerate (e.g., the master mode has the *taint* "master" to prevent workload pods from running with the control plane node) from the list of the schedulable nodes. It also removes nodes that do not have enough resources to host the pod. The default Reserve Node phase updates

ETCD [124], the Kubernetes internal database. Then, the selected node will start the pod deployment based on the information in this database.

Latency-based initial placement

For the Latency-based Initial Placement (LIP), we use the scheduler plugin framework, [75], which we have extended with our scoring plugin for latency-based initial placement. We collect the necessary latencies using the open source tools described in section 4.2.4.

Communication-based Initial Placement

The Communication-based Initial Placement (CIP) method scores nodes based on an estimation of the cost of deploying a pod on that node. This estimation includes both networking and instance costs. Knowing how much data will be transmitted is essential to estimate the networking costs. The *traffic exporter* is a component that provides an estimation of the network traffic that the pod will generate and sends it to the scoring plugin. The network traffic estimation includes communication with the other pods and communication with the end-users.

In order to estimate the communication patterns, we look at the previous communication. Based on the previous data, we try to predict the next ones, assuming a similar communication pattern in the future. The *traffic exporter* relies on two graphs: an abstract graph and a real communication graph. The abstract graph contains the prediction of communication patterns. The choice of the estimator depends on the nature of the workload and the characteristics of its communication patterns. The real communication graph represents all the volume of data exchanged between two pods or between a pod and its user. The vertices are the pods, and the edges contain the volume of data exchanged. We use the real communication graph to build the abstract graph. Once the abstract graph is generated, we can use it to estimate data traffic. That estimation is important to estimate the networking costs and, therefore, the total cost of using a node. Using the cost estimation of each node, we can rank the nodes and choose the cheapest one. If two nodes have the same cost, we choose one randomly.

The traffic exporter is doing three independent tasks:

1) Record the traffic the traffic exporter gets the traffic between the pods and stores it as the *Real Communication Graph*.

Each pod is a vertice in this graph. The edges represent the amount of data exchanged between two pods or between a pod and its end user. On each vertice, it stores the hash of the container images of the pod. It uses this hash as an identifier (a pod *type*). Two pods running the same software have the same type. The traffic exporter also attributes a type for each end-user.

2) Generate communication patterns The traffic exporter builds an abstract representation of the traffic to summarize the information in the measured communication

graph.

The traffic exporter constructs an abstract graph using the type of pods. The objective is to extract a common data traffic pattern from pods of the same type. The abstract graph uses the types of nodes as vertices and the traffic as edges. To build this graph, the traffic exporter aggregates the traffic using an aggregation function (e.g., the average).

The CIP training process corresponds to the steps of recording the real traffic and building the abstract. The training starts as soon as one pod is deployed. Then, the following pods with the same type can benefit from that learning.

3) Serve the communication patterns The traffic exporter provides communication patterns for a given type when the CIP scoring plugin requests it. The scoring plugin requests communication patterns based on the type of the evaluated pod.

Figure 4.4 illustrates the CIP mechanisms. The traffic exporter extracts communication patterns from running pods and serves them to the CIP scheduler. We store the graphs of the communication patterns in a graph database. Then, we expose them using Prometheus API. The Prometheus server collects them periodically, and the scheduler can request the Prometheus server. We provide more details on the monitoring setup and Prometheus in section 4.2.4.

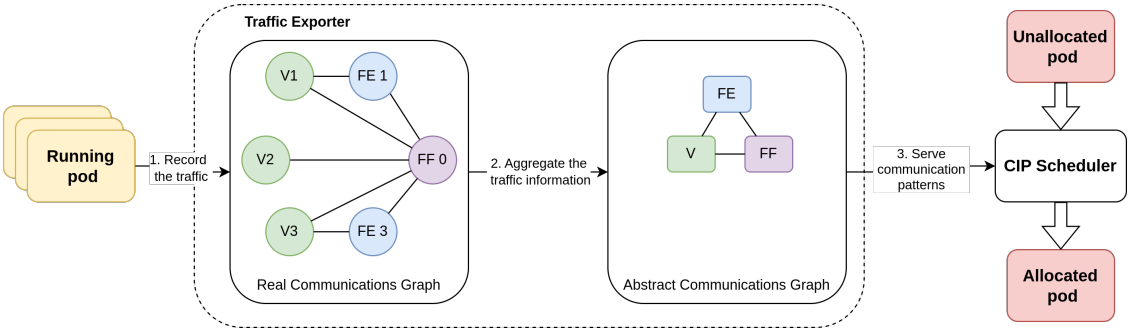


Figure 4.4.: Illustration of the CIP scheduler mechanism. The node colors represent the types.

Rescheduler

The Rescheduler is a Go application running in a dedicated pod. We use the Kubernetes and Prometheus libraries to collect all the necessary information. The traffic estimation is using linear regression. Future work can investigate more complex prediction methods to address different workloads.

To migrate a pod, the rescheduler updates its deployment manifest. This automatically triggers a rolling update with no downtime to the service. A new pod is deployed on the best node, and the old pod is deleted when it is up.

The duration of the rescheduling routine depends on the time needed to evaluate each pod configuration and the time needed to complete all pod migrations. Although we can manually set the rescheduling frequency, how often the rescheduler calls the rescheduling routine. This frequency can be low if the cluster state does not evolve quickly or higher if necessary. However, the rescheduling period cannot be lower than the duration of the rescheduling routine.

Monitoring setup

We use open-source tools to collect data about the cluster and the workload state and make it available for scheduling decisions.

Figure 4.5 presents the monitoring workflow and how data are collected and made available for scheduling decisions. This methodology can support any containerized workload.

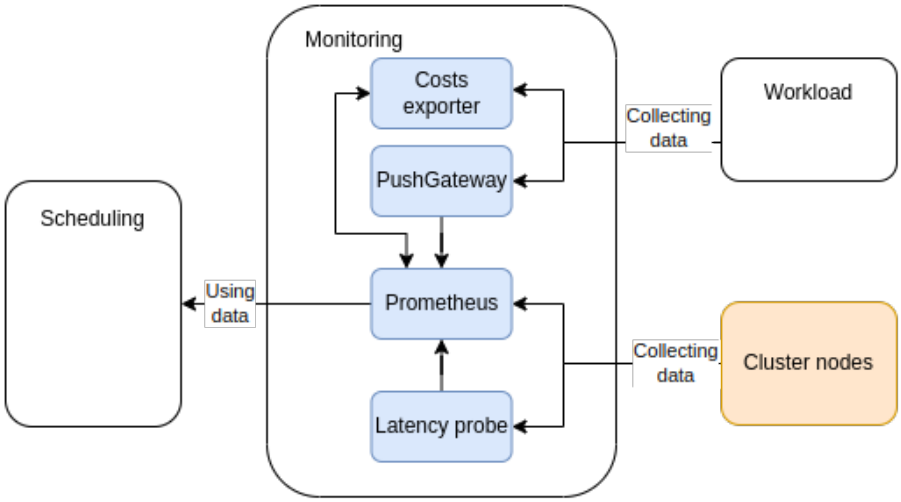


Figure 4.5.: Monitoring architecture

Prometheus [77] is the main monitoring component. It periodically pulls many metrics about the cluster state (about the nodes, the pods, and the containers). Then, this data can be requested from the scheduling modules or a monitoring dashboard.

In addition to the standard Prometheus metrics, we collect specific metrics about our workload and the network state. We can expose Key Performance Indicators (KPIs) such as the End-to-End latency. To ensure that the metrics are collected even if the pod is killed, we use the Prometheus PushGateway [125]. Data is pushed from the pods of the workload to this component. Then, Prometheus can periodically pull the PushGateway like any other module.

We have built a module: *CostExporter* to compute and export the costs. It is a Go application that exposes metrics that are accessible through Prometheus. It exports the costs of using a server and networking traffic.

The cluster latency matrix (i.e., a matrix that summarizes the latency between every node in the cluster) is a key metric in our methodology. To collect this metric, we are using a DaemonSet (i.e., a Kubernetes tool that ensures that every node in the cluster runs a copy of a pod) that deploys a pod that pings all other nodes in the cluster. These latency probes [126] make the latency matrix directly available through Prometheus.

This monitoring system is designed to have limited overhead on the workload execution. Most of the monitoring services are running outside of the workload application. Cluster state sensors run in dedicated pods. Communication probes run in sidecar containers[3]. A sidecar container has access to the same network as the containers of the workload application. It can export networking metrics to Prometheus. However, the KPIs probes run inside the workload application. A dedicated thread (to avoid interruptions) pushes the KPIs to the Prometheus push gateway.

4.3. Evaluation

To demonstrate the potential and effectiveness of our approaches, we have devised a workload that mimics the computational need and communication of a real vehicular cooperative perception application. This workload is a distributed application consisting of multiple services that are deployed in a real Kubernetes cluster in a public cloud. We control the latencies between nodes to emulate a cloud-edge continuum infrastructure. We then conducted experiments to evaluate the latency-based initial placement (LIP), communication-based initial placement (CIP), and rescheduling (RS) algorithms in a realistic environment.

This section first presents our preliminary results. It is a simulation we ran to evaluate the potential of our approach before its implementation on Kubernetes. Then, we describe the first set of experiments to evaluate the parameters of the LIP method. Finally, we compare all scheduling methods to some baselines and evaluate the limitations and overhead of these methods.

4.3.1. Preliminary results

We first build a proof of concept of our scheduling methodology before implementing it on a Kubernetes cluster with a realistic workload. We start experimenting with our scheduling methodology through simulation to understand its potential and the room for improvement.

Our service initial placement method and our rescheduler can help reduce the total costs of running an application in the edge to data center Computing Continuum. We can illustrate how to perform cost optimization with the following example. We have a car connected to a video processing service that detects other vehicles. There are three main configurations for placing this service: on a data center node (paying the data center instance price and the communication), on the edge node close to the car (paying

[3]a container running in the same pod as an application

only the instance price), or on a different edge node (paying the edge node instance price and the communication). Video processing is a bandwidth-intensive application; choosing the edge node with no communication costs (because the data stays local) is cheaper. We need the service initial placement method to find a trade-off between paying instance costs and communication costs. However, when the car is moving, the previously selected edge node is no longer optimal; it stops processing local data. We need a background process (rescheduler) to watch the cluster state and migrate the services when necessary; e.g., when a car is moving, we need to move the service to the edge node close to the car.

We designed a simulation experiment using our scheduling methodology to understand the room for improvement of distributed applications in a Cloud-to-Edge Computing Continuum. This simulation demonstrates the benefits of our scheduling methodology on a larger cluster (around 50 nodes).

The simulator is a C++ program that can generate random graphs with data center and edge nodes. We adapt the Erdős–Rényi model [127] to generate graphs with a similar topology to the Computing Continuum (e.g., interconnected data-center nodes and some edge nodes). We use a synthetic workload, a simple application with an end-user client located at the edge (e.g., a car) communicating with a server (e.g., a video processing service). The server service can be deployed anywhere in the simulated Edge-to-Cloud Computing Continuum. The clients randomly move between different edge nodes, and new clients randomly arrive, following a Poisson distribution.

We ran an experiment with graphs containing 20 data center nodes and 30 edge nodes on which we deployed 10 to 20 services. In this scenario, we consider the edge nodes to be more expensive (i.e., higher price per CPU and per hour) due to their limited resources. We are comparing three different approaches in this simulation: i) a greedy approach (choosing the cheapest node at each time step), ii) a latency-based initial placement (LIP) described in section 4.2.2 iii) the latency-based initial placement, and the rescheduler (LIP+RS) described in section 4.2.3. The LIP approach minimizes the latency between the client and the server.

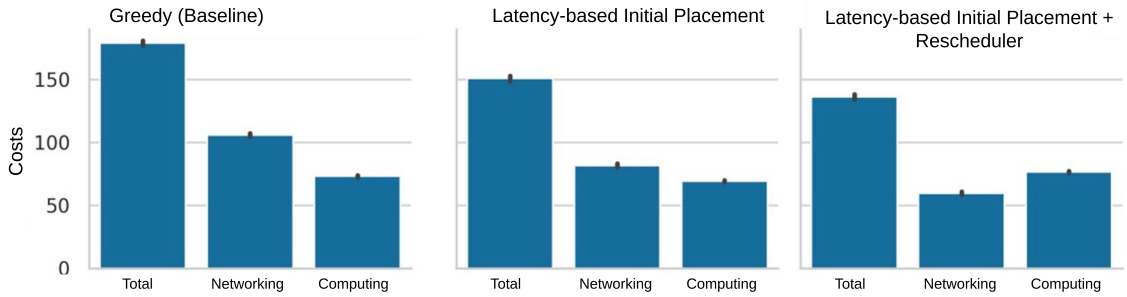


Figure 4.6.: Results of the simulation - Costs are detailed in two parts: computing and networking

Figure 4.6 presents the cost of running the workload for different approaches. Given that the simulation is a randomized process, we ran each experiment ten times and

reported the average. The lower the cost, the better. The latency-based initial placement with or without the rescheduler outperforms the greedy strategy. Greedy is taking the best decision at each time step; however, this does not lead to a globally optimal solution. Using our methodology, we can find a better global solution. The rescheduler chooses more expensive nodes to save data transfer costs, which explains why it gets better results than the initial placement approach alone.

This simulation shows that our methodology outperforms a greedy strategy on a large graph (50 nodes) with a simple workload. Using servers at the edge to save networking costs can lower the costs of running applications. This encouraged us to implement a real solution suitable for Kubernetes integration.

4.3.2. Experimental Setup

We build an experimental cluster using public cloud resources. We use Virtual Machines (VMs) to have cluster nodes with different characteristics (e.g., different number of processors and amount of memory), and we add artificial delays between them to simulate the physical distances between the nodes. The details on how this is done can be found in chapter 3. We deploy a Kubernetes cluster using all these nodes. In this infrastructure, only the latencies are emulated; we deploy the Kubernetes cluster on real nodes.

Figure 4.7 shows the infrastructure graph of the experimental cluster with all nodes and latencies between them. Our experimental cluster includes one node for the control plane and one for the monitoring tools. A workload cannot be deployed on these two nodes. The edge and data center (DC) nodes host the workload services. End-user nodes host the end-user application workload. In a real-life environment, end-user nodes would be replaced by dedicated equipment such as a smartphone or a car.

The physical distances between the edge nodes induce networking delays. These delays are not represented as direct links on the graph to improve readability. The graph summarizes the delays as they are experienced by the different nodes. E.g., there is a delay of 25ms between the nodes E1 and E3.

The difference between edge and DC nodes is the geographical location and the available resources. The nodes are AWS instances. The edge nodes have 4 CPUs and 16 GiB of memory. Other nodes have 8 CPUs and 32 GiB of memory.

4.3.3. Workload: Vehicular cooperative perception

The vehicular cooperative perception workload leverages computer vision to help detect nearby vehicles. As described by Xu et al. in [128], vehicles are sharing videos they record with their cameras to improve global knowledge of the positions of all nearby vehicles. Knowing the positions of close vehicles is helpful for drivers who cannot see others in their blind spots. Also, this technology is important for self-driving vehicle implementation, where perception is a major challenge. Getting accurate positions of surrounding vehicles helps to reduce the collision risk.

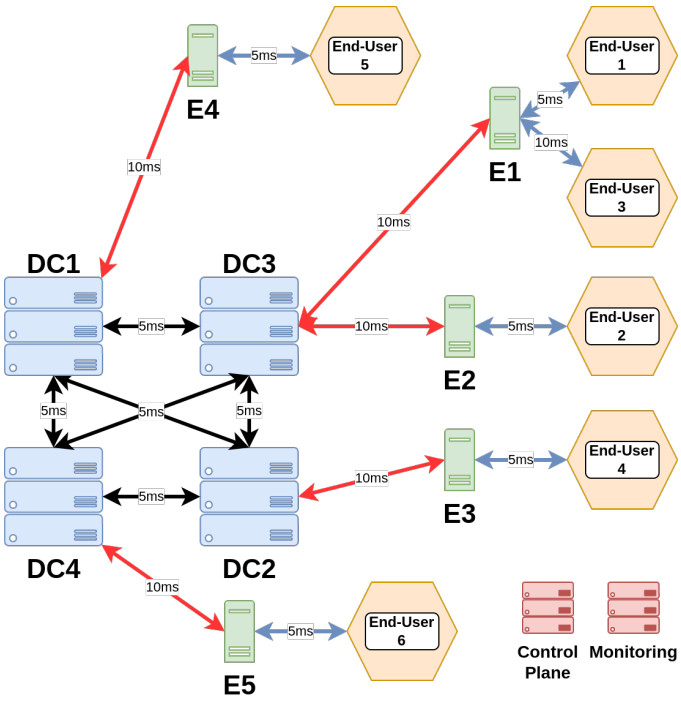


Figure 4.7.: Experimental cluster infrastructure graph (not all physical links between nodes are represented)

We implement a synthetic workload that mimics the behavior of the above-described application. The originally described application is a monolith. We break it into three services that we can deploy anywhere in the Computing Continuum. By splitting the application, we can include vehicles with limited computing resources; the services that cannot run on these vehicles can be hosted on a server at the edge or in a data center. Also, we can place the component that aggregates the data from multiple vehicles in a central location.

Three services compose the workload application. The *vehicles* generate a video stream and send it to a *feature extractor* (FE). The FE extracts the features from the video stream and sends it to a *feature fusion* (FF). FF merges the features to get accurate positions of the vehicles. Finally, the FF broadcasts the positions to nearby vehicles.

Figure 4.8 illustrates the structure and functioning of the application. This application aims to combine multiple video streams to get the accurate positions of vehicles. First, the video streams from the embedded cameras are processed to extract features. Then, features are combined to estimate the positions of the vehicles. Finally, the positions are broadcasted to nearby vehicles.

In our experiments, we have two kinds of vehicles: the ones with embedded computing capabilities (e.g., GPU) and the ones without them. The vehicles without computing capabilities send video-stream to the Feature-Extractor. The vehicle with computing capabilities sends features (already extracted) directly to the Feature-Fusion.

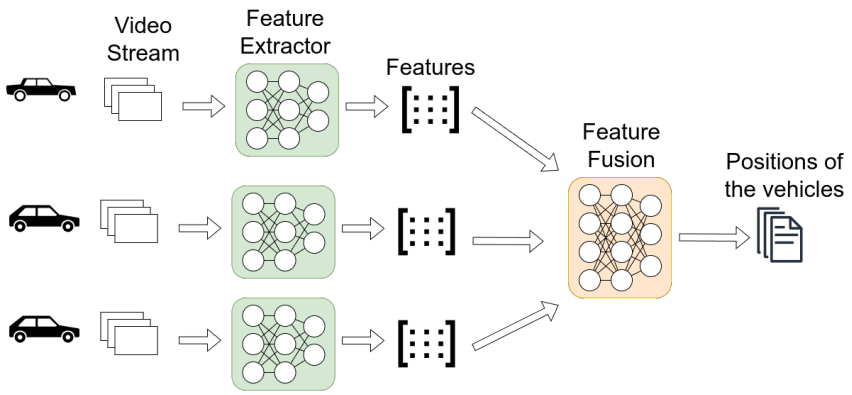


Figure 4.8.: Vehicular Cooperative Perception workload illustration

Figure 4.9 shows the interactions between the Vehicle, Feature-Extractor (FE), and Feature-Fusion (FF) services. There are three different kinds of nodes in our experiment: Data-Centre (DC), Edge (E), and End-User (EU). End-user nodes are hosting the vehicles only. The vehicles cannot be deployed on different nodes. Vehicles without GPU are sending a video stream to the FE. Vehicles with GPU are sending the features directly to the FF. FE and FF services can run only on Edge or DC nodes.

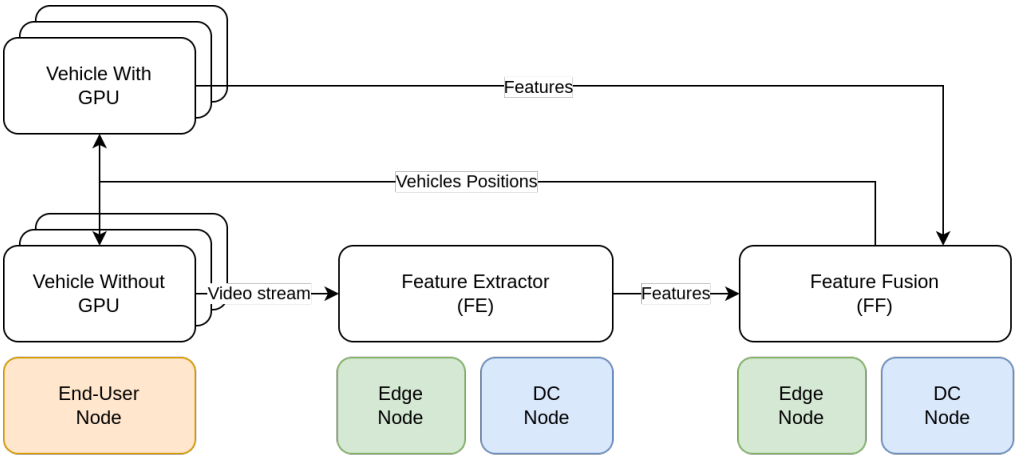


Figure 4.9.: Workload architecture: Workload pods and the nodes where they can run

There is only one instance of FF for the whole experiment, it aggregates the features from all of the vehicles. There are many vehicles, and each of them without a GPU is connected to one FE. We suppose that the vehicles are connected to the same network as the other services. However, the network delays depend on the geographic location of the vehicles.

For this workload, an end-user node represents a neighborhood or a 5G cell area where vehicles can go. In the experiment, vehicles are moving from one end-user node to another over time. A *json* configuration file defines all the movements of the vehicles

when they are starting up or shutting down when they are moving from one area to another.

We benchmark the original application to build our synthetic workload. Table 4.1 summarizes the parameters we use to configure our application. This application is using CPU instead of GPU. Future work may adapt this scheduling approach to consider accelerators such as GPU or FPGA. In this synthetic application, the CPUs are running a load generated by the *stress-ng* tool [129]. Services send randomly generated data over the network using the sizes defined in Table 4.1.

Table 4.1.: Cooperative perception workload characteristics

Parameter	**Value**
Processing time (FF)	100 μs
Processing time (FE)	100 μs
Frames size	731kB
Features size	64 kB
Frame rate	35 FPS

4.3.4. Results

In this section, we evaluate our orchestration methodology on a Kubernetes cluster. We present the costs and the end-to-end (E2E) latency of the vehicular cooperative perception workload when using the Kubernetes default scheduler and our methodology to place the services.

We first evaluate the LIP approach separately. The LIP algorithm has two heuristic parameters: α and β. We run a first set of experiments to understand how these parameters change the output of the experiments. We also run experiments to see how cost changes affect the decisions of the LIP approach. Then, we evaluate and compare all our approaches. Finally, we discuss the performances of the CIP approach when using initial data or not, and the CPU consumption overhead of running our scheduling components.

LIP approach evaluation

In this section, we evaluate the LIP approach. We use a cluster with three data centers and three edge nodes for these experiments. We define a scenario in which two vehicles embed a GPU (to extract the features locally), and three vehicles use a feature-extractor (FE) deployed in the Computing Continuum. There is one feature-fusion (FF) instance for all the vehicles.

We evaluate this scenario using three scheduling approaches. *Baseline*: the default Kubernetes scheduler. *Latency-based Initial Placement* (LIP): the initial placement algorithm described in section 4.2.2. *LIP + Rescheduler* (LIP+RS): the initial placement algorithm in addition to the rescheduling methodology described in section 4.2.3.

For each approach, we are testing different cluster configurations. Table 4.2 summarizes the experimental parameters and defines the different *experiments*. Experiments 1 to 3 test different α parameters, and experiments 4 and 5 explore different node costs. Since the experiments are run on real machines, there is significant variability between each execution. Therefore, we repeat each experiment ten times and use the average value in the figures here.

Table 4.2.: Cooperative perception workload cluster parameters

Experiment	**DC cost** (per CPU)	**Edge Cost** (per CPU)	**Network cost** (per GB)	α (ms)	β
1	0.0472	0.0472	0.01	20	1000
2	0.0472	0.0472	0.01	15	1000
3	0.0472	0.0472	0.01	40	1000
4	0.0472	0.0944	0.01	20	1000
5	0.0944	0.0472	0.01	20	1000

For every *experiment*, we measure the total costs of running the workload. The total costs are the sum of the computing and networking costs.

In addition to the costs, we also record the end-to-end latency (E2E latency) of each experiment. This is a key performance indicator (KPI) to check if the quality of service varies when performing cost optimization. The E2E latency is the duration between the time when a frame is recorded and the time when the vehicle receives the corresponding positions of the nearby vehicles. This value aggregates the network delays between each service, the duration required to send the video stream and the features, the processing time for extracting the features, the time to fusion the features, and the time to broadcast the positions.

Figure 4.10 presents the average of the total costs. The total costs are normalized to the baseline approach. Error bars represent the 95% confidence interval. The lower the costs, the better.

Figure 4.11 shows the 95^{th} percentile of the end-to-end latency for the different experiments. We can use this figure to ensure that the Quality of Service remains at the same level.

By analyzing the node allocation of the services over time, we observe that the baseline approach is choosing data center nodes in most situations. Data center nodes have more computing resources; they are the least allocated nodes. The LIP and the LIP+RS approaches use both edge and data center nodes (it depends on the cluster configuration).

Experiments 1 to 3 use different values of α. The LIP and the LIP+RS approaches show lower costs than the baseline approach. The LIP+RS provides an improvement of around 20%. According to the confidence intervals, the LIP+RS costs are significantly lower than the baseline approach. In these three experiments, the computing cost is the same for all kinds of nodes. Therefore, computing costs are the same for every

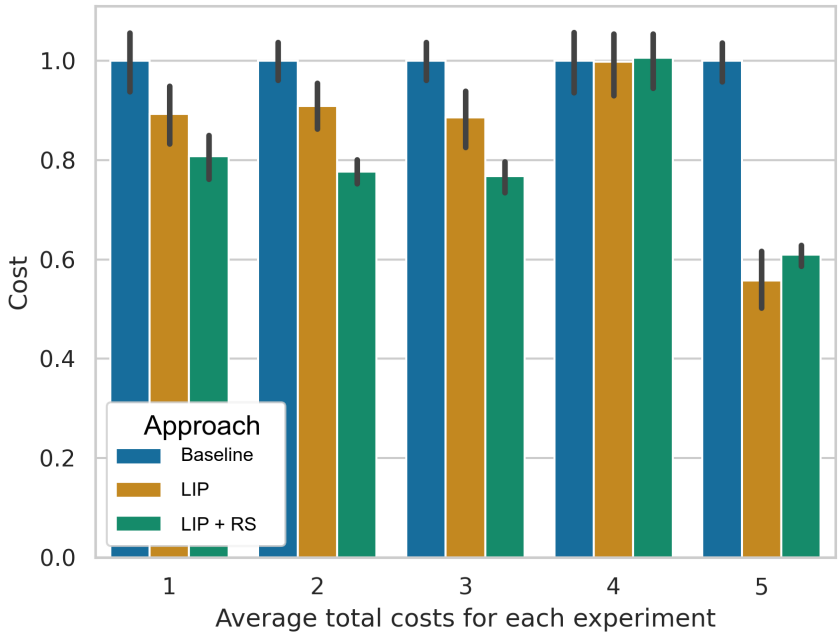


Figure 4.10.: Average of total costs for each approach: Baseline, Latency-based Initial Placement, and LIP + Rescheduler. Costs are normalized to the baseline approach

approach. The total costs can be lower only if the networking costs are lower because the computing costs are constant. The LIP and the LIP+RS get lower total costs because they managed to get smaller networking costs.

In experiment 4, the edge nodes are twice as expensive as the data center nodes. By analyzing the allocation of the services, we observe that most of the services are deployed in data centers. Data center nodes are the best solution in this configuration. Since the three approaches all choose data center nodes, the results are very similar.

In experiment 5, the data center nodes are twice as expensive as the edge nodes. When using our methodology, the services are mostly placed at the edge. The LIP and the LIP+RS achieve an improvement of around 40% compared to the baseline approach. They are significantly lower.

For each experiment, the E2E latency is similar for every approach. Even if our methodology lowers costs, the E2E latency is not impacted, and this KPI remains the same. However, the LIP approach has higher costs due to the huge optimization in experiment 5. This approach chooses only cheaper edge nodes with low latency, but the vehicles are moving, and the services stay at the same location. This location at the edge is too specific and provides higher E2E latency. Using a central location like a data center may help to avoid this issue. To benefit from the cost optimization of using edge

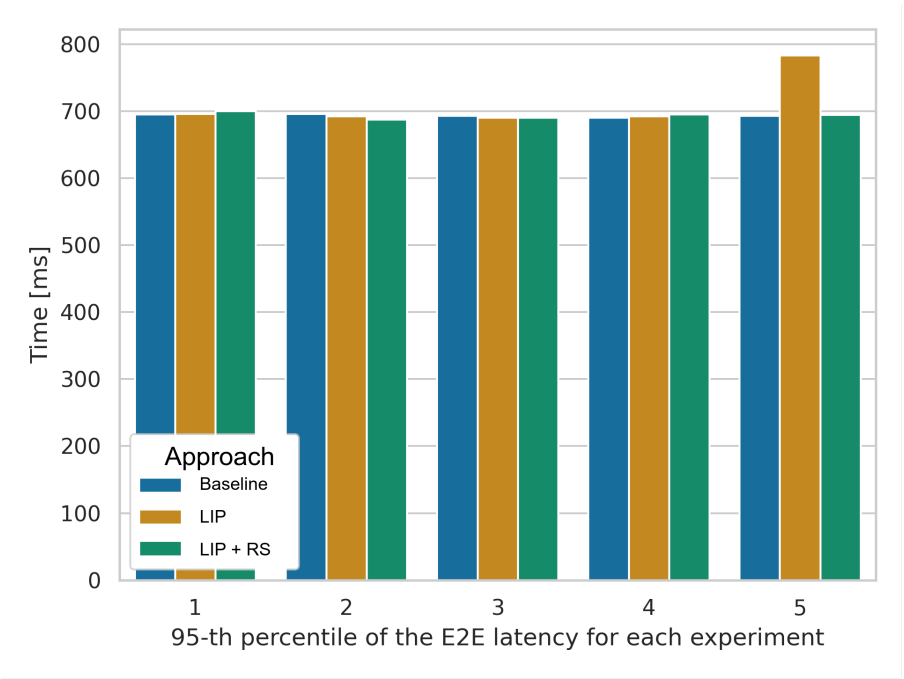


Figure 4.11.: 95^{th} percentile of the end-to-end latency for each approach: Baseline, Latency-based Initial Placement, and LIP + Rescheduler.

nodes, we need to use the rescheduler to keep the same quality of service.

Computing costs are the same for each approach; this is the expected behavior, as all instances have the same cost. Networking costs are lower for the LIP and the LIP+RS approaches, which leads to lower total costs. The LIP+RS approach performs better than the LIP approach.

The results show that we can have lower costs when running the same services with the same end-to-end latency. Our scheduling methodology can lower the costs of deployment of the Edge-to-Cloud Computing Continuum. However, to ensure lower costs, it is better to use the rescheduler in addition to our initial placement approach. α and β should be chosen respectively to the inter-node latencies and the node compute costs.

All approaches evaluation

This section evaluates the scheduling approaches presented in section 4.2.

We use a cluster with four data centers and five edge nodes for these experiments. We define a scenario in which five vehicles embed a GPU (to extract the features locally) and use a feature-extractor (FE) deployed in the Computing Continuum. There is one feature-fusion (FF) instance for all the vehicles.

We evaluate this scenario using seven scheduling approaches. *Baseline*: the default Kubernetes scheduler (least allocated node). *Most Allocated node* (MA). *Balanced allocation*

(BA): a Kubernetes approach to balance the resource usage among the nodes [130]. *Latency-based Initial Placement* (LIP): the initial placement algorithm described in section 4.2.2. *LIP + Rescheduler* (LIP+RS): the initial placement algorithm in addition to the rescheduling methodology described in section 4.2.3. *Communication-based Initial Placement* (CIP): the communication-based initial placement algorithm described in section 4.2.2. *CIP + Rescheduler* (CIP+RS): the initial placement algorithm in addition to the rescheduling methodology described in section 4.2.3.

We use three different cluster configurations for each approach: i) all nodes have the same instance cost, ii) data-center nodes are twice as expensive as the edge nodes, and iii) edge nodes are twice as expensive as the data-center nodes. The instance cost is somewhat arbitrarily chosen to 0.0472 per CPU.h, and the communication cost to 0.01 per GB. The actual values are not so important for the performance evaluation, but they are still chosen to be in the realistic realm. Since the experiments are run on real machines, there is significant variability between each execution. Therefore, we repeat each experiment ten times and use the average value in the figures here with error bars representing 95% confidence interval.

For the LIP approach, we set $\alpha = 20$ and $\beta = 1000$. We set α in relation to the observed latencies in the cluster. It is lower than the average delay between two edge nodes and higher than the average delay between a DC and an edge node. If the α value is too small, the condition with α will never be satisfied, and the LIP will select the node with the lowest instance cost. Respectively, if α is too big, the condition will always be satisfied, and the LIP will choose the cheapest node. We set β in relation to the instance costs in the cluster. It is significantly bigger than the inverse of the average instance cost. That way, LIP first ranks the nodes by the number of α constraints satisfied and then selects the cheapest node among those that satisfy the maximum number of constraints.

For every *experiment*, we measure the total costs of running the workload. The total costs are the sum of the computing, networking, and rescheduling costs. The rescheduling cost is the price to pay to migrate a pod from one node to a different one.

In addition to the costs, we also record the end-to-end latency (E2E latency) of each experiment. This is a key performance indicator (KPI) to check if the Quality of Service (QoS) varies when performing cost optimization. The E2E latency is the duration between the time when a frame is recorded and the time when the vehicle receives the corresponding positions of the nearby vehicles. This value aggregates the network delays between each service, the duration required to send the video stream and the features, the processing time for extracting the features, the time to fusion the features, and the time to broadcast the positions.

Figure 4.12, 4.13, and 4.14 presents the average of the total costs for different scenarios: same instance cost for all nodes, data center nodes twice as expensive as edge nodes and edge nodes twice as expensive as data center nodes. The total costs are normalized to the optimal approach. Error bars represent the 95% confidence interval. The lower the costs, the better.

Figure 4.15 shows the 95^{th} percentile of the end-to-end latency for the different

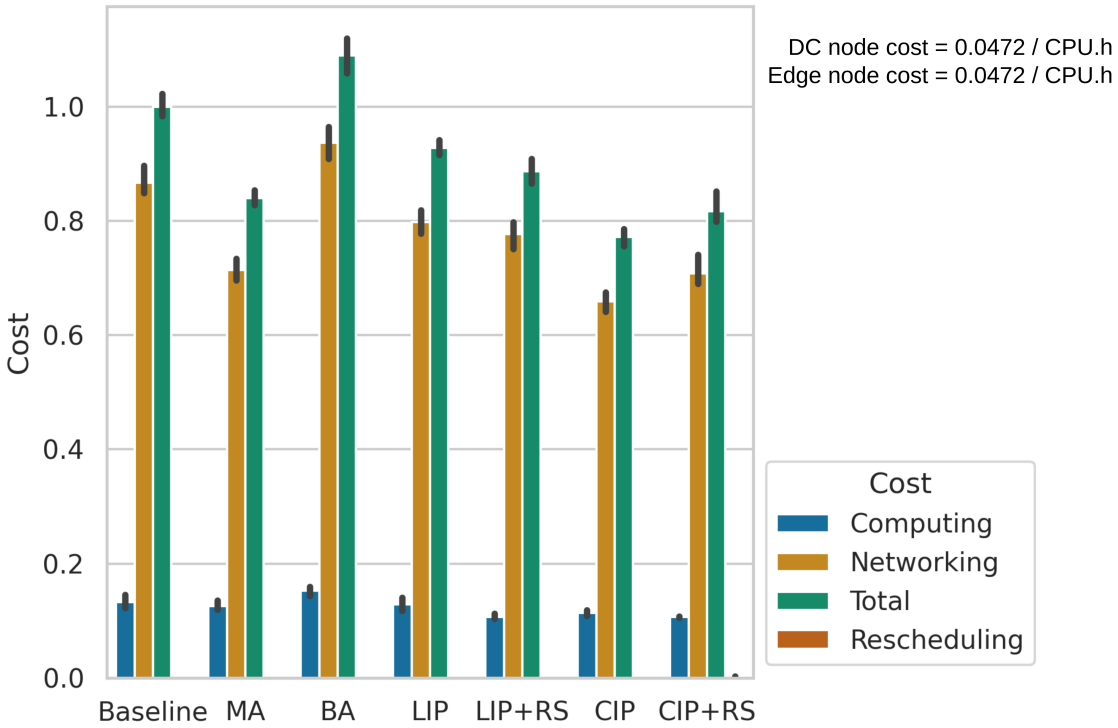


Figure 4.12.: Average of total costs for each approach: Baseline (default Kubernetes, Least Allocated), Most Allocated, Balanced Allocation, Latency-based Initial Placement, LIP + Rescheduler, Communication-based Initial Placement, and CIP + Rescheduler. Costs are normalized to the baseline approach. Data-center and edge nodes share the same price.

experiments. We can use this information to ensure that the Quality of Service remains the same.

By analyzing the node allocation of the services over time for all scenarios, we observe that the baseline approach is choosing data center nodes in most situations. Data center nodes have more computing resources; they are the least allocated nodes. The LIP, CIP, CIP+RS, and CIP+RS approaches use both edge and data center nodes (it depends on the cluster configuration).

For the first scenario where all nodes have the same instance cost, our approaches have total costs that are 7 to 23% lower than the baseline (the default Kubernetes scheduler). The optimal approach is 30% lower than the baseline approach. Our best approach is CIP; its total cost is 8% above the optimal. In this scenario, the instance costs are the same for every approach. It is the expected behavior; the node price is the same for every node. Only lower networking costs lead to lower total costs. Rescheduling costs are negligible in this experiment. They correspond to the price to pay to run the two pods during the migration; the old pod is killed only when the new one is running.

For the second scenario, where the data center nodes are twice as expensive as edge

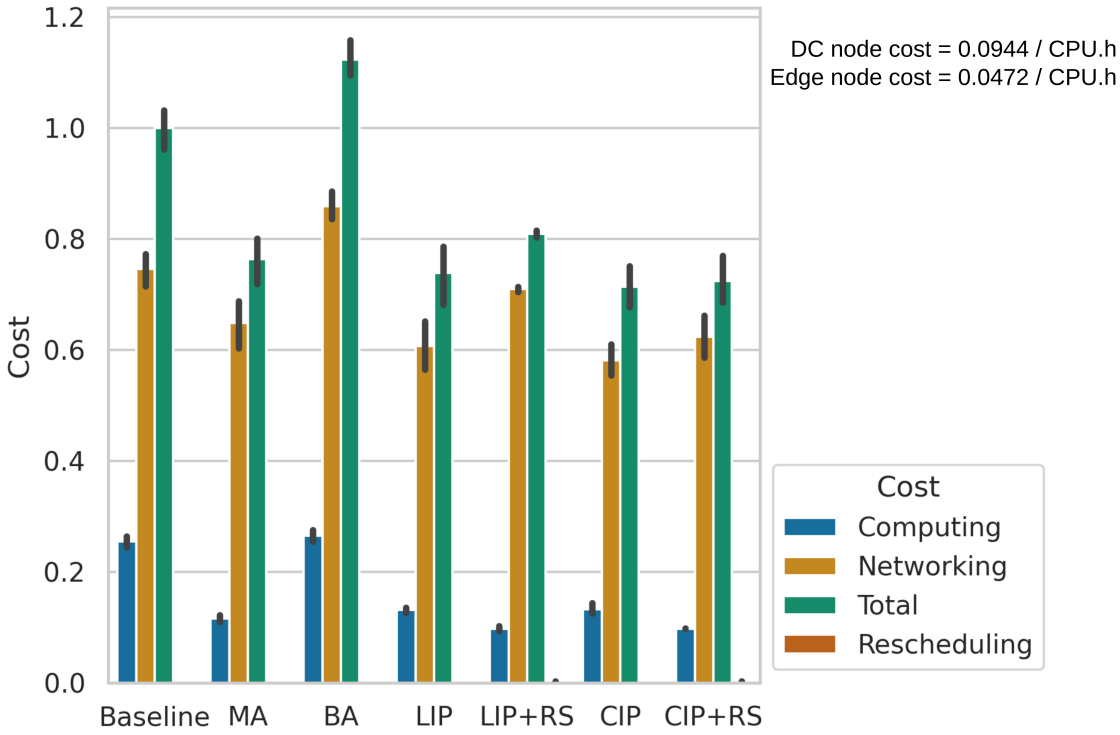


Figure 4.13.: Average of total costs for each approach: Baseline (default Kubernetes, Least Allocated), Most Allocated, Balanced Allocation, Latency-based Initial Placement, LIP + Rescheduler, Communication-based Initial Placement, and CIP + Rescheduler. Costs are normalized to the baseline approach. Data-center nodes are twice as expensive as edge nodes.

nodes, our approaches have a total cost of 19 to 29% lower than the baseline. The baseline chooses the least allocated nodes, which are the data center nodes. The baseline chooses only expensive nodes in this scenario, which leaves a lot of room for improvement. CIP and CIP+RS have similar performances using two different placement strategies. In this situation, the CIP approach uses more DC nodes (that are more expensive) than the CIP+RS. The rescheduler is using more edge nodes to save both computing and networking costs. However, the un-optimal rescheduling routine duration (described in section 4.4) leads to higher networking costs. CIP uses DC nodes for the feature fusion; it is more expensive than CIP+RS, which uses Edge nodes.

In the last scenario, where edge nodes are twice as expensive as the data center nodes, there is less room for improvement. The baseline chooses data center nodes because they are the least allocated nodes, and it gets results similar to those of the LIP and LIP+RS approaches. However, the CIP approach manages to find 9% lower total costs. But, these cost improvements imply an impact on the end-to-end latency. CIP+RS performs better than the baseline with a limited impact on the QoS.

The Balanced Allocation approach is not appropriate for the Edge-to-Cloud Comput-

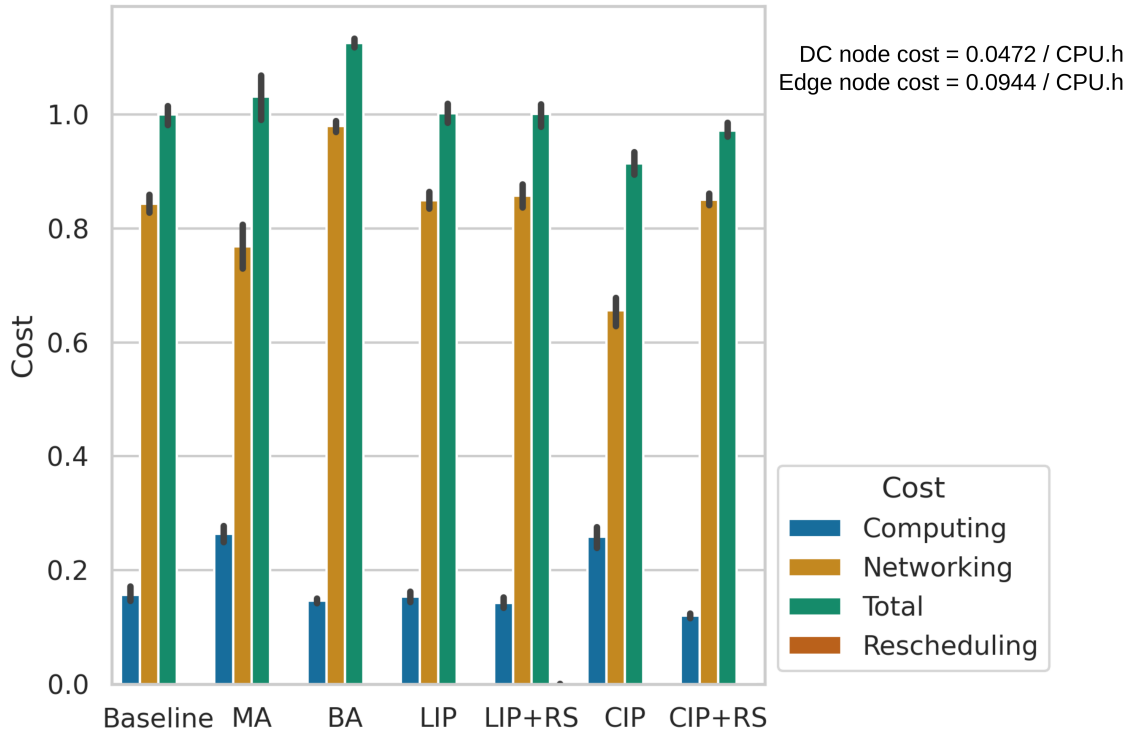


Figure 4.14.: Average of total costs for each approach: Baseline (default Kubernetes, Least Allocated), Most Allocated, Balanced Allocation, Latency-based Initial Placement, LIP + Rescheduler, Communication-based Initial Placement, and CIP + Rescheduler. Costs are normalized to the baseline approach. Edge nodes are twice as expensive as data-center nodes.

ing Continuum. It provides higher total costs and lower QoS than the baseline in every scenario. Using Balanced Allocation on heterogeneous resources is inefficient.

The Most Allocated node approach can have lower total costs than the baseline in two scenarios. The QoS of the MA approach is lower than the baseline in every scenario. The major issue of this approach is that it does not adapt to different cluster configurations. It always chooses the edge nodes (which are the most allocated nodes), even if they are not close to the end users. A first edge node is picked randomly and then filled at its maximum capacity, then the next one is selected. If the first edge node picked is close to the end users, the overall results will be better than the baseline. This approach will be even less performant if we add more edge nodes in the cluster. This method has a lot of variability due to randomly picked edge nodes. It makes that approach less reliable than our methodology.

The End-to-End latency is similar for most of the approaches, except for the above-mentioned case. Overall cost improvement does not negatively impact the Quality of Service.

The CIP approach is the best of our approaches to minimizing the total costs. However,

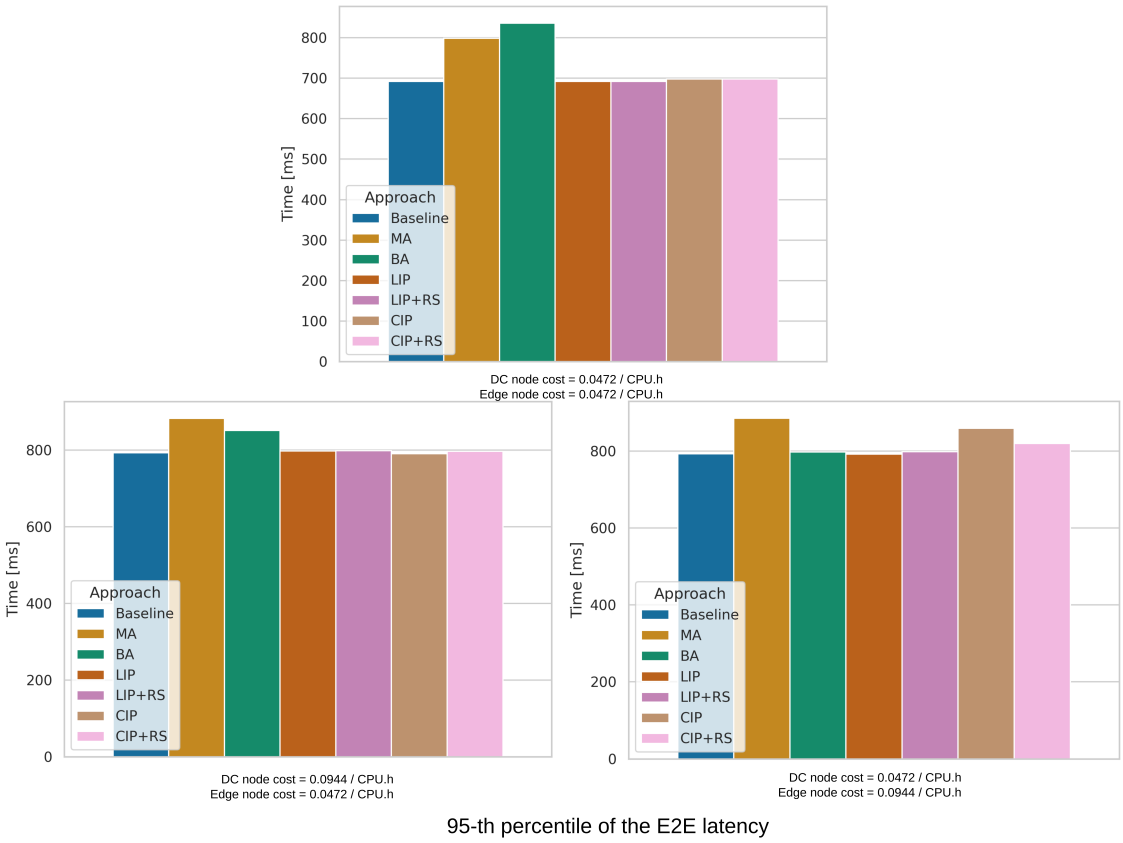


Figure 4.15.: 95^{th} percentile of the end-to-end latency for each approach (ms)

there is one scenario (when edge nodes are twice as expensive as data-center nodes) where the QoS is impacted by cost optimization. The LIP approach performs less aggressive cost optimizations, and results are lower than CIP but never negatively impacts the QoS. The rescheduler helps reduce the costs or improve the QoS. But, to propose the best performances possible, the rescheduling routine needs to be shorter than the systems evolve.

CIP performances and initial data

The performance of the CIP scheduler varies according to the available data. To quantify this variation, we run an experiment where we use the CIP with and without initial data. Table 4.3 presents the normalized total costs for different approaches. When the CIP scheduler can access initial data, the total cost is 12% lower than it is without initial data. However, the CIP approach outperforms the LIP approach in all cases, with and without initial data.

The CIP approach performs better when it already has data about the application it is deploying.

Table 4.3.: Normalized total costs for different approaches

Approach	**Optimal**	**LIP**	**CIP** (without initial data)	**CIP** (with initial data)
Normalized cost	1	1.32	1.22	1.10

Custom scheduler and rescheduler overhead

We measure the overhead of using our custom scheduling methodologies. It is important to ensure that the cost of running our approaches is lower than the savings they can achieve.

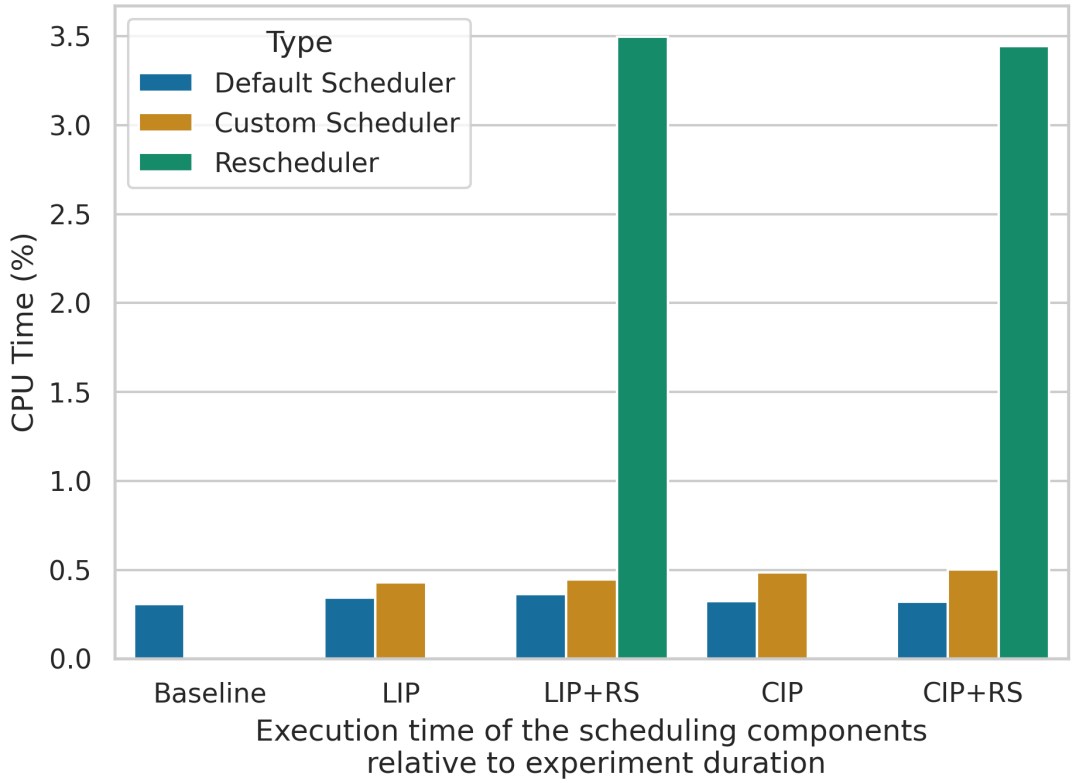


Figure 4.16.: CPU time overhead of the scheduling components

Figure 4.16 presents the execution time of the scheduling components relative to the experiment duration for each approach studied. Using our scheduling approaches has a limited impact on CPU consumption. The CPU time of the additional scheduling components is below 4% of the experiment duration. The scheduling components are idle most of the time. Our custom schedulers consume around 1.5 times more CPU than the default Kubernetes scheduler.

Overall, saving 10 to 25% of the costs of running the experiment is worth the expense

of 4% of a CPU core for the duration of the experiment.

It is worth noting that the scheduling overhead does not impact any applications in any negative way as it is executed on a separate node.

4.4. Limitations

This section presents the main limitations of our scheduling methodologies and their evaluation.

Workload

In this set of experiments, we consider that Edge and Cloud have the same processing speed (but with different numbers of CPUs). The workload consists of handling a video stream in real-time; more computing capabilities cannot make it faster, but it can improve accuracy. Further studies may investigate different framerate and resolutions, or test other workloads.

Rescheduling

The rescheduling routine sometimes is too long for optimal performance. It could be counter-intuitive that a static approach (CIP) outperforms the dynamic one (CIP+RS). The explanation is that the system evolves faster than the rescheduler can react.

For example, when a pod is placed on node E3 to be close to a vehicle located on E-U4, this solution is optimal at this given time. Nevertheless, if the vehicle moves to E-U6, the solution is no longer optimal. If the rescheduler takes a minute to move the pod to E5, the pod stays in an inadequate location for one minute, and the costs grow higher very fast. If the rescheduler cannot react fast enough, it can be better to have a static approach without a rescheduler or to improve the reactivity of the rescheduler. To avoid these situations, the period of rescheduling should be significantly lower compared to the changes that occur in the clusters (vehicle movement or new node availability).

The minimum duration of the rescheduling routine is fixed by execution and migration time. Section 4.2.4 provides more details about the rescheduling duration. Future works may investigate ways to improve the rescheduling speed.

Quality of Service

Even if our proposed methodology has similar or better Quality of Service (QoS) than the baseline, there is no mechanism to ensure that Service Level Objectives (SLOs) are met.

The LIP approach minimizes inter-service latency to lower communication costs. It reduces the end-to-end latency and improves the QoS. However, there are no hard requirements to avoid choosing nodes with high latency if it is the only choice possible.

The CIP approach minimizes the total costs. If the nodes of the cluster have similar prices, this approach will minimize the data traffic. Otherwise, it will choose a trade-off between the instance and the communication costs. Reducing data traffic by processing data locally reduces the end-to-end latency, and improves the QoS.

The CIP approach can reach lower total costs than the LIP approach, even if the QoS is lower. The LIP approach performs less aggressive optimizations, it can get higher QoS.

Future work can investigate adding a node filter in the scheduler and the rescheduler to remove the nodes that do not meet SLOs from the list of schedulable nodes.

4.5. Related work

In this section, we present some research related to the scheduling of services in the Edge-to-Cloud continuum. In this study, we define the scheduling of services as the mapping of a service to a server where it can run. We organize the different scheduling methodologies into three categories: cost-aware, network-aware, and QoS-aware scheduling.

Cost-aware scheduling Lai et al. present a cost-aware scheduler in [131]. They use a heuristic approach (most capacity first) to maximize the number of allocated edge users while minimizing the number of necessary servers at the edge. This work has no mechanism to move applications when end-users are moving. In addition, scheduling on edge nodes is outside of the scope of this study.

The authors of articles: [132, 133] present approaches to reduce costs by improving the Kubernetes scheduler. However, their main interest is in cloud computing and cannot be extended over the whole continuum without additional work. Li et al. [132] present a meta-heuristic-based scheduler that minimizes the energy costs of CPU, RAM, and network usage in addition to the networking costs of offsite nodes. They also propose a rescheduler to monitor changes in business requirements. This study only focuses on the cloud environment, where computing resources and resources are mostly homogeneous (e.g., same kind of server hardware, same latency between the nodes). Also, the scheduling decision does not consider any latency requirements. Zhong et al. [133] propose a scheduling methodology that reduces the number of allocated Virtual Machines when using the Kubernetes Autoscaler to lower the instance costs. To save costs, they propose to use a background process that checks if it is possible to shut down a server and migrate its pods to another node. They try to maximize resource utilization to save costs. This approach would have a limited impact on the scope of edge computing; node geographical location is an important parameter to consider in order to lower latency or reduce backhaul networking costs. In addition, this work does not consider the costs of the traffic going outside of data centers. Outgoing traffic is expensive when a network-intensive application is not deployed in the same location as its end users.

Network-aware scheduling Kaur et al. [134] present a scheduling algorithm that minimizes inter-service communication delays. It relies on two heuristic approaches: a greedy and a genetic algorithm. This study makes the assumption that data traffic between the services of the application is known. It is not the case in practice with most applications. Additional work would be required to specify or learn the data traffic pattern. This limitation makes their approach difficult to use with real applications. Also, this study does not use any background process to monitor the movements of the end-users at the edge. Marchese et al. have proposed using a rescheduling mechanism [135, 136]. In [135], they present a network-aware scheduler plugin and a descheduler that checks if a node with a better score can be found. The proposed scoring method is not effective for initial placement. It relies on the input from previous data traffic that is null at the initialization. It is worth mentioning that our scheduling approach is not based on this work; we independently built similar experimental setups (based on common open-source software).

In [136], the authors present a network-aware scheduler plugin to extend the InterPodAffinity module from their previous work. Their approach automatically updates the static Kubernetes manifest with real-time data collected from the cluster. In addition, they are using a Kubernetes controller that updates the manifests and triggers rolling updates if an improvement can be found. However, initial placement is not as good as the default approach in some cases. They need a few iterations or a larger workload to be better than the default approach.

Wojciechowski et al. [137] present a data traffic-aware scheduler that minimizes inter-node communications. This study does not handle the case of moving user equipment. Also, it does not consider latencies between the nodes.

In [138], Toka presents a latency-aware scheduler that maximizes resource utilization at the edge. He also introduces a rescheduler that can improve application placement over time. However, the inter-service data traffic is not considered in this work.

QoS-aware scheduling Mattia and Beraldi [139] present a reinforcement learning based scheduling approach that improves the stability of the frame rate of AR/VR applications. The experimental results are limited to a simulation; applying this methodology on a real Kubernetes would require a large dataset for the training stage.

The Polaris scheduler is presented in [140]. It is an SLO-aware (Service Level Objective) scheduler that considers many network metrics. The authors extend many Kubernetes scheduler plugins (pre-filter, filter, and score) to consider the topology of the cluster, the dependencies of the services, and the SLOs. However, no rescheduling mechanism is presented in this study. Also, the long computing time for placement is a problem in a dynamic environment where application placement needs to be often reevaluated. In [141], Orive et al. present a scheduling approach to minimize the application end-to-end (E2E) latency and maximize E2E reliability. They propose an architecture to define the application requirements. Their Kubernetes scheduler plugin uses these requirements to score the nodes. Nautilus [142] is a run-time system that maps micro-services to nodes

based on communication overhead, resource utilization, and IO pressure.

Table 4.4 summarizes the different scheduling methodologies described in this section.

Table 4.4.: Related work summary

Approach	Cost-aware	Network-aware	QoS-aware	Rescheduling mechanism	Using Kubernetes
Kubernetes [75]	-	-	-	-	✓
[131]	✓	-	-	-	-
[132, 133]	✓	-	-	✓	✓
[134]	-	✓	-	-	-
[135, 136, 137, 138]	-	✓	-	✓	✓
[139]	-	-	✓	-	-
[140, 141, 142]	-	-	✓	-	✓
Our approach	✓	✓	-	✓	✓

4.6. Conclusion

We propose a cost-effective orchestration methodology that simplifies the deployment of distributed applications in the Cloud-to-Edge Computing Continuum as it does not need manual placement of edge services. Doing that significantly lowers the costs of running the applications while keeping the same Quality of Service. This orchestration methodology works for clusters that aggregate resources from traditional data centers and the servers located at the edge of the network. We implement our orchestration methodology on a Kubernetes cluster and demonstrate its benefits using a realistic workload: a vehicular cooperative perception. Experiments on this workload show that using our approach reduces costs by 10% to 25% compared to the default Kubernetes scheduler for the same quality of service. Also, it is possible to use our methodology with any containerized workload.

Although we use monetary cost as our optimization target, any measurable metric could be used, e.g., energy consumption can be minimized instead.

In chapter 5, we study the performances of container CPU limitation mechanisms and present our tool to set CPU limitations automatically.

Chapter 5.

Understanding CPU Limitation Mechanisms in Containerized Parallel Applications

This chapter presents a detailed study of the container CPU limitation mechanisms and their impact on application performance. We propose a methodology for automatically selecting the best CPU limitation mechanism, and we evaluate this methodology on a Kubernetes cluster.

This chapter is based on the following publication (under review):

- Rac and Brorsson. "Understanding CPU Limitation Mechanisms in Containerized Parallel Applications". 2024. [55].

This chapter is organized as follows: Section 5.1 presents the main concepts of container CPU limitations. Section 5.2 presents the differences in performances we can observe on parallel applications when using Time division or Core division. It provides some guidelines for setting CPU limitations. Section 5.3 describes our CPU Limitation Setter (CLS). CLS is a methodology for automatically choosing the best CPU limitation mechanism. It also updates the number of allocated CPUs to avoid waste of resources, it allocates more CPUs only if they provide a significant speedup of the execution time. For example, allocating one more CPU core to an application if it provides a speedup of execution time of only 1% could be considered as a waste of resources. This section also describes the CLS implementation and evaluation on a Kubernetes cluster. Then, section 5.4 describes related work. Finally, we present our conclusions and future work directions in section 5.5.

5.1. Introduction

The ability to containerize applications has tremendously simplified the deployment of software applications. Containers package all dependencies in one artifact, and developers can then know that an application will execute as intended on any container runtime, provided that the architecture is the same. This also applies to parallel applications, which can use multiple CPU cores to solve a particular problem faster or a larger problem at the same time. Most container runtimes provide the possibility to specify limitations in resource usage in order to facilitate multi-tenancy of containers in the same server or to help the container orchestration system find the right sized server.

An example of resource limitations is *CPU limits*, which play a critical role in CPU resource allocation. They ensure that the CPU resources allocated to an application can be used only by this application. Having known limitations supports building efficient scheduling policies and maximizing resource utilization. Maximizing resource utilization is particularly important for edge computing, where resources are the most limited in the Computing Continuum. However, it is a challenging task to define CPU limits: allocating too many resources to an application results in unused resources, and not allocating enough CPUs deteriorates the performance of the application and lowers its Quality of Service.

Time division and *Core division* are the two main mechanisms to limit container CPU usage. Time division gives access to all cores in a CPU to the application based on a fraction of a time slot, which provides a very flexible and fine-grained resource allocation. It is also the default CPU resource limit mechanism in Docker and Kubernetes, therefore used by the majority of users. In contrast, Core division divides a CPU vertically and gives an application the entire core(s) allocated to it. As we will see, Core division can offer better performance for parallel applications for some use cases, as dedicating cores to the application helps to guarantee performance.

Some parallel applications (described in the next section) have a faster execution time when using Core division instead of Time division, even if the same number of CPUs is allocated to the application. Since Time division is the default CPU limitation mechanism, it is essential to understand how it works and what impacts its performance to avoid wasting resources. Furthermore, we wanted to investigate the possibility of automatically choosing the right CPU limit mechanism as it is not always easy for developers to understand what would be the best depending on the architecture of the available nodes in a Kubernetes cluster, for instance. Finally, we think that containers are one of the best options for deploying applications in the Computing Continuum; therefore, understanding container performances is important. Accordingly, this container performance study includes different types of CPUs, and we also experiment with our methodology on a heterogeneous cluster.

We make the following key contributions in this chapter:

Contributions

1. Explain the differences in performances according to the CPU limitation mechanism
2. Provide guidelines to set CPU limitations: using the most efficient limitation mechanism
3. Present a method to automatically set CPU limitations and its evaluation on a Kubernetes cluster.

5.2. Setting limits

This section describes the main container CPU limitation mechanisms and the performance difference we observe between the two when running parallel applications.

5.2.1. Container CPU limitation

Containers have two main mechanisms to limit CPU usage: Time division and Core division. *Control group* (*cgroup*) [143] is the Linux kernel mechanism that limits container resource usage. Cgroups can be used to restrict CPU usage, in addition to memory, IO, and other resources.

The default mechanism to limit CPU usage is Time division. This mechanism gives an application access to the whole CPU for a fraction of the time. There are two main scheduling strategies to allocate time between the different tasks: Completely Fair Scheduler (the default scheduler) and Real-Time scheduler. *Period* and *quota* are the two parameters to use for defining a Time division rule for the CFS scheduler. A containerized application limited by Time division can use the whole CPU for *quota* microseconds each *period* microseconds. Once the quota is spent, the application needs to wait until the period to access the CPU again. The default *period* is 100000 microseconds. The maximum *quota* is equal to 100000 microseconds times the number of vCPU on the machine. Allocating 1.2 vCPU to an application sets the *period* to 100000 microseconds and the *quota* to 120000 microseconds.

The *cpu-set* is the Core division implementation of cgroups. This mechanism limits the container to use only one or many CPU cores. For example, it can limit an application to access only the cores 0 and 1 of a CPU. In addition, it is possible to restrict many applications simultaneously to use cores 0 and 1. However, Kubernetes offers exclusivity guarantees. When using Kubernetes *static* CPU management policy, pods are getting exclusive usage of the CPUs restricted with cpu-set [144]. If a pod is given core 1, the Kubernetes scheduler will not allocate this core to any other pod on the node. Nevertheless, this exclusivity is only for other pods on the node; other processes, like the container runtime, can continue to run on these reserved cores.

Time division and the CFS is the default approach for most containerized workloads, it is the default configuration of Docker and most Kubernetes container runtimes. It allows fine-grain CPU allocation; allocating a fraction of a CPU core is possible. The notation mCPU refers to one-thousandth of CPU time. For example, allocating 500 mCPU to a pod means allocating half a core to the pod. On the contrary, Core division is coarse-grain. It is only possible to allocate full cores, not a fraction of a core.

Time and Core divisions perform similarly most of the time. However, we note that performances are different for parallel applications.

5.2.2. Time division and parallel application

We observe that containerized parallel applications can be significantly slower when using Time division. A parallel application's default number of threads varies from one framework or programming language to another. The default value can be set to the number of cores of the machine or a pre-defined setting. This is problematic when running parallel applications with the Time division mechanism. When the number of CPUs allocated with Time division is below the number of threads created by the parallel application, we observe performance degradation. That degradation is higher as the difference between the number of allocated CPUs and the number of threads created increases. A solution is to set that number of threads lower, but there is no general API or a global variable to set it for every parallel framework. Therefore, deploying a parallel application on a Kubernetes cluster with limits and default configuration impacts performance.

However, running the same application and configuration but using Core division instead of Time division changes the application performance. When using Core division, the application does not create more threads than the CPU limits; this improves execution time.

The rest of this section evaluates the impacts of choosing time or Core division on execution time, power consumption, and energy consumption. It also explains the difference in performance between time and Core division.

Experimental methodology

Table 5.1 describes our experimental setup. We are using Docker v27.1 and Linux v6.8.

Table 5.1.: Setting limits experimental setup

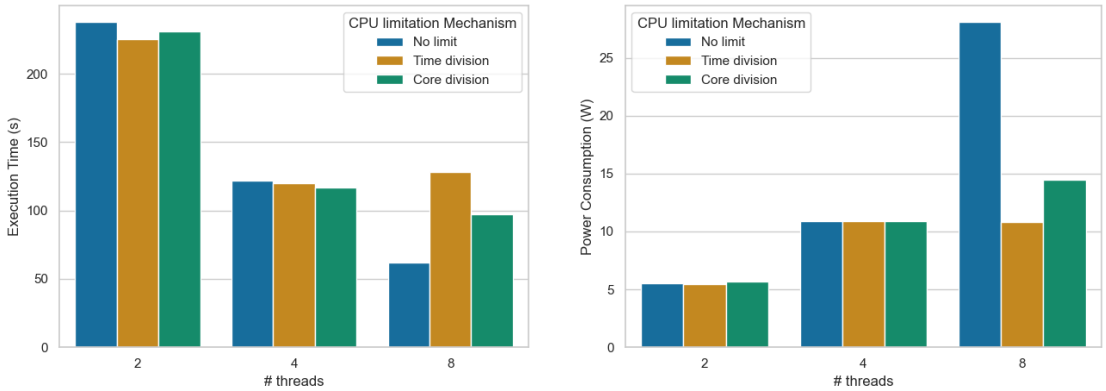
	Intel Xeon
Microarchitecture	Skylake 8175M
Architecture	x86_64
#Sockets	2
#Phys. Cores	48 (24 + 24)
#Threads	96
Clock speed	2.5 GHz
L1 Cache I/D	1.5 MiB/1.5 MiB
L2 Cache	48 MiB
L3 Cache	66 MiB
Main memory	384
Node name	intel

We measure the execution time, power consumption, and energy consumption for several benchmark applications. We use the RAPL (Running Average Power Limit) sensor available in intel processors for power and energy consumption. For the initial

benchmarks, we use the NAS Parallel Benchmark (NPB), C++ implementation [145] parallelized using OpenMP.

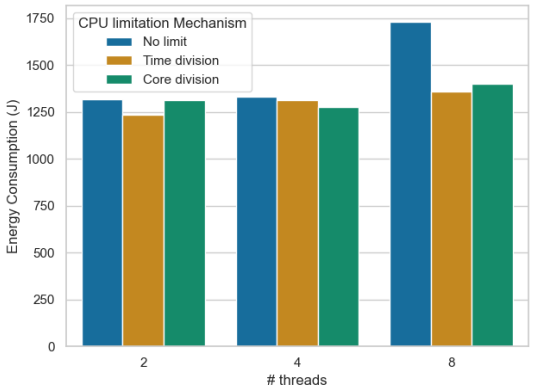
NAS parallel benchmarks

In the first experiment, we compare execution time, power, and energy consumption when we have static CPU limits but varying the number of threads, and with different CPU limitation mechanisms: No limit, Time division, and Core division. Figure 5.1 and Figure 5.2 present the results for the Embarrassingly Parallel (EP) and Conjugate Gradient (CG) benchmarks from NPB. OpenMP programs will, by default, use as many threads as there are cores on the host CPU. In this experiment, we use the environment variable `OMP_NUM_THREADS` to control the number of threads.



(a) Execution time

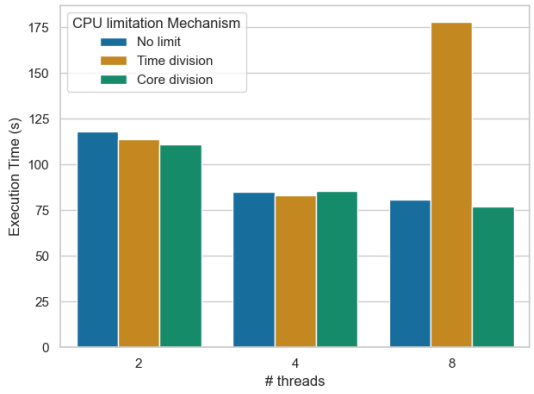
(b) Power Consumption



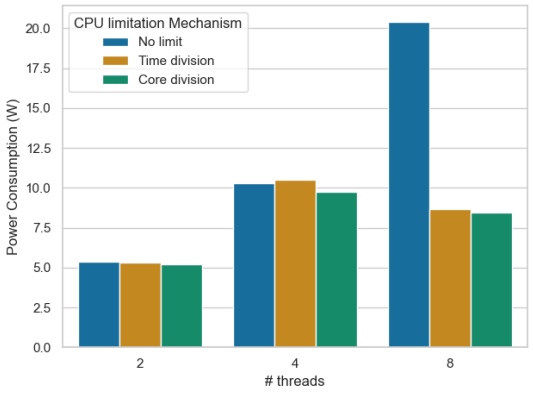
(c) Energy Consumption

Figure 5.1.: Results for a limit of 4 CPU - Embarrassingly Parallel

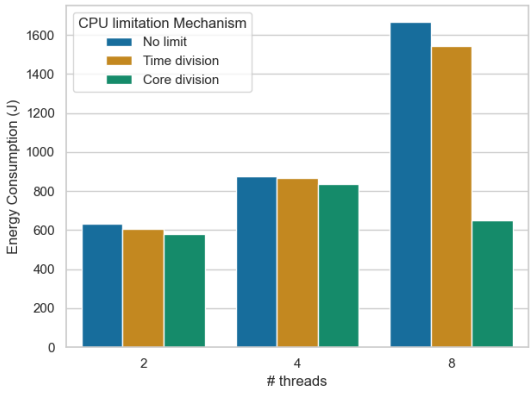
For these two applications, we note that the average execution time is similar when the number of threads is lower or equal to the CPU limits. However, it is different when the number of threads is strictly higher than the CPU limit. Execution is then considerably



(a) Execution time



(b) Power Consumption



(c) Energy Consumption

Figure 5.2.: Results for a limit of 4 CPU - Conjugate Gradient

slower when using Time division compared to Core division. For the EP application, the execution is faster when there are no limitations. That is the expected behavior; without limitations, more resources are available. For the CG application, the execution is the same when using the Core division or when there are no limitations, indicating that we have reached the practical limit of parallelism. For both of these programs, we observe a slower execution time when using Time division when the number of threads is higher than the CPU limit.

When the number of threads is below the CPU limit, the power consumption is similar for all mechanisms and and also with no limit set. However, as with execution time, the power consumption differs when the number of threads is larger than the CPU limit. Without limitation, the power consumption is almost twice as high as it is for the Time division. For Core division, power consumption is higher than Time division for EP, but it is similar for CG. We also observe that the power consumption increases when more threads are added.

The energy values are similar when using fewer threads than the CPU limit. Using

no limitations leads to higher energy consumed in every case. The energy consumed is much lower in Core division than in Time division for CG. Longer execution times lead to higher energy consumption.

Image processing workload

We also experiment with an image processing workload to confirm our observations on the NAS parallel benchmarks. We build this image processing workload using Yolov8 [146], a computer vision model. This model implementation uses PyTorch [147], one of the main machine learning frameworks. The application processes images from a video stream of a driving scene. The application identifies elements and their locations in the images. These can be helpful for autonomous driving; an embedded camera could, therefore, extract elements around the car and their locations. We want to observe if this parallel application using a framework different from the NAS benchmark presents similar performance variations when changing the CPU limitation mechanism.

Table 5.2 presents the execution times of the Image processing workload for various CPU Limitation mechanisms. We run these experiments on the Intel node.

Table 5.2.: Image Processing average executions time for Time division (TD), Core division (CD), and No limitation.

	Average Inference Time (ms)		
CPU Limit	TD	CD	No Limitations
1	798	93.2	46.2
2	251	60.9	
3	52.4	46.6	
4	42.5	41.6	
5	43.7	43.0	
6	43.8	43.7	
7	42.6	44.1	
8	44.1	44.2	

We observe a significant difference in performance between Time division and Core division when the CPU limit is equal to 3 vCPUs or less. From 4 vCPUs to 8, there is no significant difference. By default, we observe that the image-processing application creates 4 threads. When using the Time division approach, there are more threads than CPU allocated. The application adjusts the number of threads created when using the Core division approach. Bellow 4 vCPUs allocated, it creates a number of threads that matches the number of allocated vCPUs. For 4 vCPUs and above, it allocates only 4 threads. Without limitations, the application creates only 4 threads. It cannot use all the processing power of the Intel node.

Linux kernel preempt events

This section investigates why execution is slower when using the Time division mechanism. The two previous experiments have shown that parallel applications are slower when using more threads than the number of allocated CPUs and selecting Time division instead of Core division. To explain the performance difference, we study the Linux kernel preempt events. We trace the execution of the LU application from the NAS parallel benchmark suite.

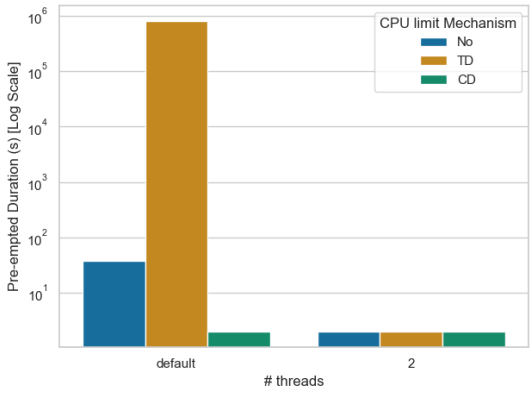
Figure 5.3 shows the duration and the number of the preempt events and the execution time of the LU application when using different CPU limitation mechanisms and a limit of 2 vCPU. We compare the results when using the default number of threads to the case where we manually set the number of threads to 2. We study Time division (TD), Core division (CD), and No-limit set (No). Figure 5.3a presents the duration of the preempt events, and Figure 5.3b their number of occurrence. Figure 5.3c shows the total execution time for the different CPU limitation mechanisms.

The duration is similar when the number of threads is set to the same value for each approach. This confirms our previous observations. The pre-empted duration is the same for all CPU limitation mechanisms, and so are the execution times. However, the number of preempt events differs, even when the same number of threads is set.

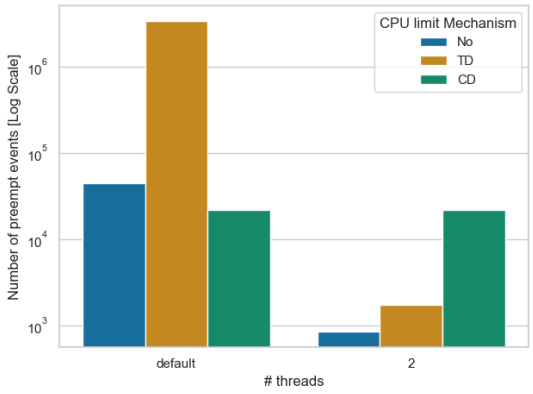
When the number of threads is set to default, we observe different behaviors for each limitation mechanism.

Execution time is much faster for the No-limit approach as it can use the 96 available CPUs. Then, using Core division over Time division gives an execution time that is 13 times faster.

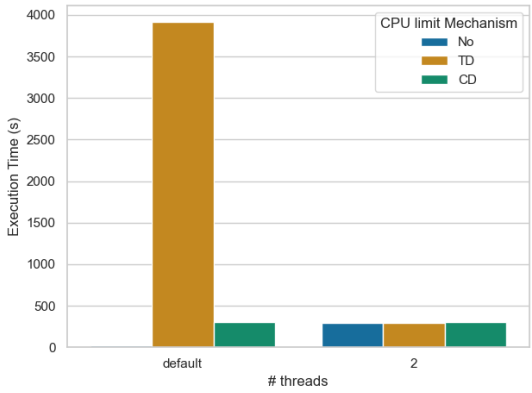
Preempted duration corresponds to the sum of each preempt event duration for each thread. The No-limit approach has a longer preemption duration than the Core division approach because it uses 96 threads, while the Core division uses only 2. Then, when all threads are preempted, the total preempt duration is longer, but the execution time is still lower. However, that preempt duration is very long for Time division. First, this duration is longer because of the 96 threads, like the No-limit approach. Then, all of the 96 threads are sharing the same time budget of 2 vCPU. That means that each thread has only a very limited time to run. When the time budget is spent, the Linux kernel scheduler runs other processes. Therefore, the threads spend most of their time switching contexts and processing them with limited time. Finally, we think that small measurement inaccuracy might inflate the total duration of preemption. The application and the tracing software are running in the same pod with a limit of 2 CPUs. That might affect the precision of the measures. However, we run all these experiments in the same conditions, with the tracing software always running with the application in a CPU-limited container.



(a) Duration of preempt events



(b) Number of preempt events



(c) Execution time

Figure 5.3.: Duration and number of preempt events and execution time for the LU application for a limit of 2 vCPU

Conclusion

We observe similar effects between these experiments. When the default number of threads is higher than the CPU limit, there is a negative impact on performance when using Time division. Using Core division adjusts the number of allocated threads. Then, we do not observe such a significant impact on performances. When the default number of threads is lower or equal to the CPU limit, Time division and Core division perform similarly. In this case, we recommend using Time division. Time division enables more flexibility in resource allocation. Also, Time division supports the in-place resize feature of the CPU limits. That helps update CPU limits without restarting the application. Core division also supports this feature in cgroup and Docker containers but not in Kubernetes. However, using in-place resize for Core division does not update the number of created threads. Therefore, the application needs to be restarted nonetheless to update the number of created threads.

In summary, using Core division helps improve performance when deploying parallel applications on Kubernetes. It is primarily essential when deploying applications on processors with a large number of cores and allocating only a few of them to the application.

5.3. CPU Limitation Setter (CLS)

This section describes our CPU Limit Setter (CLS). It is a tool that automatically adjusts the CPU limits (i.e., the number of allocated CPUs and the CPU limit mechanism). It selects limits based on the results of the previous section. Setting resource limitations is helpful for scheduling, even if it is not mandatory in Kubernetes. It helps maximize resource utilization while ensuring enough resources for applications to run. It is essential for edge computing, where computing resources are more limited than in traditional data centers.

We designed the CLS to be integrated into a Kubernetes cluster as it is the *de facto* standard for container orchestration. Setting limits can be tedious; setting limits too low leads to performance issues, and setting limits too high results in the waste of resources. We design the CLS to adjust resources automatically to find a trade-off between performance and resource utilization. According to the previous section, having well-set resources also helps reduce energy consumption. Therefore, a tool that automatically sets limits for Kubernetes deployment can be beneficial for application developers.

CLS can determine how many CPUs to allocate and which limitation mechanism to use, realizing a quick application profiling. It creates a copy of the deployment to evaluate its performances with various parameters (e.g., number of CPU allocated, limit mechanism). Then, profiling the application does not disturb the execution of the original application. In addition to ad hoc profiling, the CLS records the actual application's performances to increase the profiling's accuracy without additional costs. CLS makes sure not to profile the same application twice. It checks the hash of the container image to make sure there are no existing profiles about it. Profiles can be reused for different types of nodes at the cost of a lower accuracy. It helps reduce the number of resources required to profile an application.

The CLS proceeds as follows: it first detects which limitation mechanism to use and then profiles the application using the selected mechanism. Finally, it updates the number of CPUs allocated to the application and its CPU limitation mechanism. It only increases the number of allocated CPUs if there is an actual speedup.

Applications are profiled once and run multiple times. Then, it is possible to use application profiles multiple times. Spending a few resources to benchmark the application once and get the benefits of well-adjusted limits for all instances is an acceptable trade-off. We could even extend this approach to share profiles between multiple clusters, but it is outside of the scope of this study.

5.3.1. CLS methodology

This section describes our CPU Limitation Setter (CLS) methodology, its main algorithm, and its integration into the Kubernetes ecosystem. The main idea of this approach is to profile applications while they run and execute the additional tests in the background.

First, the CPU Limitation Setter (CLS) watches each workload running in the cluster. It maintains a list of profiled applications and already optimized pods. CLS uses the hash of the container image and the command of containers to recognize which applications are already profiled. Therefore, if an application has already been profiled, it is unnecessary to do it again. Different instances are automatically detected even if they are using different deployments. When the application is updated, the container image hash will be different, and the CLS will detect it as a new application to profile.

Profiles are stored in a database, using the hash of the container and its command as the key for the entry. A profile is a set of performance measurements (i.e., execution times, power consumption) with different labels (i.e., type of node, CPU limitation mechanism used). Once they are recorded, this measurement can be used for any application deployment. The more instances an application has deployed, the lower the cost of profiling is. We aim to profile applications that are deployed multiple times to achieve economies of scale.

We use a copy of the production deployment for profiling applications without disturbing the production pods. The idea is to create one or more pods similar to the production pod in the background. We can then change the CPU limitation mechanism and CPU limit of the application without disturbing the production environment. Updating the CPU limitation mechanism requires restarting the pod. Restarting the application multiple times would negatively impact the production environment.

We profile the application to detect the right CPU limitation mechanism. It runs two additional deployments, one using Core division and the other using a Time division. Then, it measures performance. These new deployments run in the background and do not affect the production deployment. CLS uses, by default, Time division unless it is more than ϵ percents (e.g., 5%) slower than Core division. It is better to use Time division when it does not negatively affect the performance because it can allocate fractions of a CPU. It helps maximize resource usage and set finer grain limits. However, we use Core division if there is a performance gap between the limitation mechanisms. In this case, the Core division can perform better with the same resources reserved.

The CLS starts benchmarking the application once the CPU limitation mechanism is set. It deploys new pods and measures performances similarly to the CPU limit selection process. Then, once the profiling is done, the CLS stores all the metrics in a database. We do not want to benchmark the application multiple times. The CLS can reuse this profile the next time it detects another instance of the same application. These profiles could even be available for different clusters.

When the application profile is available, the autoscaler can adjust the CPU limit and limitation mechanism of the deployment. CLS ensures that allocating more CPUs to the application really improves the execution time. The rule is to add one more CPU to the

limit only if the execution time speedup is greater than η percent (e.g., 5%). This rule avoids wasting CPU resources.

Finally, the CLS continues iterating about the workloads in the cluster, adjusting limits to the objectives.

Algorithm 3 presents the algorithm of the CLS. The CLS watches the workload deployments in the cluster. First, it checks if the deployment pods are already profiled. If not, it starts selecting the proper limitation mechanism. Then, it profiles the application for various CPU values. To select the proper limit mechanism and benchmark the application, CLS creates a copy of the deployment to avoid disturbing its execution. After the measurements, it deletes all the deployments it used for profiling the application. All the profiles are stored in a dedicated database. The pod must be restarted when changing the CPU limitation mechanism (e.g., from Time division to Core division). However, updating the number of CPUs allocated using the Time division *in-place resize* feature is possible without restarting the application. There is no similar method available for the Core division mechanism.

Algorithm 3 CPU limitation setter

Require: $\epsilon > 0$, $\eta > 0$
  for each deployment in cluster **do**
    if deployment not profiled **then** ▷ check container image hash
      profile ← CreateDeploymentCopy(deployment, td, 1) ▷ Use Time division and set CPU limit to 1
      profile.save()
      profile ← CreateDeploymentCopy(deployment, cd, 1) ▷ Use Core division and set CPU limit to 1
      profile.save()
      if $\frac{|\text{GetProfile(deployment, td, 1)} - \text{GetProfile(deployment, cd, 1}|}{\max(\text{GetProfile(deployment, td, 1)}, \text{GetProfile(deployment, cd, 1)})} < \epsilon$ **then**
        profile.limitationMechanism ← TD
      else
        profile.limitationMechanism ← CD
      end if
      cpuLimits ← 2
      while cpuLimit ≤ MaxCPULimit AND profile.lastSpeedup $> \eta$ **do**
        profile ← CreateDeploymentCopy(deployment, profile.limitationMechanism, cpuLimit)
        profile.save()
        cpuLimits ← cpuLimits + 1
      end while
    end if
    profile ← getProfile(deployment)
    updateLimits(deployment, profile.LimitMechanism, profile.NumberCPU)
  end for

Kubernetes implementation The CLS is a Go application deployed in a Kubernetes cluster in a dedicated pod. It can access the Kubernetes APIs to create, update, and delete deployments. It can also create copies of deployments, modifying only some parameters like CPU limits. CLS can pull monitoring data from Prometheus and other software described in chapter 2 section 2.1.3. Then, it stores the software profiles in a dedicated database. Each profile is identified with the hash and the command of the container of the profiled application. A profile contains information about the execution time, CPU characteristics, CPU limitation mechanism, and allocated resources. The CLS uses the profiles to set appropriate limits when deploying a new application. Profiles are created when the first instance of the application is deployed. Then, profiles are updated continuously while application instances are running. Future work can investigate rescheduling mechanisms similar to the previous chapters to update limits based on the profile updates. However, profile updates are helpful when deploying new instances of an existing application.

Figure 5.4 presents an overview of the integration of the CLS into the Kubernetes ecosystem.

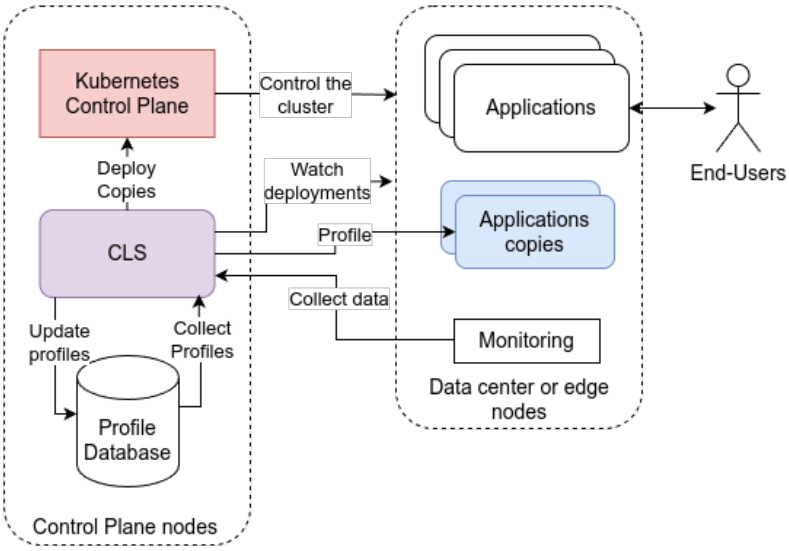


Figure 5.4.: Overview of the CPU Limitation Setter (CLS)

5.3.2. CLS evaluation

Experimental setup

We use the NAS benchmark C++ implementation [145] for these performance experiments. It uses OpenMP APIs.

Table 5.3 describes our experimental setup. All nodes use the same software configuration: Docker v27.1, Kubernetes v1.28, and Linux v6.8.

Table 5.3.: CLS evaluation experimental setup

	Intel Xeon	**AMD EPYC**	**AWS Graviton**
Microarchitecture	Skylake 8175M	EPYC 7R13	Graviton 2
Architecture	x86_64	x86_64	aarch64
#Sockets	2	2	1
#Phys. Cores	48 (24 + 24)	96 (48 + 48)	64
#Threads	96	192	64
Clock speed	2.5 GHz	2.6 GHz	2.5 GHz
L1 Cache I/D	1.5 MiB/1.5 MiB	3 MiB/3 MiB	4 MiB/4 MiB
L2 Cache	48 MiB	48 MiB	64 MiB
L3 Cache	66 MiB	384 MiB	32 MiB
Main memory	384	768	256
Node name	Intel	AMD	ARM

We use the RAPL (Running Average Power Limit) sensor to measure power and energy consumption. This sensor is only available on x86 processors from Intel and AMD. Therefore, there is no energy data available for the ARM processor.

During these benchmark experiments, we compare the execution time, power, and energy consumption.

Appendix B presents the detailed profiles of the application studied in this section.

Results

Table 5.4 presents the CLS results. It compares the execution speedup, the energy consumed, and the resource allocated for six applications from the NAS benchmark suite [145]: CG - Conjugate Gradient, EP - Embarrassingly Parallel, FT - discrete 3D fast Fourier Transform, IS - Integer Sort, LU - Lower-Upper symmetric Gauss-Seidel, MG - Multi-Grid on a sequence of meshes.

We compare the performance of each application using different CPU limitation strategies, using the Time division mechanism (the default Kubernetes approach) and comparing it to our CPU Limit Setter (CLS). The CLS can choose either Core division or the Time division mechanism. We present execution time speedup normalized relatively to the execution time of the application with only one CPU allocated.

We reproduce this experiment on three different CPUs (Intel, AMD, and ARM). We use the RAPL sensor to record power and energy consumption. This sensor is only available on Intel and AMD x86 chips. Therefore, we have energy values to discuss for the ARM node. We report the total energy consumed during the execution of each application.

For these applications, the CLS chooses the Core division mechanism every time.

The CLS has the highest execution time speedup for every application and each CPU. The higher the execution time speedup, the better. CLS gets a higher speedup than the

default approach of Kubernetes, even when they are using the same number of CPUs. That can be explained by the fact that CLS chooses to use Core division instead of the Time division mechanism. The Core division mechanism offers better performance for these parallel applications.

The CLS uses the maximum CPU for most of the cases. The lower the resources allocated, the better. It uses the most CPU possible, but there is still a significant speedup when adding more. For the Conjugate Gradient application, the AMD node gets the faster speedup with only five CPUs. For the Multi-Grid application, the AMD node gets the faster speedup with only three CPUs. The CLS allocates fewer CPUs because the relative speedup between five and six, and between three and four CPUs was less than 5%.

The CLS consumes less energy than any other approach. The lower the energy consumption, the better. The power consumption is similar when using Time division or Core division, as described in the previous section. However, the execution time when using Core division is much lower; therefore, the energy consumption is lower.

Table 5.4.: CLS Results

	Execution Time Speedup			**Resource utilization (vCPU)**			**Energy Consumed (J)**		
CG	Intel	AMD	ARM	Intel	AMD	ARM	Intel	AMD	ARM
TD 1	1,00	1,00	1,00	1	1	1	1,01 10^3	1,47 10^3	-
TD 2	2,08	1,08	1,87	2	2	2	9,73 10^2	2,72 10^3	-
TD 4	3,76	1,82	3,91	4	4	4	1,08 10^3	3,24 10^3	-
TD 8	23,3	4,17	8,03	8	8	8	3,48 10^2	2,83 10^3	-
CLS	**141**	**235**	**13,7**	8	5	8	**5,77** 10^1	**3,14** 10^1	-
EP	Intel	AMD	ARM	Intel	AMD	ARM	Intel	AMD	ARM
TD 1	1,00	1,00	1,00	1	1	1	1,59 10^2	1,41 10^2	-
TD 2	2,19	2,93	2,01	2	2	2	1,45 10^2	9,62 10^1	-
TD 4	4,63	6,27	4,01	4	4	4	1,37 10^2	8,99 10^1	-
TD 8	9,50	14,0	**8,09**	8	8	8	1,34 10^2	8,07 10^1	-
CLS	**11,2**	**17,9**	**8,09**	8	8	8	**1,14** 10^2	**6,29** 10^1	-
FT	Intel	AMD	ARM	Intel	AMD	ARM	Intel	AMD	ARM
TD 1	1,00	1,00	1,00	1	1	1	2,83 10^2	2,49 10^2	-
TD 2	2,30	2,33	1,95	2	2	2	2,46 10^2	2,14 10^2	-
TD 4	5,12	5,61	3,57	4	4	4	2,21 10^2	1,77 10^2	-
TD 8	14,0	11,3	7,15	8	8	8	1,62 10^2	1,76 10^2	-
CLS	**45,5**	**62,9**	**8,99**	8	8	8	**4,98** 10^1	**3,17** 10^1	-
IS	Intel	AMD	ARM	Intel	AMD	ARM	Intel	AMD	ARM
TD 1	1,00	1,00	1,00	1	1	1	1,44 10^1	2,04 10^1	-
TD 2	1,00	1,61	1,89	2	2	2	2,88 10^1	2,53 10^1	-

	Execution Time Speedup			**Resource utilization (vCPU)**			**Energy Consumed (J)**		
TD 4	1,94	2,93	3,65	4	4	4	2,98 10^1	2,79 10^1	-
TD 8	12,8	5,53	6,20	8	8	8	9,00 10^0	2,96 10^1	-
CLS	**40,5**	**127**	**13,1**	8	8	8	**2,85** 10^0	**1,29** 10^0	-
LU	Intel	AMD	ARM	Intel	AMD	ARM	Intel	AMD	ARM
TD 1	1,00	1,00	1,00	1	1	1	2,66 10^3	7,69 10^3	-
TD 2	0,71	4,33	1,15	2	2	2	7,51 10^3	3,55 10^3	-
TD 4	0,39	5,82	5,40	4	4	4	2,73 10^4	5,29 10^3	-
TD 8	13,8	32,8	19,2	8	8	8	1,54 10^3	1,87 10^3	-
CLS	**115**	**697**	**33,0**	8	8	8	**1,85** 10^2	**8,83** 10^1	-
MG	Intel	AMD	ARM	Intel	AMD	ARM	Intel	AMD	ARM
TD 1	1,00	1,00	1,00	1	1	1	3,37 10^2	9,04 10^2	-
TD 2	2,11	1,83	1,87	2	2	2	3,19 10^2	9,88 10^2	-
TD 4	3,22	4,89	3,91	4	4	4	4,18 10^2	7,40 10^2	-
TD 8	7,20	6,14	8,03	8	8	8	3,74 10^2	1,18 10^3	-
CLS	**41,17**	**99,26**	**13,73**	8	3	8	**6,54** 10^1	**2,73** 10^1	-

Transferable profiles

In this section, we study how to reuse application profiles when deploying new application instances on a different CPU type (e.g., with a different clock speed, generation, constructor, or even architecture). It is important to support heterogeneous clusters to be able to use the CLS in the Computing Continuum. The idea is to avoid profiling the application on every type of node in the cluster, using an approximation based on the initial profiles. Over time, profiles become increasingly complete as they also include information about the new instances running (and sometimes running on different types of nodes). Reducing the number of nodes to profile enables economies of time, costs, and energy.

To understand how similar profiles can be between two different types of CPU, we study i) the decision of the CLS for the CPU limitation mechanism and ii) the relative execution time speedups when increasing the number of allocated vCPUs. We compare these decisions and speedups for the Intel, AMD, and ARM nodes.

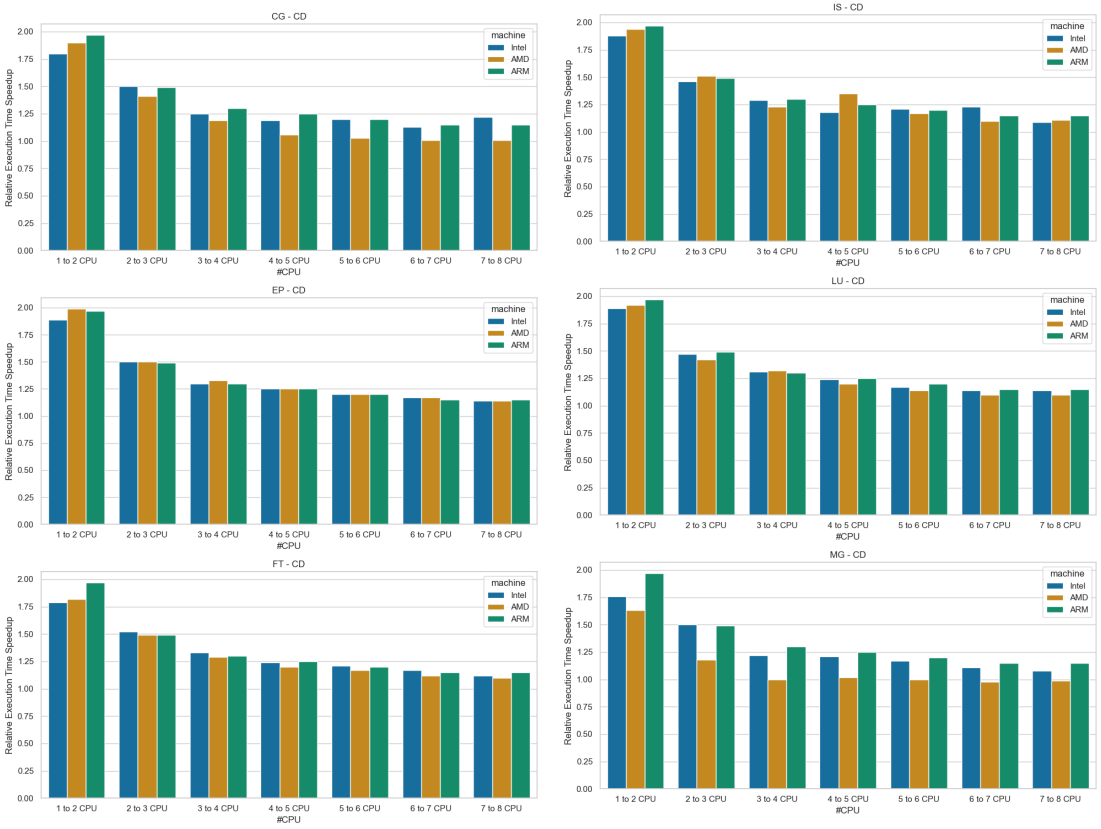
Table 5.5 presents the chosen mechanisms between Time division (TD) and Core division (CD) for each application and CPU type. The decision of the CLS is the same for all CPU types and applications except for the EP application on the ARM node. There is no significant difference between TD and CD execution time. That is why CLS chose TD over CD. However, with similar performances between TD and CD, choosing CD instead of TD is not problematic. For these six applications, the choice of the CPU Limitation Mechanism can be the same for every node. There are no negative impacts if the decision of the CLS from another node regarding the CPU Limitation Mechanism is reused.

Table 5.5.: CPU Limitation Mechanism chosen by the CLS.

	Chosen CPU Limitation Mechanism		
Application	Intel	AMD	ARM
CG	CD	CD	CD
EP	CD	CD	TD
FT	CD	CD	CD
IS	CD	CD	CD
LU	CD	CD	CD
MG	CD	CD	CD

Figure 5.5 and Figure 5.6 present the relative speedups when allocating one more CPU. Results are different for Core division and Time division.

For Core division, speedups are mainly similar from one CPU to another. The results of the CLS would be the same except for two applications (CG and MG) on the AMD node.

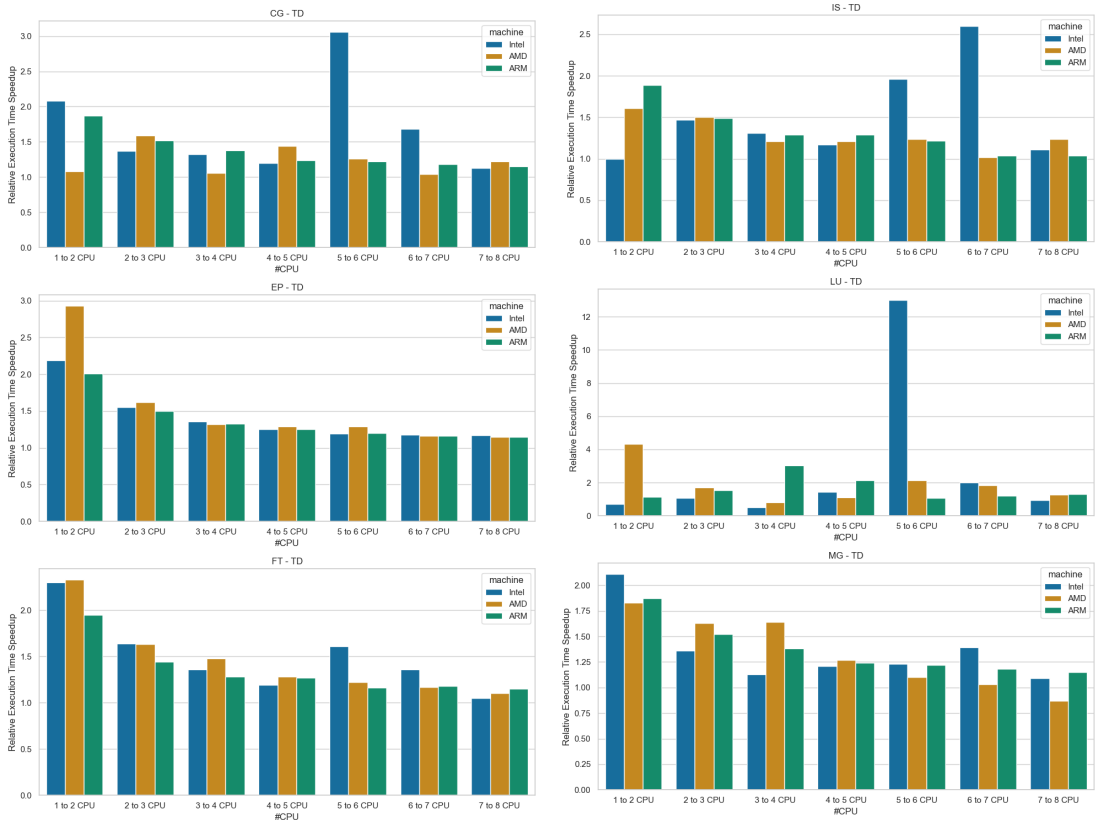


(a) CG, EP, and FT - Core division

(b) IS, LU, and MG - Core division

Figure 5.5.: Relative speedups - Core division

For Time division, speedups are different from one node to another. It would be difficult to predict the results for a different node. However, these are parallel applications; CLS chose to use Core division because of the performance difference between the two mechanisms. If CLS had chosen the Time division, we could expect performances to be similar to or better than the Core division (otherwise, the Core division would have been chosen). If performance is similar between the two, we could expect to have results similar to Core division where reusing CLS decisions for another node is not problematic.



(a) CG, EP, and FT - Time division

(b) IS, LU, and MG - Time division

Figure 5.6.: Relative speedups - Time division

Using a profile from a different type of node seems to be a promising trade-off. In most cases, the CLS decision does not change from one node type to another. The error due to the profile transfers could have a more negligible impact on resource utilization than systematically profiling applications on every node. In addition, profiles are updated with production data. That means that transfer profile approximations are only necessary until one instance runs or has run on every node type.

CLS overhead

Figure 5.7 shows the CPU usage of the Kubernetes Control Plane and CLS pods. We observe that the total CPU usage of the CLS is close to 0%. It is significantly lower than other components we can find in Kubernetes clusters. The CLS application has a very limited impact on resource usage.

However, the copies of the deployments have a larger impact. It depends on the number of CPU limits tested, but in the worst case, it doubles the resources currently allocated to the application for a few minutes. This additional usage of resources is not negligible, but applications are only profiled once and deployed multiple times often.

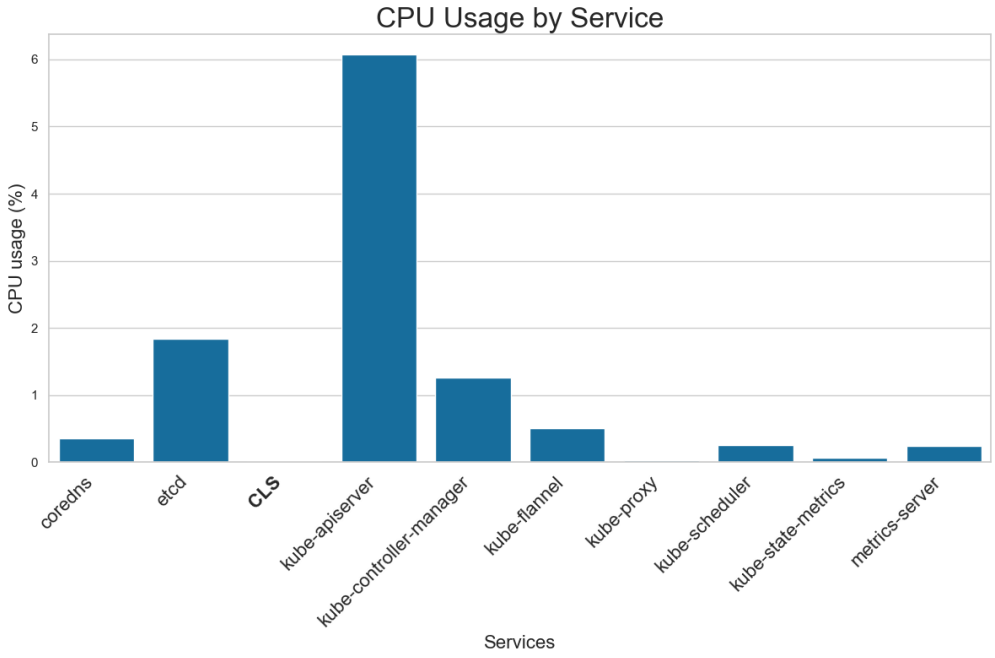


Figure 5.7.: CPU usage of the CLS and the Kubernetes control plane components

In summary, using the CLS has a cost associated with using additional resources to profile the applications. This cost should be observed in relation to the number of instances of the application that will be running. Profiling applications can save resources overall if they are deployed multiple times.

5.4. Related work

This section briefly overviews existing work related to container resource limitations.

Auto-scaling is a method for automatically adjusting allocated resources. Chapter 2 section 2.1.3 provides a general overview of autoscaling techniques and describes the Kubernetes Pod Autoscaler. Holoscale presents an approach that mixes horizontal and vertical scaling to reduce delays when creating new Virtual Machines [148]. Quattrocchi

et al. compare their two autoscaling methods to existing methods of major cloud providers [149]. One of their autoscaler is based on control theory, and the second is based on queuing theory. [150] uses machine learning to predict how many pod replicas should be deployed to achieve the expected response time. [151] predicts the pod resource limits using decision tree regression.

There are many approaches to finding a **suitable number of threads**. De Lima et al. use Artificial Neural Networks to predict thread count combinations that provide optimal energy-delay product [152]. Da Silva et al. present an autoscaler that first tunes the number of threads during the runtime to find a trade-off between energy consumption and performance and then reallocates freed resources (if any) to other containers [153]. One limitation is that this approach only supports OpenMP workloads. Balla et al. have created an adaptative autoscaler to adjust CPU limits to have just enough processing power to be below execution time requirements [154]. [155] uses machine learning to automatically tune the degree parallelism for query optimization.

Performances of containers and the Linux CFS has been studied in different studies [156, 157, 158, 159]. [156] proposes to extend the notion of namespace to include more detailed information about available resources to the containers. This could improve resource management for parallel runtimes and programming languages. Zhang et al. propose a joint optimization of CPU scaling and core sharing to address a limitation of the CFS [159]. The CFS forces threads from different containers to share cores, which can waste resources. Liu et al. observe performance degradations in multi-tenant environments [157]. They find that containers are not getting all the resources they requested due to the CFS forcing the threads of different containers to share CPU cores. [158] studies limitations of the CFS for burstable containers. The authors present two OS mechanisms to mitigate the performance issues.

5.5. Conclusion

This chapter analyzes the performances of different Container CPU limitation mechanisms. We observe that the Core division limitation mechanism performs better than Time division for some containerized parallel applications. Using Core division can provide lower execution time and energy consumption for the same CPUs allocated.

We propose a CPU Limitation Setter (CLS) method for automatically selecting the CPU limitation mechanism that offers the best performance. Our Kubernetes implementation of this method demonstrates faster execution time and lower energy consumption compared to the default approach using the Time division mechanism.

Profiling applications with the CLS has an overhead worth paying when deploying many instances of the same application. In addition, profiles are updated over time with already deployed applications to reduce this overhead. Also, using a profile for different machines is a reasonable approximation. Transferring profiles from one machine type to another helps support heterogeneous clusters with different kinds of servers. Supporting heterogeneous clusters is important for achieving unified orchestration over the

Computing Continuum. Computing resources at the edge might be different from those in traditional data centers.

Future work can investigate ways to update CPU limitation decisions during application runtime to react to workload variation. New CPU limitation decisions could rely on data acquired during runtime to improve the accuracy of the decision.

Chapter 6.

Conclusions

In this dissertation, we studied new tools and methods to improve the experience of deploying applications in the Cloud-to-Edge Computing Continuum. In our vision, deploying applications at the edge or anywhere in the Computing Continuum should be as easy as deploying applications in traditional data centers. Traditional cloud computing techniques need to evolve to address the new challenges of the Computing Continuum. Traditional orchestration techniques are unsuitable for handling geographically distributed nodes. For example, orchestration methods for the Computing Continuum should be aware of networking and end users' locations to offer ultra-low latency.

First, in chapter 3, we improved how we can evaluate orchestration methods in the Cloud-to-Edge Computing Continuum. We built an experimental methodology to evaluate new orchestration methods efficiently without dedicating much time and money to building a realistic testbed from scratch. Evaluating orchestration methods for the Computing Continuum requires to experiments with the effects of the geographic distribution of nodes. Testbeds built using this methodology can include any kind of nodes and then constitute a heterogeneous cluster. We conducted a 5G core study to illustrate this experimental methodology. This 5G study provides advice for setting up 5G infrastructure in the Computing Continuum. Also, telecommunication actors can quickly reproduce our experimental setup to run additional experiments.

Then, we developed a new orchestration approach to minimize the costs of deploying applications in the Cloud-to-Edge Computing Continuum in chapter 4. This methodology includes a rescheduling mechanism that can update scheduling decisions to adapt to changing environments. Our approach is cost- and network-aware in adapting to specificities of the Computing Continuum, like geographically distributed nodes. This approach works for heterogeneous clusters; any kind of node can be managed, and only the characteristics of the node matter for this scheduling approach. We evaluated this approach with a realistic 5G use case: vehicular cooperative perception. Our scheduling approach finds a trade-off between networking costs to transfer video streams and computing costs to process the frames from the video streams. It shows its adaptability by reactively moving the video processing application from one node to another when vehicles move.

Finally, in chapter 5, we evaluated CPU limitation mechanisms and their performance when running containerized parallel applications. Also, we propose a methodology to automatically choose the best CPU limitation mechanism and set limits that do

not waste computing resources. Maximizing resource utilization is important in the Computing Continuum because resources can be limited in some areas, for example, at the edge. Therefore, allocating the correct number of CPUs is important to avoid over- or under-provisioning. We observe that when deploying parallel applications in a containerized environment, the choice of the CPU limitation mechanism is crucial for performance. Then, we built a tool that automatically set the mechanism that offers the best performance. Also, our CPU Limitation Setter Tool only allocates more CPUs to an application if it really improves its performance. Finally, we studied how this methodology works on heterogeneous clusters. It is important that our method works for clusters with different types of nodes to simplify access to the Computing Continuum resources.

In summary, this dissertation presents many tools and methodologies to facilitate the adoption of edge computing. The tools presented help improve testing, scheduling, and setting CPU limits in the Computing Continuum. We make sure that all the tools we built can work with Kubernetes, the *de facto* standard for container orchestration. We think it is important to facilitate the usage of these tools by other academic and industrial actors.

Future research directions

We conclude this manuscript with a presentation of possible future research directions. It is not an exhausting list, but ideas we find interesting.

Better accelerator support Orchestration and scheduling techniques could be improved to offer better support to accelerators in the Cloud-to-Edge Computing Continuum. Orchestration frameworks like Kubernetes already support accelerators like GPGPUs and FPGAs, but this support needs to be extended for the Computing Continuum. Chapter 4 shows that the geographical position of the nodes is crucial for orchestration in the Computing Continuum. These techniques could be extended to support accelerators in addition to CPUs. The geographical location of these devices is essential to achieve ultra-low latency.

Accelerator-aware orchestration approaches should be able to map applications to available accelerators in the Computing Continuum, close to the end users. Orchestration could reprogram FPGAs to replace a missing accelerator when nothing else is available in an area. Finally, future research can investigate software that can run on most accelerators without additional compilation.

Running the same software everywhere Running the same software everywhere would help distribute software in the Cloud-to-Edge Computing Continuum. We can find different kinds of processors in the Computing Continuum, and a wide variety of accelerators. WebAssembly could be a good candidate for running the same binary code everywhere. It would avoid creating a container image for each different type of device

in the Computing Continuum. Its smaller size is beneficial for small devices at the edge. WebAssembly modules would be easier to distribute than other containers.

Scheduling approaches could be runtime-aware to ensure the QoS when executing WebAssembly workloads in the Computing Continuum. There are many WebAssembly runtimes; they offer different performances depending on the host architecture and the compilation parameters (i.e., Interpreted, Just In Time, Ahead of Time). These various parameters have a direct impact on the application performance. Therefore, a runtime-aware scheduler could select nodes and adjust allocated resources to ensure the QoS. Larger nodes could run different runtimes to offer more versatility; smaller nodes would be limited in the choice of runtime they can run. Future research in orchestration systems for selecting the best runtime, node, and resources to allocate would be valuable to facilitate the adoption of edge computing technologies.

Users or nodes mobility prediction In chapter 4, we presented an orchestration methodology that reactively moves applications when it detects that users are in a different location. Future research can investigate proactive methods that can predict where users will be when they are moving. That way, an application can be moved to the new areas before the users. Having applications always close to traveling users is helpful to keep ultra-low latency, even during mobility. For example, people on a train are more likely to be in a position following the railway. Modeling a train could help to estimate the traveler's movement reasonably. Similar estimations could be done for vehicles and highways.

Optimal CPU limitations Future research could investigate a way to create software profiles for the CPU Limitation Setter described in chapter 5 using source code or binaries. Using resources to profile applications is worth it when they are deployed multiple times. However, it is more expensive to use it when applications do not have so many replicas or when they are updated very often (e.g., many times per day). Therefore, using machine learning techniques or large language models could help predict the performance of an application on a dedicated processor using the source code or the binary of the application. They also might be able to automatically detect the best CPU limitation mechanism to choose without running the application.

Cluster federation Future research could investigate cluster federation and distributed orchestration algorithms to help deploy workloads anywhere in the Cloud-to-Edge Computing Continuum. Federated clusters can help to make the Computing Continuum resources more accessible. However, cluster federation raises challenges like preserving users' privacy, reliability, or scalability. The geographical position of the users and resources must be known to achieve ultra-low latency, but these should not be globally available to ensure privacy.

Appendix A.

5G setup validation

This section presents procedures based on the traces recorded for the experiments of chapter 3.

Figure A.1 presents the User Equipment (UE) registration procedure.

Figure A.2 presents the Protocol Data Unit (PDU) session establishment procedure.

Figure A.3 and Figure A.4 illustrate how UEs can access data networks. A data network can be the internet or a local network host edge nodes. A UE sends a package to the gNB using the radio network. Then, the gNB transmits the package to the UPF using the GTP protocol. Finally, the UPF transfers the package to the data network.

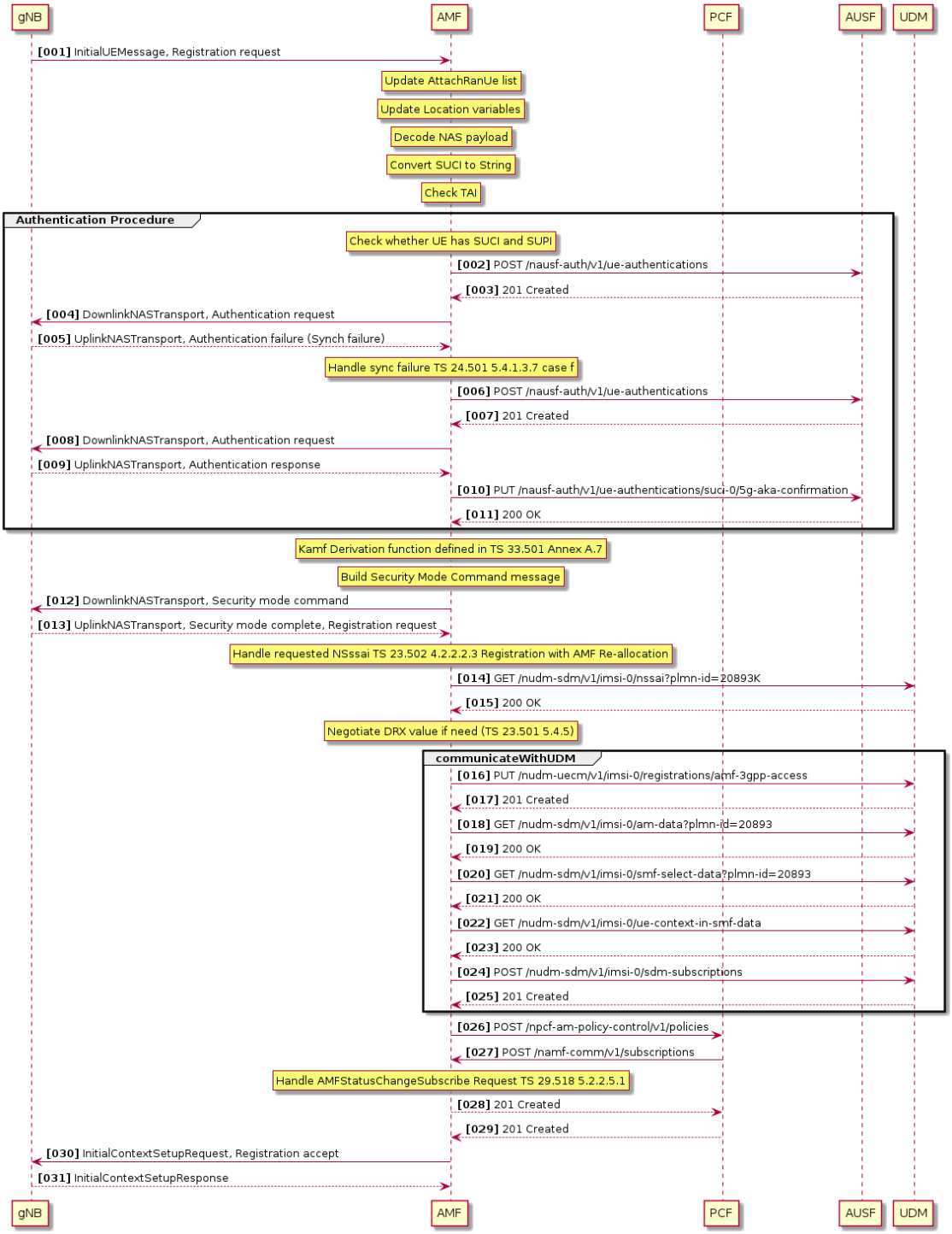


Figure A.1.: UE registration procedure

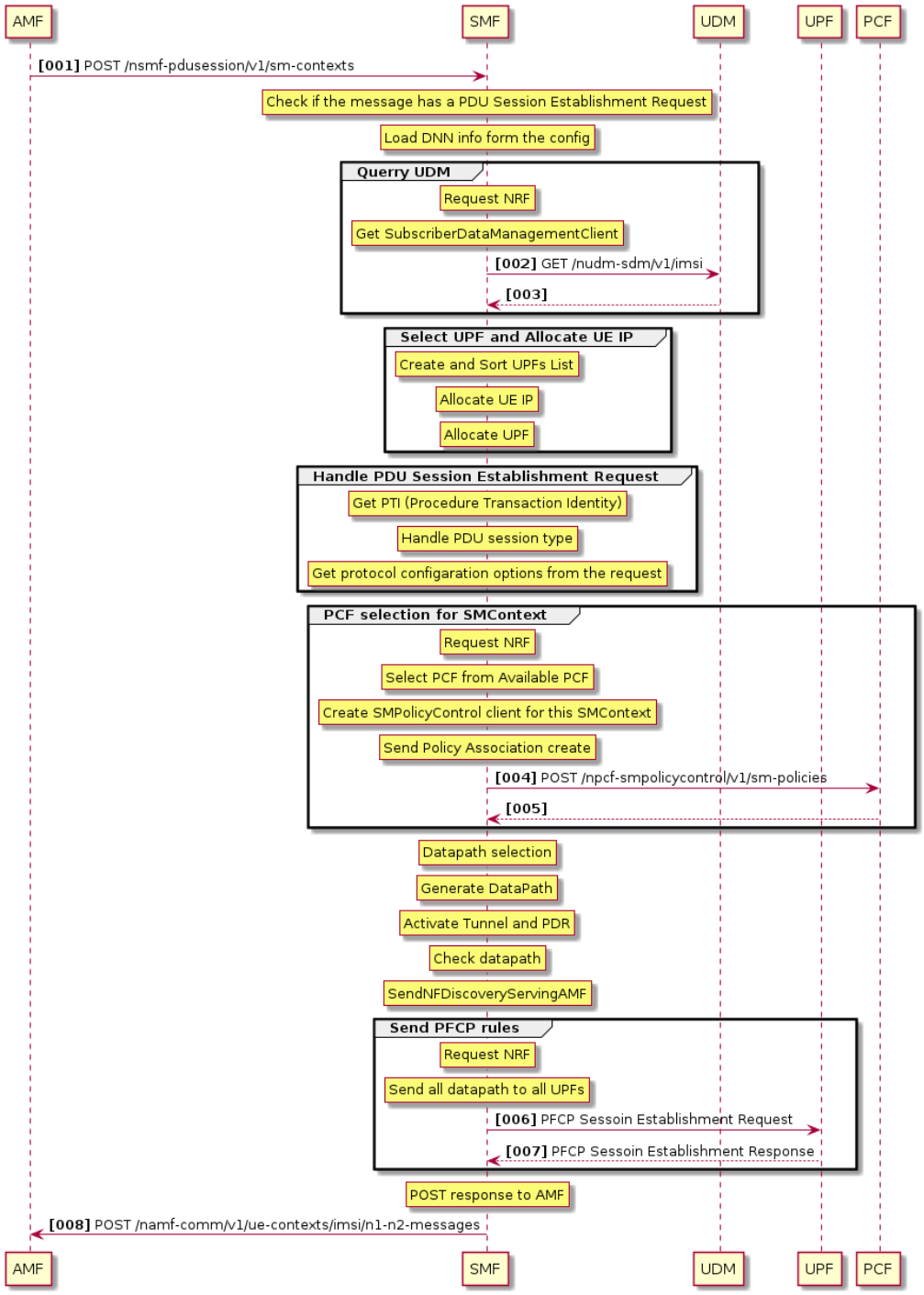


Figure A.2.: PDU session establishment procedure

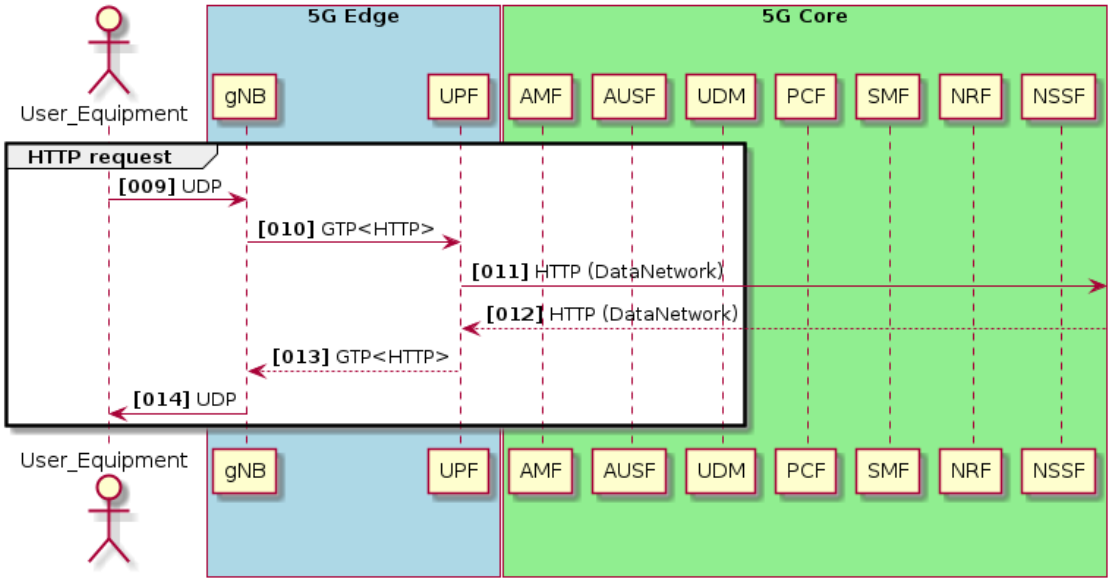


Figure A.3.: Http requests

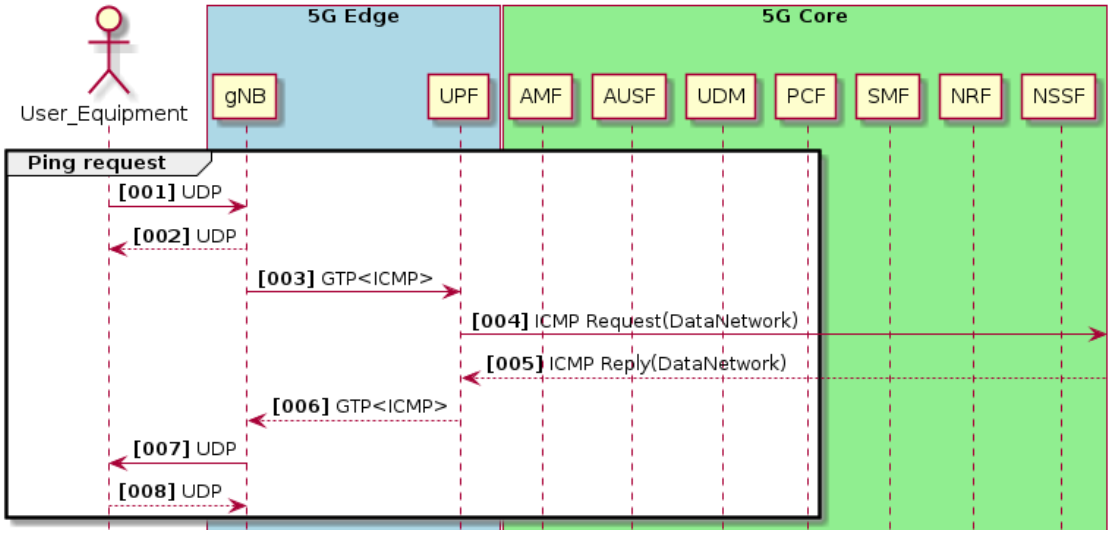
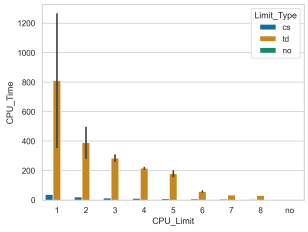


Figure A.4.: Ping requests

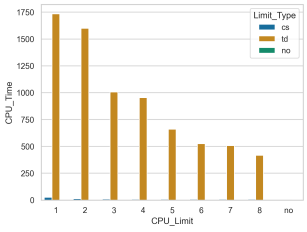
Appendix B.

Detailed application profiles

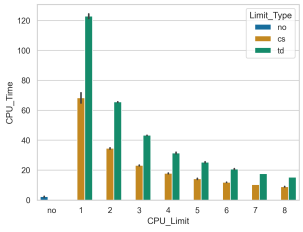
Figure B.1, Figure B.2, Figure B.3, Figure B.4, Figure B.5, and Figure B.6 presents the profiles of six applications from chapter 5 experiments. The experimental setup is described chapter 5 section 5.3.2.



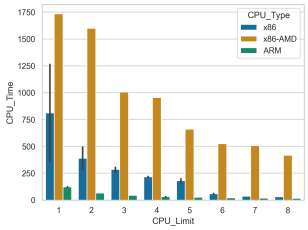
(a) Execution time for intel



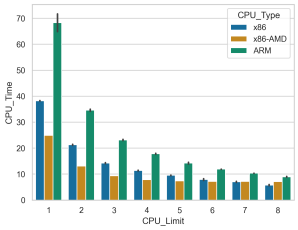
(b) Execution time for amd



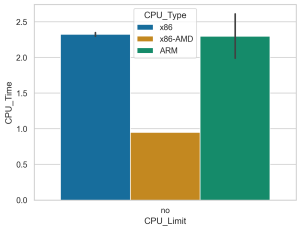
(c) Execution time for arm



(d) Execution time for time division

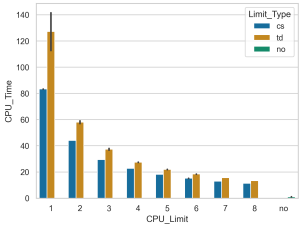


(e) Execution time for cpu-set

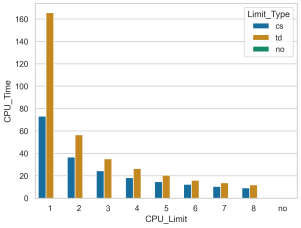


(f) Execution time for no limitation

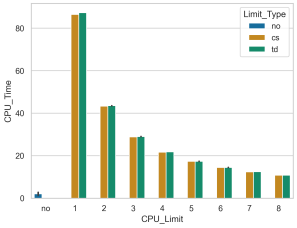
Figure B.1.: Conjugate Gradient (CG)



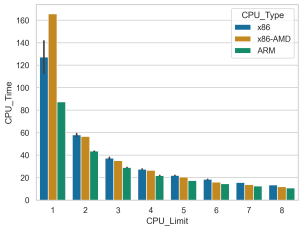
(a) Execution time for intel



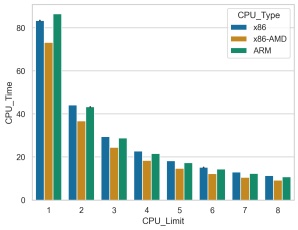
(b) Execution time for amd



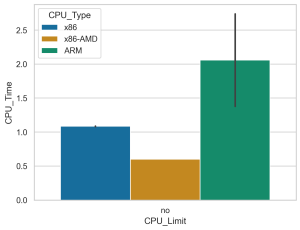
(c) Execution time for arm



(d) Execution time for time division

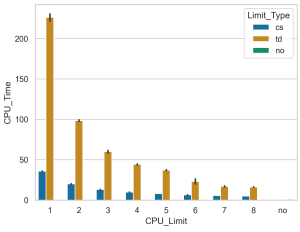


(e) Execution time for cpu-set

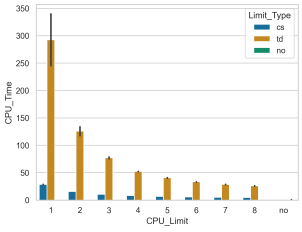


(f) Execution time for no limitation

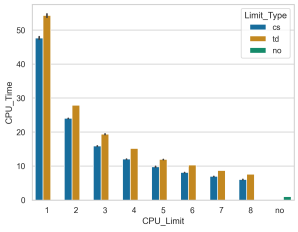
Figure B.2.: Embarrassingly Parallel (EP)



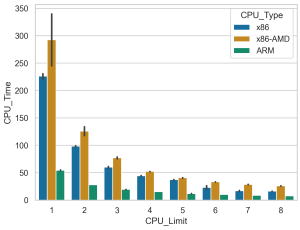
(a) Execution time for intel



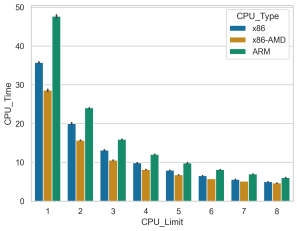
(b) Execution time for amd



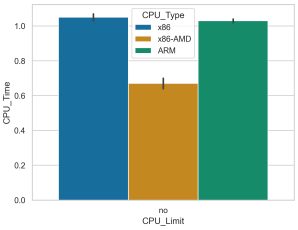
(c) Execution time for arm



(d) Execution time for time division

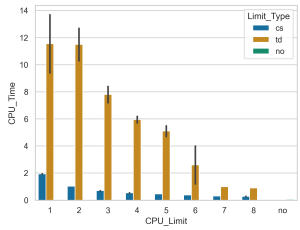


(e) Execution time for cpu-set

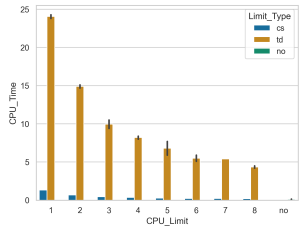


(f) Execution time for no limitation

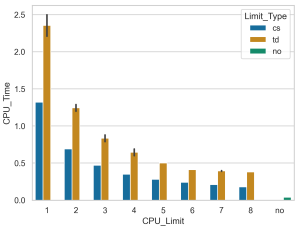
Figure B.3.: Discrete 3D fast Fourier Transform (FT)



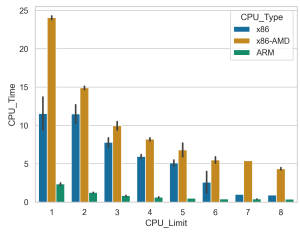
(a) Execution time for intel



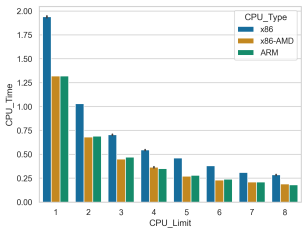
(b) Execution time for amd



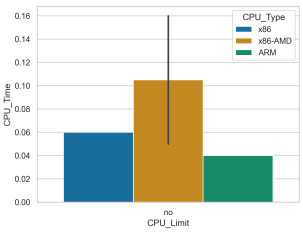
(c) Execution time for arm



(d) Execution time for time division

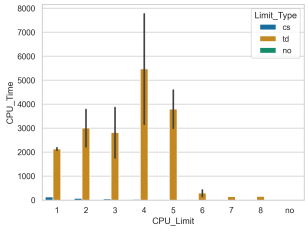


(e) Execution time for cpu-set

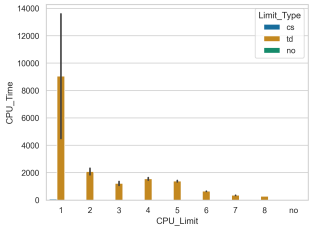


(f) Execution time for no limitation

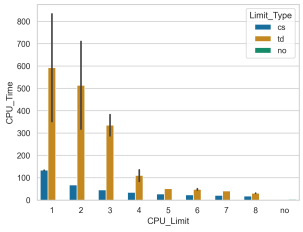
Figure B.4.: Integer Sort (IS)



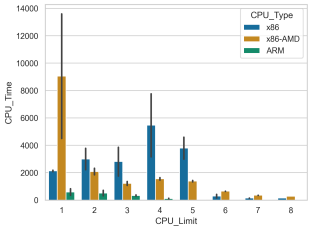
(a) Execution time for intel



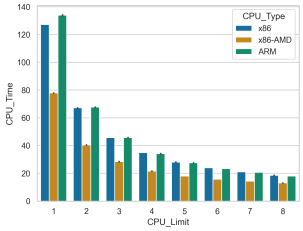
(b) Execution time for amd



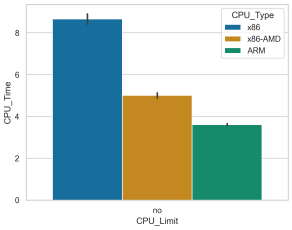
(c) Execution time for arm



(d) Execution time for time division

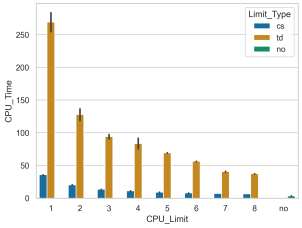


(e) Execution time for cpu-set

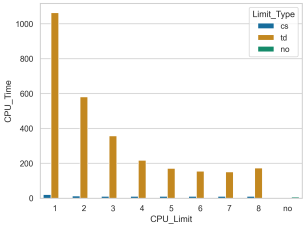


(f) Execution time for no limitation

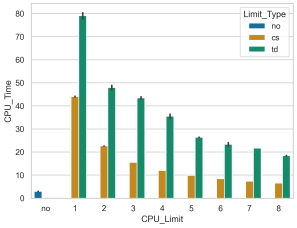
Figure B.5.: Lower-Upper Gauss-Seidel solver (LU)



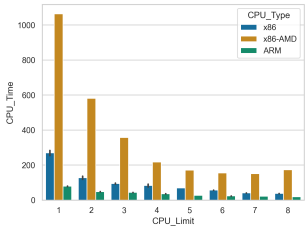
(a) Execution time for intel



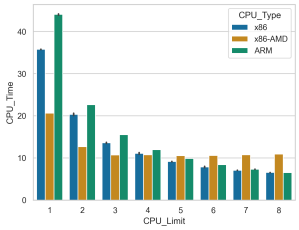
(b) Execution time for amd



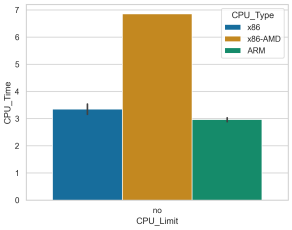
(c) Execution time for arm



(d) Execution time for time division



(e) Execution time for cpu-set



(f) Execution time for no limitation

Figure B.6.: Multi-Grid (MG)

Bibliography

[1] Gartner. Accessed: 2024-05-06. 2023. URL: https://www.gartner.com/en/newsroom/press-releases/11-13-2023-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-679-billion-in-20240.

[2] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. "A Break in the Clouds: Towards a Cloud Definition". In: *SIGCOMM Comput. Commun. Rev.* 39.1 (Dec. 31, 2009), pp. 50–55. ISSN: 0146-4833. DOI: 10.1145/1496091.1496100. URL: https://doi.org/10.1145/1496091.1496100 (visited on 06/27/2024).

[3] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi. "Cloud Computing — The Business Perspective". In: *Decision Support Systems* 51.1 (Apr. 1, 2011), pp. 176–189. ISSN: 0167-9236. DOI: 10.1016/j.dss.2010.12.006. URL: https://www.sciencedirect.com/science/article/pii/S0167923610002393 (visited on 05/06/2024).

[4] M. Mukherjee, L. Shu, and D. Wang. "Survey of Fog Computing: Fundamental, Network Applications, and Research Challenges". In: *IEEE Communications Surveys & Tutorials* 20.3 (2018), pp. 1826–1857. ISSN: 1553-877X. DOI: 10.1109/COMST.2018.2814571. URL: https://ieeexplore-ieee-org.proxy.bnl.lu/abstract/document/8314121 (visited on 06/27/2024).

[5] M. Laroui, B. Nour, H. Moungla, M. A. Cherif, H. Afifi, and M. Guizani. "Edge and fog computing for IoT: A survey on current research activities & future directions". In: *Computer Communications* 180 (2021), pp. 210–231.

[6] A. Maia, A. Boutouchent, Y. Kardjadja, M. Gherari, E. G. Soyak, M. Saqib, K. Boussekar, I. Cilbir, S. Habibi, S. O. Ali, W. Ajib, H. Elbiaze, O. Erçetin, Y. Ghamri-Doudane, and R. Glitho. "A Survey on Integrated Computing, Caching, and Communication in the Cloud-to-Edge Continuum". In: *Computer Communications* 219 (Apr. 1, 2024), pp. 128–152. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2024.03.005. URL: https://www.sciencedirect.com/science/article/pii/S0140366424000847 (visited on 04/22/2024).

[7] S. Moreschini, E. Younesian, D. Hästbacka, M. Albano, J. Hošek, and D. Taibi. "Edge to Cloud Tools: A Multivocal Literature Review". In: *Journal of Systems and Software* 210 (Apr. 1, 2024), p. 111942. ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111942. URL: https://www.sciencedirect.com/science/article/pii/S0164121223003370 (visited on 04/22/2024).

[8] J. Palomares, E. Coronado, C. Cervelló-Pastor, and S. Siddiqui. "Enabling Intelligence Inclusiveness in Edge to Cloud Continuum: Challenges and Opportunities". In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. 2023 IEEE 9th International Conference on Network Softwarization (NetSoft). June 2023, pp. 362–365. DOI: 10.1109/NetSoft57336.2023.10175414. URL: https://ieeexplore.ieee.org/abstract/document/10175414 (visited on 04/22/2024).

[9] K. Oztoprak, Y. K. Tuncel, and I. Butun. "Technological Transformation of Telco Operators towards Seamless IoT Edge-Cloud Continuum". In: *Sensors* 23.2 (2 Jan. 2023), p. 1004. ISSN: 1424-8220. DOI: 10.3390/s23021004. URL: https://www.mdpi.com/1424-8220/23/2/1004 (visited on 04/22/2024).

[10] P. Gkonis, A. Giannopoulos, P. Trakadas, X. Masip-Bruin, and F. D'Andria. "A Survey on IoT-Edge-Cloud Continuum Systems: Status, Challenges, Use Cases, and Open Issues". In: *Future Internet* 15.12 (12 Dec. 2023), p. 383. ISSN: 1999-5903. DOI: 10.3390/fi15120383. URL: https://www.mdpi.com/1999-5903/15/12/383 (visited on 04/22/2024).

[11] AWS. *AWS Wavelength*. Accessed: 2024-05-09. 2024. URL: https://aws.amazon.com/wavelength/.

[12] Google. *Google Distributed Cloud*. Accessed: 2024-05-09. 2024. URL: https://cloud.google.com/distributed-cloud.

[13] IBM. *IBM Edge Application Manager*. Accessed: 2024-05-09. 2024. URL: https://www.ibm.com/products/edge-application-manager.

[14] Akamai. *Akamai Edge Workers*. Accessed: 2024-05-09. 2024. URL: https://techdocs.akamai.com/edgeworkers/docs/welcome-to-edgeworkers.

[15] Intel. *Intel Smart Edge*. Accessed: 2024-05-09. 2024. URL: https://smart-edge-open.github.io/.

[16] AWS. *AWS Greengrass*. Accessed: 2024-05-09. 2023. URL: https://aws.amazon.com/greengrass/.

[17] Azure. *Azure IoT Edge*. Accessed: 2024-05-09. 2023. URL: https://learn.microsoft.com/en-us/azure/iot-edge/about-iot-edge?view=iotedge-1.4.

[18] Alibaba. *Alibaba IoT Edge*. Accessed: 2024-05-09. 2024. URL: https://www.alibabacloud.com/en/product/linkiotedge?_p_lc=1.

[19] AWS. *AWS Outpost*. Accessed: 2024-05-09. 2023. URL: https://aws.amazon.com/outposts/.

[20] Azure. *Azure Stack Edge*. Accessed: 2024-05-09. 2024. URL: https://azure.microsoft.com/en-us/products/azure-stack/edge.

[21] KubeEdge. *KubeEdge*. Accessed: 2024-05-09. 2024. URL: https://kubeedge.io/.

[22] Kubernetes. *k0s*. Accessed: 2024-05-09. 2024. URL: https://k0sproject.io/.

[23] Rancher. *k3s*. Accessed: 2024-05-09. 2024. URL: https://k3s.io/.

[24] Canonical. *MicroK8s*. Accessed: 2024-05-09. 2024. URL: https://microk8s.io/.

[25] L. Fondation. *EVE*. Accessed: 2024-05-09. 2024. URL: https://lfedge.org/projects/eve/.

[26] L. Fondation. *eKuiper*. Accessed: 2024-05-09. 2024. URL: https://ekuiper.org/.

[27] FogFlow. *FogFlow*. Accessed: 2024-05-09. 2024. URL: https://github.com/smartfog/fogflow.

[28] N. Computing. *Nearby One*. Accessed: 2024-05-09. 2024. URL: https://www.nearbycomputing.com/nearbyone/.

[29] N. Orchestrator. *Nebula Container orchestrator*. Accessed: 2024-05-09. 2024. URL: https://nebula-orchestrator.github.io/.

[30] Nomad. *Nomad*. Accessed: 2024-05-09. 2024. URL: https://www.nomadproject.io/.

[31] ONAP. *Open Network Automation Platform*. Accessed: 2024-05-09. 2024. URL: https://www.onap.org/.

[32] L. Fondation. *Open Horizon*. Accessed: 2024-05-09. 2024. URL: https://lfedge.org/projects/open-horizon/.

[33] R. Hat. *Red Hat Openshift at the edge*. Accessed: 2024-05-09. 2024. URL: https://www.redhat.com/en/technologies/cloud-computing/openshift/edge-computing.

[34] StarlingX. *StarlingX*. Accessed: 2024-05-09. 2024. URL: https://www.starlingx.io/.

[35] B. Cohen, G. Csatári, S. Huang, B. Jones, A. Lebre, D. Paterson, and I. Váncsa. *Edge Computing: Next Steps in Architecture, Design and Testing*. Accessed: 2024-05-09. 2024. URL: https://www.openstack.org/use-cases/edge-computing/edge-computing-next-steps-in-architecture-design-and-testing/.

[36] P. M. Mell and T. Grance. "The NIST Definition of Cloud Computing". In: *NIST* (Sept. 28, 2011). URL: https://www.nist.gov/publications/nist-definition-cloud-computing (visited on 05/13/2024).

[37] F. Tusa and S. Clayman. "End-to-End Slices to Orchestrate Resources and Services in the Cloud-to-Edge Continuum". In: *Future Generation Computer Systems* 141 (Apr. 1, 2023), pp. 473–488. ISSN: 0167-739X. DOI: 10.1016/j.future.2022.11.026. URL: https://www.sciencedirect.com/science/article/pii/S0167739X22003971 (visited on 04/22/2024).

[38] A. Al-Qamash, I. Soliman, R. Abulibdeh, and M. Saleh. "Cloud, Fog, and Edge Computing: A Software Engineering Perspective". In: *2018 International Conference on Computer and Applications (ICCA)*. IEEE, Aug. 2018.

[39] A. Ullah, T. Kiss, J. Kovács, F. Tusa, J. Deslauriers, H. Dagdeviren, R. Arjun, and H. Hamzeh. "Orchestration in the Cloud-to-Things Compute Continuum: Taxonomy, Survey and Future Directions". In: *Journal of Cloud Computing* 12.1 (Sept. 27, 2023), p. 135. ISSN: 2192-113X. DOI: 10.1186/s13677-023-00516-5. URL: https://doi.org/10.1186/s13677-023-00516-5 (visited on 04/22/2024).

[40] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. Goren, and C. Mahmoudi. *Fog Computing Conceptual Model*. NIST SP 500-325. Gaithersburg, MD: National Institute of Standards and Technology, Mar. 2018, NIST SP 500–325. DOI: 10.6028/NIST.SP.500-325. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-325.pdf (visited on 07/05/2024).

[41] J. Vergara, J. Botero, and L. Fletscher. "A Comprehensive Survey on Resource Allocation Strategies in Fog/Cloud Environments". In: *Sensors* 23.9 (9 Jan. 2023), p. 4413. ISSN: 1424-8220. DOI: 10.3390/s23094413. URL: https://www.mdpi.com/1424-8220/23/9/4413 (visited on 04/22/2024).

[42] Q. Duan, J. Huang, S. Hu, R. Deng, Z. Lu, and S. Yu. "Combining Federated Learning and Edge Computing Toward Ubiquitous Intelligence in 6G Network: Challenges, Recent Advances, and Future Directions". In: *IEEE Communications Surveys & Tutorials* 25.4 (2023), pp. 2892–2950. ISSN: 1553-877X. DOI: 10.1109/COMST.2023.3316615. URL: https://ieeexplore.ieee.org/abstract/document/10258360 (visited on 05/15/2024).

[43] H. G. Abreha, M. Hayajneh, and M. A. Serhani. "Federated Learning in Edge Computing: A Systematic Survey". In: *Sensors* 22.2 (2 Jan. 2022), p. 450. ISSN: 1424-8220. DOI: 10.3390/s22020450. URL: https://www.mdpi.com/1424-8220/22/2/450 (visited on 05/15/2024).

[44] R. Hussain and S. Zeadally. "Autonomous Cars: Research Results, Issues, and Future Challenges". In: *IEEE Communications Surveys & Tutorials* 21.2 (2019), pp. 1275–1313. ISSN: 1553-877X. DOI: 10.1109/COMST.2018.2869360. URL: https://ieeexplore.ieee.org/document/8457076 (visited on 05/23/2024).

[45] Y. Wang, Z. Su, N. Zhang, R. Xing, D. Liu, T. H. Luan, and X. Shen. "A Survey on Metaverse: Fundamentals, Security, and Privacy". In: *IEEE Communications Surveys & Tutorials* 25.1 (2023), pp. 319–352. ISSN: 1553-877X. DOI: 10.1109/COMST.2022.3202047. URL: https://ieeexplore.ieee.org/document/9880528 (visited on 05/23/2024).

[46] A. Clemm, M. T. Vega, H. K. Ravuri, T. Wauters, and F. D. Turck. "Toward Truly Immersive Holographic-Type Communication: Challenges and Solutions". In: *IEEE Communications Magazine* 58.1 (Jan. 2020), pp. 93–99. ISSN: 1558-1896. DOI: 10.1109/MCOM.001.1900272. URL: https://ieeexplore.ieee.org/document/8970173 (visited on 05/23/2024).

[47] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. "All One Needs to Know about Fog Computing and Related Edge Computing Paradigms: A Complete Survey". In: *Journal of Systems Architecture* 98 (Sept. 1, 2019), pp. 289–330. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2019.02.009. URL: https://www.sciencedirect.com/science/article/pii/S1383762118306349 (visited on 06/27/2024).

[48] A. J. Ferrer, J. M. Marques, and J. Jorba. "Ad-Hoc Edge Cloud: A Framework for Dynamic Creation of Edge Computing Infrastructures". In: *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. July 2019, pp. 1–7. DOI: 10.1109/ICCCN.2019.8847142.

[49] A. J. Ferrer, J. Panadero, J.-M. Marques, and J. Jorba. "Admission Control for Ad-hoc Edge Cloud". In: *Future Generation Computer Systems* 114 (2021), pp. 548–562. ISSN: 0167-739X.

[50] F. Bonomi, R. Milito, J. Zhu, and S. 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*. SIGCOMM '12: ACM SIGCOMM 2012 Conference. Helsinki Finland: ACM, Aug. 17, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. URL: https://dl.acm.org/doi/10.1145/2342509.2342513 (visited on 05/14/2024).

[51] S. Rac and M. Brorsson. "At the Edge of a Seamless Cloud Experience". In: *CoRR* abs/2111.06157 (2021). arXiv: 2111.06157. URL: https://arxiv.org/abs/2111.06157.

[52] S. Rac, R. Sanyal, and M. Brorsson. "A Cloud-Edge Continuum Experimental Methodology Applied to a 5G Core Study". In: *arXiv preprint arXiv:2301.11128* (2023).

[53] S. Rac and M. Brorsson. "Cost-Effective Scheduling for Kubernetes in the Edge-to-Cloud Continuum". In: *2023 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2023, pp. 153–160.

[54] S. Rac and M. Brorsson. "Cost-aware Service Placement and Scheduling in the Edge-Cloud Continuum". In: *ACM Trans. Archit. Code Optim.* 21.2 (Mar. 2024). ISSN: 1544-3566. DOI: 10.1145/3640823. URL: https://doi.org/10.1145/3640823.

[55] S. Rac and M. Brorsson. "Understanding CPU Limitation Mechanisms in Containerized Parallel Applications". In: (2024). Work under review.

[56] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante. "The Seminal Role of Edge-Native Applications". In: *2019 IEEE International Conference on Edge Computing (EDGE)*. 2019 IEEE International Conference on Edge Computing (EDGE). July 2019, pp. 33–40. DOI: 10.1109/EDGE.2019.00022. URL: https://ieeexplore-ieee-org.proxy.bnl.lu/abstract/document/8812200 (visited on 06/27/2024).

[57] K. Asanović and D. A. Patterson. "Instruction sets should be free: The case for risc-v". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[58] RISC-V Foundation. Accessed: 2024-05-09. 2024. URL: https://live-risc-v.pantheonsite.io/technical/technical-forums/.

[59] A. Dörflinger, M. Albers, B. Kleinbeck, Y. Guan, H. Michalik, R. Klink, C. Blochwitz, A. Nechi, and M. Berekovic. "A comparative survey of open-source application-class RISC-V processor implementations". In: *Proceedings of the 18th ACM international conference on computing frontiers*. 2021, pp. 12–20.

[60] R. A. Cooke and S. A. Fahmy. "A model for distributed in-network and near-edge computing with heterogeneous hardware". In: *Future Generation Computer Systems* 105 (2020), pp. 395–409. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.11.040.

[61] Z. Zhu, J. Zhang, J. Zhao, J. Cao, D. Zhao, G. Jia, and Q. Meng. "A Hardware and Software Task-Scheduling Framework Based on CPU+FPGA Heterogeneous Architecture in Edge Computing". In: *IEEE Access* 7 (2019), pp. 148975–148988. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2943179.

[62] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou. "A Survey on Edge Computing Systems and Tools". In: *Proceedings of the IEEE* 107.8 (Aug. 2019), pp. 1537–1562. ISSN: 1558-2256. DOI: 10.1109/JPROC.2019.2920341.

[63] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. "All one needs to know about fog computing and related edge computing paradigms: A complete survey". In: *Journal of Systems Architecture* 98 (Sept. 2019), pp. 289–330.

[64] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed. "Edge computing: A survey". In: *Future Generation Computer Systems* 97 (Aug. 2019), pp. 219–235.

[65] W. W. C. G. .-. W. S. I. S. Charter. *WebAssembly System Interface (WASI)*. Accessed: 2024-05-09. 2024. URL: https://wasi.dev/.

[66] C.-H. HONG and B. VARGHESE. "Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms". In: *ACM Computing Survey* 52.5 (2019). URL: https://doi.org/10.1145/3326066.

[67] A. Randazzo and I. Tinnirello. "Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way". In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. Oct. 2019, pp. 209–214. DOI: 10.1109/IOTSMS48152.2019.8939164.

[68] T. L. Foundation. *Open Container Initiative*. Accessed: 2024-05-09. 2024. URL: https://opencontainers.org/.

[69] Kubernetes. *Container Runtime Interface*. Accessed: 2024-05-09. 2024. URL: https://kubernetes.io/docs/concepts/architecture/cri/.

[70] Containerd. Accessed: 2024-05-09. 2024. URL: https://containerd.io/.

[71] CRI-O. Accessed: 2024-05-09. 2024. URL: https://cri-o.io/.

[72] B. Xu, S. Wu, J. Xiao, H. Jin, Y. Zhang, G. Shi, T. Lin, J. Rao, L. Yi, and J. Jiang. "Sledge: Towards Efficient Live Migration of Docker Containers". In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE. 2020, pp. 321–328.

[73] S. Kakati and M. Brorsson. "WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum". In: *2023 3rd International Conference on Intelligent Technologies (CONIT)*. 2023 3rd International Conference on Intelligent Technologies (CONIT). June 2023, pp. 1–8. DOI: 10.1109/CONIT59222.2023.10205816. URL: https://ieeexplore.ieee.org/abstract/document/10205816 (visited on 04/22/2024).

[74] Containerd. Accessed: 2024-05-09. 2024. URL: https://github.com/containerd/runwasi.

[75] Kubernetes. Accessed: 2024-05-09. 2024. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[76] OpenTelemetry. *Open Telemetry*. Accessed: 2024-05-09. 2024. URL: https://opentelemetry.io/.

[77] Prometheus. Accessed: 2024-05-09. 2024. URL: https://prometheus.io/.

[78] Jaeger. Accessed: 2024-05-09. 2024. URL: https://www.jaegertracing.io/.

[79] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu. "Serverless Computing: State-of-the-Art, Challenges and Opportunities". In: *IEEE Transactions on Services Computing* 16.2 (Mar. 2023), pp. 1522–1539. ISSN: 1939-1374. DOI: 10.1109/TSC.2022.3166553. URL: https://ieeexplore.ieee.org/document/9756233 (visited on 05/14/2024).

[80] A. Khandelwal, A. Kejariwal, and K. Ramasamy. "Le Taureau: Deconstructing the Serverless Landscape & A Look Forward". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2641–2650. ISBN: 9781450367356. DOI: 10.1145/3318464.3383130. URL: https://doi-org.proxy.bnl.lu/10.1145/3318464.3383130.

[81] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. "Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge". In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 265–279. ISBN: 9781450381536. DOI: 10.1145/3423211.3425680. URL: https://doi-org.proxy.bnl.lu/10.1145/3423211.3425680.

[82] Nulio. *Nuclio*. Accessed: 2024-05-09. 2024. URL: https://nuclio.io/.

[83] OpenFaaS. *OpenFaaS*. Accessed: 2024-05-09. 2024. URL: https://github.com/openfaas/faas.

[84] Knative. *Knative*. Accessed: 2024-05-09. 2024. URL: https://knative.dev/docs/.

[85] Kubernetes. *Kubernetes Autoscaler*. Accessed: 2024-05-09. 2024. URL: https://github.com/kubernetes/autoscaler?tab=readme-ov-file.

[86] B. Costa, J. Bachiega, L. R. de Carvalho, and A. P. F. Araujo. "Orchestration in Fog Computing: A Comprehensive Survey". In: *ACM Computing Surveys* 55.2 (Jan. 18, 2022), 29:1–29:34. ISSN: 0360-0300. DOI: 10.1145/3486221. URL: https://dl.acm.org/doi/10.1145/3486221 (visited on 05/13/2024).

[87] N. Kazemifard and V. Shah-Mansouri. "Minimum delay function placement and resource allocation for Open RAN (O-RAN) 5G networks". In: *Computer Networks* 188 (2021), p. 107809. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2021.107809.

[88] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76. ISSN: 1558-2256. DOI: 10.1109/JPROC.2014.2371999. URL: https://ieeexplore.ieee.org/abstract/document/6994333 (visited on 05/14/2024).

[89] A. C. Baktir, A. Ozgovde, and C. Ersoy. "How Can Edge Computing Benefit From Software-Defined Networking: A Survey, Use Cases, and Future Directions". In: *IEEE Communications Surveys Tutorials* 19.4 (2017), pp. 2359–2391. DOI: 10.1109/COMST.2017.2717482.

[90] S. D. A. Shah, M. A. Gregory, and S. Li. "Cloud-Native Network Slicing Using Software Defined Networking Based Multi-Access Edge Computing: A Survey". In: *IEEE Access* 9 (2021), pp. 10903–10924. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3050155. URL: https://ieeexplore.ieee.org/document/9317860 (visited on 05/14/2024).

[91] S. C. Ergen. "ZigBee/IEEE 802.15. 4 Summary". In: *UC Berkeley, September* 10.17 (2004), p. 11.

[92] P. Drahoš, E. Kučera, O. Haffner, R. Pribiš, and L. Beňo. "Trends in Industrial Networks Including APL, TSN, WiFi-6E and 5G Technologies". In: *2022 Cybernetics & Informatics (K&I)*. 2022 Cybernetics & Informatics (K&I). Sept. 2022, pp. 1–7. DOI: 10.1109/KI55792.2022.9925965. URL: https://ieeexplore-ieee-org.proxy.bnl.lu/abstract/document/9925965 (visited on 07/09/2024).

[93] E. Mozaffariahrar, F. Theoleyre, and M. Menth. "A Survey of Wi-Fi 6: Technologies, Advances, and Challenges". In: *Future Internet* 14.10 (10 Oct. 2022), p. 293. ISSN: 1999-5903. DOI: 10.3390/fi14100293. URL: https://www.mdpi.com/1999-5903/14/10/293 (visited on 05/09/2024).

[94] L. U. Khan, I. Yaqoob, M. Imran, Z. Han, and C. S. Hong. "6G Wireless Systems: A Vision, Architectural Elements, and Future Directions". In: *IEEE Access* 8 (2020), pp. 147029–147044. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3015289. URL: https://ieeexplore.ieee.org/abstract/document/9163104 (visited on 05/07/2024).

[95] sigfox. Accessed: 2024-05-09. 2024. URL: https://www.sigfox.com/.

[96] L. Alliance. Accessed: 2024-05-09. 2024. URL: https://lora-alliance.org/.

[97] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. "Mobile edge computing—A key technology towards 5G". In: *ETSI white paper* 11.11 (2015), pp. 1–16.

[98] S. Kekki, W. Featherstone, Y. Fang, P. Kuure, A. Li, A. Ranjan, D. Purkayastha, F. Jiangping, D. Frydman, G. Verin, et al. "MEC in 5G networks". In: *ETSI white paper* 28.2018 (2018), pp. 1–28.

[99] F. Giust, X. Costa-Perez, and A. Reznik. "Multi-access edge computing: An overview of ETSI MEC ISG". In: *IEEE 5G Tech Focus* 1.4 (2017), p. 4.

[100] R. Eidenbenz and T. Locher. "Task Allocation for Distributed Stream Processing". In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications. Apr. 2016, pp. 1–9. DOI: 10.1109/INFOCOM.2016.7524433. URL: https://ieeexplore-ieee-org.proxy.bnl.lu/abstract/document/7524433 (visited on 06/27/2024).

[101] Y.-K. Kwok and I. Ahmad. "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors". In: *ACM Comput. Surv.* 31.4 (Dec. 1, 1999), pp. 406–471. ISSN: 0360-0300. DOI: 10.1145/344588.344618. URL: https://doi.org/10.1145/344588.344618 (visited on 06/27/2024).

[102] I. Cohen, C. F. Chiasserini, P. Giaccone, and G. Scalosub. "Dynamic Service Provisioning in the Edge-Cloud Continuum With Bounded Resources". In: *IEEE/ACM Transactions on Networking* 31.6 (May 10, 2023), pp. 3096–3111. ISSN: 1063-6692. DOI: 10.1109/TNET.2023.3271674. URL: https://dl.acm.org/doi/10.1109/TNET.2023.3271674 (visited on 04/22/2024).

[103] HetBed. Accessed: 2022-06-07. 2022. URL: https://github.com/srac0/HetBed.

[104] 3GPP. *System architecture for the 5G System (5GS)*. Technical Specification (TS) 23.501. Version 17.2.0. 3GPP, Sept. 2021.

[105] Y. Siriwardhana, P. Porambage, M. Ylianttila, and M. Liyanage. "Performance Analysis of Local 5G Operator Architectures for Industrial Internet". In: *IEEE Internet of Things Journal* 7.12 (Dec. 2020), pp. 11559–11575. ISSN: 2327-4662. DOI: 10.1109/JIOT.2020.3024875.

[106] free5GC. Accessed: 2022-06-07. 2022. URL: https://www.free5gc.org/.

[107] UERANSIM. Accessed: 2022-06-07. 2022. URL: https://github.com/aligungr/UERANSIM.

[108] 3GPP. *Procedures for the 5G System (5GS)*. Technical Specification (TS) 23.502. Version 17.2.1. 3GPP, Sept. 2021.

[109] Kubernetes. *Device Plugins*. Accessed: 2024-05-09. 2024. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/.

[110] E. Goshi, M. Jarschel, R. Pries, M. He, and W. Kellerer. "Investigating Inter-NF Dependencies in Cloud-Native 5G Core Networks". In: *2021 17th International Conference on Network and Service Management (CNSM)*. Oct. 2021, pp. 370–374. DOI: 10.23919/CNSM52442.2021.9615565.

[111] O. Arouk and N. Nikaein. "Kube5G: A Cloud-Native 5G Service Platform". In: *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*. 2020, pp. 1–6. DOI: 10.1109/GLOBECOM42002.2020.9348073.

[112] H. C. C. de Resende, M. A. K. Schimuneck, C. B. Both, J. A. Wickboldt, and J. M. Marquez-Barja. "COPA: Experimenter-Level Container Orchestration for Networking Testbeds". In: *IEEE Access* 8 (2020), pp. 201781–201798. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3035619.

[113] A. Recse, R. Szabo, and B. Nemeth. "Elastic Resource Management and Network Slicing for IoT over Edge Clouds". In: *Proceedings of the 10th International Conference on the Internet of Things*. IoT '20. Malmö, Sweden: Association for Computing Machinery, 2020. ISBN: 9781450387583. DOI: 10.1145/3410992.3411015.

[114] H. Rahimi, A. Zibaeenejad, and A. A. Safavi. "A Novel IoT Architecture based on 5G-IoT and Next Generation Technologies". In: *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. Nov. 2018, pp. 81–88. DOI: 10.1109/IEMCON.2018.8614777.

[115] G. Patounas, X. Foukas, A. Elmokashfi, and M. K. Marina. "Characterization and Identification of Cloudified Mobile Network Performance Bottlenecks". In: *IEEE Transactions on Network and Service Management* 17.4 (Dec. 2020), pp. 2567–2583. ISSN: 1932-4537. DOI: 10.1109/TNSM.2020.3018538.

[116] N. Gupta, S. Sharma, P. K. Juneja, and U. Garg. "SDNFV 5G-IoT: A Framework for the Next Generation 5G enabled IoT". In: *2020 International Conference on Advances in Computing, Communication Materials (ICACCM)*. ISSN: 2642-7354. Aug. 2020, pp. 289–294. DOI: 10.1109/ICACCM50413.2020.9213047.

[117] I. Sarrigiannis, E. Kartsakli, K. Ramantas, A. Antonopoulos, and C. Verikoukis. "Application and Network VNF migration in a MEC-enabled 5G Architecture". In: *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. Sept. 2018, pp. 1–6. DOI: 10.1109/CAMAD.2018.8514943.

[118] M. Ejaz, T. Kumar, M. Ylianttila, and E. Harjula. "Performance and Efficiency Optimization of Multi-layer IoT Edge Architecture". In: *2020 2nd 6G Wireless Summit (6G SUMMIT)*. Mar. 2020, pp. 1–5. DOI: 10.1109/6GSUMMIT49458.2020.9083896.

[119] B. C. Şenel, M. Mouchet, J. Cappos, O. Fourmaux, T. Friedman, and R. McGeer. "EdgeNet: A Multi-Tenant and Multi-Provider Edge Cloud". In: *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. EdgeSys '21. Online, United Kingdom: Association for Computing Machinery, 2021, pp. 49–54. ISBN: 9781450382915. DOI: 10.1145/3434770.3459737.

[120] R.-A. Cherrueau, M. Delavergne, A. van Kempen, A. Lebre, D. Pertin, J. R. Balderrama, A. Simonet, and M. Simonin. "EnosLib: A Library for Experiment-Driven Research in Distributed Computing". In: *IEEE Transactions on Parallel and Distributed Systems* 33.6 (2022), pp. 1464–1477. DOI: 10.1109/TPDS.2021.3111159.

[121] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu. "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends". In: *2015 IEEE 8th International Conference on Cloud Computing*. ISSN: 2159-6190. June 2015, pp. 621–628. DOI: 10.1109/CLOUD.2015.88.

[122] Docker. Accessed: 2024-05-09. 2024. URL: https://www.docker.com.

[123] L. Fondation. Accessed: 2024-05-09. 2024. URL: https://opencontainers.org.

[124] ETCD. Accessed: 2024-05-09. 2024. URL: https://etcd.io.

[125] Prometheus. Accessed: 2024-05-09. 2024. URL: https://github.com/prometheus/pushgateway/.

[126] Bloomberg. Accessed: 2024-05-09. 2024. URL: https://github.com/bloomberg/goldpinger.

[127] P. Erdos and A. Renyi. "On random graphs I. Math". In: *Debrecen* 6 (1959), pp. 290–297.

[128] R. Xu, Z. Tu, H. Xiang, W. Shao, B. Zhou, and J. Ma. *CoBEVT: Cooperative Bird's Eye View Semantic Segmentation with Sparse Transformers*. July 2022. DOI: 10.48550/arXiv.2207.02202.

[129] C. I. King. Accessed: 2024-05-09. 2024. URL: https://github.com/ColinIanKing/stress-ng.

[130] Kubernetes. Accessed: 2024-05-09. 2024. URL: https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins.

[131] P. Lai, Q. He, J. Grundy, F. Chen, M. Abdelrazek, J. G. Hosking, and Y. Yang. "Cost-Effective App User Allocation in an Edge Computing Environment". In: *IEEE Transactions on Cloud Computing* (2020). Conference Name: IEEE Transactions on Cloud Computing, pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/TCC.2020.3001570.

[132] H. Li, J. Shen, L. Zheng, Y. Cui, and Z. Mao. "Cost-efficient scheduling algorithms based on beetle antennae search for containerized applications in Kubernetes clouds". en. In: *The Journal of Supercomputing* (Feb. 2023). ISSN: 1573-0484. DOI: 10.1007/s11227-023-05077-7. URL: https://doi.org/10.1007/s11227-023-05077-7 (visited on 03/21/2023).

[133] Z. Zhong and R. Buyya. "A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources". In: *ACM Transactions on Internet Technology* 20.2 (Apr. 2020), 15:1–15:24. ISSN: 1533-5399. DOI: 10.1145/3378447. URL: https://dl.acm.org/doi/10.1145/3378447 (visited on 03/20/2023).

[134] K. Kaur, F. Guillemin, V. Q. Rodriguez, and F. Sailhan. "Latency and network aware placement for cloud-native 5G/6G services". In: *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*. ISSN: 2331-9860. Jan. 2022, pp. 114–119. DOI: 10.1109/CCNC49033.2022.9700582.

[135] A. Marchese and O. Tomarchio. "Network-Aware Container Placement in Cloud-Edge Kubernetes Clusters". In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. May 2022, pp. 859–865. DOI: 10.1109/CCGrid54584.2022.00102.

[136] A. Marchese and O. Tomarchio. "Extending the Kubernetes Platform with Network-Aware Scheduling Capabilities". en. In: *Service-Oriented Computing*. Ed. by J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2022, pp. 465–480. ISBN: 978-3-031-20984-0. DOI: 10.1007/978-3-031-20984-0_33.

[137] L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong. "NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh". In: *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*. ISSN: 2641-9874. May 2021, pp. 1–9. DOI: 10.1109/INFOCOM42981.2021.9488670.

[138] L. Toka. "Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes". en. In: *Journal of Grid Computing* 19.3 (July 2021), p. 31. ISSN: 1572-9184. DOI: 10.1007/s10723-021-09573-z. URL: https://doi.org/10.1007/s10723-021-09573-z (visited on 07/15/2022).

[139] G. P. Mattia and R. Beraldi. "Leveraging Reinforcement Learning for online scheduling of real-time tasks in the Edge/Fog-to-Cloud computing continuum". In: *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*. ISSN: 2643-7929. Nov. 2021, pp. 1–9. DOI: 10.1109/NCA53618.2021.9685413.

[140] T. Pusztai, S. Nastic, A. Morichetta, V. C. Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang. "Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge". In: *2022 IEEE/ACM 15th International Conference on Utility*

and Cloud Computing (UCC). Dec. 2022, pp. 61–70. DOI: 10.1109/UCC56403.2022.00017.

[141] A. Orive, A. Agirre, H.-L. Truong, I. Sarachaga, and M. Marcos. "Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications". en. In: *Sensors* 22.5 (Jan. 2022). Number: 5 Publisher: Multidisciplinary Digital Publishing Institute, p. 1755. ISSN: 1424-8220. DOI: 10.3390/s22051755. URL: https://www.mdpi.com/1424-8220/22/5/1755 (visited on 07/15/2022).

[142] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo. "Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum". In: *IEEE Transactions on Parallel and Distributed Systems* 33.8 (Aug. 2022). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 1825–1840. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3128037.

[143] Linux Kernel. *Control Group*. Accessed: 2024-05-09. 2024. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.

[144] Kubernetes. *Control CPU Management Policies on the Node*. Accessed: 2024-05-09. 2024. URL: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/#static-policy.

[145] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes. "The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures". In: *Future Generation Computer Systems* 125 (2021), pp. 743–757.

[146] Ultralytics. *Ultralytics YOLOv8*. Accessed: 2024-05-09. 2024. URL: https://github.com/ultralytics/ultralytics.

[147] PyTorch Foundation. *PyTorch*. Accessed: 2024-05-09. 2024. URL: https://pytorch.org/.

[148] V. Millnert and J. Eker. "HoloScale: Horizontal and Vertical Scaling of Cloud Resources". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). Dec. 2020, pp. 196–205. DOI: 10.1109/UCC48980.2020.00038. URL: https://ieeexplore-ieee-org.proxy.bnl.lu/abstract/document/9302812 (visited on 06/25/2024).

[149] G. Quattrocchi, E. Incerto, R. Pinciroli, C. Trubiani, and L. Baresi. "Autoscaling Solutions for Cloud Applications under Dynamic Workloads". In: *IEEE Transactions on Services Computing* (2024), pp. 1–17. ISSN: 1939-1374. DOI: 10.1109/TSC.2024.3354062. URL: https://ieeexplore.ieee.org/abstract/document/10419899 (visited on 02/19/2024).

[150] A. A. Pramesti and A. I. Kistijantoro. "Autoscaling Based on Response Time Prediction for Microservice Application in Kubernetes". In: *2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)*. 2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA). Sept. 2022, pp. 1–6. DOI: 10.1109/ICAICTA56449.2022.9932943. URL: https://ieeexplore.ieee.org/abstract/document/9932943.

[151] Z. Bouflous, F. Haraka, M. Ouzzif, and K. Bouragba. "Enhanced Vertical Pod Auto Scaling with Decision Tree Regressor-Max in Kubernetes". In: *2024 11th International Conference on Wireless Networks and Mobile Communications (WINCOM)*. 2024 11th International Conference on Wireless Networks and Mobile Communications (WINCOM). July 2024, pp. 1–5. DOI: 10.1109/WINCOM62286.2024.10654970. URL: https://ieeexplore.ieee.org/abstract/document/10654970.

[152] E. C. de Lima, F. D. Rossi, M. C. Luizelli, R. N. Calheiros, and A. F. Lorenzon. "A Neural Network Framework for Optimizing Parallel Computing in Cloud Servers". In: *Journal of Systems Architecture* 150 (May 1, 2024), p. 103131. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2024.103131. URL: https://www.sciencedirect.com/science/article/pii/S1383762124000687 (visited on 07/04/2024).

[153] V. S. da Silva, E. C. de Lima, J. Schwarzrock, F. D. Rossi, M. C. Luizelli, A. C. S. Beck, and A. F. Lorenzon. "Synergistically Rebalancing the EDP of Container-Based Parallel Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 35.3 (Mar. 2024), pp. 484–498. ISSN: 1558-2183. DOI: 10.1109/TPDS.2024.3357353. URL: https://ieeexplore-ieee-org.proxy.bnl.lu/abstract/document/10412171 (visited on 06/25/2024).

[154] D. Balla, C. Simon, and M. Maliosz. "Adaptive Scaling of Kubernetes Pods". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium. Apr. 2020, pp. 1–5. DOI: 10.1109/NOMS47738.2020.9110428. URL: https://ieeexplore-ieee-org.proxy.bnl.lu/abstract/document/9110428 (visited on 07/22/2024).

[155] Z. Fan, R. Sen, P. Koutris, and A. Albarghouthi. "Automated Tuning of Query Degree of Parallelism via Machine Learning". In: *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. aiDM '20. New York, NY, USA: Association for Computing Machinery, June 14, 2020, pp. 1–4. ISBN: 978-1-4503-8029-4. DOI: 10.1145/3401071.3401656. URL: https://dl.acm.org/doi/10.1145/3401071.3401656.

[156] H. Huang, J. Rao, S. Wu, H. Jin, K. Suo, and X. Wu. "Adaptive Resource Views for Containers". In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '19. New York, NY, USA: Association for Computing Machinery, June 17, 2019, pp. 243–254. ISBN: 978-1-4503-6670-0. DOI:

10.1145/3307681.3325403. URL: https://dl.acm.org/doi/10.1145/3307681.3325403.

[157] L. Liu, H. Wang, A. Wang, M. Xiao, Y. Cheng, and S. Chen. "Mind the Gap: Broken Promises of CPU Reservations in Containerized Multi-tenant Clouds". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '21. New York, NY, USA: Association for Computing Machinery, Nov. 1, 2021, pp. 243–257. ISBN: 978-1-4503-8638-8. DOI: 10.1145/3472883.3486997. URL: https://dl.acm.org/doi/10.1145/3472883.3486997.

[158] H. Huang, Y. Zhao, J. Rao, S. Wu, H. Jin, D. Wang, S. Kun, and L. Pan. "Adapt Burstable Containers to Variable CPU Resources". In: *IEEE Transactions on Computers* 72.3 (Mar. 2023), pp. 614–626. ISSN: 1557-9956. DOI: 10.1109/TC.2022.3174480. URL: https://ieeexplore-ieee-org.proxy.bnl.lu/abstract/document/9773966 (visited on 07/16/2024).

[159] Y. Zhang, H. Zhang, Y. Wu, and H. Ma. "Joint Optimization of CPU Scaling and Core Sharing in Container Clouds". In: *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. 2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS). Dec. 2023, pp. 1031–1038. DOI: 10.1109/ICPADS60453.2023.00152. URL: https://ieeexplore.ieee.org/abstract/document/10476265#full-text-header.