Enabling Zero-Touch Cloud Edge Computing Continuum Management

Mohamed Mekki

Sorbonne University

Doctoral School of Informatics, Telecommunications and Electronics of Paris

EURECOM

Enabling Zero-Touch Cloud Edge Computing Continuum Management

---

Presented by **Mohamed Mekki**

Dissertation for Doctor of Philosophy in Information and Communication Engineering

Directed by **Prof. Adlen Ksentini**

The Jury committee is composed of:

Prof. Nadjib Ait-Saadi	(Université de Versailles Saint Quentin en Yvelines)	Reviewer
Prof. Pascal Lorenz	(Université Haute Alsace)	Reviewer
Prof. Miloud Bagaa	(Université du Québec à Trois-Rivières)	Examiner
Dr. Amadou Ba	(IBM)	Examiner
Dr. Nancy Perrot	(Orange labs)	Examiner
Prof. Jérome Harri	(EURECOM)	Jury President
Prof. Adlen Ksentini	(EURECOM)	Thesis Director

Sophia Antipolis, July 12th, 2024

Sorbonne université

Ecole doctorale Informatique, télécommunications et électronique de Paris

EURECOM

Enabling Zero-Touch Cloud Edge Computing Continuum Management

---

Présenté par **Mohamed Mekki**

Thèse de doctorat en Sciences de l'Information et de la Communication

Dirigée par **Prof. Adlen Ksentini**

Le jury est composé de :

Prof. Nadjib Ait-Saadi	(Université de Versailles Saint Quentin en Yvelines)	Rapporteur
Prof. Pascal Lorenz	(Université Haute Alsace)	Rapporteur
Dr. Miloud Bagaa	(Université du Québec à Trois-Rivières)	Examinateur
Dr. Amadou Ba	(IBM)	Examinateur
Dr. Nancy Perrot	(Orange labs)	Examinateur
Prof. Jérome Harri	(EURECOM)	Président du Jury
Prof. Adlen Ksentini	(EURECOM)	Directeur de Thèse

Abstract

The maturation of cloud computing and edge computing infrastructure provisioning and management has given rise to what is termed as Cloud Edge Computing Continuum (CECC). CECC facilitates a seamless spectrum for applications' deployment and migration between centralized cloud infrastructures and decentralized edge infrastructure. This evolution has spurred new use cases across industries, including Industrial Internet of Things (IIoT), autonomous vehicles, and augmented reality, all of which benefit from this distributed architecture. These use cases demand a combination of scalability and storage from massive data centers typical of traditional cloud computing, as well as the low latency and high bandwidth offered by edge computing infrastructures. The development and deployment of applications to fully leverage CECC are made possible by several factors: advancements in application deployment technologies, primarily virtualization and containerization; a shift in application architecture and development methodology from monolithic to microservices architectures; and innovations in networking technologies such as 5G mobile networks, which provide larger bandwidth and reduced latency.
Efficiently orchestrating applications within the CECC framework, especially those comprised of multiple microservices spread across the continuum, is essential for meeting performance requirements and optimizing infrastructure resource utilization. This thesis proposes solutions to enable zero-touch management of CECC. These solutions are categorized into three primary aspects of automated monitoring: Monitoring and metric collection, which involves continuously monitoring and collecting metrics related to both the running applications and the underlying infrastructure, providing a comprehensive view of the managed CECC; utilizing collected metrics to gain insights into the functioning of applications and infrastructure, including profiling both applications and infrastructure components; and making informed decisions regarding application placement, migration, and resource scaling. These decisions are driven by algorithms or machine learning models that generate actionable insights based on monitoring metrics and profiling data. By addressing these aspects, the proposed solutions aim to automate the management of CECC applications, facilitating seamless orchestration and optimization of resources.
In the first contribution, we propose a novel monitoring system for multi-domain services. The framework is technology agnostic as abstract the underlying technologies Specificities using a unified structure for Key Performance Indicator (KPI). The monitoring system is scalable and support high number of running services in parallel. To our best knowledge, it is the first monitoring system to monitor end-to-end network slices including Radio Access Network (RAN), Core Network (CN) and Cloud/Edge domain.
In the second contribution of this thesis, We present the results of our experimental study aiming to detect if a tenant's configuration allows running its service optimally. To this aim, we run several experiments on a cloud-native platform, using different types of applications under different resource configurations. The obtained results provide insights on how to detect and correct performance degradation due to misconfiguration of the service resource.
Then, in our third contribution, we move towards the decision making part of a Cloud Edge Computing Continuum Manager (CECCM). We propose a novel ZSM framework featuring a fine-granular computing resource scaler in a cloud-native environment. The proposed scaler algorithm uses AI/ML models to predict micro service performances. More specifically, we use eXtreme Gradient Boosting (XGBoost) as ML algorithm to predict any violations related to the latency performance of running applications; if a service degradation is detected, then a root-cause analysis is conducted using an eXplainable Artificial Intelligence (XAI) module. Based on the XAI output, the proposed framework scales only the needed resources (i.e., CPU or memory)

to overcome the service degradation. The proposed framework and resource scheduler have been implemented on top of a cloud-native platform. The results shows that the introduction of the XAI allows the scheduler to achieve the same service quality as the default scheduler but with lesser resources allocated to applications.
Finally, in the last contribution of the thesis, we propose an architecture of CECC Application Orchestrator. The architecture leverages applications and infrastructures profiling to efficiently manage the CECC applications. This profiling is done using the monitoring information, including energy metrics and carbon intensity of infrastructures. We define a structural model for the application profile, the goal of this model is to represent the current and requirements of the application in terms of compute and network resources. The representation is structured to be used by the CECCM to decide on the Life-Cycle Management (LCM) of applications including placement, migration and resources reconfiguration. The decision of the CECCM aims to minimize the carbon footprint and cost of the CECC deployment.

Résumé

La maturation de la provision et de la gestion des infrastructures de cloud computing et d'edge computing a donné naissance à ce que l'on appelle le Cloud Edge Computing Continuum (CECC). Le CECC facilite un spectre fluide pour le déploiement et la migration d'applications entre les infrastructures cloud centralisées et les infrastructures edge décentralisées. Cette évolution a engendré de nouveaux cas d'utilisation dans divers secteurs, notamment l'Internet Industriel des Objets (IIoT), les véhicules autonomes et la réalité augmentée, qui bénéficient tous de cette architecture distribuée. Ces cas d'utilisation exigent une combinaison de scalabilité et de stockage provenant des centres de données massifs typiques du cloud computing traditionnel, ainsi que de la faible latence et de la bande passante élevée offertes par les infrastructures edge computing. Le développement et le déploiement d'applications pour exploiter pleinement le CECC sont rendus possibles par plusieurs facteurs : les avancées dans les technologies de déploiement d'applications, principalement la virtualisation et la conteneurisation ; un changement dans l'architecture et la méthodologie de développement des applications, passant des architectures monolithiques aux microservices ; et les innovations dans les technologies de réseau telles que les réseaux mobiles 5G, qui offrent une bande passante plus grande et une latence réduite.

Orchestrer efficacement des applications dans le cadre du CECC, en particulier celles composées de plusieurs microservices répartis sur le continuum, est essentiel pour répondre aux exigences de performance et optimiser l'utilisation des ressources d'infrastructure. Cette thèse propose des solutions pour permettre la gestion automatisée du CECC. Ces solutions sont catégorisées en trois aspects principaux de surveillance automatisée : la collecte de données de surveillance et de métriques, qui implique la surveillance continue et la collecte de métriques liées aux applications en cours d'exécution et à l'infrastructure sous-jacente, fournissant une vue complète du CECC géré ; l'utilisation des métriques collectées pour comprendre le fonctionnement des applications et de l'infrastructure, y compris le profilage des applications et des composants d'infrastructure ; et la prise de décisions éclairées concernant le placement, la migration et l'adaptation des ressources des applications. Ces décisions sont guidées par des algorithmes ou des modèles d'apprentissage automatique qui génèrent des idées exploitables basés sur les métriques de surveillance et les données de profilage. En abordant ces aspects, les solutions proposées visent à automatiser la gestion des applications CECC, facilitant ainsi l'orchestration et l'optimisation transparentes des ressources.

Dans la première contribution, nous proposons un système de supervision novateur pour les services multi-domaines. Notre systeme est technologiquement agnostique car il abstrait les spécificités des technologies sous-jacentes en utilisant une structure unifiée pour les Key Performance Indicator (KPI). Le système de surveillance est évolutif et prend en charge un grand nombre de services en cours d'exécution en parallèle. À notre connaissance, il s'agit du premier système de surveillance à contrôler des tranches de réseau de bout en bout, y compris le Radio Access Network (RAN), le Core Network (CN) et le domaine Cloud/Edge.

Dans la deuxième contribution de cette thèse, nous présentons les résultats de notre étude expérimentale visant à détecter si la configuration d'un locataire permet d'exécuter son service de manière optimale. À cette fin, nous avons mené plusieurs expériences sur une plateforme cloud-native, en utilisant différents types d'applications avec différentes configurations de ressources. Les résultats obtenus fournissent des informations sur la manière de détecter et de corriger la dégradation des performances due à une mauvaise configuration des ressources du service.

Ensuite, dans notre troisième contribution, nous nous tournons vers la prise de décision d'un Cloud Edge Computing Continuum Manager (CECCM). Nous proposons un nouveau cadre ZSM doté d'un ajusteur de ressources informatiques à granularité fine dans un environnement cloud-native. L'algorithme de l'ajusteur proposé utilise des modèles d'IA/ML pour prédire les performances des microservices. Plus précisément, nous utilisons le Boosting Extrême par Gradient (*eXtreme Gradient Boosting* ou XGBoost) comme algorithme de ML pour prédire toute violation liée aux performances de latence des applications en cours d'exécution ; si une dégradation du service est détectée, une analyse des causes profondes est effectuée à l'aide d'un module eXplainable Artificial Intelligence (XAI). Sur la base de la sortie XAI, l'approche proposée ajuste uniquement les ressources nécessaires (c'est-à-dire CPU ou mémoire) pour surmonter la dégradation du service. Cette approche et l'ordonnanceur de ressources proposés ont été implémentés sur une plateforme cloud-native. Les résultats montrent que l'introduction du XAI permet à l'ordonnanceur d'atteindre la même qualité de service que l'ordonnanceur par défaut mais avec moins de ressources allouées aux applications.
Enfin, dans la dernière contribution de la thèse, nous proposons une architecture de l'Orchestration d'Applications CECC. L'architecture exploite le profilage des applications et des infrastructures pour gérer efficacement les applications CECC. Ce profilage est réalisé à l'aide des informations de surveillance, y compris les métriques énergétiques et l'intensité carbone des infrastructures. Nous définissons un modèle structurel pour le profil de l'application ; l'objectif de ce modèle est de représenter les capacités actuelles et les exigences de l'application en termes de ressources de calcul et de réseau. La représentation est structurée pour être utilisée par le CECCM afin de décider de la gestion du cycle de vie des applications, y compris le placement, la migration et la reconfiguration des ressources. Les décisions du CECCM visent à minimiser l'empreinte carbone et le coût du déploiement CECC.

Acknowledgements

This thesis summarizes the research I have conducted over the past three and a half years. Achieving this milestone would not have been possible without the support of several key people in my life.

I am profoundly grateful to my parents. My father was deeply invested in everything I pursued, consistently motivating me to pursue excellence. Although he passed away two months before my thesis defense, his support and belief in my potential remain the foundation of all my achievements. My mother, with her constant support and love, has been a source of strength and encouragement at every stage of my journey. I am forever indebted to both of them.

Since the final year of my engineering degree, I have had the chance of working with Professor Adlen Ksentini, who offered me the opportunity to pursue a PhD. His expertise and mentorship throughout my time at EURECOM have been essential to the successful completion of my thesis. I am truly thankful for the opportunity to learn and grow under his guidance.

Last but not least, I want to thank everyone who has influenced my personal and professional life, including my extended family, friends, and colleagues. Their support and contributions have made a lasting impact on me and this work.

Contents

1 General Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 1
- 1.3 Thesis Challenges and Contributions . . . 2
- 1.4 Thesis Structure . . . 6

2 Background **7**
- 2.1 Introduction . . . 7
- 2.2 CECC Management Landscape . . . 7
  - 2.2.1 Cloud Edge Computing . . . 7
  - 2.2.2 Network Function Virtualization . . . 9
- 2.3 Zero-touch Service Management . . . 12
  - 2.3.1 ETSI ZSM Reference Architecture . . . 12
  - 2.3.2 Closed Control Loops . . . 14
- 2.4 Machine Learning (ML) and eXplainable AI (XAI) . . . 16
  - 2.4.1 Transparent machine learning Models . . . 17
  - 2.4.2 Post-hoc explainability techniques . . . 18
- 2.5 Conclusion . . . 20

3 A scalable monitoring framework for network slicing in 5G and beyond mobile networks **21**
- 3.1 Introduction . . . 21
- 3.2 Related works . . . 22
- 3.3 A monitoring framework for end-to-end network slices . . . 24
  - 3.3.1 Architecture . . . 24
  - 3.3.2 Data Transfer . . . 31
  - 3.3.3 Data presentation . . . 34
  - 3.3.4 Data privacy and isolation . . . 34
- 3.4 Performance evaluation . . . 35
  - 3.4.1 Testing environment . . . 35
  - 3.4.2 System Scalability . . . 36
  - 3.4.3 End-to-end messages latency . . . 39
- 3.5 Conclusion . . . 40

4 Microservices Configurations and the Impact on the Performance in Cloud Native Environments **41**
- 4.1 Introduction . . . 41
- 4.2 Related Works . . . 42

4.3 Motivation . . . 43
4.4 Performance evaluation . . . 44
4.4.1 Testing environment . . . 44
4.4.2 Obtained results . . . 45
4.4.3 Discussion . . . 49
4.5 Conclusion . . . 52

5 XAI-Enabled Fine Granular Vertical Resources Autoscaler 53
5.1 Introduction . . . 53
5.2 Related Work . . . 54
5.3 Design and specification of the proposed autoscaler framework . . . 55
5.3.1 The envisioned ZSM architecture . . . 56
5.3.2 Analytical Engine (AE) . . . 57
5.4 Performance evaluation . . . 60
5.4.1 Testing Environment . . . 60
5.4.2 Performance results . . . 60
5.5 Conclusion . . . 64

6 Energy Aware Applications Deployment on Cloud Edge Continuum Infrastructure Using Applications Profiles 65
6.1 Introduction . . . 65
6.2 Related works . . . 66
6.3 Application profiling . . . 66
6.4 Design and specification of the proposed application scheduling framework . . . 69
6.4.1 Architecture . . . 69
6.4.2 Network solutions for seamless migration between the cloud edge and far edge . . . 71
6.5 Application scheduling taking into account the cost and carbon footprint of the deployment . . . 72
6.5.1 Objective function . . . 74
6.5.2 Constraints . . . 75
6.5.3 Application scheduling algorithm . . . 76
6.6 Results . . . 78
6.6.1 Test results comparison . . . 79
6.6.2 Discussion . . . 80
6.7 Conclusion . . . 81

7 Conclusion & Perspectives 85
7.1 Conclusion . . . 85
7.2 Future Perspectives . . . 86
7.2.1 Profile creation and management for applications through collaboration . . . 86
7.2.2 Serverless computing and WebAssembly edge modules management . . . 86
7.2.3 Intent Driven Application Management . . . 87

8 Résumé 89
8.1 Contexte de la Thèse . . . 89
8.2 Motivation . . . 89
8.3 Les contributions de la Thèse . . . 90
8.3.1 Supervision de bout en bout des applications et services multi-domaines . . . 91

8.3.2 Profilage des applications dans des environnements cloud-native 92
8.3.3 Mise à l'échelle automatique des ressources des applications dans des environnements cloud-native 93
8.3.4 Gestion du cycle de vie des applications prenant en compte l'énergie dans le cloud edge computing continuum 94
8.4 Conclusion 95

List of Figures

1.1 PhD journey 3

2.1 NFV-MANO framework reference architecture with support for containers [6] . . 10
2.2 Cloud-native lightweight slice orchestration framework (Cloud related components) [8] 11
2.3 ZSM framework reference architecture [10] 13
2.4 Closed-control loop components: MS, AE, and DE [11] 15
2.5 Explainability techniques classification [13] 17

3.1 Network slicing architecture 24
3.2 Architecture of the monitoring system 27
3.3 RAN KPIs collection sub-system 30
3.4 KPI message structure 32
3.5 The RAM consumption of the system as a whole in relation to the number of slices and the polling interval 36
3.6 The RAM consumption of the slice-specific data collectors in relation to the number of slices and the polling interval 37
3.7 The RAM consumption of the data collection servers (data brokers) in relation to the number of slices and the polling interval 37
3.8 The CPU consumption of the system as a whole in relation to the number of slices and the polling interval 38
3.9 The CPU consumption of the slice-specific data collectors in relation to the number of slices and the polling interval 39
3.10 The RAM consumption of the data collection servers (data brokers) in relation to the number of slices and the polling interval 39
3.11 The end-to-end messages latency in relation to the number of slices and the polling interval 40

4.1 Trade-off between resource efficiency and application's performance 44
4.2 The components of the testbed 45
4.3 Web Server's 95% latency in relation to the allocated CPU 46
4.4 Web Server's 95% latency statistical distribution 47
4.5 AMF's 95% registration time in relation to the allocated CPU 48
4.6 5G AMF's 95% UE registration time statistical distribution 48
4.7 RabbitMQ broker's 95% message delay in relation to the allocated CPU 49
4.8 RabbitMQ broker's 95% message delay statistical distribution 50
4.9 RabbitMQ last recorder relative memory before pod failure 51
4.10 Database's 95% operation duration statistical distribution 51

5.1 Zero touch network framework architecture 55
5.2 Web Server's latency in relation to the allocated CPU 57
5.3 Web Server's latency statistical distribution 58
5.4 Shapley values provided by the SHAP method 60
5.5 Evolution of CPU and allocated RAM using the XAI-assisted autoscaler 61
5.6 Evolution of CPU and allocated RAM using the autoscaler without XAI module's assistance 62
5.7 Evolution of the application performance (response time) using the autoscaler with and without XAI module's assistance 63
5.8 Highest values of CPU and RAM allocated to an instance of the applications in relation to the number of concurrent clients 63
5.9 Mean response time in relation to the number of concurrent clients 64

6.1 Proposed application profile 67
6.2 choice of the load threshold 68
6.3 The overall architecture of the Cloud Edge Computing Continuum Applications Orchestrator 69
6.4 Dynamic application deployment on the CECC infrastructure over time 75
6.5 The gap in Carbon Footprint (%) produced by the models for a duration of 10, 100, and 1000 80
6.6 The gap in the Cost (%) produced by the models for a duration of 10, 100 and 1000 81
6.7 The gap in Carbon Footprint (%) produced by the different heuristic configurations for a duration of 10, 100 and 1000 82
6.8 The gap in the Cost (%) produced by the different heuristic configurations for a duration of 10, 100 and 1000 83
6.9 The execution time of the models for a duration of 10, 100 and 1000 84

7.1 WASM application deployment on edge devices 87

8.1 Parcours de la thèse 91

List of Tables

3.1 Comparison between [27], [30], [31], and our monitoring system 25
3.2 RAN KPIs list 29
3.3 NFVO KPIs list 31
3.4 KPI message fields description 32
3.5 The implemented components and the used technologies 35

6.1 Measured downtime from the application perspective during migration. 72
6.2 Summary of Notations & Variables. 73

Acronyms

3GPP 3rd Generation Partnership Project.

5G 5th generation of mobile network.

AE Analytical Engine.

AI Artificial Intelligence.

AMF Access and Mobility Management Function.

API Application Programming Interface.

AppD Application Descriptor.

BSS Business Support System.

CCM CIS Cluster Management.

CECC Cloud Edge Computing Continuum.

CECCM Cloud Edge Computing Continuum Manager.

CIR Container Image Registry.

CIS Container Infrastructure Service.

CISM Container Infrastructure Service Management.

CN Core Network.

CNCF Cloud Native Computing Foundation.

CNF Cloud-native or Container Network Function.

CQI Channel Quality Indicator.

CSMF Communication Service Management Function.

CU Centralized Unit.

DB Database.

DE Decision Engine.

DL Downlink.

DMO Domain-specific Management and Orchestration.

DNS Directory Name Service.

DU Distributed Unit.

E2E End-to-End.

EM Element Manager.

eMBB enhanced Mobile BroadBand.

eNB evolved Node B.

ESSO Edge Sub-Slice Orchestrator.

ETSI European Telecommunications Standards Institute.

FaaS Function as a Service.

FL Federated learning.

GBM Gradient Boosting Machines.

gNB next Generation Node B.

GSMA Global System for Mobile Communications.

GST Generic Slice Template.

IBN Intent-based networking.

ICT Information and Communication Technology.

IDMO Inter-Domain Management and Orchestration.

IIoT Industrial Internet of Things.

IMSI International Mobile Subscriber Identity.

IoT Internet of Things.

ISM In-Slice Manager.

KPI Key Performance Indicator.

LCM Life-Cycle Management.

LLM large language model.

LSTM Long Short-Term Memory.

MANO Management and Orchestration.

MEC Multi-access Edge Computing.

ML Machine Learning.

mMTC massive Machine-Type Communication.

MS Monitoring System.

NBI Northbound API.

NFV Network function virtualization.

NFVI NFV Infrastructure.

NFVO Network Function Virtualization Orchestrator.

NR New Radio.

NS Network Slice.

NSA NonStandalone.

NSD Network Service Descriptor.

NSMF Network Slice Management Function.

NSSMF Network Sub Slice Management Function.

NSSO Network Sub-Slice Orchestrators.

NSST Network Sub Slice Templates.

NST Network Slice Template.

O-RAN Open RAN.

OAI OpenAirInterface.

OCI Open Container Initiative.

OOM Out Of Memory.

OSS Operations Support System.

PNF Physical Network Function.

PRB Physical Resource Block.

QoS Quality of Service.

RAN Radio Access Network.

RANO Radio Access Network Orchestrator.

RL Reinforcement Learning.

RNN Recurrent Neural Network.

RRU Radio Remote Unit.

SA Standalone.

SBA Service Based Architecture.

SDN Software-Defined Networking.

SHAP SHapley Additive exPlanations.

SLA Service Level Agreement.

SLO Service Level Objective.

SO Slice Orchestrator.

SOA Service-Oriented Architecture.

SVM Support Victor Machine.

UE User Equipment.

URLLC Ultra-Reliable and Low-Latency Communication.

VIM Virtualized Infrastructure Manager.

VLAN Virtual Local Area network.

VM Virtual Machines.

VNF Virtual Network Function.

VNFD VNF Descriptor.

VNFM Virtual Network Function Manager.

VPN Virtual Private Network.

WAN Wide Area Network.

WASI WebAssembly System Interface.

WASM WebAssembly.

WIM WAN Infrastructure Manager.

XAI eXplainable Artificial Intelligence.

XGBoost eXtreme Gradient Boosting.

ZSM Zero-touch Service Management.

Chapter 1

General Introduction

1.1 Context

Cloud Edge Computing Continuum (CECC) have seen light thanks to the paradigm shift in applications development and deployment, integrating the scalability and resource availability of cloud computing with the real-time processing capabilities of edge computing. This continuum is made possible due to the maturation of the cloud and edge computing provisioning and management, providing a spectrum wherein workloads can seamlessly migrate between centralized cloud infrastructures and decentralized edge and far edge infrastructures or devices based on factors such as compute resources requirements, latency, and bandwidth constraints. At one end of this continuum lies traditional cloud computing, characterized by massive data centers and centralized processing resources, offering resources scalability and storage capabilities. Conversely, at the opposite end, edge computing leverages local computing infrastructures composed of small data centers or far-edge devices. Reducing latency and enhancing responsiveness for time-sensitive applications. Additionally, edge and far edge devices may be mobile or volatile, meaning that they are not available permanently due to either battery restrictions or mobility, such as for drone-based far edge devices or single board computers powered by batteries.
The continuum between these extremes enables new architecture that takes advantage of the dynamic distribution of computing tasks across cloud and edge nodes based on workload demands and network conditions. This paradigm encourages the development of innovative applications spanning multiple domains, such as Internet of Things (IoT), autonomous systems, and augmented reality. In this case, the seamless orchestration of resources across the cloud edge continuum is crucial for meeting the performance requirements and enabling pervasive connectivity for the deployed applications on the one hand and making efficient use of the underlying infrastructure resources.

1.2 Motivation

A CECC infrastructure spans over multiple locations and domains, including public cloud, dedicated edge cloud and computing capable devices. Here, a Cloud Edge Computing Continuum Manager (CECCM) is a key element to handle application's Life-Cycle Management (LCM) and manage the federated infrastructure resources. The CECCM should be self-managed and self-configured to enable zero-touch management and configuration by making decisions based on the collected monitoring data about the application behaviour and infrastructure resources usage as well as the network status. These decisions should ensure the respect of application's requirements based on the available infrastructure capabilities. To achieve this end, the first

building block of an CECCM is a monitoring system that offers visibility over the deployed application's behavior and infrastructure resources. Indeed, the monitoring system should cover the entirety of the CECC infrastructure.
Then, to efficiently manage the applications, the CECCM should understand the applications' behavior and requirements. We denote as an application profile the representation of the application's behavior, performance, resource usage, and dependencies within the deployment environment. By profiling cloud applications, the CECCM can optimize the placement of the applications to improve their performance and optimize the infrastructure's resource usage. Profiling techniques may include monitoring system metrics and analyzing network traffic patterns. While, Artificial Intelligence (AI) techniques can be used in order to predict the future needs of the application in terms of compute resources and network requirements. Application's LCM procedures include mainly the initial placement, the resources scaling, and migration of the applications. Since the CECC management system makes decisions based on the collected metrics and data on the application behavior, it uses models based on Machine Learning (ML) or mathematical optimization problems. It is important that the decisions of the system can be trusted to ensure the respect of the Service Level Agreement (SLA) required by the applications. Finally, each compute location of the CECC infrastructure is composed of nodes that consume electricity and water to function as well as for cooling. Based on the sources of energy used by the infrastructure nodes, the CECC infrastructure and the workloads running on top of it produce a carbon footprint. Indeed, recently, more and more governments are taking action towards reducing carbon emissions; for instance, the European Union aims to be climate-neutral by 2050; this objective is at the heart of The European Green Deal [1]. Communication technology is no exception, as in 2021, an ACM technology brief [2] estimated that the Information and Communication Technology (ICT) sector contributed between 1.8% and 3.9% of global carbon emissions. This motivates the availability of green energy sources to supply the compute nodes, producing the necessity for making carbon-aware decisions when managing the applications deployment and the infrastructures that are part of the CECC.

1.3 Thesis Challenges and Contributions

This thesis aims to enable the zero-touch management of multi-domain CECC deployments. We started by tackling the challenge of end-to-end unified monitoring of multi-domain deployments where the application's microservices (or cloud-native applications) can be deployed in different infrastructures using different technologies. We propose a novel monitoring system utilizing metrics collectors designed to gather sub-service applications' metrics in a deployment region (or infrastructure). These metrics are then aggregated to offer an End-to-End (E2E) overview on service Key Performance Indicator (KPI), abstracting away the underlying technological details. Next, we use the monitoring data to understand the behavior of the applications under different scenarios. This allows us to build datasets about the resource requirements of various types of applications under different loads. Thus providing a basic profile for the application type, which includes the requirements of the application workload depending on the traffic load coming towards the services. The next station of the thesis was to close the control loop by producing and enforcing LCM decisions based on the monitoring data. For this end, we propose Zero-touch Service Management (ZSM) framework containing a fine-granular resource autoscale. The scaler leverages eXplainable Artificial Intelligence (XAI) to make decisions on which resource to scale, based on the output of an ML model that predicts microservice performance; the model was trained on the dataset generated by the second contribution. Finally, we model the problem of the application's placement and migration using a multi-objective optimization problem with

two objectives: the first is to reduce the carbon footprint of the deployment, and the second is to reduce the cost of the deployment. Figure 8.1 represents the PhD journey from collecting metrics to enforcing data-driven decisions, thus closing the control loop.

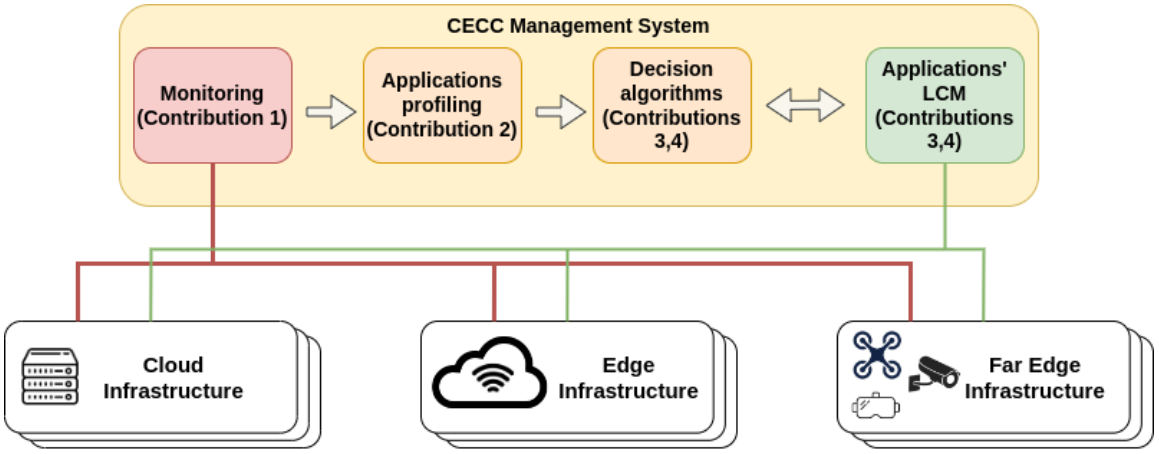


Figure 1.1: PhD journey

1. **End to end monitoring of multi-domain applications and services:**

   (a) *Challenge description:* An efficient and scalable monitoring system is a critical component for any cloud computing provision or network to monitor and validate the functioning of the running services and the underlying infrastructure. This is more valid in 5G, as it relies on the network slicing concept, which adds many challenges to the monitoring system. Among these challenges is the fact that network slices use resources from different technological domains involving different entities based on different technologies, which require monitoring different types of resources, like Radio Access Network (RAN), computing, memory, and network data rate. Indeed, the monitoring of RAN components is completely different from monitoring Cloud or Network function virtualization (NFV) Infrastructure. Another challenge pertains to the monitoring system's scalability, as the network operator is expected to run several parallel network slices on top of its 5G infrastructure. Finally, multi-tenancy and isolation among network slices need to be enforced; slice-related data should be seen only by its owner.

   (b) *Proposed solutions:* We devise a novel monitoring framework multi-domain network slicing ready mobile networks including CECC deployments. The framework is technology agnostic as it provides metrics in a uniform manner, leveraging a unified structure of KPIs, effectively abstracting away any underlying technological intricacies. The monitoring system is scalable and supports a high number of running services in parallel.

   (c) *Publications:*
   A Scalable Monitoring Framework for Network Slicing in 5G and Beyond Mobile Networks
   Mohamed Mekki; Sagar Arora; Adlen Ksentini
   IEEE Transactions on Network and Service Management (Volume: 19, Issue: 1, March 2022)

A multi-technological domains KPIs Monitoring System for Network Slicing in 5G
Mohamed Mekki; Sagar Arora; Adlen Ksentini
Cloud Days, 25-26 November 2021, Brest, France

2. **Application behavior profiling in cloud native environments:**

   (a) *Challenge description:* The emergence of cloud-native and containerization changed the way applications are developed and deployed. Current applications decompose the service's functionalities and features into multiple microservices; each microservice is responsible for a subset of those overall functionalities. When packaged into containers, to be run on the cloud or edge infrastructure, the tenant needs to specify computing resources to run their workload. Indeed, the infrastructure owner has to indicate the amount of CPU and memory limit for a container running a microservice. It happens that a container that exceeds these limits is killed or experiences a drop in performance. Accordingly, determining the limit to assign to a container and configure a service resource is a challenge. On the one hand, the tenant does not clearly understand the environment in which the application will be deployed; on the other hand, the platform provider gets the application as a packaged container in which the workload is seen as a black box. In many situations, the configuration ends by using default configurations that are not appropriate for the application's requirements. Indeed, tenants naturally request a larger limit than what the application needs, which, in turn, for a constrained environment (like the edge), results in resource wastage.

   (b) *Proposed solution:* We perform an experimental study aiming to detect if a tenant's configuration allows running its service optimally. To this aim, we run several experiments on a cloud-native platform, using different types of applications under different resource configurations. The obtained results provide insights on how to detect and correct performance degradation due to misconfiguration of the service resource.

   (c) *Publications:*
   Microservices Configurations and the Impact on the Performance in Cloud Native Environments
   Mohamed Mekki; Nassima Toumi; Adlen Ksentini
   2022 IEEE 47th Conference on Local Computer Networks (LCN), 26-29 September 2022, Edmonton, AB, Canada.

3. **Automatic resources scaling of applications in cloud native environments:**

   (a) *Challenge description:* Containerizing microservice applications is emerging as a new paradigm to optimize the portability, flexibility, and management of such applications. Running cloud applications based on microservice architecture creates new challenges. Cloud applications require different Quality of Service (QoS) as well as various resource demands, which require the design of reliable scaling frameworks. In this context, ML, and in particular Reinforcement Learning (RL), algorithms have been widely leveraged to design intelligent and autonomous scaling frameworks. They aimed to determine the right values for the different resource requirements of applications' microservices and hence meeting applications' QoSs. However, ML-based models are becoming more and more complex, and their decisions are hardly interpreted by users, especially companies' executive staff as well as container orchestration tools.

Therefore, the corresponding users (tools) cannot neither trust and understand ML models' outputs, nor optimize their decisions based on ML models' outputs. Fine-granular management of cloud-native computing resources is one of the key features sought by cloud and edge operators. It consists of giving the exact amount of computing resources needed by a micro-service to avoid resource over-provisioning, which is, by default, the adopted solution to prevent service degradation. Fine-granular resource management guarantees better computing resource usage, which is critical to reducing energy consumption and resource wastage (vital in edge computing).

(b) *Proposed solution:* In our solution, we propose a novel ZSM framework featuring a fine-granular computing resource scaler in a cloud-native environment. The proposed scaler algorithm uses AI/ML models to predict micro service performances. More specifically, we use eXtreme Gradient Boosting (XGBoost) as ML algorithm to predict any violations related to the latency performance of running applications; if a service degradation is detected, then a root-cause analysis is conducted using XAI. Based on the XAI output, the proposed framework scales only the needed (exact amount) resources (i.e., CPU or memory) to overcome the service degradation. The proposed framework and resource scheduler have been implemented on top of a cloud-native platform based on the well-known Kubernetes [3] tool. The proposed scheduler with lesser resources achieves the same service quality as the default scheduler of Kubernetes.

(c) *Publications:*
XAI-Enabled Fine Granular Vertical Resources Autoscaler
Mohamed Mekki; Bouziane Brik; Adlen Ksentini; Christos Verikoukis
2023 IEEE 9th International Conference on Network Softwarization (NetSoft), 19-23 June 2023, Madrid, Spain.

4. **Energy aware application life-cycle management in cloud edge computing continuum**

(a) *Challenge description:*

Deploying applications on a CECC infrastructure requires detailed information about each application's profile to be considered by the CECC application orchestrator or manager. Application profiling refers to the systematic analysis of the characteristics of an application. This process aims to gain insights into an application's behavior, performance, resource utilization, and dependencies within the cloud infrastructure. By having access to these profiles, the CECCM can optimize the placement of applications to enhance performance and maximize the efficient utilization of infrastructure resources. Profiling techniques include a range of techniques, such as monitoring system metrics and analyzing network traffic patterns. Additionally, the integration of AI techniques facilitates predictive modeling to anticipate future demands by the application workloads in terms of compute resources and network requirements.

Furthermore, CECC infrastructure produces a carbon footprint, which increases with the scale of the infrastructure. However, amid the global movement towards sustainability, initiatives such as the European Green Deal [1] accelerate the transition towards greener energy sources. This shift is increasing the availability of infrastructure powered by green energy. Consequently, when determining the optimal placement of microservices, energy consumption considerations and the utilization of green energy sources must be carefully balanced while still satisfying the application's SLA. Also,

the CECC application orchestrator (i.e., CECCM) can take advantage of the application profile to predict the resource usage and decide on migrating application microservices if a better carbon footprint can be achieved with a new configuration and execution location of applications.

(b) *Proposed solution:* We propose an architecture of CECC Application Orchestrator. The architecture leverages applications and infrastructures profiling to efficiently manage the CECC applications. This profiling is done using the monitoring information, including energy metrics collected from energy monitoring systems such as Kepler [4] and carbon intensity of infrastructures. Then, we define a modeling method for applications profiles from the point of view of the CECCM, the profile represents the application's current and future compute and network requirements. The profile is constructed based on the historical usage of the application using statistical methods. Finally, we model the problem of selecting the best locations to deploy or migrate applications while minimizing the deployment's cost and the carbon footprint. The model decides the current and future placements of each application based on the application profile, the availability constraints of the application and the available infrastructures resources. We further propose an heuristic solution to solve the problem rapidly and we compare the different trade-off between carbon and cost efficiency.

(c) *Publications:*
Energy-Aware Application Life-Cycle Management in Cloud Edge Computing Continuum Using Applications Profiles
Mohamed Mekki; Adlen Ksentini
Under submission.

1.4 Thesis Structure

This thesis is structured as follows: Chapter 2 is dedicated to the presentation of the background and concepts that our research work was based on, which are mainly: application management landscape, ZSM and ML/XAI techniques. Then, in the next four chapters, we present the contributions of the thesis: First, in Chapter 3, we introduce the multi-domain monitoring framework. Then, in Chapter 4, we present the work on applications benchmarking a cloud-native environment. Afterwards, in Chapter 5, we present our ZSM system featuring an XAI-enabled fine granular resource autoscaler. Lastly, in Chapter 6, we introduce our proposed architecture and model for carbon footprint aware CECC application deployment and migration.o Finally, in Chapter 7 we conclude the Thesis while in Chapter 8 we introduce the perspectives of our work.

Chapter 2

Background

2.1 Introduction

Cloud Edge Computing Continuum (CECC) management cannot be achieved using lower level resources management solutions alone as it also requires service and application level orchestration frameworks. In addition, Machine Learning (ML) is pivotal in automating the application lifecycle management process. However, it is imperative that ML models are trusted to make decisions that do not compromise the application requirements as well as the CECC infrastructure state. eXplainable Artificial Intelligence (XAI) must be seamlessly integrated with ML algorithms to furnish additional insights into the model's decisions. These insights serve multiple purposes: ensuring the model operates as anticipated, extracting new rules and relationships from available data, and facilitating the verification of model functionality against expectations. In this Chapter, we provide an overview of the Cloud Edge Computing Management Frameworks enablers, as well as ML/XAI techniques that part of it were used in this thesis to achieve zero-touch LCM of CECC applications.

2.2 CECC Management Landscape

2.2.1 Cloud Edge Computing

Cloud Edge Computing is a distributed computing paradigm that extends the capabilities of traditional cloud computing by decentralizing computation and data storage to the network edge. Unlike centralized cloud architectures, which rely on remote data centers for processing and storage, edge computing leverages computing resources located closer to the user or point of data generation and consumption. This approach reduces latency, improves responsiveness, and enables real-time processing of data-intensive applications, particularly in scenarios where low latency, high bandwidth, and localized data processing are critical requirements. Edge computing encompasses a diverse ecosystem of edge devices, edge servers, and edge data centers, interconnected through low-latency networks, and supports a wide range of use cases across industries, including Industrial Internet of Things (IIoT), autonomous vehicles, augmented reality, industrial automation, healthcare, video surveillance, and smart cities.

Enabling technologies

The deployment of use cases leveraging Cloud Edge Computing Continuum is supported by three key elements: advancements in application deployment technologies, evolutions in applications' architecture and development methodology, and innovations in networking technologies.

First, the combination of virtualization and containerization has transformed how applications are deployed and managed. Virtualization facilitates the partitioning of physical hardware into multiple virtual machines (VMs), each functioning as an isolated instance of an operating system, complete with its own CPU, memory, storage, and network interfaces. This is typically orchestrated by a hypervisor, which abstracts the underlying physical hardware and orchestrates resource allocation to each VM. Hypervisors are usually categorized into two types:

1. Type 1, or Bare Metal Hypervisor, involves running a hypervisor directly on the physical hardware without needing a host operating system. This approach offers high performance, eliminating overhead from an additional operating system layer and enabling direct access to hardware resources by VMs.

2. Type 2, or Hosted Hypervisor, where the hypervisor runs on top of a host operating system. Type 2 hypervisors rely on the underlying operating system to manage hardware resources, making them suitable for desktop virtualization and development/testing environments.

In addition to virtualization, containerization provides an alternative approach to sharing physical hardware among multiple applications. Containers encapsulate applications along with their required libraries and some OS packages. While running, containers share the host operating system's kernel and resources, but they are isolated from each other, providing process-level isolation. Containerization offers rapid startup times, efficient resource utilization, and consistent deployment across different environments.
Second, there is a paradigm shift in application architectures and development methodologies, moving away from monolithic applications to Service-Oriented Architecture (SOA) and microservices architecture. In a monolithic architecture, the entire application is constructed as a single, tightly integrated unit deployed together on one machine or VM. This architecture lacks the flexibility to deploy application parts in different locations. Conversely, microservices architecture decomposes the application into smaller, loosely coupled services, each responsible for a specific business function or capability. These services communicate with each other through well-defined APIs or messaging protocols, enabling them to be independently developed, deployed, and scaled. This architectural approach facilitates the separation of different parts of an application based on their functionality. It enables applications to have microservices requiring low latency and higher bandwidth to run at the Cloud Edge, while other microservices run in the Central cloud or other locations. This flexibility in deployment optimizes performance and resource utilization.
Lastly, the evolution of networking solutions such as 5G mobile networks, which revisited the mobile network system radically. Besides improving the throughput and data rate, 5G introduces a new network architecture relying on the concept of network slicing. A network slice is composed of sub-slices that use resources from different technological domains: Radio Access Network (RAN), Core Network (CN), Edge/Cloud. 5G provides larger bandwidth (up to 400 MHz) and reduces latency. Combined with edge cloud capabilities in the vicinity of the radio network, 5G can achieve a very low latency for services.

Services and application management

Kubernetes is the de facto choice for orchestrating containerized applications reliably and efficiently in cloud-native environments. It is an open-source container orchestration platform that streamlines the deployment and management of containerized applications. Kubernetes automates tasks such as scaling, load balancing, and service discovery. Originally developed by Google, Kubernetes is now maintained by the CNCF. However, Kubernetes alone cannot

be sufficient to manage multi-domain services; for this reason, several frameworks, standards, and management architectures have been introduced. First, the ETSI Multi-access Edge Computing (MEC) framework [5] provides a set of specifications and guidelines for implementing edge computing capabilities within telecommunication networks. It defines architectural principles, interfaces, and protocols that enable the deployment and management of applications and services at the network edge. Another initiative is ETSI NFV MANO [6]. It provides an architectural framework that enables the deployment and management of Cloud-native or Container Network Function (CNF) and VNF. Other initiatives include TM Forum's Open API program, which is a global initiative to enable end-to-end seamless connectivity, interoperability, and portability across complex ecosystem-based services [7]. The program is creating an Open API suite, which is a set of standard REST-based APIs enabling rapid, repeatable, and flexible integration among operations and management systems, making it easier to create, build and operate complex, innovative services. Finally, Zero-touch Service Management (ZSM), a framework designed to automate network and service management processes without human intervention. In the rest of this chapter, we will describe two main management frameworks: NFV MANO and ZSM. NFV MANO framework has several open-source implementations, and its components are well suited for managing cloud applications' life cycles. Meanwhile, ZSM allows automated service management.

2.2.2 Network Function Virtualization

Network Function Virtualization (NFV) is a paradigm shift in the networking industry, transforming traditional hardware-based network functions into software-based entities. NFV leverages virtualization technologies to decouple network functions from proprietary hardware, enabling them to run on standard servers and data centers.

NFV Management and Orcherstration

ETSI NFV Management and Orchestration (MANO) is a framework to orchestrate and manage the life cycle of network services comprised of VNFs and PNF. With the rise of Cloud-Native Network Functions (CNFs), designed to leverage containerization and microservices architectures, NFV MANO architectures are evolving to accommodate these lightweight and agile network functions. This evolution involves integrating container orchestration platforms, like Kubernetes, into the NFV MANO stack to manage the lifecycle of CNFs alongside traditional VNFs, enabling operators to efficiently deploy and manage a diverse range of network functions in modern cloud-native environments. Fig. 2.1 shows the architecture of the ETSI NFV MANO framework with support for CNFs.
The role of the different building blocks of the framework is defined below:

- **Network Function Virtualization Orchestrator (NFVO):** Responsible for managing the life cycle of network services.
- **Virtual Network Function Manager (VNFM):** Responsible for managing the life cycle of VNFs, their configuration, performance, and fault management.
- **Virtualized Infrastructure Manager (VIM):** Responsible for controlling and managing the virtual resources of NFVI to provide them to VNFs. In short, it is responsible for the life cycle management of VMs.

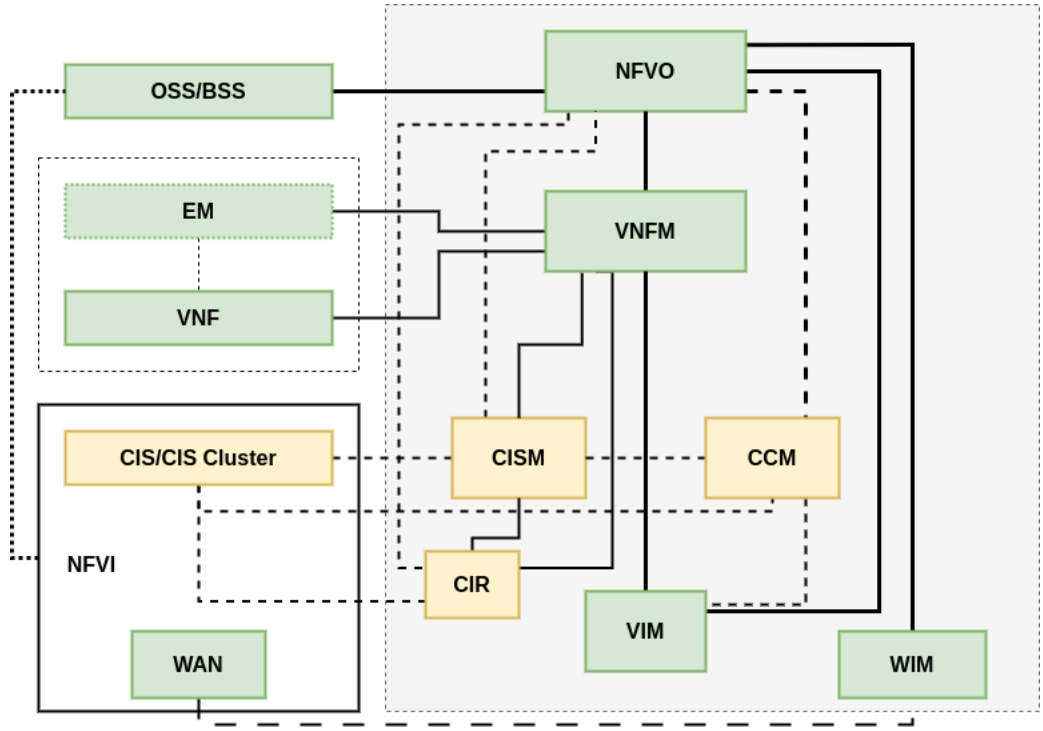


Figure 2.1: NFV-MANO framework reference architecture with support for containers [6]

- **Element Manager (EM):** These are designed to monitor and configure the VNFs. They can be used in some aspects of VNF's life cycle management. For example, fault management.
- **Container Infrastructure Service (CIS):** Container Infrastructure Service (CIS) provides run-time infrastructural dependencies, computational, storage, and networking resources for one or more containerization technologies. It can be considered the cloud-native equivalent of a virtual-machine hypervisor. Hypervisors provide infrastructure to host virtual machines.
- **Container Infrastructure Service Management (CISM)**: Manages containers executed by CIS. It is responsible for container deployment, monitoring, and life cycle management.
- **CIS Cluster Management (CCM):** CCM is responsible for managing the life cycle of CISM.
- **Wide Area Network (WAN):** Refers to the transport network used to connect multiple NFVI sites.
- **WAN Infrastructure Manager (WIM):** It provides management of Multi-Site Connectivity Services.
- **Container Image Registry (CIR):** Its role is to store all the container images of the container-based VNFs.
- **NFV Infrastructure (NFVI):** The hardware infrastructure on which virtual machines or containers will be hosted.

Cloud resources orchestration ready systems

The existing service orchestrators were designed to orchestrate VNFs. Later, they adapted their architecture and service descriptors to orchestrate CNFs. The orchestrators are capable of placing the applications at the cloud edge. Some of the well-known service orchestrators we can mention include [8]:

1. Open Source MANO (OSM): A network service orchestrator proposed by ETSI as a reference design based on ETSI NFV standard. Designed to orchestrate VM-based network services, it supports container-based network functions. OSM can deploy network functions at the edge cloud as well.

2. Open Network Automation Platform (ONAP): Designed to manage VM-based VNFs, and with the recent release, it follows cloud-native principles to orchestrate CNFs. ONAP's complicated architecture, which has a large number of components, results in high resource consumption [8]. Furthermore, the complex realization of ONAP demands the involvement of a big team in taking care of service orchestration. The high resource consumption makes it unsuitable to deploy in a resource-constrained environment.

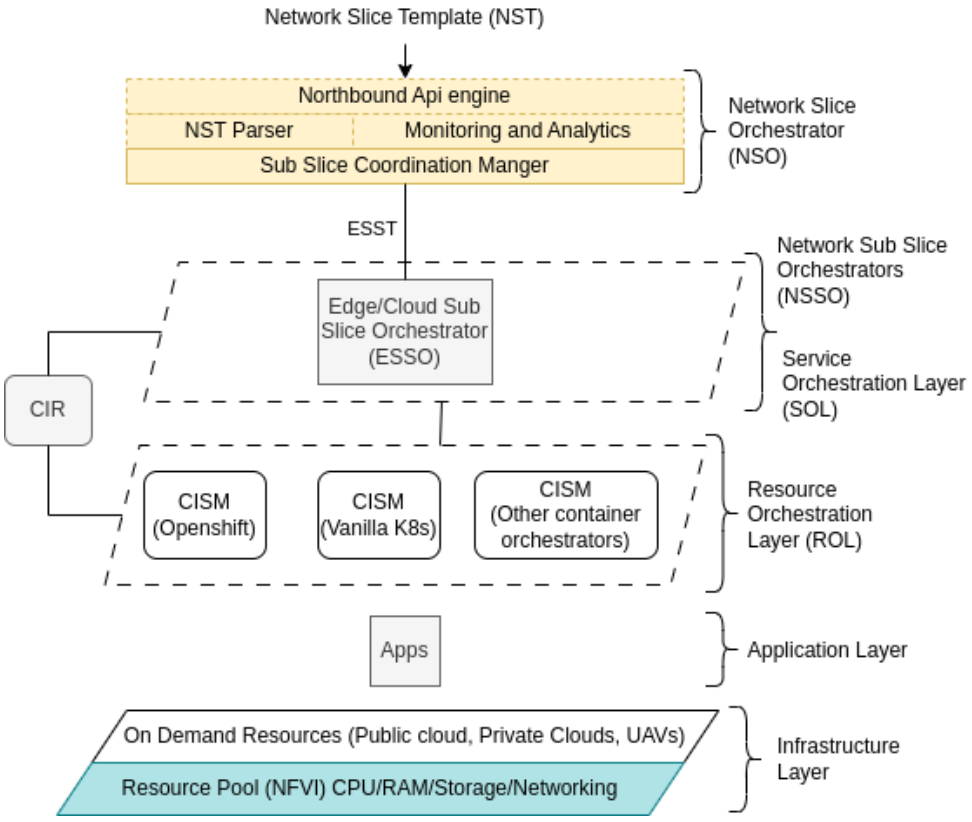


Figure 2.2: Cloud-native lightweight slice orchestration framework (Cloud related components) [8]

Building zero-touch CECC application management solutions necessitates a cloud-ready service orchestrator. The work developed in this thesis was tested on the Cloud-native Lightweight Slice Orchestration (CLiSO) framework [8]. The framework has a hierarchical architecture starting from the top: Network Slice Orchestrator (NSO), Network Sub-Slice Orchestrator (NSSO) (ran, edge, and core domain), CISM, and Container Image Registry (CIR). While the framework

allows the deployment of end-to-end network slices spanning over the RAN, Core Network, and Cloud domain, we focus on the cloud edge management components of its architecture as shown in Fig. 2.2. The Service Orchestration Layer is responsible for translating Service Level Objective (SLO) to Resource Level Objectives and coordinating with resource controllers. Whereas, the Resource Orchestration Layer manages the resources by communicating with the underlying resource pool. The purpose of each component of the framework is as follows:

- **Network Slice Orchestrator (NSO):** It is responsible for creating Network Sub Slice Templates (NSST) for different domains and coordinating the life cycle management of sub-slices. It receives the monitoring data from different sub-slice orchestrators to extract slice-level monitoring information.
- **Network Sub-Slice Orchestrators (NSSO):** Or Service Orchestrators, They are responsible for handling sub-slices of their respective domains and collecting monitoring data to share with NSO. For Cloud orchestration, the Edge Sub-Slice Orchestrator (ESSO) [9] handles the life cycle of edge sub-slices composed of MEC Applications. It coordinates with the MEC Platform to provide the necessary services like traffic redirection, DNS-based redirection, or RNIS to the MEC Apps. ESSO only handles container-based MEC Apps.
- **CISM:** Responsible for orchestrating containers. It creates the necessary communication links between applications and network functions to deliver the required service behavior. The framework can orchestrate containers on different distributions of Kubernetes, Openshift, Vanilla Kubernetes (also known as K8s), and K3s. A new distribution can be supported by creating a plugin.
- **Container Image Registry (CIR):** Manages and stores Open Container Initiative (OCI) format container images. It can pull images from public or private repositories and build images from source code.

2.3 Zero-touch Service Management

In this section, we introduce the concept of Zero-touch Service Management (ZSM), which enables automated reconfiguration of cloud applications deployments and networks. In such an environment, the management system continually learns from infrastructure and service metrics, as well as feedback, in order to dynamically optimize service configurations to meet its evolving requirements.

2.3.1 ETSI ZSM Reference Architecture

ETSI Zero touch network & Service Management (ZSM) is a framework designed to automate network and service management processes without human intervention. It enables the seamless provisioning, monitoring, and optimization of services across various vendor technologies. By automating routine tasks, ZSM reduces operational costs and accelerates service delivery. It promotes agility and flexibility in managing complex network environments, adapting to dynamic changes with minimal disruption. ZSM aligns with industry trends towards Software-Defined Networking (SDN) and Network function virtualization (NFV), offering a scalable and adaptable approach to service management. Overall, ZSM enhances efficiency, reliability, and scalability in

modern service management operations. In this section, we introduce the ZSM architecture and its key components. Figure 2.3 depicts the framework reference architecture where every management domain, as well as the E2E service management domain, provides a set of ZSM service capabilities by management functions that expose and/or consume a set of service endpoints. The inter-domain integration fabric facilitates providing capabilities and accessing endpoints cross-domain. The main building blocks of the ETSI ZSM framework are:

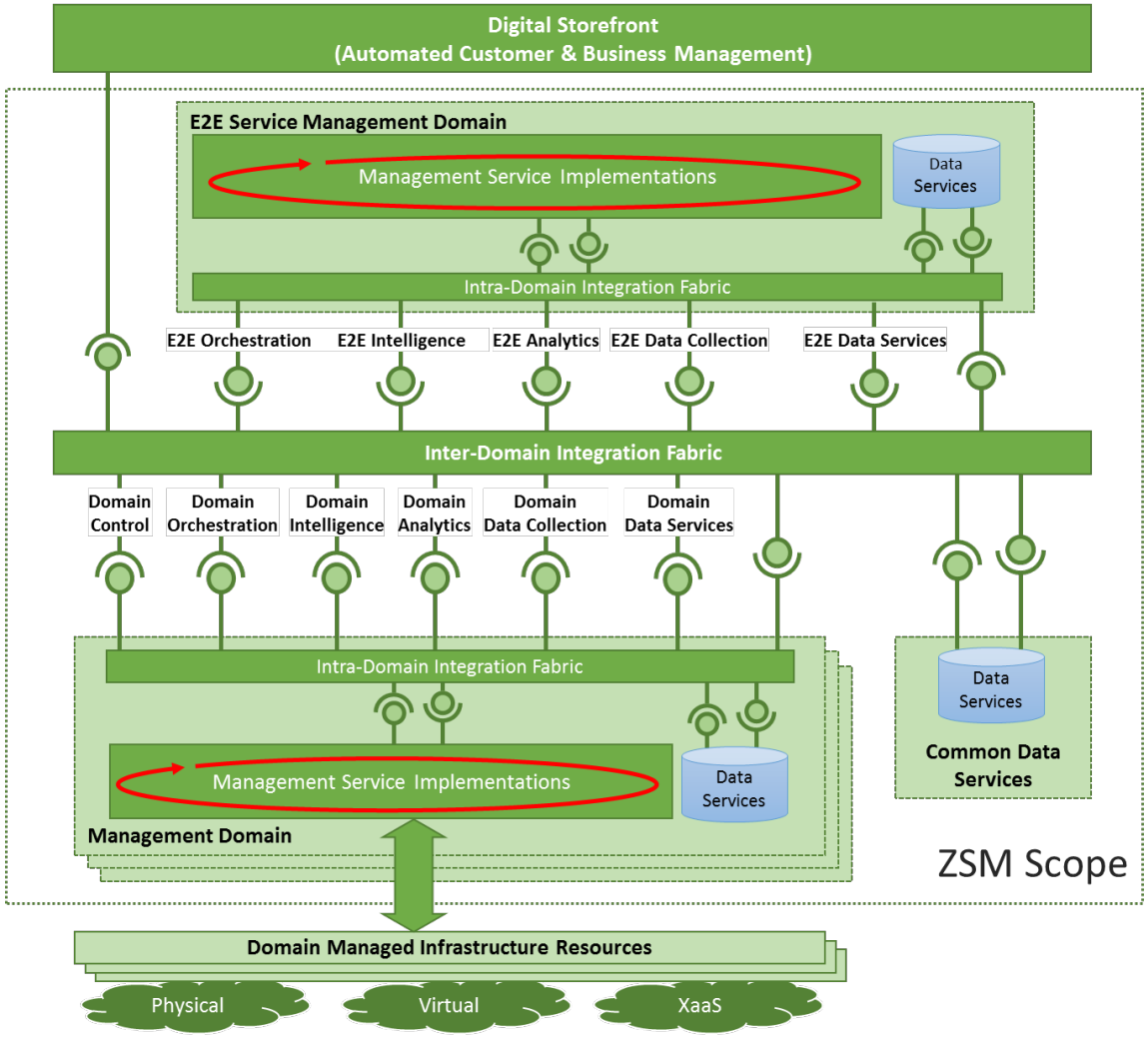


Figure 2.3: ZSM framework reference architecture [10]

- **Management services:** A "management service" is the most fundamental building block used as part of the ZSM framework reference architecture. Management services offer capabilities for consumption by service consumers via standardized management service endpoints. The capabilities of a given management service collectively define its function with respect to the entities it manages. Service capabilities may be offered for consumption by multiple service consumers. One or more service capabilities can be mapped to one or more service endpoints. All management services offer consistent capabilities for invocation and communication purposes. This enables a high degree of automation and continuity regarding interactions between management domains. Offered management services can be combined into new management services. In the composition hierarchy, each higher layer supports management services with a higher abstraction and a broader scope.

- **Management functions:** Management functions are entities that produce and consume management services. A management function can be a service producer when it offers specific capabilities of one or more management services. Conversely, it can be a service consumer when it utilizes certain capabilities of one or more management services.
- **Management Domains:** Management domains are used to partition administrative responsibilities and to create separation of concerns within a given ZSM deployment, considering various implementation, organizational, governance, and functional limitations. These domains federate management services equipped with the necessary capabilities to oversee resources or resource-facing services within a specific domain. For instance, certain management services may require approvals for consumption within the authorized consumer base of a management domain, while others remain consistently available to authorized consumers both within and outside the domain. Management domains oversee one or more entities and deliver service capabilities by utilizing service endpoints.
- **The End-to-End (E2E) service management domain:** The E2E service management domain is a special management domain that provides end-to-end management of customer-facing services, composed of the customer-facing or resource-facing services provided by one or more management domains. However, it does not directly manage infrastructure resources.
- **Integration fabric:** The integration fabric enables inter-operation and communication between management functions within and across management domains, including registering, discovering, and invocating management services. It also offers management service integration, inter-operation, and communication capabilities, and consumption capabilities.
- **Data services:** Data services enable consistent means of shared management data access and persistence by consumers across management services within or across management domains. It also allows data persistence to be separated from data processing.

2.3.2 Closed Control Loops

The ZSM Industry Specification Group(ISG) ISG is focused on the definition of generic enablers, closed-loop enhancements, and operations for the next generation of AI-driven autonomous networks. In this thesis, we focus on proposing the enablers of a closed control loop to enable zero-touch CECC management. Our work does not implement ETSI ZSM standard, but it implements all the components of the closed control loop defined in [11].
Figure 2.4 shows the components of the closed-control loop, which, according to [11], are:

- **Monitoring System (MS):** The MS role is to collect critical information on the functioning of a system and provide this information, after aggregation or normalization, to the Analytical Engine (AE), which in turn uses this information to detect and react to applications' LCM events, such as performance degradation, performance optimization, and security threats. The MS interacts with different entities that orchestrate and manage the per technological domain sub-service (i.e., CISM or cloud/edge Infrastructure local management system in the context of CECC). Indeed, there is a difference between information that monitors the state of the infrastructure shared by the running services and information that monitors the service applications. The MS has to interact with CISM to collect information on:

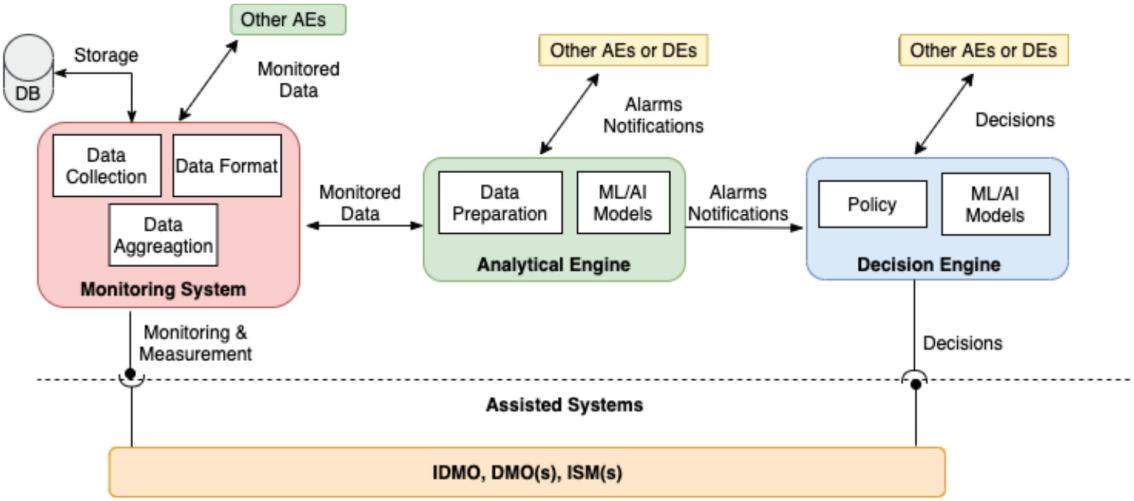


Figure 2.4: Closed-control loop components: MS, AE, and DE [11]

  - Infrastructure, such as computing platforms and hardware.
  - Applications' microservices.
  - Or any common entities such as 5G Core Network (CN) functions or Directory Name Service (DNS).
  - The MS can also interact with applications through API exposed by the latter that allows extraction of information on the application's state, such as events, alarms, and logs.

  The principal consumer of MS information is AE, which is in charge of triggering the monitoring of needed information from the MS. The latter starts the monitoring process by connecting to the appropriate source. Accordingly, the MS exposes two APIs: control API and data collection API. The AE uses the control API to request the KPI to monitor the periodicity, duration, and so on. Meanwhile, the data collection API is the interface from which data is provided to the AE as requested through the control API. The control API also indicates how data is provided: publish/subscribe, request/response, the data format, and so on.

- **Analytical Engine (AE):** As opposed to the MS, the AE does not store but processes data gathered from the same or lower-level MS or AE and exposes the result to any requester (i.e., the DE or another AE) in an on-demand or periodic fashion. AE-to-AE communication makes it possible to build a learning model using federated learning (FL) techniques. The main functions of AE are:

  - Identify performance degradation of a network slice.
  - Optimize the performance of a service or the infrastructure resources.
  - React to security threats.

  To this aim, the AE subscribes to data types in which it is interested using the control API exposed by the MS. The data type will be determined according to the logic of the LCM application execution. Then, the AE starts receiving the stream of data or uses a request/response mechanism, depending on the purpose of the analysis. The AE may adapt

the monitoring data rate or stop the precedent request and request other related monitoring information. The AE heavily relies on AI/ ML to complete an inference task locally, extract features, analyze these features, and send alerts and notifications to the Decision Engine (DE). AEs can collaborate to build distributed learning (based on Federated learning (FL)) models to realize the analysis and notify the DE accordingly. Examples of features extracted and analyzed are prediction of SLA violation, prediction of service migration, prediction of service faults, attack identification, and anomaly detection.

- **Decision Engine (DE):** The DE is the decision-making element of the loop. It analyzes alerts and notifications from AE(s) and considers a decision to make. The decisions are either derived using a local ML algorithm, based mainly on reinforcement learning (RL), or a predefined policy enforced by the service owner or Service Orchestrator. The DE uses exposed APIs by the Service Orchestrator to enforce the considered decisions.

Note that: Inter-Domain Management and Orchestration (IDMO) is a centralized component with full-scope service management and orchestration decision capabilities it can be mapped to the NSO from Fig. 2.2. Domain-specific Management and Orchestration (DMO) (e.g., cloud infrastructure, edge, RAN) is the technological domain management and orchestration entity, equivalent to the ESSO from Fig. 2.2. While, for each service, the In-Slice Manager (ISM), a logical entity, handles the autonomous management of the applications or network functions (i.e., CNF).
Finally, according to [12], since closed-loop automation leverages AI/ML, it is necessary to ensure the trustworthiness of the AI algorithms. One way to ensure it is to use XAI to explain the decision of the ML models and ensure that the models are not malfunctioning. In the next section, we present the ML and XAI techniques that can be used to manage zero-touch CECC management systems deployments

2.4 Machine Learning (ML) and eXplainable AI (XAI)

Machine learning is a subset of artificial intelligence that focuses on developing algorithms and statistical models that enable machines to perform tasks without being explicitly programmed for each step. At its core, machine learning aims to enable systems to learn from data and improve their performance over time.
Subfields of machine learning include:

- **Supervised Learning:** In supervised learning, the algorithm is trained on a labeled dataset, where each input data point is associated with a corresponding target output. The algorithm learns to map inputs to outputs by generalizing from the labeled examples provided during training. Common tasks in supervised learning include classification (predicting categories) and regression (predicting continuous values). Such algorithms include: Linear Regression; Logistic Regression; Decision Trees; Random Forest; Support Victor Machine (SVM); Gradient Boosting Machines (GBM) such as Extreme Gradient Boosting (XGBoost).

- **Unsupervised Learning:** Unsupervised learning involves training algorithms on unlabeled data, where the algorithm must identify patterns or structures within the data on its own. Unlike supervised learning, no target outputs are provided during training. Instead, the algorithm discovers hidden patterns or groups in the data, such as clustering similar data points together or dimensionality reduction to represent data in a lower-dimensional space.

- **Semi-Supervised Learning:** Semi-supervised learning techniques combine supervised and unsupervised learning elements. These methods leverage a small amount of labeled data along with a larger amount of unlabeled data to improve model performance. Semi-supervised learning is particularly useful when obtaining labeled data is expensive or time-consuming.
- **Reinforcement Learning:** Reinforcement learning involves training agents to make sequential decisions in an environment to maximize cumulative rewards. The agent learns by interacting with the environment and receiving feedback in the form of rewards or penalties. Reinforcement learning algorithms aim to discover the optimal strategy, policy, or behavior for the agent to achieve its goals over time.
- **Deep Learning:** Deep learning is a subset of machine learning focusing on artificial neural networks with multiple layers (deep architectures). These networks can learn intricate patterns and representations from data, often achieving state-of-the-art performance in various tasks such as image recognition, natural language processing, and speech recognition. Relevant DL algorithms include: Recurrent Neural Network (RNN) including variations like Long Short-Term Memory (LSTM).

Success in Machine Learning has led to a wave of AI applications in several domains. However, because ML models are seen as black boxes, the usage of these models to make decisions in critical systems requires trust in the model. Therefore, to increase the trustworthiness of ML models, XAI techniques have been devised in order to explain why and how an ML model arrives at a specific decision. This means that XAI is the result of efforts to make AI systems intelligible to their users, Whether the user is human or another computer program.
There is a clear distinction between models that are interpretable by design and those that can be explained by means of external XAI techniques. In this section, we present some types of Transparent machine learning models and post-hoc interpretability techniques classified (as shown in Fig. 2.5) into model-specific/model-agnostic and global/local explanation techniques.

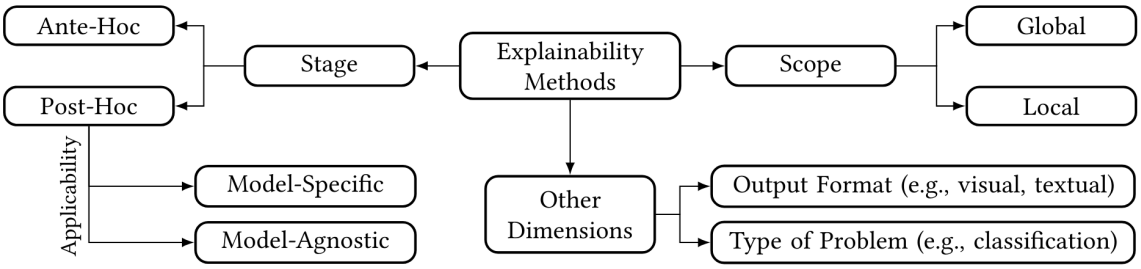


Figure 2.5: Explainability techniques classification [13]

2.4.1 Transparent machine learning Models

A transparent model is a model that conveys some degree of explainability by itself. According to [14], those models can have three levels of transparency: Simulayability, which denotes the ability of a model to be simulated by its user. Decomposability, or the ability to explain parts of the model (input, parameter, calculations), which requires every input to be readily interpretable. Finally, algorithmic transparency presents the user's ability to understand the process followed by the model to produce any given output from its input data. Examples of machine learning models that are considered transparent are:

- **Linear/logistic regression:** Logistic regression is a classification model that predicts representing the category, and its output is a binary variable. For the same if the predicted variable is continuous, Linear regression is used instead. Linear regression models are capable of being simulated because the predictors are human readable. Its decomposability can be determined by the variables used to make the prediction and their understandability by the users.

- **Decision trees** Decision trees are models that can easily be considered as explainable. In the simplest structures, decision trees are simulable models. However, for manageability, the size of the decision tree is a determining factor. The features and their meaning are easily understandable. Since the model comprises rules that do not alter data, decision trees are decomposable. Further, their human-readable rules that explain the knowledge learned from the data make them Algorithmically transparent.

- **k-nearest neighbors** It deals with classification problems simply: It predicts the class of a sample by voting the class of its k-nearest neighbors. This neighborhood relation is induced by the measure of distance between samples. KNN can be simulated as the model's complexity matches human simulation capabilities.

- **Rule-based learning** Rule-based learning encompasses models that generate rules to characterize the data. Rules can take the form of simple conditional if-then rules or more complex combinations of simple rules to form their knowledge. Fuzzy rule-based systems are designed to address a broader scope of actions. Because variables included in rules are readable, and the size of the rule set is manageable by a human user without external help, Rule-based learners can be considered simulable.

- **General additive models** Linear models in which the value of the variable to be predicted is given by the aggregation of a number of unknown smooth functions defined for the predictor variables. For a General additive model to be simulated, variables and the interaction among them as per the smooth functions involved in the model must be constrained within human capabilities for understanding.

- **Bayesian models** Bayesian models usually take the form of a probabilistic directed acyclic graphical model whose links represent the conditional dependencies between a set of variables. Its categorization leaves it simulatable, decomposable, and algorithmically transparent. However, it is worth noting that a model may lose these first two properties under certain circumstances(overly complex or cumbersome variables).

Note that even if a model is considered transparent, the ability to explain it can be reduced as its complexity increases. In this case, post-hoc explainability techniques can be used.

2.4.2 Post-hoc explainability techniques

When a machine learning model cannot be declared transparent, separate methods must be devised and applied to the model to explain its decisions. This is the purpose of post-hoc explainability. These methods vary from model-agnostic techniques to post-hoc methods tailored to define a certain ML model. Simultaneously, post-hoc techniques may be categorized as either Local or Global.

Model agnostic techniques

Model-agnostic techniques for post-hoc explainability are designed to extract information from a model prediction procedure. This can be done using explanations by simplification, which represent the broadest techniques under the category of model-agnostic post-hoc methods. Feature relevance explanation techniques are also used for model agnostic explanations. These techniques aim to describe the function of a model by measuring the influence of each feature on the prediction output of the model to be explained. Moreover, in some cases, visual explanations are possible.

Model specific techniques

Post-hoc explainability techniques for explaining the decisions and that are adapted to a type of ML model are used for models that rely on more sophisticated learning algorithms that require additional layers of explanation. Such techniques are used to explain Tree ensembles and random forests, support vector machines, multi-layer neural networks, convolutional neural networks, and recurrent neural networks.

Global explanation techniques

XAI techniques are classified as global explanation techniques or local ones. Global explanation techniques are applied to obtain the general behavior of a model. Global models try to explain the whole logic of a model by inspecting its structure [15]. Techniques in this category include:

- **SHAP:** a unified framework for interpreting predictions, SHAP [16] (SHapley Additive exPlanations). SHAP assigns each feature an importance value for a particular prediction. Its components include the identification of a new class of additive feature importance measures and theoretical results showing a unique solution in this class with a set of desirable properties.
- **Causal Dataframe[17]:** which is a causal inference tool for data explainability. Causal inference refers to an intellectual discipline that considers the assumptions, study designs, and estimation strategies that allow researchers to draw causal conclusions based on data [18]
- **PI[19]:** introduces a heuristic for normalizing feature importance measures that can correct the feature importance bias. It can be used as a post-processing step with other machine learning models that provide measures of feature relevance. The PIMP (Permutation Variable Importance Measure) algorithm's P-values can be used to compare feature relevance.

Local explanation techniques

Local explanation techniques tackle explainability by segmenting the solution space and giving explanations to less complex solution subspaces that are relevant to the whole model. These explanations can be formed by means of techniques with the differentiating property that these only explain part of the whole system's functioning. Techniques falling into this category include:

- **LIME[20]:** which builds a local linear model around the predictions near a particular point for an opaque model to explain it.
- **ELI5[21]:** is a tool for explaining and debugging machine learning classifiers.

- **CFProto[22]:** propose a fast, model-agnostic method for finding interpretable counterfactual explanations of classifier predictions by using class prototypes. It offers actionable counterfactual explanations describing concrete steps to change a model's prediction.

Note that SHAP can also be used for local explanations.

2.5 Conclusion

In this Chapter, we defined Cloud Edge Computing and explored essential technologies and frameworks facilitating application Life-Cycle Management (LCM). Additionally, we outlined various XAI techniques applicable to explaining ML models aiding LCM can elucidate the assistance provided by ML models in LCM. In the next chapter, we introduce our first contribution, which proposes a framework to monitor multi-domain applications and services KPIs in order to achieve a uniform end-to-end view of the services metrics.

Chapter 3

A scalable monitoring framework for network slicing in 5G and beyond mobile networks

3.1 Introduction

Mobile networks have seen a significant shift in the last decade, with the development of a new generation (5G) and the next generation's foundation (6G). Several commercial deployments are available using the Standalone (SA) or NonStandalone (NSA) model, i.e., using 5G New Radio (NR) with 4G Core Network (CN). 5G introduces a radical evolution of the mobile network system. Besides improving the throughput and data rate, 5G introduces a new network architecture relying on the concept of network softwarization. On one hand, 5G NR provides larger bandwidth (up to 100 MHz in $<$ 6 GHz frequency band, and up to 400 MHz in $>$ 6 GHz frequency band) [23] to accommodate high data-rate demanding applications. Moreover, 5G NR introduces new physical layer numerologies that drastically reduce Radio Access Network (RAN) latency. Combined with Multi-access Edge Computing (MEC) capabilities at the vicinity of the radio network [24], 5G NR will allow achieving a very low latency for services. On the other hand, 5G network architecture builds on the concept of network softwarization, which advocates for the usage of Software-Defined Networking (SDN) and Network function virtualization (NFV) to build an agile CN and shed light on the concept of Network Slicing. The latter is a novel concept that aims at the partitioning of mobile network infrastructure into virtual network instances that are individually tuned to accommodate diverse services characterized by different requirements in terms of communication Quality of Service (QoS) within a common physical infrastructure. Network Slicing concept allows mobile operators to efficiently support the three envisioned classes of services using the same physical infrastructure: 1) enhanced Mobile BroadBand (eMBB) for applications requiring high data rates, 2) massive Machine-Type Communication (mMTC) intended to cover IoT applications that require support for a massive number devices, and 3) Ultra-Reliable and Low-Latency Communication (URLLC) for applications with strict requirements of communication latency and reliability.
A network slice is composed of sub-slices that use resources from different technological domains: Radio Access Network (RAN), CN, Edge/Cloud. The RAN sub-slice is constituted of Physical Network Function (PNF) (Radio Remote Unit (RRU)) and Virtual Network Function (VNF) (Centralized Unit (CU), Distributed Unit (DU)), which are usually shared with other slices. The RAN sub-slice is also reserving enough Physical Resource Block (PRB) to guarantee its performances [25]. The CN sub-slice is composed of a set of VNF, which are shared with other

network slices or dedicated to the network slice. Finally, the Cloud/Edge sub-slice is where applications the network service functions are run as VNF; all the VNFs are slice specific. The sub-slices are stitched together to build the end-to-end slice that runs a 5G network service.
Monitoring the performances (or Key Performance Indicator (KPI)) of the running network slices is vital for both the network operator and the slice owner. For the former, measuring the KPI allows checking and troubleshooting the performances of the entities running the components of network slices, such as RRU, CU, CU, CN, Cloud/Edge, routers, etc; while for the latter, measuring KPI allows for checking the performance of the running services and validating the SLA signed with the network operator. It is obvious that the KPIs to measure for the network operator and for the slice owner are different. The slice owner is more interested in service level KPI [26] that corresponds to the service's performances such as the end-to-end latency, achieved throughput, consumed CPU, and memory of the VNF running the service, etc. While, the network operator is more interested in collecting KPI on the infrastructure components, such as radio resource usage, the radio latency, computing usage of shared VNF among slices, etc.
However, monitoring network slice KPI introduces several challenges. Among these challenges is the fact that network slices use resources from different technological domains involving different entities based on different technologies. Indeed, the monitoring of RAN components is completely different from monitoring a NFV Infrastructure. Another challenge pertains to the scalability of the monitoring system, as it is expected that the network operator will run several parallel network slices on top of its 5G infrastructure. Finally, multi-tenancy and isolation among network slices need to be enforced; slice-related data should be seen only by its owner.
In this Chapter, we tackle these challenges by proposing a novel monitoring platform for 5G and beyond. The proposed framework natively supports network slicing and features a scalable architecture to handle a high number of running slices in parallel. To achieve this objective, we introduce metrics collectors deployed per network slice and follow the life cycle of the network slice (they are created when the network slice is deployed and are removed after the network slice is deleted). Besides, the proposed framework is technology agnostic when it comes to data collection, where a novel and technologically agnostic monitoring protocol is introduced. Indeed, we propose a metric structure that unifies the management of the collected metrics and allows the association of the metric with the part of the network slice from which it was collected, hence linking between metrics from different domains. This will tackle the problem related to the heterogeneous monitoring system used by the technological domains where a network slice is running. Finally, the proposed monitor framework supports multi-tenancy and is cloud-native compliant. Indeed, besides supporting services that run in a cloud-native environment, all the framework components run as containers.

3.2 Related works

Several works addressed the challenge related to monitoring, especially in the context of cloud computing, which has been extensively studied in the literature. Different monitoring solutions have been designed to monitor traditional IT infrastructures and cloud environments, such as Prometheus[1], Zabbix[2], and collectd[3]. In [14] a survey of cloud monitoring tools is conducted. It presents common characteristics, differences, strengths, and weaknesses of each reviewed monitoring tool. However, these solutions alone are not suitable for 5G relying on network slicing. Firstly, these solutions cover only one technological domain, the Cloud domain, while

[1]https://prometheus.io/
[2]https://www.zabbix.com/
[3]https://collectd.org/

a network slice spans over different technological domains. Secondly, they cannot easily scale with the number of network slices, as only one component is in charge of collecting data, hence it constitutes the system bottleneck.
As stated earlier, our objective is to provide a framework featuring an end-to-end monitoring solution for 5G supporting network slicing. To the authors' best knowledge, the proposed framework in this work is the first end-to-end monitoring solution for a 5G network relying on network slices, covering all the needed technological domains to deploy end-to-end network slices.
The work in [27] introduces a system that monitors multiple domains of a 5G infrastructure. The system consists of a metrics extraction function (MEF) that extracts and translates metrics, one MEF per monitored infrastructure component. The MEF extracts the metrics from the monitored component and exposes them to the upper layer where a broker system is deployed. A metrics aggregation component consumes the metrics provided by the MEFs from the broker and provides them to other tools responsible for metrics analysis and data visualization. Finally, a metric management entity is responsible for configuring the system entities. The system does not take into account network slicing and does not differentiate metrics from different slices. In addition, metrics are transferred without specific format or identifying elements, making it more suitable for one network slice at a time since deploying multiple services or slices will result in multiple unidentified data being stored in the system. In [28] the authors introduce a prototype for RAN monitoring implemented on top of ElasticSearch[4] and FlexRAN [29]. It includes a producer API that writes measured data and statistics from the southbound control plane (using the FlexRAN controller) to the data store. The SDK has a filtering module that performs filtering operations such as selecting data or aggregating results. The solution focuses on RAN KPIs and can be considered a source of metrics for the RAN domain. In [30], the authors propose an elastic monitoring solution for end-to-end cloud slices. The proposed system relies on other monitoring systems, called Monitoring Entities (MEs), and deploys adapters per ME and slice. The adapters collect the KPIs from the monitoring entities and send them to a distribution mechanism using RabbitMQ[5] to be then consumed by a slice aggregator that stores the KPI in a dedicated database. The slice identification is made at the adapter level, which creates a coupling between the management and deployment levels. In contrast, in our solution, the metric is mapped with the network slice part at the SO slice-specific collector using information coming from the sub-slice orchestrators (technological domain orchestrators), which allows keeping independence between the different entities involved in the monitoring process and the technological domain orchestrators. Last but not least, this solution does not collect RAN metrics.
In [31], the authors introduce a slice monitoring abstraction mechanism for data center slices relying on Lattice [32]. Monitoring a network slice resources is done via slice monitoring adapters using Lattice data sources, which are in charge of collecting the different KPI using probes. The latter collects relevant measurements for a segment of the end-to-end slice. Besides disregarding the slice elasticity, the solution is designed for data center slices ignoring the RAN monitoring.
Work in [33] proposes a flexible monitoring framework that creates monitoring slices integrating cloud-specific monitoring solutions like Openstack Ceilometer[6] and non-cloud monitoring solutions like Nagios[7] and MRTG[8]. Again, the authors did not discuss the scenarios of multi-clouds and multiple technological domains. DASMO, introduced in [34], proposes the modification of the NFV architecture to support monitoring by embedding the monitoring elements into the

[4]https://www.elastic.co/elasticsearch/
[5]https://www.rabbitmq.com/
[6]https://docs.openstack.org/ceilometer
[7]https://www.nagios.com/
[8]https://www.mrtg.com/

Element Manager (EM) of a VNF. The work also tackles the monitoring's scalability, but no implementation nor performance evaluation of the solution has been conducted; it stays at the conceptual level.
Table 3.1 presents a comparison of our solution with [27], [30] and [31]. The three solutions share the same objective, which is monitoring platforms for network slicing.

3.3 A monitoring framework for end-to-end network slices

In this section, we will introduce the proposed monitoring framework of network slices in 5G and beyond networks. The section is divided into three parts: (1) the considered architecture for enabling monitoring in 5G; (2) the data collection servers and the monitoring protocol; (3) the data presentation.

3.3.1 Architecture

Network Slicing Management and Orchestration architecture

Before detailing the proposed monitoring framework architecture, we first introduce the network slicing architecture we build on. The latter is a generic architecture that allows orchestrating and managing the life-cycle of a network slice, including the monitoring process. It is based on the architecture introduced in [35] and shown in Fig. 3.1. The proposed architecture is composed of a Slice Orchestrator (SO), which is in charge of the Life-Cycle Management (LCM) of network slices and monitoring its performance. The SO is equivalent to the 3GPP Network Slice Management Function (NSMF) [36] and exposes Northbound API (NBI) for the OSS/BSS or Communication Service Management Function (CSMF) as specified by 3rd Generation Partnership Project (3GPP). The NBI covers the LCM of a network slice and the management of the monitoring process. The NS LCM API is already specified in [36] and manages the different steps of the network slice life-cycle, like commissioning, operation, and decommissioning. However, no API is specified to manage the monitoring of network slices. Therefore, in this work, we devised a new NBI to be exposed by SO to manage the monitoring of network slices.

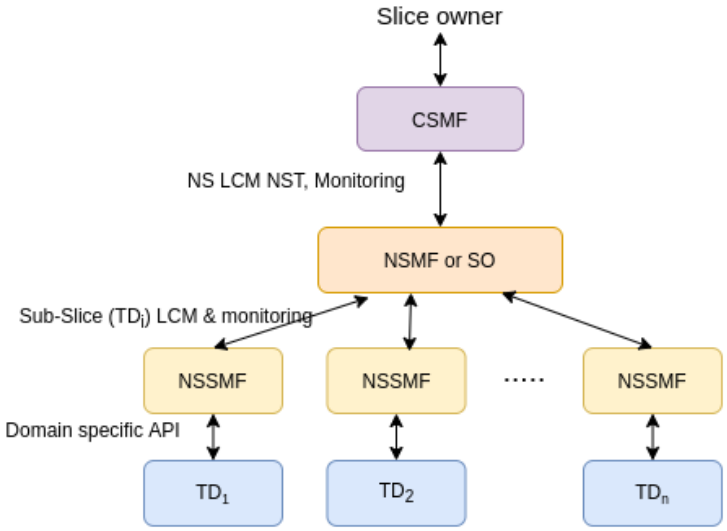


Figure 3.1: Network slicing architecture

Table 3.1: Comparison between [27], [30], [31], and our monitoring system

Aspect	**[30]**	**[31]**	**[27]**	**Our system**
Metrics source	From Monitoring Entities (Prometheus, NetData).	Based on lattice data sources.	Metrics generated by Infrastructure Components	From Kubernetes API, FlexRAN with possibility of adding plugins for Prometheus, Ceilometer.
Slice deployment	Support multiple slice parts on top of the same VIM.	Consider a VIM per slice.	Does not support network slicing	Support multiple slice parts on top of the same VIM.
Monitoring management	Controlled by an Engine Controller that communicates with the Slice Orchestrator.	Use a monitoring controller that communicates with the Slice Orchestrator.	Metrics Management Entity configures the system entities	Use a monitoring engine integrated with the SO and communicate with the NFVO and RANO. The NFVO and the RANO participate in the creation and the configuration of the collection agents for the slice parts.
Dynamic deployment	Use adapters deployed for each slice part.	For each slice a uniform on-demand monitoring layer is built.	For each infrastructure component the system have a Metrics Extraction Function	Uses collection agents for each slice part that run as containers.
Slice elasticity	Horizontal elasticity is handled by deploying new adapters and monitoring entities. Vertical elasticity is handled by default.	Does not consider slice elasticity techniques.	Introduction of a new infrastructure componenet is by deploying a new Metrics Extraction Function	Slice horizontal elasticity and the integration of new technological domains is handled by the deployment of new metrics collectors. Vertical elasticity is handled by default.
Metrics exposition	Expose the metrics to the Slice Orchestrator for elasticity operations.	The metrics can be consumed by the Slice Orchestrator as feedback for the execution of its services and functions.	Expose the metrics to data visualisation and analysis tools	Alerts can be sent to the SO based on the collected KPIs. The KPIs are exposed to the slice tenant using RabbitMQ for publish/subscribe and Grafana for visualization.
Technological domains	Cloud, with support of multiple clouds.	Cloud, with support of multiple clouds.	Cloud and RAN	Cloud with support of multiple clouds, and RAN with support of the FlexRAN controller.
Metrics abstraction	provide the element ID with the metrics	No abstraction	No abstraction	Introduces a data collection protocol for multi-domain network slices

In the envisioned network slice architecture, each technological domain is managed and orchestrated by its own entity, known in 3GPP as Network Sub Slice Management Function (NSSMF). Depending on the technological domain, a NSSMF may correspond to Network Function Virtualization Orchestrator (NFVO) for Cloud/Edge domain, Radio Access Network Orchestrator (RANO) for RAN domain, and SDN controller for the case of the transport network domain. Examples of existing tools covering these functions are ONAP[9] and OSM[10] for NFVO, FlexRAN controller[29] for RANO, and OpenDayLight[11] and ONOS[12] for transport orchestrators.
To deploy and instantiate a network slice, the CSMF or OSS uses the NBI exposed by SO to describe the needed resources (Compute and network) through a Network Slice Template (NST). It includes attributes and meta-data on the network slice (ex. the start date and end date, slice owner, type of slice, etc.), and information on each sub-slice composing the network slice. GSMA is providing examples of NST available in [37], namely Generic Slice Template (GST). To enable on-demand monitoring, we propose to extend the NST to indicate if monitoring is needed. If so, the slice owner should include the KPI list to monitor each technological domain where the network slice is deployed. The template is created by the slice owner using an existing Blueprint provided by the operator or defining a new one through Intent. Each sub-slice (technological domain) composing the network slice is described in NST. For instance, in the case of the computing resource (i.e., Cloud/Edge domain), the NST may include information such as the number of CPUs, memory, and the virtualization technology (i.e., Virtual Machines (VM) or containers) to be used. For the RAN domain, resources may be related to the functional split type [38], the MAC scheduler algorithm, the number of PRBs, and others. Finally, for the transport domain, resources may include the type of link (bandwidth, latency), number of Virtual Local Area network (VLAN), front haul link capacity, Virtual Private Network (VPN) links, and QoS. Each technological domain needed resources is enclosed in the NST in the form of a technological domain-specific descriptor. For instance, for the Cloud/Edge domain, the resources are described using a Network Service Descriptor (NSD) that includes the VNF(s) list, the link to their VNF Descriptor (VNFD) or Application Descriptor (AppD) [39], how they should be interconnected, and the number of computing resources needed by each VNF. NST is passed by CSMF to NSMF when a network slice creation is requested.

The proposed monitoring framework architecture

Building on the network slicing architecture (Fig. 3.1), we introduce in Fig. 3.2 the overall architecture of the proposed monitoring framework, where two technological domains (RAN and Edge/Cloud) are considered and detailed. Besides the management and orchestration entities mentioned earlier (i.e., SO, NFVO, and RANO), the proposed architecture includes Data Collection Servers, a Data Presentation Server, and network slice-specific components. The data collection servers are composed of Brokers deployed at two different levels: a high-level Broker is used to expose collected monitoring data to the end-users in RAW format, and a low-level Broker to collect data from the different technological domains. The high-level Broker is based on RabbitMQ, while the low-level Broker is relying on Kafka. In one of the next sections, we will explain in more detail the motivation behind this choice. On the other hand, the presentation server is in charge of displaying the collected data per slice to the slice owners via a dashboard and graphical user interfaces. The presentation server consumes the monitored data per network slice stored in a Database (DB) at the SO, known as KPI DB.

[9]https://www.onap.org/
[10]https://osm.etsi.org/
[11]https://www.opendaylight.org/
[12]https://opennetworking.org/onos/

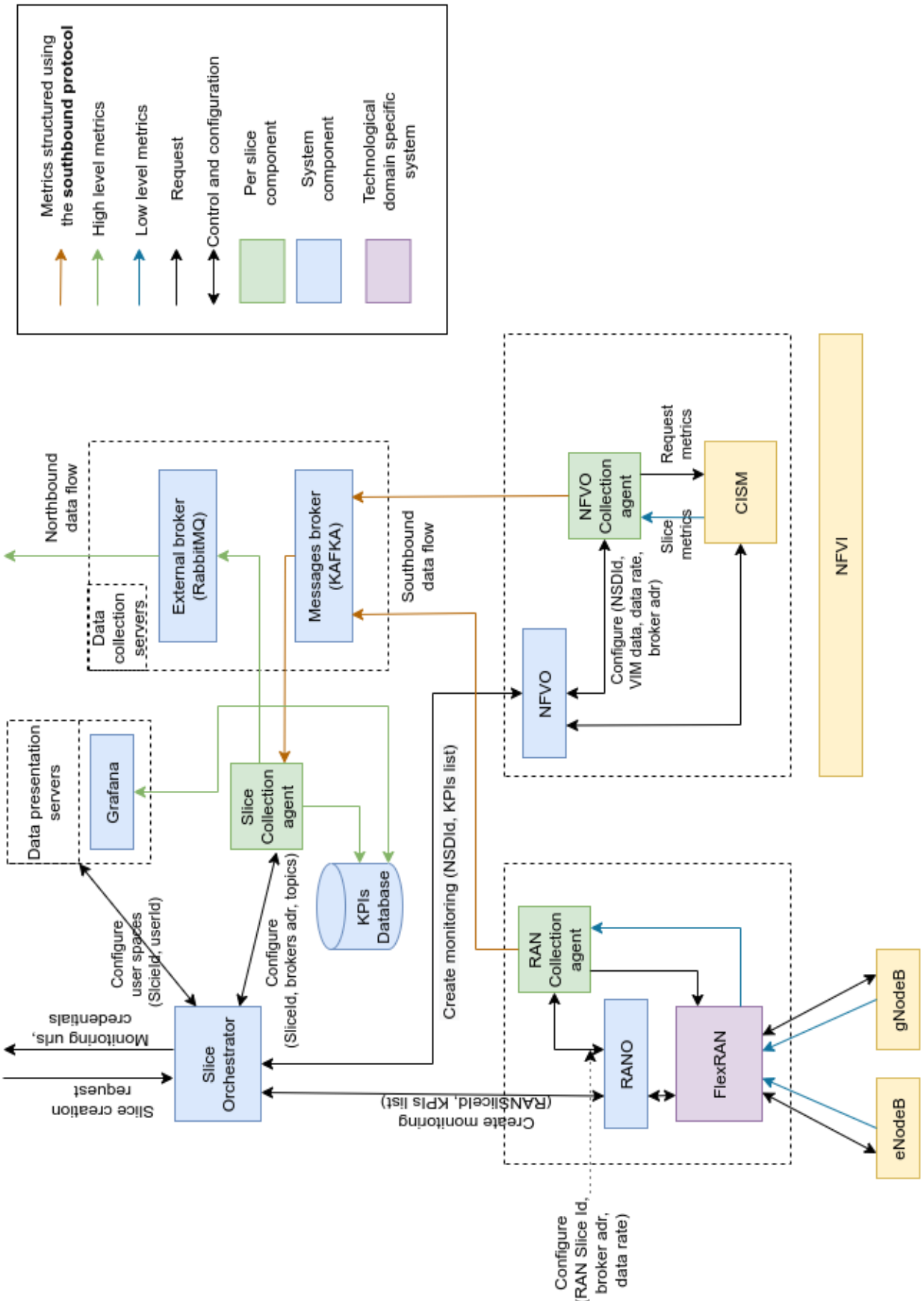


Figure 3.2: Architecture of the monitoring system

It should be noted that the proposed framework's critical components are the slice collection agents that collect monitoring data of each sub-slice and aggregate them per slice. These agents are instantiated when the network slice is created and active throughout its life-cycle. In the considered scenario of two technological domains, three agents are instantiated per network slice, one by the SO and two by the technological domain orchestrators, i.e., at the RANO and NFVO. The role of the SO-level agent is to consume the data published by the slice-specific agents, add meta-data on the network slice, and push it to the high-level Broker. After that, the data will be consumed by (i) the slice owner as RAW data; (ii) the data presentation server to be displayed via the dashboard. Whereas the role of the agents instantiated by the technological domain orchestrators is to collect data from the infrastructure, format the obtained data by adding meta-data, and push it to the lower-level Broker of the data collection server, i.e., Kafka. Once the network slice is terminated, these three elements are deleted. Note that we assume that the slice agents are instantiated as containers and run in the Cloud/Edge domain. Obviously, using slice specific agents will guarantee that the proposed monitoring framework scales with the number of slices. Besides, if another technological domain is used, such as the transport network domain, only another collector agent is added; without impacting the overall system and the other agents' functions.

Data Collection procedures

As mentioned in the precedent section, the data collection is done by the slice specific collectors, which are instantiated by the different entities involved in the network slice LCM, i.e., SO, and technological domain orchestrators when the network slice is created. In this section, we will detail the instantiation process of the slice specific collectors and their functions, considering two technological domains RAN and Cloud/Edge, managed by RANO and NFVO, respectively.

SO SO is the entry point of the system. It interfaces with the slice owner via OSS/BSS or via CSMF. It receives requests to deploy a network slice in the form of a NST. To recall, the NST includes details and attributes on the network slice to deploy, including the list of KPI to measure for each technological domain. The SO relies on the API exposed by the NSSMF (in our case RANO and NFVO) to first deploy the network slice. Once NFVO and RANO create the sub-slices, an Id for each sub-slice is returned to the SO, namely RANId and NSDId, respectively. SO stores that information on a local DB and links the RANId as well as NSDId to the NSId of the created slice. The second step is the instantiation of the three slice-specific collectors. To do so, SO sends a request to both RANO and NFVO to create the technological domain slice-specific collectors by indicating the sub-slice Id and the list of KPI to collect per domain. Once the RANO and NFVO validate the request, SO instantiates and configures the slice collector agent. The configuration consists of providing information on the Slice ID, the IP addresses of the Brokers, DB where to store the data pushed by the technological domain collectors, and the topics to be used to fetch data at the low-level Broker (i.e., sub-slices ID) as well as topics to publish to at the high-level Broker (i.e., slice ID). SO also needs to inform the presentation server about the creation of a new slice by communicating the Slice ID. As a response, the presentation server communicates a URL and credentials to observe the measured KPI in real-time via a dashboard. SO acknowledges the creation of the network slice and the monitoring system to CSMF by providing (1) URL and credentials for the dashboard; (2) URL and the topic where to fetch raw data of the collected monitoring KPI.

RANO Once receiving the request to create a slice-specific data collector from SO for a sub-slice identified by its RANId, RANO instantiates a slice-dedicated collector agent. The latter is

configured with the list of KPI to measure as well as the IP address of the message Broker and the topic to publish to.
To collect KPI from the RAN infrastructure (i.e., eNB/gNB), we rely on the concept of programmable RAN under investigation by the O-RAN initiative [40]. Programmable RAN allows opening RAN through a NBI to be exposed by the eNB/gNB to extract monitoring information or update the configuration on the run-time of eNB/gNB. O-RAN is currently standardizing the different interfaces between the eNB/gNB and a remote RAN controller or a RAN orchestrator. In the proposed framework, we use FlexRAN, a programmable RAN initiative on top of the OpenAirInterface (OAI) eNB/gNB. FlexRAN is composed of an agent sitting at the eNB/gNB. Its role is to extract information from the eNB/gNB, send them to the FlexRAN controller, and enforce at run-time an eNB/gNB configuration policy received from the FlexRAN controller; for instance, to create a new radio slice and assign resources to the network slice. Fig. 3.3 illustrates the interaction between the RAN data collector and FlexRAN controller as well as RANO. The RAN data collector periodically requests data on the performance of the sub-slice using the RANId. The collected data includes all information regarding the sub-slices. Therefore, some KPIs can directly be mapped to information provided by FlexRAN, while others need to be obtained by combining different information. The list of RAN KPI supported by our framework is summarized in Table 3.2.
The RAN slice-specific data collector first extracts the data regarding KPI from FlexRAN, or uses a local logic to deduce the KPI if not directly available with FlexRAN. Then, it formats the data by adding meta-data and creates a message using the monitoring protocol described later. Finally, it publishes the message to the low-lever Broker using RANId as a topic.

Table 3.2: RAN KPIs list

KPI	**Level**	**Details**
Latency-RAN	RAN	Latency at the RAN (aggregated per slice)
Uplink-data-rate	RAN	Uplink data rate (aggregated per slice)
Downlink-data-rate	RAN	Downlink data rate (aggregated per slice)
Packet-Loss-rate	RAN	Packet loss at the RAN after attempts (RLC layer) (aggregated per slice)
IP-rate	RAN	Packet rate (at PDCP) (aggregated per slice)
Latency-eNB-CN	RAN	Measured RTT between the RAN and CN
Bandwidth	RAN	Bandwidth of cells

NFVO The Cloud/Edge slice-specific data collector shares similar features with the RAN data collector. It is instantiated by NFVO per the request of SO using the NSDId as an identifier. All the collected data on KPI are published on the same message Broker (the low-level) using the NSDId as a topic. It formats data by adding meta-data and generates a message using the same monitoring protocol. In contrast, the Cloud/Edge slice-specific collector relies on other tools to collect KPI on the running sub-slices, particularly for the VNF. Indeed, the KPI to measure for Cloud/Edge domain covers the computing resources and virtual network resources used by the running VNF. Therefore, to collect this data, we rely, in our proposed framework, on the API provided by VIM. However, as we consider in the proposed framework that all VNFs run as Cloud-native Network Function (CNF), i.e., using container-based virtualization, Container

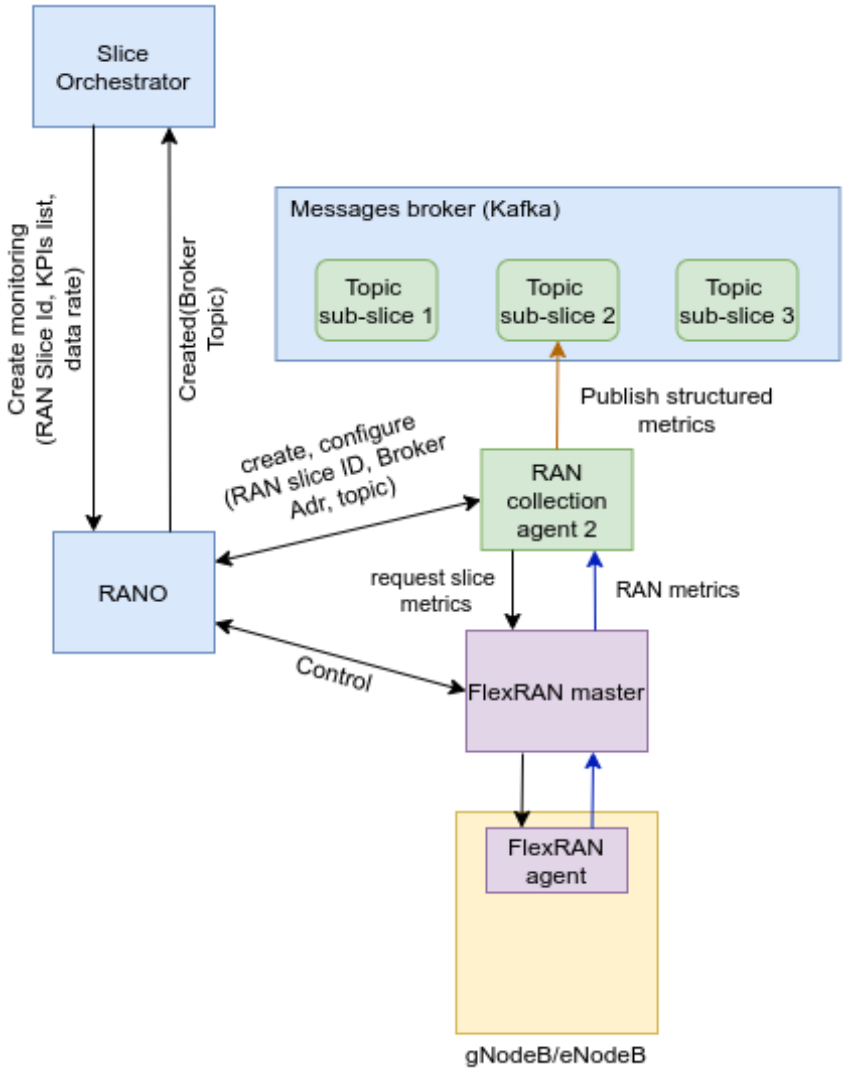


Figure 3.3: RAN KPIs collection sub-system

Infrastructure Service (CIS) as defined by ETSI [41] replaces the VIM. The CIS is managed by CIS Manager (CISM), which, technologically speaking, corresponds to the Kubernetes[13]. Indeed, Kubernetes provides an NBI that allows collecting several KPIs on the running containers, hence on VNFs. The Cloud/Edge data collector collects the KPI list communicated by the NFVO by consuming the Kubernetes NBI API. It is worth noting that Cloud/Edge sub-slice is composed of a set of VNFs. Thus, NFVO, when instantiating the Cloud/Edge data collector, it communicates the list of VNF instances (Id, names) known by NFVO and linked to the NSD Id. At the CISM level (i.e., Kubernetes), VNFs are identified with their Instance ID. It should be noted that in this work, we relied on Kubernetes NBI to collect KPI, but the platform can use other monitoring collection systems on top of Kubernetes, such as Prometheus.
Table 3.3 summarizes the list of some KPIs that we consider important in the context of network slicing and can be measured by the Cloud/Edge data collector.

Table 3.3: NFVO KPIs list

KPI	Level	Details
CPU-utilization	NFVO	Aggregated per CNF
Memory-utilization	NFVO	Aggregated per CNF
Number-instances	NFVO	Number of ReplicaSets per CNF
Network-Rx	NFVO	Aggregated per CNF
Network-Tx	NFVO	Aggregated per CNF

3.3.2 Data Transfer

Monitoring protocol and message format

One of the big challenges that we need to overcome when monitoring the network slice performances (i.e., KPI) is the span of resources over different technological domains. Indeed, each domain has its own system to collect data from the infrastructure. Besides, to the authors best knowledge, there is no existing data collection protocol available in the literature that can address the mentioned concern.Therefore, we propose a novel data collection protocol that defines common monitoring messages for all the technological domains, aiming to abstract lower-level technological domains' infrastructure specificities. Each domain slice-specific data collector will use these messages to encapsulate a monitoring measure and publish it. By doing so, we first simplify the aggregation process to be done by the slice data collector at the SO level. Second, adding a new technological data collector does not impact the other components of the monitoring platform, making it easy to extend the platform in the future. The monitoring communication protocol relies on the Publish/Subscribe concept, where the data collectors produce monitoring data while the slice data collector consumes that data.
All data collected by the technological domain collectors are encapsulated in a common message format shown in Fig. 3.4. All the fields are detailed in Table 3.4.
Each message shall include the sub-slice ID, Controller ID, Timestamp, metric (or KPI) name, and value. The latter corresponds to the measured data. The sub-slice ID indicates the ID of the sub-slice used as a topic for the Publish/Subscribe protocol. The NSSMF field indicates the name of the orchestrator of the technological domain where the data is coming from. The Timestamp field indicates the time when the message is generated. This information is very

[13]https://kubernetes.io/

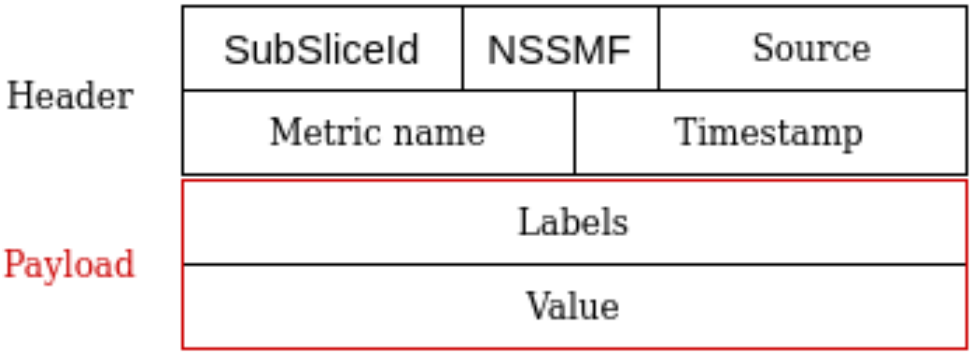


Figure 3.4: KPI message structure

Table 3.4: KPI message fields description

Field	Description	Type
SubSliceId	(required) The Id of the subslice (the slicing part) from which the metric was collected.	String
NSSMF	(required) The type of the subslice orchestrator (NFVO, RANO)	String
Timestamp	(required) The time of the metric collection	Long
Metric name	(required) The name of the metric	String
Source	(required) If the Controller Id == NFVO then it should include the VNF name	String
Labels	A set of key-value pairs that help in adding information to the metrics	Key: string value: string, integer, boolean
Value	(required) The value of the metrics	Double, Integer

important as it allows us to correlate the KPI coming from different technological domains, which will help to understand the network slice performances' behavior from an end-to-end perspective. Therefore, all the technological domains should be synchronized using the same clock based on GPS and IEEE Precision Time Protocol (PTP). The metric name indicates to which KPI the measured data belongs. The Source field is mandatory only if NSSMF corresponds to NFVO. Indeed, the source field indicates the VNF name concerned with the measured data. The last field, namely the label, is not mandatory. It helps to provide multidimensional metrics; for example, if we consider measuring the Channel Quality Indicator (CQI) as calculated by UE in RAN, the label field should include the UE Id (International Mobile Subscriber Identity (IMSI) or other identifiers).

Data collection servers

The data collection servers are in charge of collecting data from the domain slice-specific data collector deployed for each running network slice. The data collection servers are hard components of the framework as they need to be run in parallel to SO, NFVO, and RANO. The data collection servers use the Publish/Subscribe protocol with two levels. The first level (high-level) allows the slice owner to consume the monitoring data regarding its run slice in RAW format. The Broker used at this level is based on RabbitMQ, a push-based system. This choice is motivated by the fact that RabbitMQ allows more control over the message routing. It offers more elaborate routing capabilities by providing various exchanges (direct, fan-out, headers, topic). Therefore, using this type of Broker allows the monitoring platform to control what metrics to send to each user and avoid requesting that the consumer manage the messages offset from which it needs to consume. Moreover, the messages are deleted after consumption, which avoids storing redundant data on a metric for a long period of time. Finally, in RabbitMQ, an interesting feature is the possibility of creating a virtual host (vHost) containing the exchanges and their corresponding queues of each user. Those vHosts are used to define the users' permissions and constitute the user space in the Broker. Based on vHosts, we can ensure multi-tenancy and isolation between network slices.
The second level (low-level) is internal to the framework components. It allows collecting the monitoring data generated by the technological domain slice-specific data collector and pushing the data to the concerned slice data collector (at SO level). The Broker at this level is based on Kafka. We argue this choice by the fact that Kafka provides routing by topic, which gives a simple and robust model for internal metrics transfer. Kafka is a pull-based system where the consumers use an offset to access the messages in a topic, and the messages in a topic have a retention period, which allows the consumption of the messages multiple times; this cannot be done using RabbitMQ in which the messages are deleted after consumption. Such a feature allows more control over the consumed messages. SO can change the offset of the messages to consume, hence allowing the collection of messages multiple times if needed (in case of database writing problems or collector failure, which results in the deployment of a new collector that can access the metrics that were not handled correctly). Another feature is the notion of partitions within a topic. It allows a group of competitive consumers to get metrics from the same topic in a round-robin way, making the system's scaling easier while keeping a simple model of message routing. Indeed, if the metrics' rate gets higher, new slice collectors can be instantiated for the concerned slice.

3.3.3 Data presentation

How data is presented to the user is a very important criterion of a monitoring system. Indeed, the presentation of monitoring data should be adapted to the level of technical knowledge of users. Some users may be satisfied only by visualizing the data through a friendly GUI or dashboard and reacting to any degradation. Other users may want to have access to the RAW data to store it, and later on, do further analysis using Machine Learning (ML) tools to build models predicting performances or detecting when a parameter is causing an issue. Therefore, the proposed platform covers both ways to present data, i.e., using a GUI and providing RAW data.
Regarding the RAW data, as previously explained, we used the high-level Broker, which is based on RabbitMQ, to expose the collected data, aggregated by the slice data collector. The user can fetch the queue identified by the Slice ID, which stores the monitoring messages. It is then the slice owner's responsibility to write a program or use a RabbitMQ client to connect to the message queue and consume data.
Regarding the graphical presentation, the data presentation server entity is in charge of this task. The data presentation server relies on Grafana[14] to expose via a Dashboard the monitoring messages. Grafana is an open-source, multi-platform, interactive web application for metrics analysis and visualization. It offers fast and flexible visualizations with a multitude of options like tables, graphs, and alerts. A large number of data sources are supported by Grafana, from which we use the InfluxDB[15] data source.
Once the data is available at the lower-level Broker, the slice data collector consumes the message, identifies the subslice, and adds the Slice Id to the metric. The metric then is stored in the KPIs database, and a measurement per metric name is provided and sent to the corresponding topic in the external Broker from which the slice owner can consume it. Grafana is configured when a Network Slice is created by the SO. The SO's monitoring engine creates a dedicated space for the slice owner and its running slices by preparing a folder that contains the dashboards that represent the performances of the user's slices.
It is worth noting that the network operator is granted full access to all the platform's monitoring information. Through Grafana, the network operator can see all the collected monitoring data, aggregated per slice, or aggregated per technological domain. The same information is available as RAW data in the KPI DB that can be used to run ML algorithms for troubleshooting prediction and mitigation, and resource usage optimization.

3.3.4 Data privacy and isolation

Multi-tenancy and network slice isolation are an important feature that the devised monitoring framework ensures. It is vital that a slice owner has access to monitored data corresponding to only its running slices. The network slice isolation needs to be mainly enforced when presenting the data to the network slice owners. In the proposed framework, the presentation server and the high-level Broker need to guarantee isolation, as they are interfacing with the network slice owners.
In order to ensure data isolation at the high-level Broker, the Slice ID and vHost are used to segregate the monitored data on running network slices. The Slice ID is unique and known only by SO and shared by the latter with the slice owner. The slice owner uses the Slice ID as a topic to fetch the monitoring data. The Slice ID is also communicated to the presentation server along with the User ID (Identifier of the Slice owner). By using a combination of User ID, Slice

[14] https://grafana.com/
[15] https://www.influxdata.com/

ID, and vHost, the presentation server can ensure that a slice owner (User ID) can access only data with respect to its running slices (identified via the Slice ID). It is worth recalling that the User ID is communicated by CSMF to SO when creating a network slice. SO stores and associates the User ID with the Slice ID when a network slice is created.

3.4 Performance evaluation

3.4.1 Testing environment

We have implemented the proposed monitoring framework on top of the 5G facility of EURECOM deployed in the context of the 5GEvE[16] and 5G!Drones[17] projects. EURECOM 5G facility includes all the element introduced in Fig. 3.1, i.e., SO, RANO and NFVO. Besides, it uses OpenAirInterface (OAI) [18] for the RAN infrastructure and a Openshift/Kubernetes for Cloud/Edge. We have implemented all the components described in Fig. 3.2. Table 3.5. summarizes the used technologies, which have been adapted and improved.

Table 3.5: The implemented components and the used technologies

Component	**Technology**
SO (NSMF)	Python-based
RANO (NSSMF)	Python-based
NFVO (NSSMF)	Python-based
CISM	Kubernetes
Data collection servers	RabbitMQ, Kafka
Data presentation server	Grafana
NBI API	REST

Our experiments were performed on two hosts (Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz 32GB RAM, 6 CPUs without hyper-threading: 1 Thread per core), which form a Kubernetes cluster using Kubeadm v1.19.5. The cluster represents the CIS infrastructure on top of which the cloud sub-slices will be deployed. All the platform components run as containers. It should be noted that the data collector servers (Kafka and RabbitMQ servers), the Data presentation server (Grafana), KPI DB, and the KPIs engine run in the same Kubernetes name-space to measure their resource consumption more precisely.
To get insight into the monitoring platform's performance, we have focused on evaluating two critical aspects. Firstly, we concentrate on the scalability issue when the number of running slices is high, where we measured the resource consumption of the whole system when increasing the number of deployed slices. Secondly, we shed-light on the performances of the system, where we measured the latency to present a collected data to the external consumer that a tenant deploys to consume a slice metrics. For all the experiments we used two values of the polling interval (i.e., interval of time to collect data), 1s and 5s. This will allow us to see the impact of polling interval on the measured KPI, knowing that 1sec is very demanding in terms of computing and networking resources. Finally, we collected eight KPI, four on the RAN and four on the Cloud/Edge.

[16]https://www.5g-eve.eu/
[17]https://5gdrones.eu/
[18]https://openairinterface.org/

3.4.2 System Scalability

The first presented results correspond to the performance of the monitoring system in terms of scalability. To obtain these results, we increase the number of network slices to monitor and measure the consumed computing resources, i.e., CPU and memory. We measured the consumed CPU and memory of the whole monitoring system, but also per component: data presentation, data collection, and the slice-specific collectors.

Memory

Fig. 3.5 represents the RAM consumption of the whole system (including the slice-specific collectors, the data collector serves, and the presentation server) in respect to the number of network slices. As it is expected, the RAM consumption increases (linearly) with the number of network slices to monitor. Besides, when the poling interval is small, the CPU consumption is high. We also remark that when the number of network slices is equal to 50, the RAM consumption exceeds 7 Gb and 8 Gb, for 5s and 1s of polling interval, respectively.

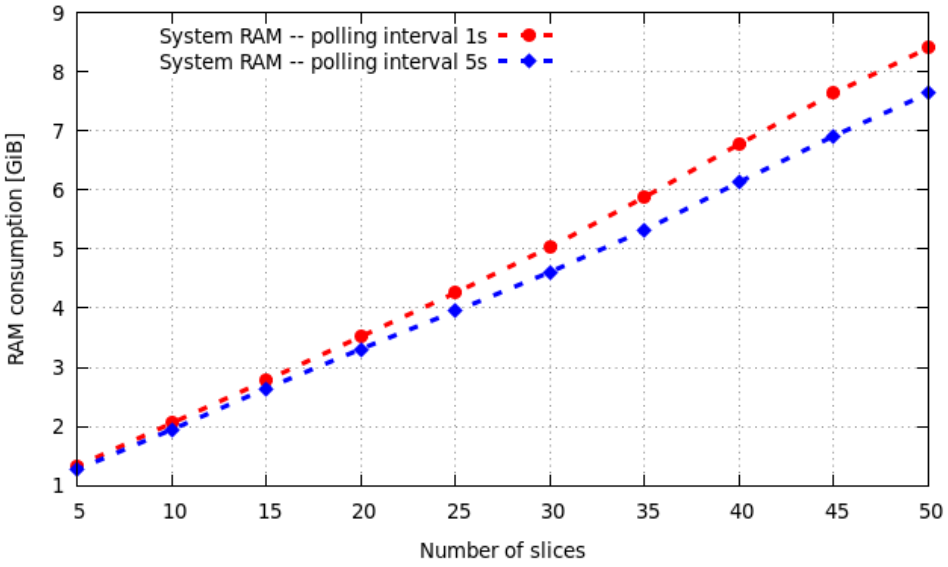


Figure 3.5: The RAM consumption of the system as a whole in relation to the number of slices and the polling interval

Now we investigate the RAM consumption per component. Fig. 3.6 and Fig. 3.7 illustrate the RAM consumption of the slice-specific data collectors and the data collection servers (Brokers), respectively. Notice that 1MiB = 2^{20} Bytes and 1GiB = 2^{30} Bytes. It is worth noting that only one curve is shown for both polling intervals as the RAM consumption is merely the same (only 0.1 MiB of difference). This is explained by the fact the poling interval does not impact the consumption of the collectors. Indeed, the memory space used by the collector is not affected by the polling interval, since its role is to request the measurements, structure them, and send them to the Brokers of the data collection servers, which do not require high computing resources. We recall that three collectors are instantiated per slice; one at SO, and one for each technological domain. We remark that the slice-specific collectors are consuming merely 80% of the RAM of the whole system. Obviously, the RAM consumption increases linearly with the number of monitored network slices; the RAM consumption reaches 6 GiB for a high number of monitored network slices. On the other hand, we remark that the brokers consume less RAM; less than 2

GiB and 2.4 GiB, when the number of slices is equal to 50 and for a polling interval of 5s and 1s, respectively.
For the Data presentation server, the RAM consumption is practically constant and is around 20 MiB.

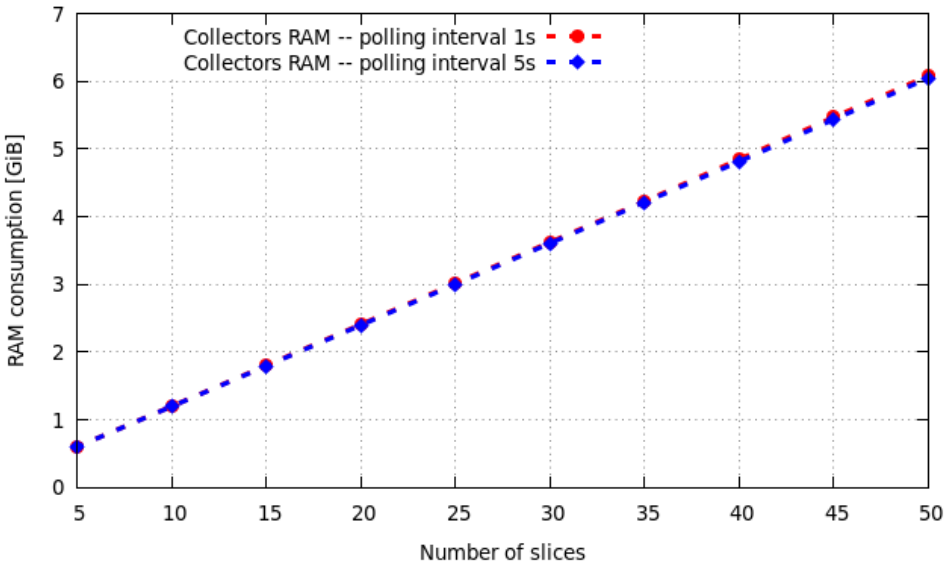


Figure 3.6: The RAM consumption of the slice-specific data collectors in relation to the number of slices and the polling interval

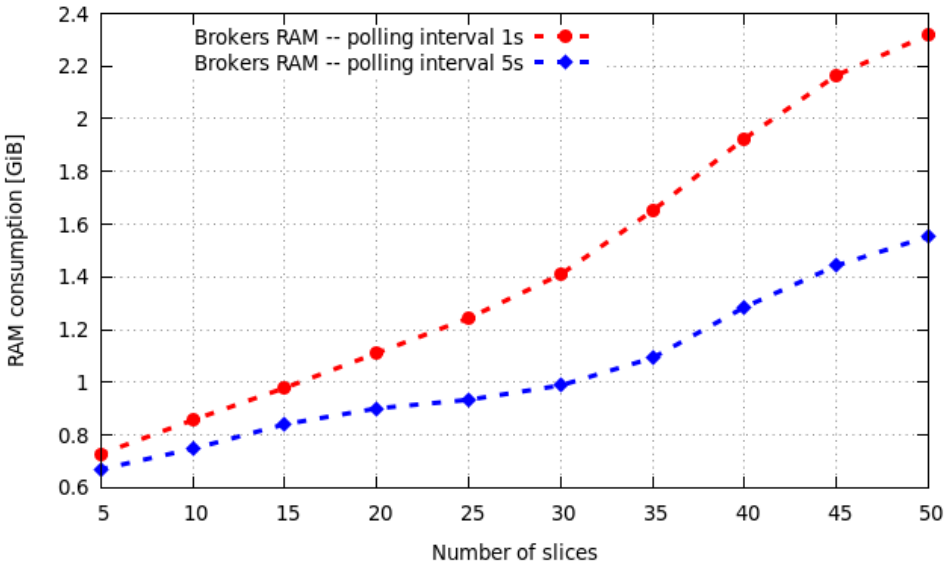


Figure 3.7: The RAM consumption of the data collection servers (data brokers) in relation to the number of slices and the polling interval

CPU

To calculate the CPU consumption of the monitoring system elements, we rely on cAdvisor[19], which is a daemon that collects, aggregates, processes, and exports information such as resource usage and performance characteristics about running containers. CAdvisor is integrated into Kubernetes and provides a "container_cpu_usage_seconds_total" metric that shows the cumulative CPU time consumed by a given container. This metric is scrapped periodically using Prometheus. Then, we apply a rate on the time series, i.e., we use a function that calculates the per-second average rate of increase of the time series in the given time interval, representing the CPU consumption of a container. Using this way of computing CPU explains the decimal values of CPU shown in the figures.
Fig. 3.8 shows the entire monitoring system CPU consumption with respect to the number of monitored network slices for the two polling intervals. Clearly, we observe a similar behavior as for the RAM, where the CPU consumption increases with the increase of the number of monitored slices. Besides, the CPU consumption is higher when the polling interval is small; it reaches 4.5 CPU and 3 CPU for 1s and 5s polling intervals, respectively, while the number of monitored slices is 50. This means that in a saturated situation, the whole system uses merely 5 CPU and 9 GiB of RAM, which is an acceptable value in modern hardware.

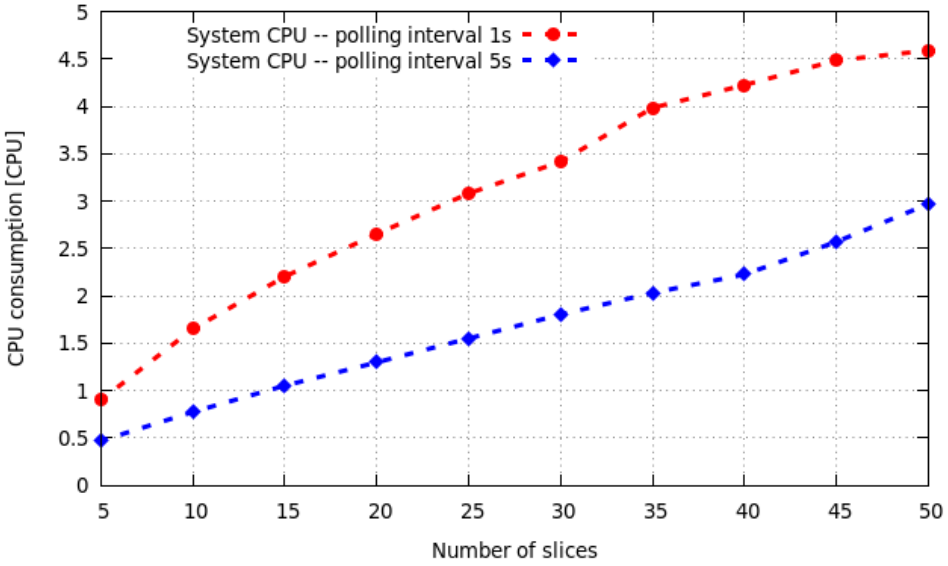


Figure 3.8: The CPU consumption of the system as a whole in relation to the number of slices and the polling interval

As for the RAM, we will investigate which components are consuming more CPU. Fig. 3.9 and Fig. 3.10 show CPU consumption of the slice-specific collectors and the data collection servers, respectively. As for the RAM, we remark that the slice-specific collectors consume more CPU, reaching 2.5 CPU and 3.7 CPU when the number of monitored network slices is 50 and for a polling interval of 5s and 1s, respectively. We also observe that reducing the polling interval has a strong impact on CPU consumption. When the number of monitored slices is high, the difference could reach merely double. Meanwhile, the difference is less obvious for the Brokers, where CPU consumption is not highly impacted by the polling interval. We remark the same behaviour when increasing the number of monitored slices; i.e., a small impact on CPU consumption. For instance, for a polling interval of 1s, the difference of consumed CPU between

[19]https://github.com/google/cadvisor

10 monitored slices and 50 is around 0.4 CPU. This indicates that the Brokers scale well with the number of network slices.

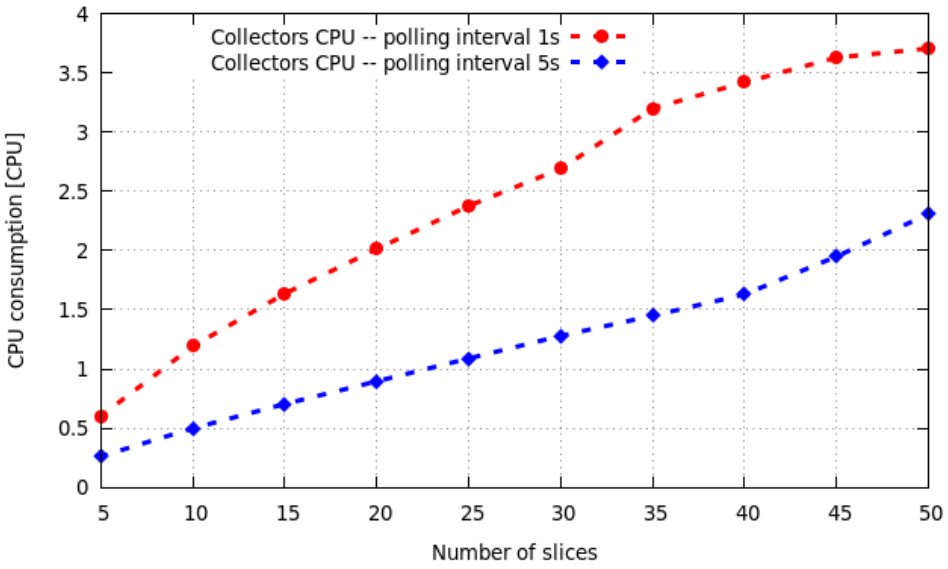


Figure 3.9: The CPU consumption of the slice-specific data collectors in relation to the number of slices and the polling interval

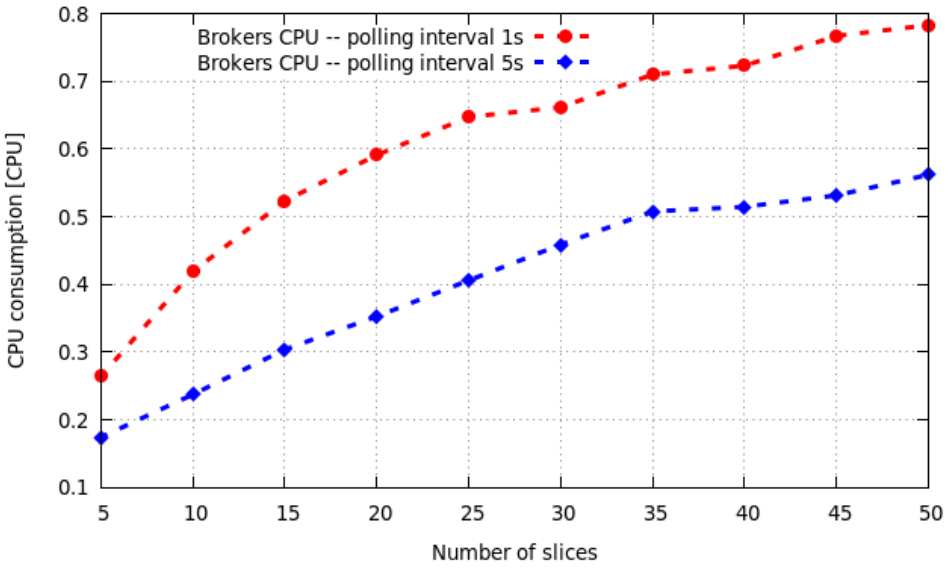


Figure 3.10: The RAM consumption of the data collection servers (data brokers) in relation to the number of slices and the polling interval

Regarding the data presentation server the CPU usage is very low and is around 0.004 CPU.

3.4.3 End-to-end messages latency

Now we turn our attention to the performance of the monitoring system in terms of latency to deliver the monitored data to the slice owner. To this aim, we measured the time taken

by the system to collect data and present them to the slice owner. We measure this latency for the two data collection polling intervals while increasing the number of monitored network slices. We clearly observe that the latency increases linearly in the case of a polling interval of 5s but exponentially for 1s. It reaches 160 ms and around 40 ms for a polling interval of 1s and 5s, respectively. This clearly proves the strong impact of the polling interval on the latency. However, it remains acceptable, less than 1s in each case, and even when the number of monitored network slices is very high.

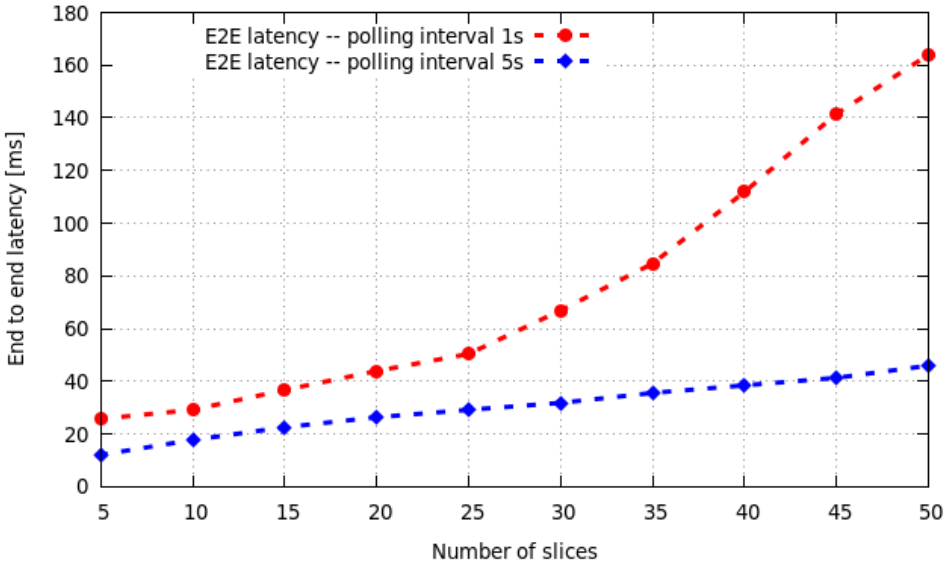


Figure 3.11: The end-to-end messages latency in relation to the number of slices and the polling interval

3.5 Conclusion

In this Chapter, we introduced a novel monitoring framework for network slicing in 5G. The proposed framework solves many issues that arise when monitoring the performances of network slices. First, it is a scalable framework, as slice data collectors are instantiated within a network slice and deleted when the network slice ends. Second, it allows monitoring resources of different technological domains and abstracts each domain's specificity by devising a novel data collection protocol. Third, aggregate the measured data at the slice level and provide it to the slice owner as raw data or via a graphical interface. The proposed framework has been implemented in a 5G facility and extensively evaluated. Obtained results indicate that even if a high number of network slices are deployed and monitored, the CPU and memory consumption remain sustainable by current hardware. In addition, the latency to present the data to the slice owner remains under 1s when a high number of network slices are deployed.
In the next Chapter, we will use the metrics collected by the monitoring system to study the impact of resource configuration on the microservices performance in cloud-native environments.

Chapter 4

Microservices Configurations and the Impact on the Performance in Cloud Native Environments

4.1 Introduction

In recent years, software development models have shifted from monolithic architectures to loosely coupled microservices. In the monolithic architecture model, all the components of the system are part of the same application, making it harder to deploy, manage, and improve its functionality due to the high coupling and dependence between the components of the application. While, in micro-service architecture, an application is decoupled into many loosely distributed services that have independent and straightforward functions following the single responsibility principle. Microservices, combined with containerization and container orchestration solutions such as Kubernetes [3], allowed the emergence of the cloud-native ecosystem. A recent model in which applications are developed to take full advantage of the distributed computing offered by the cloud. This evolution led many vertical industries to consider migrating their applications to cloud and Edge environments [42] to get full advantage of the cloud-native deployment and the benefits that it offers. Indeed, cloud-native offers reliable and self-healing deployment as containers are deployed using advanced container orchestration solutions such as Kubernetes, Openshift, etc. These solutions, in addition to the decomposability of the applications, reduce the points of failures and speed up the recovery in cases where failures occur in a set of microservices. The cloud-native has an impact not only on the vertical industry but also on other industries, such as telecommunication. Indeed, the 5th generation of mobile network (5G) relies on cloud-native to deploy the 5G core network functions, known as Service Based Architecture (SBA); all the network functions are cloud-ready and can be deployed in containers on a cloud or edge infrastructure. But despite this shift in application design, one fundamental problem is setting the configuration (needed computing resources) of individual microservices (i.e., container) to allow optimal running of the service on the one hand and to optimize the usage of the available resources on the other hand. Usually, the users or tenants settle for default configurations and update them manually, which either results in inferior performance or overuse of the resources in a cloud node. For this, reliance on users for providing resource configuration is not optimal as users do not have enough knowledge about the workloads, infrastructure, or orchestration systems [43].
In this Chapter, we run several experiments using a cloud-native platform to find solutions to overcome the resource configuration challenge. To this end, we study the behaviour of a set of

applications using different resource configurations and under different loads. We measure the performance of different types of applications, and we deduce the relationship between the latter and the resource allocated to the workload (CPU and memory) in both absolute and relative forms. This relation allows the detection of faulty configurations and the provision of more optimal configurations for the deployed applications. For the conducted tests, we considered four representative applications: three represent vertical applications, namely web servers, data brokers, and database; one represents a 5G core network service, namely Access and Mobility Management Function (AMF). Besides deriving in a generic way the threshold of CPU and memory limit from which an application does not run optimally and hence the probability of failure is high (as a container is killed if it exceeds its limit of resources), the results open several perspectives. Indeed, we have also constructed different datasets using the experiment results, which may be used to run Machine Learning (ML) to predict the performances of the workloads according to their configuration.

4.2 Related Works

Several works tackled the problem of optimising application performance while reducing the amount of resources used by the latter. Most of the works propose reactive and proactive auto-scaling methods in containerized deployments. The work in [44] studied the relevance of a set of metrics to be used as threshold metrics for scaling up and down a container. They concluded that CPU utilisation is a suitable metric for scaling all classes of micro-services unless the microservice experiences a change in its characteristics, such as I/O intensive applications. The paper also proposed a queue-based auto-scaling. However, this solution requires additional modules in the container orchestrator to queue the requests destined for each service. Another work on the relevance of metrics [45] studied the difference between absolute versus relative metrics, i.e., metrics providing the percentage of resources used by the application from allocated resources, in microservices autoscaling. They concluded that absolute metrics, such as CPU utilisation permit more accurate decisions on horizontal autoscaling than relative metrics. This is in the case of CPU-intensive applications when the relative CPU usage is between 85% and 90%. Work in [46] proposes a container load prediction model based on the application's historical data and using a bidirectional LSTM neural network. The model is trained on selected metrics offered by a Metrics Selection Module on Kubernetes. However, the solution requires a large amount of historical data about the performance of the target application, and it does not provide the best resource configuration for the application in order to handle the surge in load. In [47], the authors propose a Reinforcement Learning approach to autoscale microservices in the cloud. It uses two modules: the first one is a threshold-based auto-scaling algorithm deployed on Kubernetes (GKE), and the second module uses Reinforcement Learning to tune the autoscaler threshold values to obtain better threshold values to trigger autoscaling. The authors did not consider the relevance of the initial configuration. Work in [48] argues that a threshold-based scaling policy like the default Kubernetes HPA is not well suited to satisfy QoS requirements of latency-sensitive applications, which requires identifying the relationship between a system metric such as CPU utilisation and application metrics such as response time as well as to know the application bottlenecks. Their work focuses on CPU utilisation. Autopilot [49] is the autoscaler used by Google for horizontal and vertical auto-scaling of its internal workloads running on top of Borg [50]. The paper focuses on vertical auto-scaling of memory allocated to the workloads. Autopilot uses two main algorithms for vertical autoscaling: the first relies on an exponentially-smoothed sliding window over historic usage; the other is a Reinforcement Learning model that runs many variants of the sliding window algorithm and chooses the one

that would have historically resulted in the best performance for each job. For telecom use cases, [51] proposes a prediction model that leverages the time series data models and exploits the relationship of historical data to predict the future workload. The proposed solution scales horizontally the network function, such as the SGW (Serving Gateway function of the 4G Core Network). However, the authors did not study the behaviour of the network functions under different loads and did not study the effects of the single function configuration since the under-provisioning of memory or CPU can cause service deterioration even if multiple instances are deployed. Such deterioration can be caused by the container getting Out-of-memory errors or CPU throttling for individual functions.

4.3 Motivation

As stated earlier, most of the vertical applications are cloud-native and deployed in cloud and edge environments. However, the migration of services into a new environment introduces resource allocation difficulties. First, the tenant or the application owner generally does not have an expert understanding of both the application behaviour and functioning and the cloud-native environment in which it will be deployed to offer the desired services. Second, although resource over-provisioning might seem like an easy fix for this problem, this alternative introduces additional costs over long periods, which is worse when considering multiple instances of a service to deploy. Furthermore, overprovisioning can lead to resource wastage which is not acceptable in an edge environment. An over-provision of an edge application will cause an overload of the Edge server, thus restricting the number of services that can run in the latter. Overprovision is not optimal when considering node consolidation to reduce energy consumption, as the number of services to run on a node is not optimal.
On the other hand, if only horizontal scalability is available, the initial configuration of the service becomes crucial in determining the performance/resources used ratio, and a faulty configuration of the memory and CPU resources for an application can cause the container to run into Out Of Memory (OOM) errors and experience CPU throttling or be stopped (which is harmful on the service availability). The latter can be fixed by increasing the number of instances of the application, but this will come with the cost of doubling the memory allocated to the service even though the application can perform as expected using lower memory.
From the above, we investigate the behaviour of different types of services under variable load and resource limits. Our goal is to improve the initial resource configuration process and provide vertical scalability based on the limiting resource (CPU or memory). We perform tests to validate the assumption that CPU and memory relative values can be a strong indicator of the application performance and explore the cases where it is not.
The trade-off between resource efficiency and the application's performance is shown in Fig. 4.1. An application can have different resource configurations (1) an under-provisioning in which the application's performance is insufficient while the allocated resources to the application are lower than required; (2) An over-provisioning in which the application performs with respect to the Quality of Service (QoS). However, over-provisioning consumes much more resources than it needs. The last situation needs to be avoided, resulting in higher costs and higher energy consumption as more resources are needed to run the service.
In our work, we consider a cloud-native edge environment scenario, where tenants (mostly verticals) provide edge applications without indicating the needed computing resources to run them adequately, which often results in default configurations used for different applications and other virtualization platforms. However, using these configurations sometimes results in a degradation of the performance of the service. To tackle this issue and guarantee the necessary

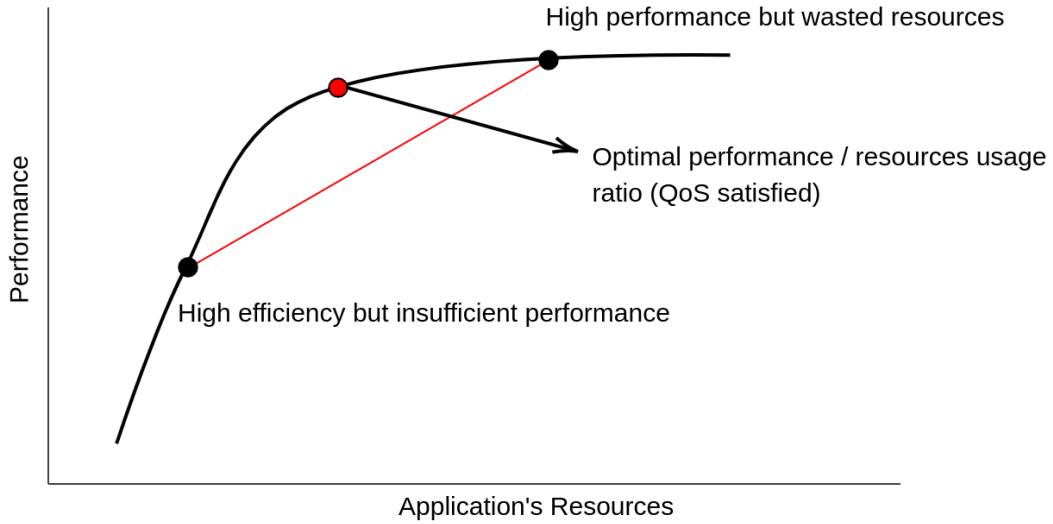


Figure 4.1: Trade-off between resource efficiency and application's performance

memory and CPU to run applications in near-optimal performance while reducing the allocated resources, we run experiments using 4 types of applications: 1) a Golang web server and a Python web server, since web applications are widely used and deployed as the frontend of any complex service; 2) a RabbitMQ message broker, as many applications use the publish-subscribe model for communication; 3) OpenAirInterface's 5G Core Network (CN), more specifically the AMF (Access and Mobility Management Function); 4) MongoDB database as most microservices are stateless and use a database to store data, and MongoDB is a widely used NoSQL database. For each application, we experiment with different configurations by changing the CPU limit, Memory limit, relative CPU usage (i.e. ratio), memory usage ratio, and the number of service requests received in parallel. For each configuration, we measured the latency to treat a service request, which reflects the performance of the service in terms of Quality of Service (QoS) seen by a service consumer.

4.4 Performance evaluation

In this section, we describe the performed tests and the obtained results. We have performed the tests on top of the cloud-native edge facility of EURECOM. The facility uses a Kubernetes cluster for container orchestration. We have implemented a benchmarking program that automatically deploys the application while allocating different resources for each deployment, mainly changes in CPU and memory allocated. We focused on these two metrics as they are the key resources used by the application since we did not consider access to storage or network latency and restrictions as the tested applications are run in the same cluster as the tester process. For each test, we measured the latency of the service to treat a number of requests received in parallel.

4.4.1 Testing environment

The test facility includes a Kubernetes cluster, which is deployed on top of an Intel server PowerEdge T440 with 128GB of RAM and 64 Core (Intel(R) Xeon(R) Silver 4216 CPU @

2.10GHz) with hyperthreading enabled. The cluster was bootstrapped using Kubeadm v1.20.1, and the host operating system is Ubuntu 18.04.5. All the tested applications (web servers, RabbitMQ, InfluxDB, and 5G core functions) run as containers in the cluster.
The testbed, shown in Fig. 4.2, includes a Prometheus [1] deployment for metrics collection, a Container Infrastructure Service Manager (CISM) [52] to manage workloads (automatic creation and deletion of the applications pods and its necessary Kubernetes services) and software for load tests such as ApacheBench [2] and RabbitMQ PerfTest [3].

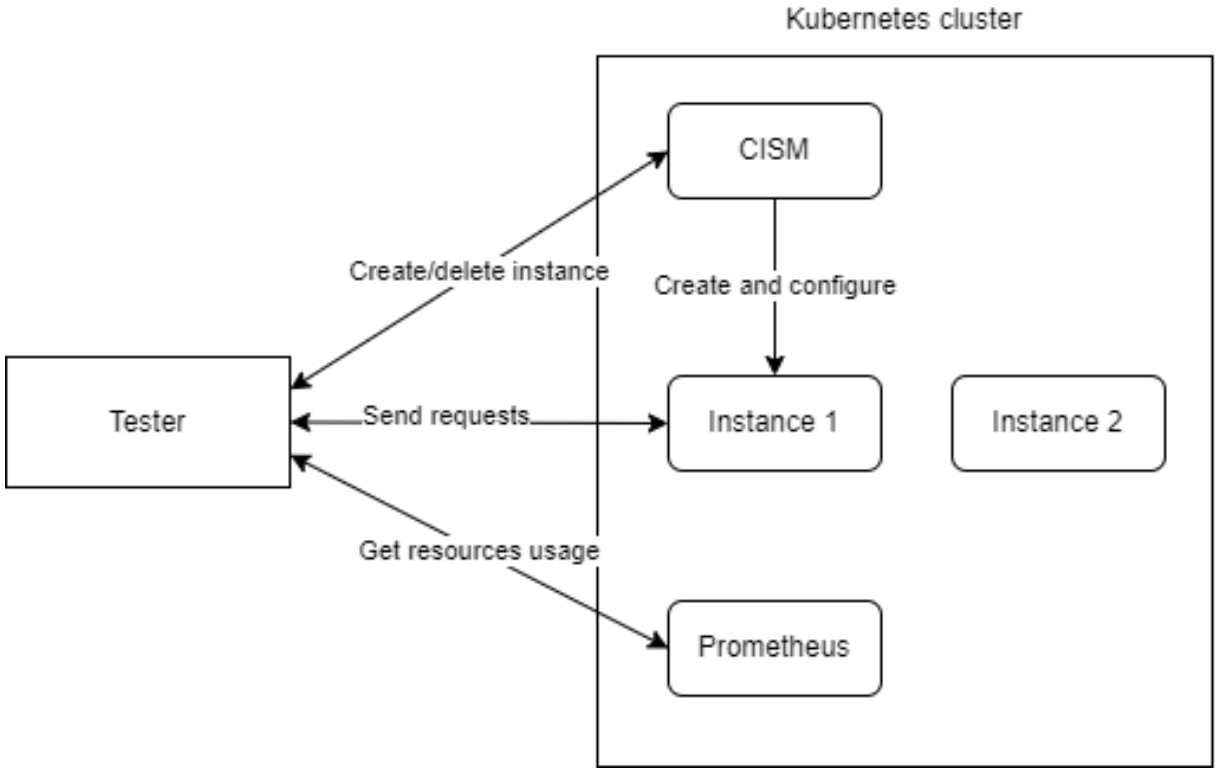


Figure 4.2: The components of the testbed

4.4.2 Obtained results

We performed the tests using the following procedure. First, the test program is configured by specifying the different combinations of allocated resources. The application is then deployed with one initial configuration of resources as a container in a Kubernetes pod. Once the container is running, the application is tested with a gradually increasing load (i.e., number of service requests in parallel) until all the requests are performed or the pod fails. While the tests are running, the tester collects resource usage from the Prometheus instance in the cluster and stores it in a file. Once all the tests are performed and the desired results are collected, the pod is deleted, and the next pod is deployed with the next configuration.

[1] https://prometheus.io/
[2] https://httpd.apache.org/docs/2.4/programs/ab.html
[3] https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/

Web servers

We used Golang and Python-based web servers for the test. Each request to the web server returns a video of size 43 MB, which allowed us to overload the server with a lower number of parallel requests. We used ApacheBench, a command-line program used for benchmarking HTTP web servers, to produce high traffic. ApacheBench permits to specify the number of requests that need to be sent to the web server and the level of concurrency, which allows parallel requests from multiple clients. We used a number of requests ranging from 100 to 1000 and a concurrency level between 1 and 100.
For allocated resources variation, we used configurations with the CPU value ranging between 0.5 and 4 CPUs with an increment of 0.5 CPU, and memory between 70 MB and 500 MB with 5MB increment between 70-100MB and 50MB increment from 100 to 500 MB. The obtained results from the two web servers were similar. The Golang-based server results are shown in Fig. 4.3 and Fig. 4.4. The performance/resources trade-off is shown in Fig. 4.3. We can notice that under different loads, represented by the number of requests sent to the server, the more CPU the application has, the lower the response latency is. This is true for CPU allocation between 0.5 and 2 CPUs; afterward, providing more CPU does not improve the latency of the web application. Fig. 4.4 shows the distribution of the response time. In this experiment, we consider the relative CPU and memory, which represent the percentage of resources used from the provided limit. We notice that the higher the relative CPU (shifting vertically from one graph to the one below) is, the greater the percentage of high response times is. In contrast, the memory percentage (moving horizontally from left to right) does not change the distribution of latency values.

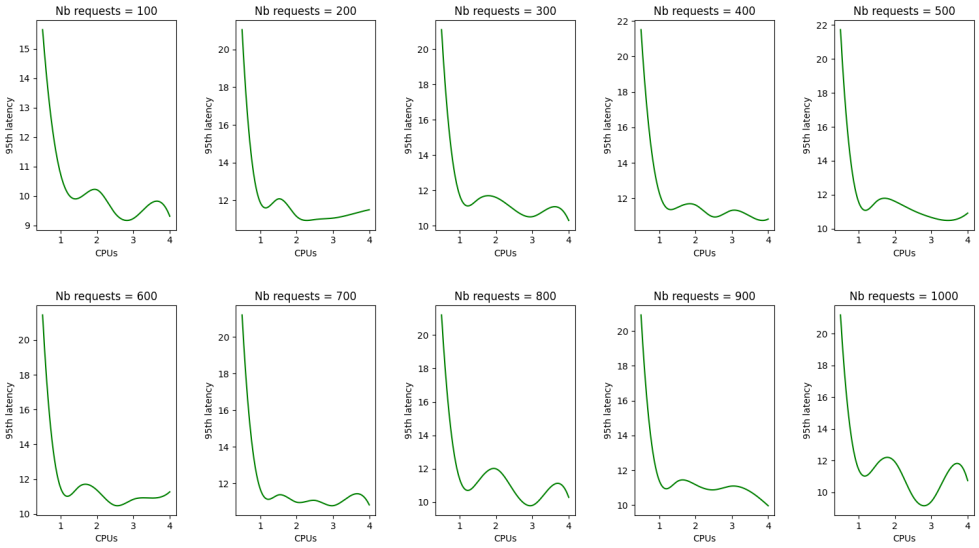


Figure 4.3: Web Server's 95% latency in relation to the allocated CPU

5G AMF

For the second use case, we study the performance of a 5G core network function: the AMF, which is a control plane function. Its main functions and responsibilities are registration, connection, as well as mobility management and access authentication and authorization. To obtain the

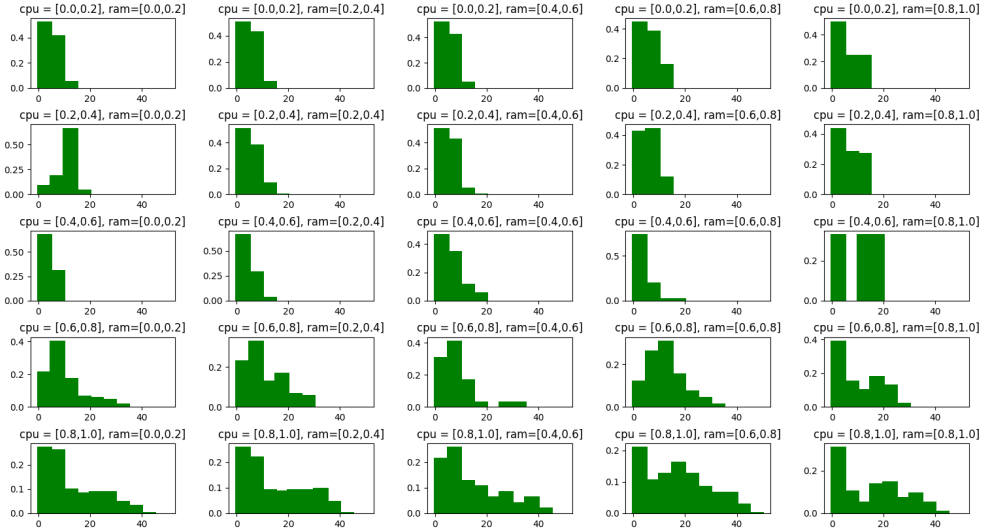


Figure 4.4: Web Server's 95% latency statistical distribution

performance of the AMF, we measure the registration time, which represents the time between the sending of the attachment requests from the UE (User Equipment) until the UE receives the authentication request.
To perform the test and generate 5G attach requests, we use my5G-RANTester [4], my5G-RANTester is a tool for emulating control and data planes of the UE and gNB (5G base station). my5G-RANTester follows the 3GPP Release 15 standard for NG-RAN (Next Generation-Radio Access Network). my5G-RANTester allows the generation of different workloads and testing of several functionalities of a 5G core, including its compliance with the 3GPP standards. Scalability is also a relevant feature of the my5G-RANTester, which is able to mimic the behaviour of a large number of UEs and gNBs accessing a 5G core simultaneously. We deploy an OpenAir-Interface [53] core network and specify the configuration of the AMF. The explored resource allocations vary from 0.5 to 4 CPUs with an increment of 0.5 for each test and memory from 256 MB to 4096 MB with an increment of 256 MB for each test. The number of simultaneous registration requests that are sent to each instance varies between 10 and 400.
The obtained results are shown in Fig. 4.5, where the mean values for registration time (i.e., the time needed to complete the User Equipment registration to the network) in relation to the allocated CPU are displayed. We can observe that AMF performances have the same behaviour as the webserver. Indeed, we remark that the higher the CPU allocation to the AMF is, the lower the registration time is. This is valid between 0.5 and 2 CPU, after which the performance is constant. Fig. 4.6 shows the distribution of registration time. We see that an increase in the relative CPU results leads to high registration times.

RabbitMQ broker

The third microservice that we tested is a RabbitMQ messages broker. We used RabbitMQ PerfTest, which is a throughput testing tool that simulates basic workloads and provides the throughput and the time that a message takes to be consumed by a consumer. For the test, we

[4]https://github.com/my5G/my5G-RANTester

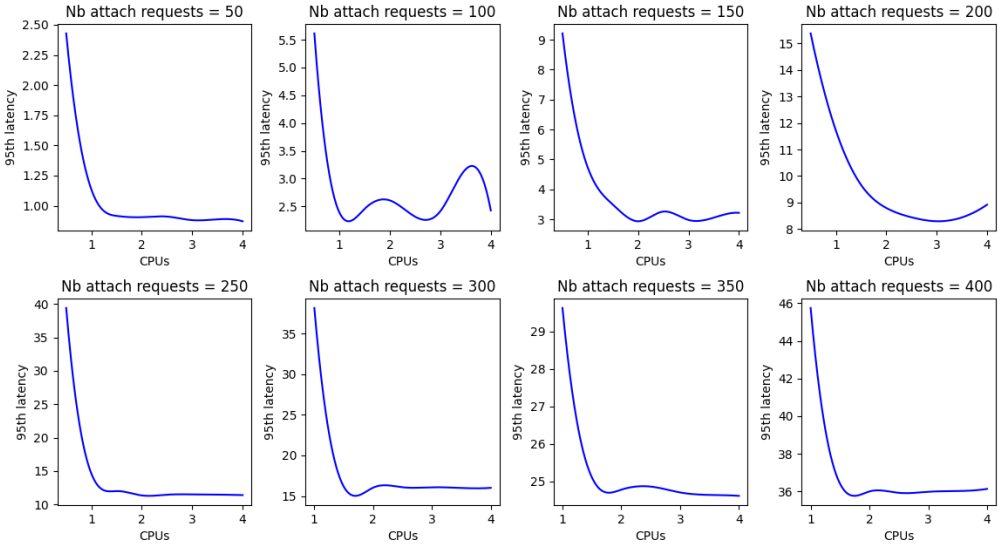


Figure 4.5: AMF's 95% registration time in relation to the allocated CPU

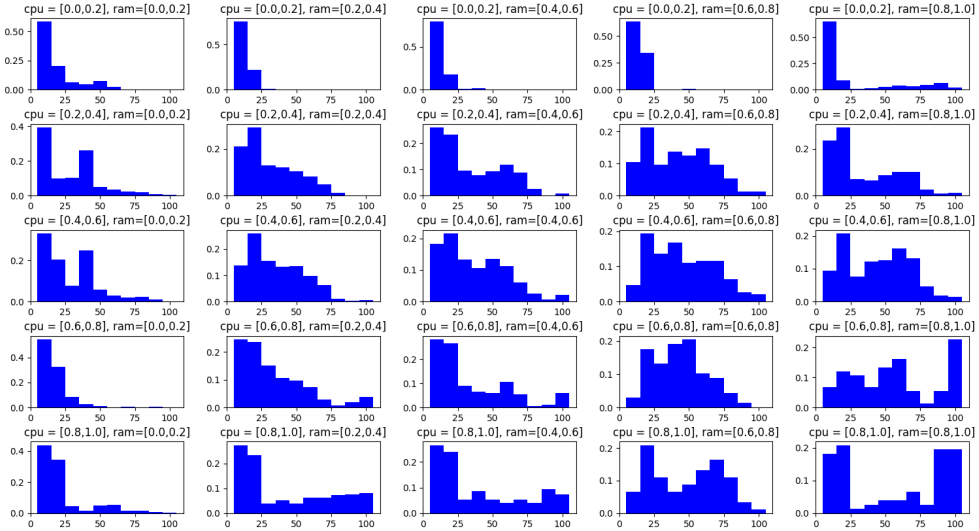


Figure 4.6: 5G AMF's 95% UE registration time statistical distribution

used a number of producers and consumers that ranged from 50 to 500. Each producer sends messages to the broker at a rate of 100 messages per second for a period of time of 90 seconds. Similar to the precedent tests, we vary the CPU configuration of the RabbitMQ pod from 0.5 to 4 CPU, while we vary the memory from 1024 MB to 4096 with increments of 256 MB.
The results are shown in Fig. 4.7.a, we observe that the response time follows the same trend which is that the message's latency gets lower when more CPU is allocated to the broker. Fig. 4.8 shows that the proportion of high latencies is largest when the relative CPU utilization is between 0.8 and 1.
Interestingly, while testing the broker, and as it is a memory intensive application, even if the relative memory does not correlate with the message delay time (Fig. 4.8.b, when the allocated memory was low (around 1024 MB) it resulted in several container restarts due to an OOM signal. Fig. 4.9 shows the distribution of relative memory values for a memory allocation of less than 1536 MB. The values shown are the last collected memory before the failure of the pod, indeed not all failures were caused by OOM exception, but the graph shows that memory usage was approaching the set limit before the occurrence of the failure.

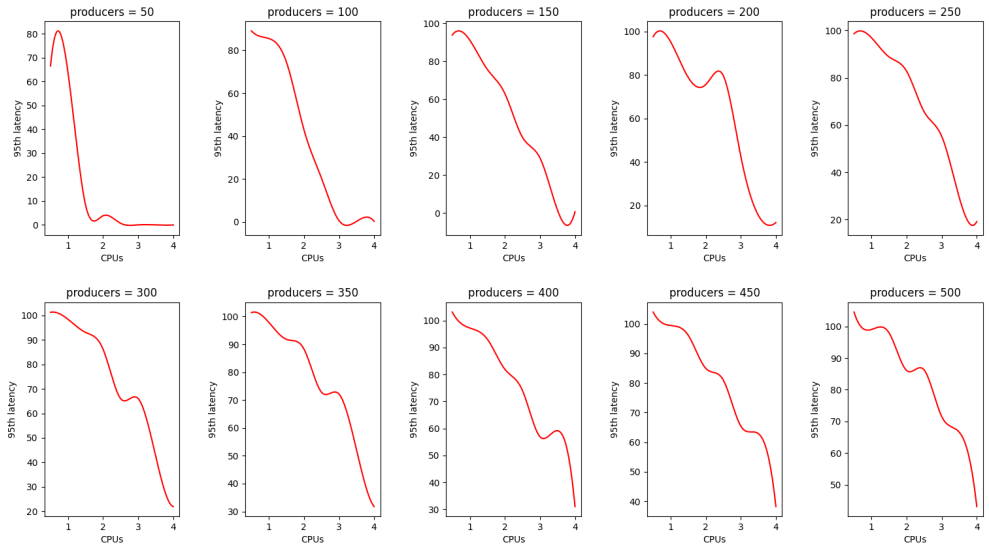


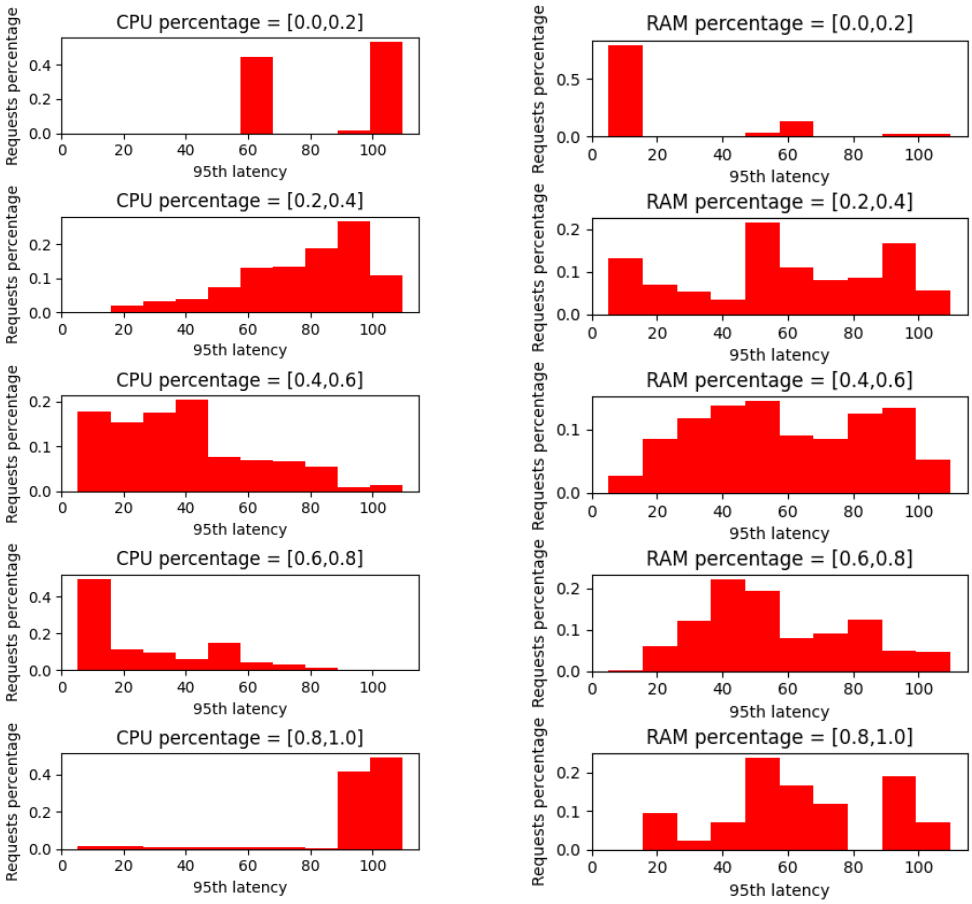
Figure 4.7: RabbitMQ broker's 95% message delay in relation to the allocated CPU

MongoDB database

The last application tested is a MongoDB instance. We execute a batch of requests ranging from 1000 to 10000 with a read write ratio of 90:10. Similar to the previous tests, the CPU allocated to the pod of MongoDB ranges from 0.5 to 4 CPU, and memory allocation from 256 MB to 4096 MB. Fig. 4.10 shows the distribution of operations time in relation to the relative CPU and memory. We can observe the same trend as the previous tests: when the percentage of CPU consumed gets higher, the proportion of high operation times increases.

4.4.3 Discussion

The obtained results obviously show a strong relationship between the allocated resources and the performance of the service. The more resources the application has, the better its perfor-



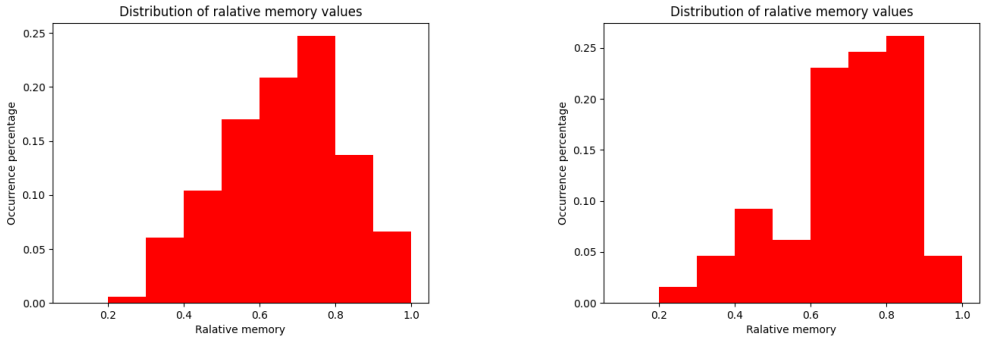
(a) Distribution according to CPU relative value

(b) Distribution according to Memory relative value

Figure 4.8: RabbitMQ broker's 95% message delay statistical distribution

mance, which is valid until it reaches peak performance. The results confirm the assumption of Fig. 4.1. We showed the effect of CPU allocation on the performance, where we clearly observed that for the same load, the response time of the service is lower when more CPU is allocated. It continues to decrease until it reaches an optimum performance, after which additional resources do not make the performance better, which means that a proportion of allocated resources is wasted.

Another important measure is the relative CPU. From the obtained results, we remarked that when an application's CPU usage approaches the limit allocated to the latter, the distribution of response times tends toward higher response time. Consequently, in order to detect performance degradation of applications that do not provide measures about their performances, like the latency of treated requests that is available only at the application level and difficult to derive by monitoring the infrastructure. The relative CPU and relative memory values are good indicators for (1) the application performance, where the relative CPU values indicate the service load and, consequently, the probability of occurrence of high response times. Indeed, the higher the relative



(a) Distribution when allocated Memory ≤ 1536 MB

(b) Distribution when allocated Memory = 1024 MB

Figure 4.9: RabbitMQ last recorder relative memory before pod failure

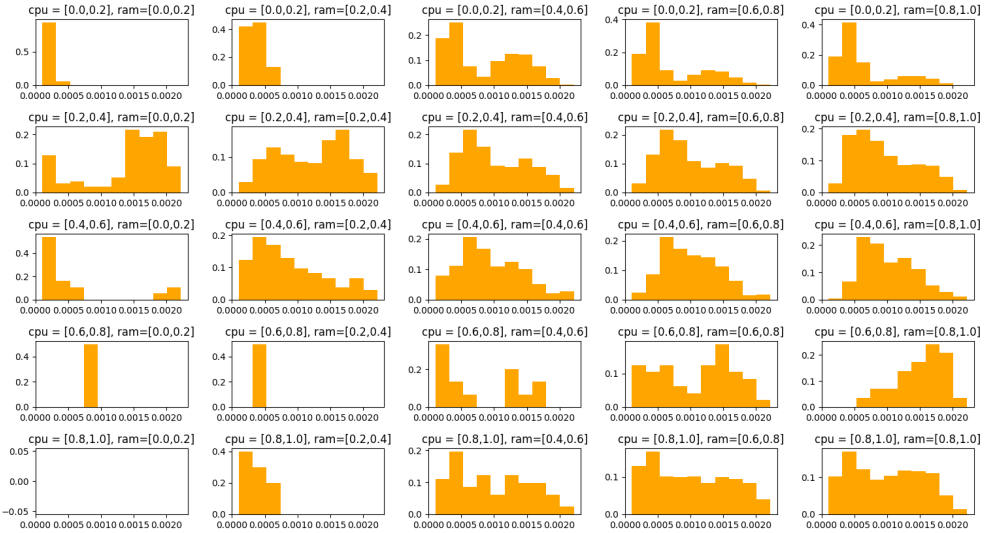


Figure 4.10: Database's 95% operation duration statistical distribution

CPU is, the higher the response time of a server is, whatever the type of service and the number of service requests. This finding is very important to cloud/edge providers in order to detect early misconfiguration of the service in terms of needed resources and anticipate any Service Level Agreement (SLA) issues with the tenant. Second, the relative memory consumption alerts on the occurrence of OOM signals that result in the restart of the container deploying the service, which leads to disturbing the service continuity and hence the service availability.
Finally, using the obtained results, we have implemented monitoring in our edge facility that alerts on the values of the relative CPU and memory values by aiming to keep the relative CPU and memory values less than 0.8, which anticipates any misconfiguration of application resources. The alerts provide feedback about the application's configuration. This process helps validate the vertical resources request and assists him in finding the best configuration for the service according to the relative values of CPU and memory. If the resource usage/limit ratio approaches one, we update the Edge application with more resources to avoid performance

degradation and service disturbance.

4.5 Conclusion

In this Chapter, we presented a study on application performance in a cloud-native and containerized environment. We have run different experiments using representative vertical applications, including a telecommunication network function. All the applications were tested under different resource configurations (CPU and memory) and loads. The obtained results provide useful insights into the behaviour of the workloads and the relation between resource usage and application performance. From those insights, we concluded that relative CPU usage is an important indicator of the relevance of the initial application resources configuration. The higher this value is, the more likely the applications will experience performance deterioration. In comparison, relative memory usage is an important indicator of the risk of occurrence of out of memory errors and, hence, service disruption.
In the next Chapter, we use the different datasets we have constructed using these experiments to build ML and XAI-enabled applications to anticipate and correct resource misconfiguration by autoscaling applications' resources in a fine-granular manner.

Chapter 5

XAI-Enabled Fine Granular Vertical Resources Autoscaler

5.1 Introduction

The cloud-native concept was born with the advent and the success of cloud resource virtualization (computing, storage, and networking) relying on container technology [54]. In a cloud-native architecture, cloud applications and services are no longer deployed as monolithic blocks but rather as loosely coupled microservices. Each microservice is deployed as a container and managed using container orchestration engines and platforms like Kubernetes. In contrast to the traditional monolithic model, running applications as containerized microservices permits more agility and flexibility by facilitating development, upgrades, maintenance, and hence DevOps operations. However, running microservices in a cloud-native architecture opens the door for new challenges when managing both the application life-cycle and cloud resources [55] [56]. One of the critical challenges is scaling the microservices resources under dynamic workload variations and resource demands. Indeed, before deploying users' applications or services, cloud operators identify the number of virtual instances needed during the execution of the workload as well as the required resources for each application instance, such as the type and number of virtual machines or the number of pod replicas and resources needed for each pod in the Kubernetes cluster. It is well-accepted that estimating the needed resources of an application, i.e., the adequate combination of memory, CPU, and the number of concurrent instances, to avoid service degradation is a challenging task. The number of replicas and the needed computing resources to optimally handle a given application may vary over time due, for instance, to the time period or the application's popularity (i.e., an increase in popularity). Accordingly, it is important to design an intelligent and efficient management system that, throughout the life-cycle of a microservice, computes the needed resources of microservices to run optimally and avoid service degradation.

One of the key functions of the cloud-native management system is the scaler algorithm. The latter's role is to compute: (1) the needed application instances or virtual instances, known as "horizontal scaling"; or (2) the amount of computing resources per instance, known as "vertical scaling". Several solutions have been proposed to devise intelligent and autonomous scaler solutions, which largely leverage machine learning (ML) algorithms and, in particular, Reinforcement Learning (RL) [57][58]. These solutions mainly learn microservices' load patterns as well as traffic changes and generate the learning models able to predict each microservice's needs and scale the vertical resources. However, the scaling approach triggered after the ML algorithm prediction consists of simply doubling or multiplying by a factor the dedicated CPU

and memory assigned to the microservice [59]. Scaling the CPU and memory is not optimal, as not all applications are simultaneously sensitive to CPU and memory. Some microservices are sensitive to CPU or memory, while only a few are sensitive to both [60]. Therefore, scaling both computing resources may negatively impact resource usage (waste of resources) and, hence, on other critical metrics, particularly energy consumption.
In this Chapter, we propose a Zero-touch Service Management (ZSM) framework featuring a fine-granular resource scaler algorithm to run microservices in a cloud-native environment optimally. The proposed scaler algorithm relies on ML models to predict the performances of the run microservice. When a service degradation is detected, eXplainable Artificial Intelligence (XAI) algorithms are used to interpret the ML prediction and deduce which features led to that bad performance. More specifically, our framework relies first on an ML algorithm based on eXtreme Gradient Boosting (XGBoost) [61] to predict any violations related to the performance of running applications. Here, we use the application response time metric to characterize the application performance. The trained ML model considers many features related to CPU and memory, namely CPU usage, CPU limit, memory usage, and memory limit. Parallelly, an XAI algorithm is run, namely SHapley Additive exPlanations (SHAP) [16], to deduce the most important features that yield such violation using ML outputs. By knowing the root cause of the performance violation, the autoscaler algorithm scales the CPU, memory, or both. Regarding the scale-down process, we consider a threshold-based approach for CPU and memory, in which a scale-down is possible. But, in order to avoid a ping-pong effect (repetitive scale down and scale up), we also consider a stabilization period after a scale-up where a scale-down process is not allowed. The proposed vertical autoscaling framework can be combined with existing horizontal scaling mechanisms in order to achieve both vertical and horizontal resources autoscaling.

5.2 Related Work

In this section, we briefly overview some of the most significant works on ML-based resource autoscaling in cloud-native systems.
In [62], the authors designed Autopilot as horizontal and vertical autoscaling of resources at Google. It mainly aims to minimize slack, i.e., the difference between capacity and real-time resource usage, while ensuring the stable performance of running tasks. Autopilot leveraged machine learning algorithms on top of historical data related to tasks/jobs executions. In particular, Autopilot relies on two main algorithms. The first one enables an exponentially-smoothed sliding window, while the second one is based on reinforcement learning (RL) to select the suitable sliding window algorithm, which gives better performance for each task/job. Practical results show that Autopilot succeeds in reducing, on one hand, the slack to 23%, against 46% for manually-scaled tasks; on the other hand, the number of tasks impacted by out-of-memory by a factor of 10. The authors in [55] proposed a model-based RL scheme to enable both horizontal and vertical auto-scaling mechanisms of container-based applications. The designed model considers several criteria in its auto-scaling, including application performance, resource cost, and adaptation cost. Furthermore, the proposed scheme is integrated into Docker Swarm to realize an Elastic Docker Swarm (EDS). The obtained results show the efficiency and flexibility of EDS in leveraging RL with respect to other existing elasticity policies. The challenge of ensuring end-to-end Service Level Agreement (SLA) while improving resource allocation to microservices is addressed in [56]. The authors proposed a novel SLA-Aware scheme based on Bayesian Optimization to assign necessary resources to meet applications' performance. This scheme is evaluated on top of a real microservice workload, where the results clearly demonstrate the ability to meet the requirements of each microservice and find Pareto-optimal solutions. In

the same context, a smart autoscaling system for cloud microservices-based applications considering applications' constraints has been introduced in [63]. The proposed autoscaler comprises two main modules: (i) the first one monitors the microservices requirements regarding resources through a generic autoscaling scheme integrated into the Google Kubernetes Engine. This module auto-scales Kubernetes with respect to the running application needs; (ii) based on the resource requirements and applications QoS, the second one leverages RL and deploys a set of agents to learn and determine the autoscaling thresholds concerning resource utilization and the maximum number of pods.
The above works mainly leverage deep learning (DL) algorithms, in particular reinforcement learning, to optimize the horizontal/vertical autoscaling of cloud applications. However, the main drawback of such works is the black-box deployment of DL/RL-based models. Specifically, DL/RL models are becoming more and more complex, i.e., it is hard to understand their inner workings, especially by non-expert users. Moreover, the DL/RL models give predictions/decisions about scaling up/down without any interpretations or explanations on how and why such outputs are made. Hence, the corresponding users (or container orchestration tools) can neither trust and understand DL/RL models' outputs nor optimize their decisions with respect to DL/RL model outputs. To overcome these limits, we leverage the emerging explainable AI paradigm to design a novel vertical autoscaling framework. XAI enables not only interpreting and explaining predictions made by ML models but also helps in making suitable decisions, e.g., scaling up or down CPU, memory, or both, based on the provided explanations. To the best of our knowledge, this is the first work that combines ML and XAI models to design a vertical and explainable autoscaling framework.

5.3 Design and specification of the proposed autoscaler framework

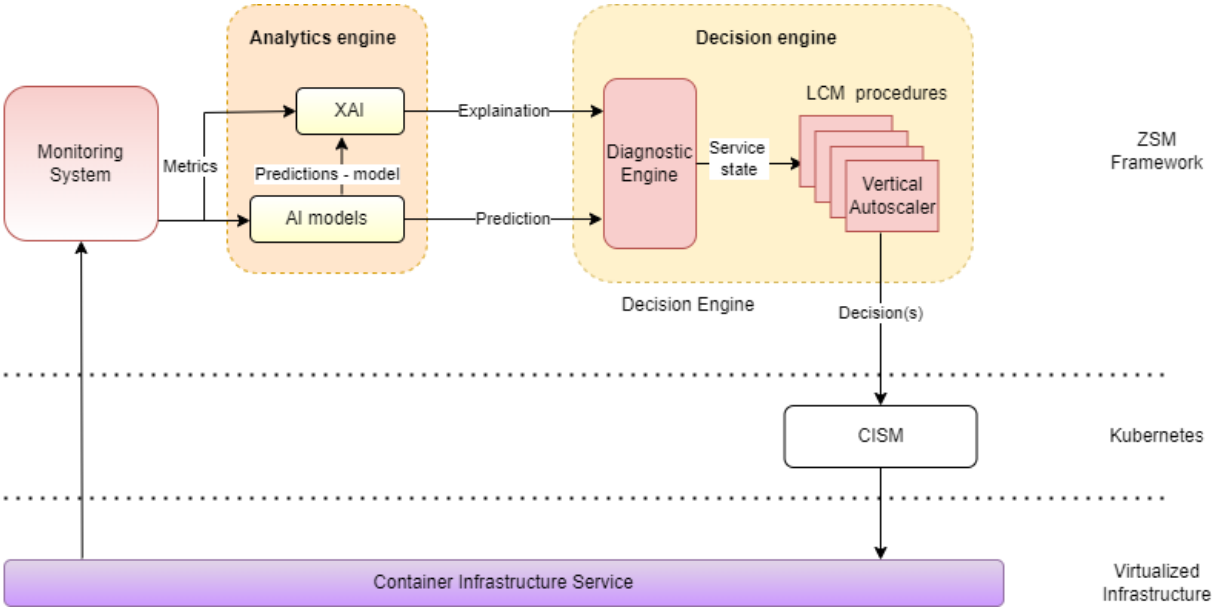


Figure 5.1: Zero touch network framework architecture

5.3.1 The envisioned ZSM architecture

To achieve the concept of a fine-granular vertical auto-scaler in a cloud-native environment, we propose a novel ZSM framework that combines both ML and XAI. The proposed ZSM framework encloses a closed-control loop that monitors, analyses, and derives appropriate life-cycle decisions regarding a microservice, mainly the vertical scaling of computing resources. The component in charge of the vertical scaling (noted vertical scaler) is part of the decision-making function of the closed-control loop. The latter runs autonomously without external intervention and follows the same design of the closed-control loop introduced in [11]; i.e., composed of three key components: (1) Monitoring System (MS) that collects Key Performance Indicator (KPI) regarding the performance of running microservices, such as CPU and memory consumption. (2) Analytical Engine (AE) that uses the collected KPI by MS to analyze the microservice performance behavior and detect Quality of Service (QoS) degradation. To run the analysis, AE may rely on an ML algorithm, which is, in our case, based on XGBoost, to predict the latency performance of a microservice. In contrast to [11], our AE runs, in parallel, the XAI algorithm to interpret the ML output. (3) Decision Engine (DE) runs the life-cycle decision-making process to overcome service degradation. It relies on AE analysis and takes the ML prediction (QoS performance) and its explanation as input. In our solution, DE contains the vertical autoscaler algorithm that scales up resources when a service degradation is detected. The closed-control loop system enables the automatic scaling of vertical microservice resources. Fig. 6.3 illustrates a generic architecture of the proposed ZSM framework. We assume that all microservices run in a cloud-native environment. The figure separates between the closed-control loop components described in the preceding paragraph and the virtualized infrastructure and its manager. The latter is known as the assisted system based on ETSI Experiential Networked Intelligence (ENI) group's notation [64]. According to ETSI cloud-native report [65], the cloud-native equivalent of a hypervisor is Container Infrastructure Service (CIS), which provides all the runtime infrastructural dependencies for one or more container virtualization technologies. In contrast, Container Infrastructure Service Management (CISM) is a cloud-native equivalent of Virtualized Infrastructure Manager (VIM). Technologically speaking, CSIM may correspond to Kubernetes. Regarding the closed-control loop, MS monitors the KPI from CIS regarding the container's resource usage, such as CPU and memory consumption. In our case, we extracted information regarding computing resource consumption (CPU, memory) that AE will use to predict the performance of the microservice at the service level. Here, we are interested in predicting QoS as perceived by the end-users. In the context of a web server, the metric reflecting the QoS can be the response time, i.e., the time a web server takes to answer a client request. Usually, high service time means the server is overloaded and cannot handle the requests in a bounded time, hence degrading the user's quality of experience. AE runs the trained ML model along with the XAI algorithm to predict whether the response time corresponds to service degradation. The XAI module uses both the collected KPI as well as the ML prediction to provide an explanation. Both the explanation and the prediction are transmitted to DE and, more precisely, to the Diagnostic Engine module (Fig. 6.3). The latter uses the output of the AI model responsible for detecting whether the application response time is appropriate or not. It also receives explanations from the XAI module about inference. The explanation gives the contribution of the features to the model output, which means that if the model detects a high response time occurrence (i.e., QoS degradation), the XAI output indicates the contribution of the application's resource features in this result. These characteristics are related to either CPU usage or memory usage. The diagnostic engine then detects the element that caused the high response time. This information is then passed to the vertical autoscaler algorithm, which makes a decision on which resource to scale, hence performing a fine-granular scaling rather

than blindly scaling both CPU and memory. It is worth noting that the autoscaler decision is enforced using the northbound API exposed by CISM that allows updating the resources dedicated to the container running the application by modifying the application's controller object of Kubernetes. Afterward, the Kubernetes controller will rollout a new instance with the new resources definition and delete the old instance.
Regarding the scaling down process, we use a stabilization period after a scale-up in which scaling down will not be performed in order to avoid resource scaling oscillations, i.e., the autoscaler performs one action and, after a short period, performs the opposite action. If no performance drop has been detected during the last few seconds of the stabilization period, a scaling down is possible. To perform this operation, we rely on historical data on resource usage. For memory, if during the last stabilization period, the maximum memory usage was under a chosen percentage, then a scaling down of memory resources is possible. For CPU, if the mean CPU usage during the last stabilization period was less than a certain percentage, then a scale down of CPU resources can be performed. It is worth noting that the scaling down process has no impact on the granularity of the resource allocation. Indeed, when scaling down an application, the resources are released and can be used by another running application instance.

5.3.2 Analytical Engine (AE)

The analytical engine is responsible for analyzing the microservices' performance and detecting QoS degradation. It is composed of two main components: (1) the AI model, which is based on XGBoost, to predict the latency performance of a microservice. (2) The XAI model interprets the output of the AI model using SHAP. Before describing the functioning of these components, we describe the data generation process where we perform a benchmarking of different types of applications.

Data generation

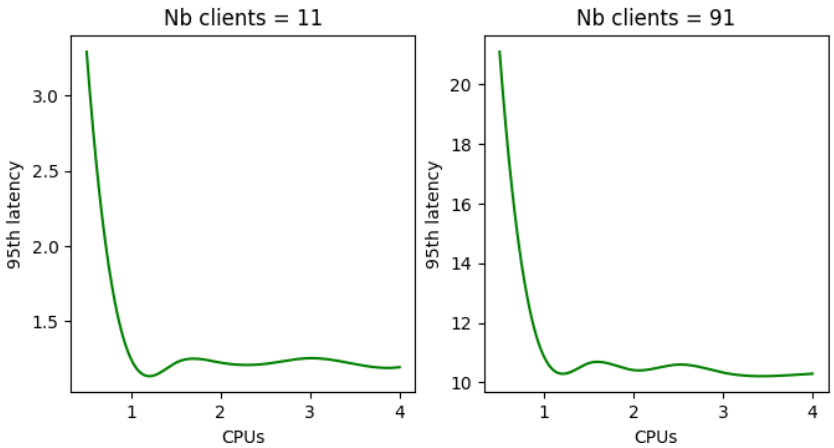


Figure 5.2: Web Server's latency in relation to the allocated CPU

In the work described in the previous Chapter (4), we built a dataset containing information about different applications' resource usage and performance, including web servers, data brokers, and 5G Core Network functions. The performance of the applications is measured using the response time to the client requests. Each tuple of the dataset contains the following information: the memory and CPU allocated to the workload, the memory, and CPU used by the application, the application response time, and the load on the application. The latter is

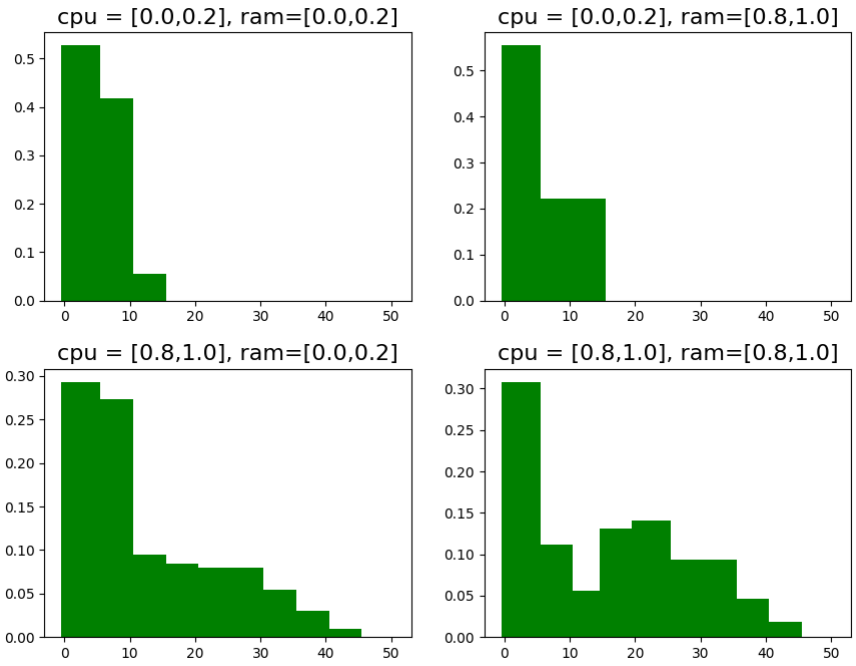


Figure 5.3: Web Server's latency statistical distribution

measured by the number of concurrent requests received by the application during an interval of time. Fig. 5.2 shows the response time of the webserver in relation to the CPU allocated, we can notice that under different loads, represented by the number of concurrent clients sending requests to the server, the more CPU the application has, the lower the response latency is. Moreover, in Fig. 5.3 we show the distribution of the response time of the webserver. We consider the relative CPU and memory, which represent the percentage of resources used from the provided limit. We notice that the higher the relative CPU is, by comparing the distribution while the relative CPU is between 0 and 0.2 and between 0.8 and 1, the greater the percentage of high response times is. In contrast, the memory percentage does not change the distribution of latency values. For more information the complete dataset is available in [66].

ML training

Considering the collected dataset, we can observe that the resources allocated to the application and the relative usage of resources are related to the performance of the application. First, the allocated resources show the limit of performance; the application with fewer available resources will perform worse. Second, the relative resource utilization indicates the possibility of the occurrence of high response times, which means that the degradation of the application performance is more likely to occur when the resource utilization approaches the limit allocated to the application. Therefore, we implement an ML model using the XGBoost classifier to detect performance deteriorations of the application. The model uses resource usage and limits information which can be collected on the running applications via the MS.
XGBoost is a scalable ML system for tree boosting. It implements the gradient-boosted trees algorithm, a supervised learning algorithm that can be used for regression or classification tasks. We train the XGBoost classifier to detect the application's performance drop based on resource usage patterns.
To train the XGBoost classifier on the web server's dataset, we label the dataset's lines as QoS

respected or QoS not respected when the response time is lower or higher than a threshold, respectively. Finally, the model gets the following information as input: memory limit, memory usage, CPU limit, CPU usage, relative CPU, and relative memory. Those metrics can be collected for all the running workloads via MS during runtime. Based on the label and the resource usage, the model classifies the performance of the application, using the resource consumption of the workload, into respecting QoS or not based on the resource usage and limit.
During training, we compared several classification algorithms: K-Nearest Neighbors classifier, Artificial neural network classifier, logistic regression, Random forests, and XGBoost classifier. The XGBoost model was selected based on the classification report by comparing the precision and recall for class 0, which represents the performance degradation of the service. The model's accuracy was 0.95, and the precision and recall for both classes (0 for Qos not respected and 1 for QoS respected) were respectively 0.86, 0.74 for class 0, and 0.97, 0.99 for class 1.

XAI

The second element of the analytical engine is the XAI module, which is responsible for interpreting the output of the AI model. Several XAI techniques exist and can be classified into global or local explanation techniques. Global explanation techniques, such as SHAP, are applied to obtain the general behavior of a model by attempting to explain the whole logic of a model by inspecting its structure. On the other hand, local explanation techniques, such as SHAP and LIME [20], tackle explainability by segmenting the solution space and giving explanations to less complex solution subspaces that are relevant to the whole model. These explanations can be formed through techniques with the differentiating property that only explain part of the whole system's functioning.
The XAI module of the AE relies on the local explanation method based on SHAP to compute the scores of the features contributing to the model's output. The module's output is the contribution score values of the features to the output. Fig. 5.4 represents a visualization of an output of the SHAP method for an ML prediction. The negative values indicate that the feature pushes the model's output towards the output 0. while the positive values signify that the feature pushes the output of the model towards the positive output 1. For this inference, the XAI module reports the following Shapley values or scores of the features, ordered by decreasing contribution to the model output: CPU_percentage -2.56, meaning that the CPU_percentage value pushed the model towards the output 0 (SLA not respected) with a score of 2.56, the second affecting feature is RAM_limit with a score of -1.81, the next contributing feature is RAM_usage with a score of +0.99 meaning that this feature pushed the model towards the output 1 (SLA respected) with a score of 0.99; for the less impacting features, the XAI module indicates CPU_usage +0.71, RAM_percentage -0.38, and CPU_limit -0.13.
Afterward, the selection of the resources to scale up is made at the DE level. This is done by comparing the weighted sum of the contribution of the features related to CPU resources with the weighted sum of the contribution of resources related to memory resources. We can deduce from the previous scores that the combined score of CPU-related features is -1.98 and memory -1.2, meaning that CPU-related features have more influence on the model decision. Therefore, the DE decides that the cause of the performance drop is insufficient CPU allocation. This information allows the vertical autoscaler to decide to allocate more CPU resources to the workload.

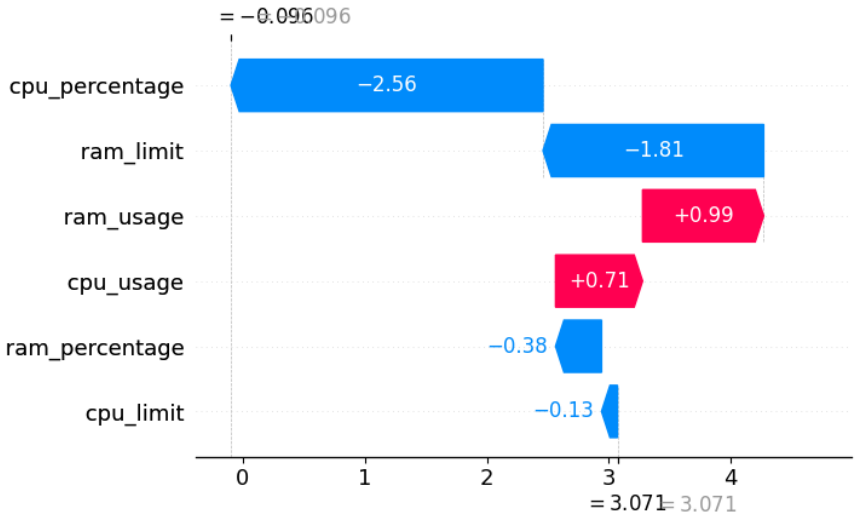


Figure 5.4: Shapley values provided by the SHAP method

5.4 Performance evaluation

In this section, we present the results of validating the ZSM components (i.e., MS, AE, and DE) and testing the XAI-based vertical auto scaling framework. For the sake of comparison, we implemented two versions of the vertical scaler. The first one includes the XAI module, whereas the second implementation does not. Next, we provide metrics on the efficiency of autoscaling, i.e., the amount of resources allocated to the workload regarding the performance achieved by the workload. We demonstrate the benefits of introducing XAI into resource management both at the application level by achieving better performance and at the infrastructure level by reducing resource usage, achieving the goals of both the service owner and the network provider.

5.4.1 Testing Environment

The test facility includes a Kubernetes cluster, which is deployed on top of an Intel server PowerEdge T440 with 128GB of RAM and 64 Core (Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz) with hyper-threading enabled. The cluster was bootstrapped using Kubeadm v1.21.1, and the host operating system is Ubuntu 18.04.5. All the framework components and the tested application run as containers in the cluster. The cluster has a Prometheus deployment for pods and node metrics collection used by MS to collect KPI regarding the infrastructure (i.e., CPU and memory usage of applications).
During the performance evaluation, we run a web server as the target application to be scaled. The application instances are deployed on the Kubernetes cluster as pods with an initial resource configuration of 64MB of memory and 0.25 CPU core.

5.4.2 Performance results

For the tests, we use two versions of the vertical autoscaler, an XAI-based autoscaler to vertically scale the web server instances resources and one without using the XAI output. The running application is exposed to requests load produced by a test component that uses ApacheBench[1]

[1]https://httpd.apache.org/

(a) CPU limit and CPU used by the pod in relation to the round number

(b) Memory limit and memory used by the pod in relation to the round number

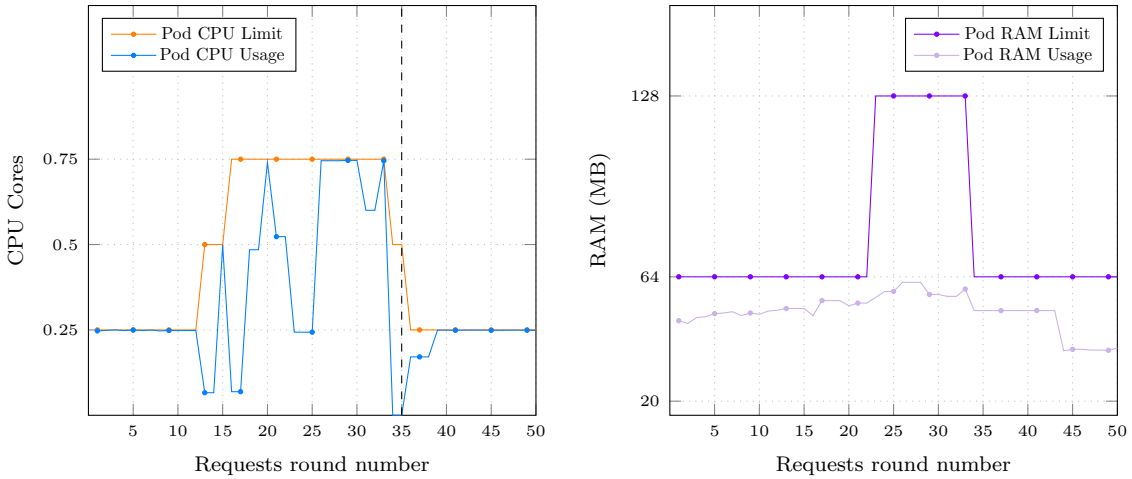


Figure 5.5: Evolution of CPU and allocated RAM using the XAI-assisted autoscaler

to make a number of concurrent HTTP requests. We refer to a test round as a set of N requests made by C concurrent clients.
It is worth recalling that AE runs the ML model to predict QoS degradation. The latter was trained on the dataset related to the performance of web servers under changing configurations. If the model detects degradation, the XAI module is called. The XAI takes as input both the ML output as well as the data set to return the Shapley (or score) values of the features as a numerical score. Then the diagnostic engine of DE compares the weighted sum of the memory-related features scores with the weighted sum of the CPU-related features scores. This output will allow the autoscaler to decide what type of resources need to be scaled.
In case the XAI module is not involved, the autoscaler obtains information about the service's state using only the ML module's output (XGBoost); it has no information about the contribution of the features to the model output. If degradation is detected, both CPU and memory resources are scaled.

Application's performance evolution

In this test, we run a web application with an initial configuration of 0.25 CPU core and 64 MB of memory. We perform 35 rounds of requests to the application with a load that uses a concurrency level of 50 clients, making 250 total requests, followed by 15 rounds of requests with a concurrency level of 10 clients, making 200 requests in total. Parallelly, we note the vertical autoscaling decisions (i.e., scale down or up) and measure the application's response time and resource utilization during each round of requests.
Figs. 5.5 and 5.6 show the results of the two key metrics regarding resource usage, CPU and RAM, obtained when the autoscaler decision is taken with and without the help of XAI, respectively. Both figures represent the evolution of the PoD's CPU_limit, CPU_usage, RAM_limit, and RAM_usage according to the request round number. The vertical dashed line after round 35 indicates a reduced load on the application. Regarding the case of the autoscaler using XAI (Fig. 5.5) and based on the response time, we observe both changes in the resources allocated to the application and their effect on its performance shown in Fig. 5.7. The application initially receives the first twelve rounds of requests, after which, based on the metrics on pod resource

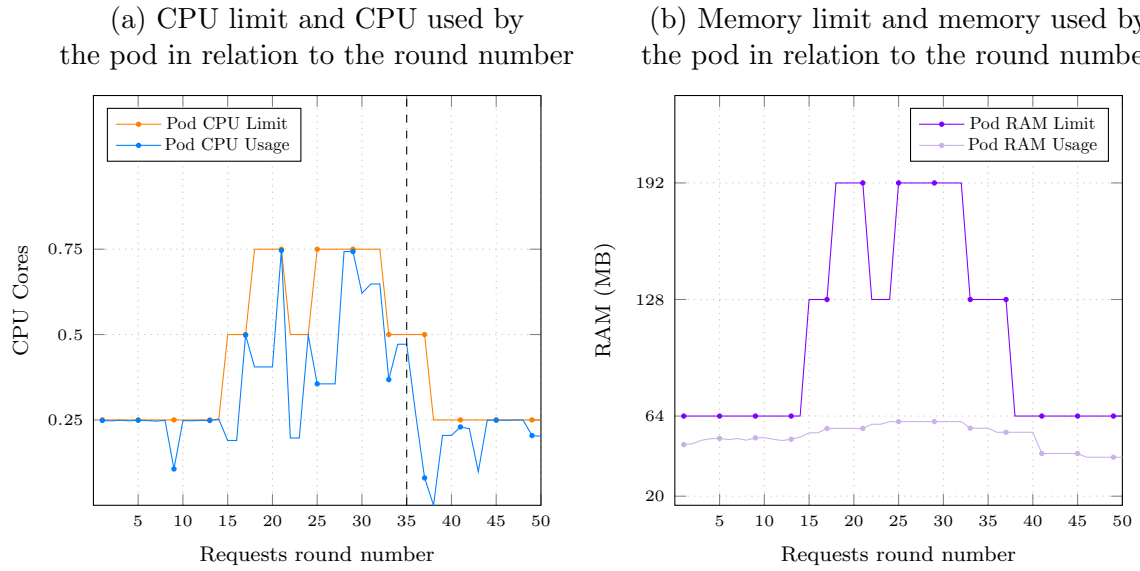


Figure 5.6: Evolution of CPU and allocated RAM using the autoscaler without XAI module's assistance

usage that are collected by the monitoring system, the ML module detects that the application performances degrade (i.e., an increase in the response time). For this ML prediction (i.e., inference), the XAI module reports the Shapley values or scores of the features shown in Fig. 5.4. As described in section 5.3.2, the DE concludes that insufficient CPU allocation is the cause of the performance drop. Consequently, the vertical autoscaler decides to allocate more CPU resources to the workload. This operation increases the CPU limit for the application; it can be observed in the metrics of round 13 as we notice in Fig. 5.5.a that the amount of CPU allocated to the application is 0.5 Core instead of the previous 0.25 Core. Similarly, we notice changes in the resources allocated to the pod at round 15, as the CPU has increased from 0.5 Core to 0.75 Core, and at round 22 from Fig. 5.6.b, the memory allocated to the application has increased from 64MB to 128MB. At these same points, we notice the effect of the autoscaler decisions on the application response time. It drops from 20 seconds to about 4 seconds after round 16.
After round 35, the load on the application is reduced to a concurrency level of 10 clients. We observe that the vertical autoscaler took decisions to scale down both CPU and memory at (1) round 34, the CPU allocated to the container is reduced to 0.5 Core, and the memory is reduced to 64 MB; (2) round 37, the CPU allocated to the container is decreased to 0.25 Core. We can also remark, from Fig. 5.7, that the application's response time remains constant after round 35.
When the diagnostic engine has no access to the XAI module output (i.e., no information about the contribution of the features to the model output) (Fig. 5.6), the cause detection at this level is less granular, which leads the autoscaler's decision to be less specific to the type of resource (CPU or memory); hence both resources are increased.
In this scenario, and by comparing the response time of the application in both cases, shown in Fig. 5.7, we can conclude that the introduction of the XAI module allowed the vertical autoscaler to provide lesser resources to the application (less memory) to achieve the same level of performance (a response time of around 4 seconds).

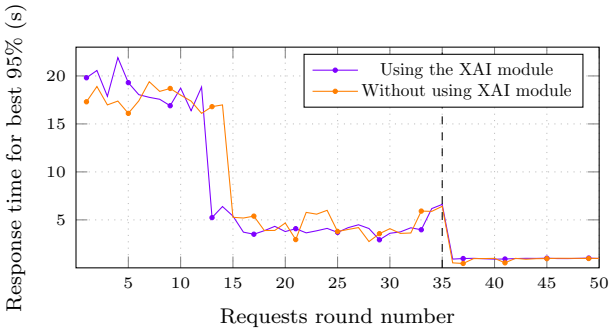


Figure 5.7: Evolution of the application performance (response time) using the autoscaler with and without XAI module's assistance

Testing multiple instances while varying the load on the auto-scaled applications

This time we perform an extensive test regarding the performances of the two vertical autoscalers. Hence, we deploy 30 applications and we vary the load to which each application is exposed. The application is first deployed with an initial resource configuration of 0.5 Core of CPU and 128MB of memory. The number of concurrent clients sending requests to each application varies from 10 to 100, while the number of requests varies from 90 when using 10 concurrent clients to 450 when a concurrency level of 100 is used. Finally, we perform 100 rounds for each concurrency level. For each application, the pod configuration is reinitialized afterward. Resulting in a total of 300 application instances to be scaled by each autoscaler (30 services exposed to a load varying from 10 to 100).

(a) CPU allocated to the workload in relation to the load on the service

(b) Memory allocated to the workload in relation to the load on the service

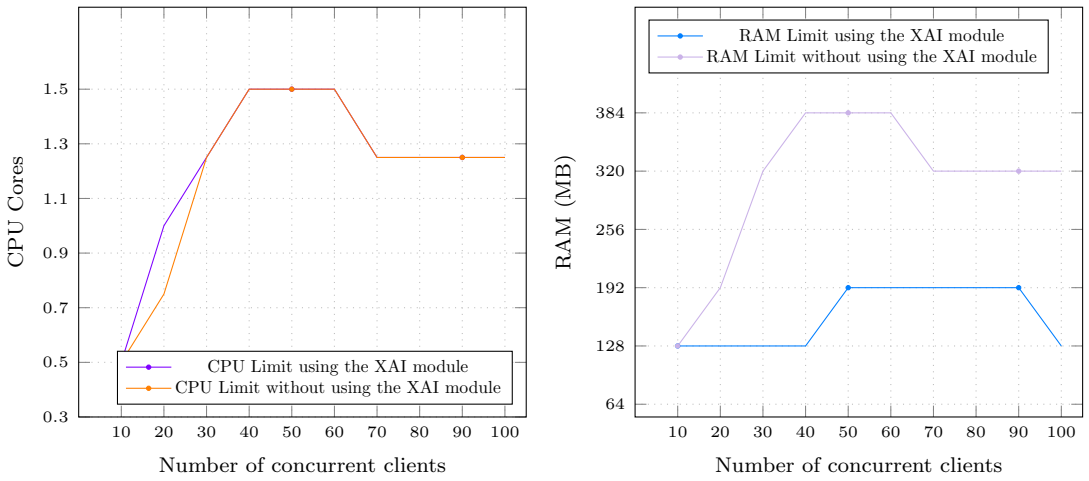


Figure 5.8: Highest values of CPU and RAM allocated to an instance of the applications in relation to the number of concurrent clients

Fig. 5.8 shows the highest amount of CPU allocated to the pod running the application for each load (Fig.5.8.a) and the highest amount of memory allocated to the pod (Fig.5.8.b) for both vertical autoscalers. Whereas, Fig. 5.9 illustrates the mean response time of the 30 applications for the 100 rounds of requests that each scaled application receives.
During the experimentation, the 300 instances that have been scaled using the XAI-based au-

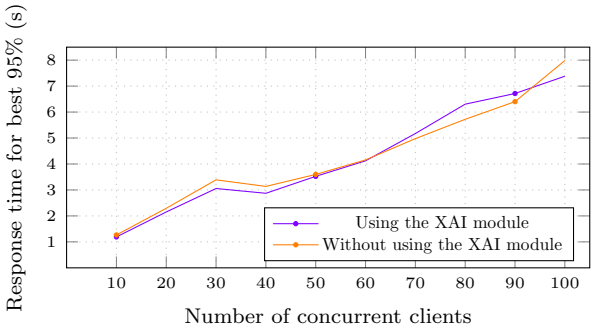


Figure 5.9: Mean response time in relation to the number of concurrent clients

toscaler employed a total of 184.25 cores of CPU and 29.312 GB of memory, while the 300 instances that have been scaled using the non-XAI-based autoscaler used a total of 188.0 core of CPU and 48.128 GB of memory. Thus, the percentage of memory gained is 39%, while the percentage of CPU resources gained is 1%.
By comparing the decisions of the two vertical autoscalers we observe that the XAI-based one allocates less memory to the application for all amounts of load. In contrast, it allocates the same or more CPU than the non-XAI-based autoscaler. These results clearly show the fine granularity of the resource allocation achieved by the XAI-based autoscaler, thanks to the ability of the latter to determine the factors that led to performance degradation. Moreover, from the response time plot, we observe that the mean response time of the applications while being managed by both vertical autoscalers is approximately equal, meaning that the allocation of lower resources by the XAI-based autoscaler did not affect the applications' performances.

5.5 Conclusion

In this Chapter, we introduced a ZSM framework featuring ML and XAI to achieve fine-granular resource management in a cloud-native environment. The proposed framework relies on a novel closed-control loop to ensure vertical scaling of application resources (i.e., computing). Unlike the existing solutions, the proposed closed-control loop combines ML and XAI to detect service degradation and select the appropriate resource to scale up instead of scaling all types of computing resources. All the proposed framework components (including the closed-control loop) have been implemented on top of Kubernetes, where the focus was to evaluate the vertical scaler that relies on XAI using, as an example, a web server. The obtained results showed the benefit of introducing XAI in resource auto-scaling, allowing the latter to achieve fine-granular resource allocation. Indeed, for the same performance (same response time), XAI-based autoscaler allows using lesser computing resources (CPU, memory). Therefore, knowing the root cause of application degradation via XAI enables more efficient use of the resources available in the infrastructure.

Chapter 6

Energy Aware Applications Deployment on Cloud Edge Continuum Infrastructure Using Applications Profiles

6.1 Introduction

In the previous chapters, we presented a monitoring system for applications, performed profiling of a set of applications, and implemented the functionality of fine-granular scaling of applications' resources. Thus allowing the Cloud Edge Computing Continuum Manager (CECCM) to have an overview of the behavior of applications and the evolution of their requirements based on their workloads.
In this chapter, we leverage both application profiles and infrastructure profiles to operate the Life-Cycle Management (LCM) of applications, from location of execution, resource allocation, and migration. The LCM procedures are done while taking into consideration the carbon footprint of the running application. In order to achieve this objective:

1. We propose an architecture of CECC Application Orchestrator. The architecture leverages applications and infrastructure profiling to efficiently manage the CECC applications. This profiling is done using the monitoring information, including energy metrics and carbon intensity of infrastructures.

2. We propose a modeling for applications profiles from the point of view of the CECC Manager, the profile represents the application's current and future compute and network requirements. The profile is constructed based on the historical compute and network resources usage of the application using statistical methods.

3. We profile infrastructure energy consumption and carbon footprint using the energy monitoring system Kepler [4] and the energy sources' carbon intensity.

4. We model the problem of selecting the best locations to deploy or migrate applications to while minimizing the deployment's cost and the carbon footprint. The model decides the current and future placements of each application based on the application profile, the availability constraints of the application, and the available infrastructure resources. Further, we propose a heuristic solution to solve the problem rapidly. We compare the different trade-off between carbon and cost efficiency in different scenarios.

6.2 Related works

Several works tackled the challenge of application placement and migration. First, taking into consideration the energy consumption while deploying microservice in a Kubernetes environment, in [67], the authors present a scheduler for Kubernetes, KEIDS, in edge-cloud environments for IIoT with the view to minimize the carbon footprint rate while enhancing the performance and energy efficiency. KEIDS is a multi-cluster scheduling controller for Kubernetes, which aims to enhance performance and energy efficiency. While operating inside Kubernetes clusters to reduce the energy footprint, KEIDS can be used to schedule pods in CECC infrastructures managed by Kubernetes. Another work, [68], focused on minimizing carbon footprints. It present an approach that exploits the elasticity of batch workloads in the cloud to optimize their carbon emissions. The authors' approach is based on the notion of carbon scaling, similar to cloud autoscaling, where a job dynamically varies its server allocation based on fluctuations in the carbon cost of the grid's energy. However, this method overlooks the downtime associated with job migration. Moreover, in [69], the authors present a microservice placement method employing workload profiling over multiple clusters. The work proposes a framework that uses empirical workload profiling, by resource variation monitoring, to acquire micro services resources consumption while responding to workloads repeatedly. Then, the obtained results are fed to the placement model, which selects the best location to run the microservice. However, the framework does not manage the application's life cycle, meaning that it only takes care of the initial deployment of the application. Thus, it does not consider the possible change of load on the microservices, which can increase the application resource requirements.
Another work, [70], introduces a system that enables services migration triggered by the users mobility by facilitating docker container's migration. Their proposed framework consists of three main components: a single cloud component used to facilitate communications from behind NATed networks, one or more edge computing platforms on which the migration server and the docker containers are run, , and The user's mobile device. The framework migrates microservices by transferring their containers in a halted state to the destination location. However, the work does not take into account the evolution of application requirements as it only considers the users' mobility to trigger migration.
The management of containerized cloud-native applications cannot be achieved without orchestration tools. Over the past decade, Kubernetes[3] has emerged as the de facto cloud operating system, facilitating container orchestration. To address resource constraints in environments with limited resources, lighter versions of Kubernetes have been developed such as MicroK8s[1], K3s [2]. Work proposed in [71] presents a system that tackles the scaling of nodes at the far edge. addressing the challenge of managing all the data plane servers via the few control plane node. The work propose a push systems where the control plane nodes push the configuration to the data plane nodes, to avoid the overwhelming of the control plane by configuration pull requests. In contrast to Kubernetes, which follows a pull model where data planes request information from the API servers available at the control nodes.

6.3 Application profiling

Application profiling has gained interest in recent years. It aims to provide knowledge about the application behavior in different environments. This behavior can be affected by several factors, such as the allocated resources and the load on the application. In our work, we focus

[1] https://microk8s.io/
[2] https://k3s.io/

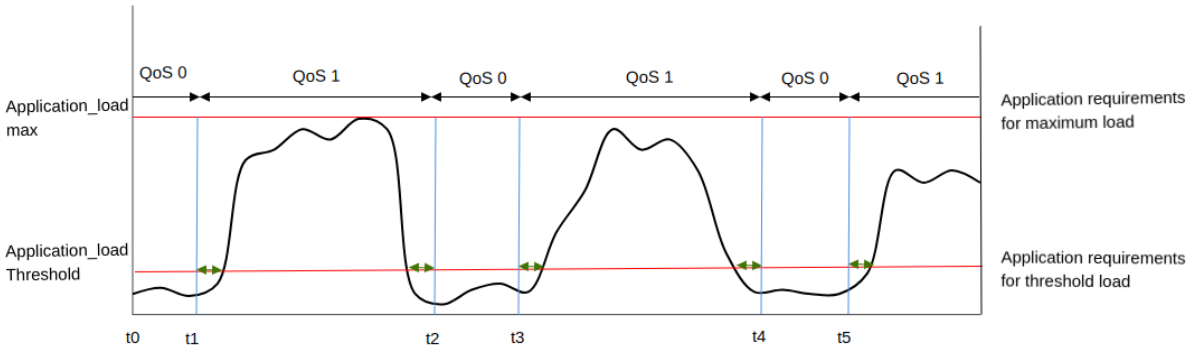


Figure 6.1: Proposed application profile

on profiling the application resources, which means the amount of computing and network resources required by the application now and in the near future. This profiling can be done in several ways, and many works tackled the challenge of predicting the load on the application, the resources needed by the application to handle a certain load using both statistics and AI methods, and benchmarking systems. The main statistical methods include (1) the Autoregressive Integrated Moving Average (ARIMA), which is a combination of the autoregressive model and the moving average model with the goal of predicting future values of a time series; (2) Exponential Smoothing is a time series forecasting method that assigns exponentially decreasing weights over time to past observations; (3) Long short-term memory (LSTM) is a deep neural network that memorizes long-term dependencies of time series data.
We start by describing some of these works, mainly those proposing models to predict application load based on historical trends. Afterwards, we will present the application profile model that we propose and how it can be used to achieve more efficient utilization of CECC resources. A first work published in[72] proposes a hybrid model that combines ARIMA with triple exponential smoothing to accurately predict linear and non-linear relationships in the time series of resource workload of Docker containers. The authors claim that the proposed hybrid model improves prediction accuracy over ARIMA or triple exponential smoothing, which is used alone. The system aims to be used directly by users to improve resource utilization by predicting the resources required by containerized workload and scaling the resources accordingly. On the other hand, and focusing on workload prediction, the authors of [73] propose CloudInsight, an online workload prediction framework that addresses dynamic and highly variable cloud workloads. CloudInsight relies on a number of local predictors based on ARIMA, Linear Support Vector Machine (L-SVM), and linear regression, to name a few, and dynamically determines the weights of each local predictor in order to provide a more accurate prediction. Based on the results, the authors claim that the framework can be used to predict real-world cloud workloads. Finally, on empirical profiling methods, work [69] cited earlier proposes a framework to obtain fine-grained resource requirements depending on workload characteristics. More concretely, the framework proposes a testing deployment, allowing the capture of the behavior of the application in terms of used resources under varying loads. The observed resource usage can be used afterward to profile the application; the profile would have the number of resources needed to handle certain workloads.
Back to our proposed application profile model, illustrated by Fig. 6.1, an application would have a future load; this load can be predicted using the methods described in the existing works that we cited a few of them. The predicted application load could be divided into: 1) load below a threshold and 2) load over the threshold. Then, for each case, we define the application requirements that allow the application to handle the maximum load in that period.

For instance, in the example illustration, between [$t0$ and $t1$] and [$t2$ and $t3$] and [$t4$ and $t5$], the load on the application is predicted to be below the threshold load. During these periods, the application requirements that allow the application to handle the threshold load are sufficient to handle all the coming workloads to the application since the load is always lower than the threshold. Similarly, between [$t1$ and $t2$] and [$t3$ and $t4$] and [$t5$ and t_{end}], the load is higher than the threshold, meaning that the application requirements that allow the application to handle the threshold load is not sufficient to handle the present load. Consequently, the resources allocated to the application should be increased to meet the requirements of handling the maximum load. From here, we define the application profile as the following: during the time between $t0$ and t_{end}, the application will have two sets of requirements that we can call QoS requirements. The first one is the normal QoS of the application, which allows it to handle all the incoming workloads that we denote QoS_{max}. The second QoS in the resources requirements that are sufficient to handle the threshold load, we denote it QoS_{th}. The application profile should also contain the timestamps at which the application requirement changes. For instance, it will have $times = [t0,\ t1,\ t2,\ t3,\ t4,\ t5,\ t_{end}]$ and $req = [QoS_{th},\ QoS_{max}]$. The times can be either minutes or hours, granular based on the use cases. For instance, a week contains 168 hours or 10080 minutes; this granularity will affect the complexity of the scheduling model described in section 6.5.

One of the most important elements of this approach is the choice of the threshold load, based on which we will change the application requirements. For this end, we propose a method based on the compute of the area $\mathcal{S}$ covered by the rectangles shaped by QoS_{th} or QoS_{max}, the timestamps of the change in QoS and the y axis as shown in Fig. 6.2 by the green and orange rectangles. The area covered by those rectangles represents the resource usage over time for each chosen threshold. In Fig. 6.2, the change of the selected threshold load produces a change in the QoS_{th}, so one way to determine if QoS_{th} or QoS'_{th} is more resources efficient is by comparing the areas $\mathcal{S}$ and $\mathcal{S}'$ and chose the threshold producing the minimum area. Finally, given the continuous nature of the load and requested resources, we can choose discrete values of load collected from profiling frameworks like the one described in [69]. As a matter of course, the proposed model can be scaled to have N cascading thresholds with N different application requirements; in our work, we stick with one threshold.

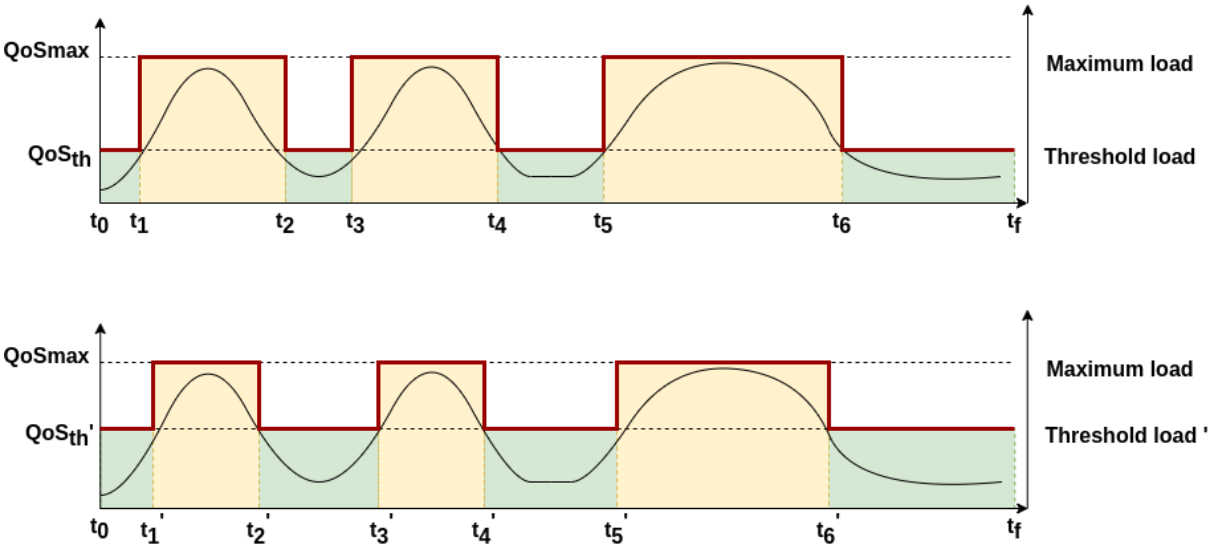


Figure 6.2: choice of the load threshold

6.4 Design and specification of the proposed application scheduling framework

6.4.1 Architecture

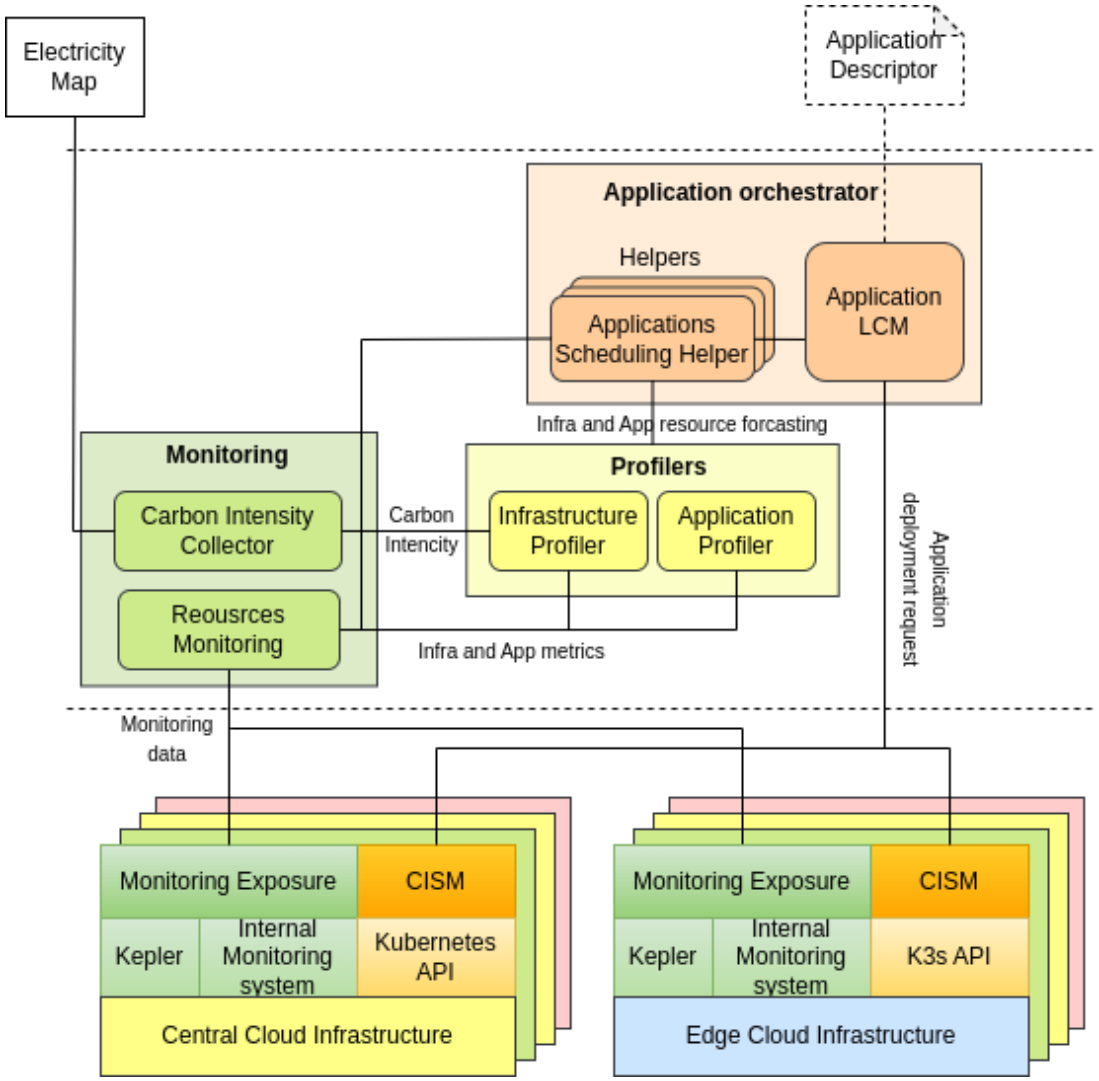


Figure 6.3: The overall architecture of the Cloud Edge Computing Continuum Applications Orchestrator

This work aims to provide a resource-efficient application scheduling on top of the Cloud Edge Computing Continuum (CECC) Infrastructures. We consider that CECC spans multiple regions and infrastructures, ranging from cloud infrastructure, including public and private cloud, to edge cloud infrastructure, including far edge devices. Each of these infrastructures is able to run containerized microservices, meaning that it contains a cluster, mainly a Kubernetes cluster or one of its variants, such as the lightweight Kubernetes K3s. These clusters are managed by a Container Infrastructure Service Manager (CISM); as described in [9], the CISM is the cloud-native equivalent of a Virtualized Infrastructure Manager (VIM), according to ETSI cloud-nativ report [65]; it is responsible for maintaining the containerized workloads. Moreover, each infrastructure exposes monitoring metrics on the infrastructure and the running applications. Those metrics can be provided by internal monitoring systems such as Prometheus [74], which

provide metrics on resource usage, applications, and networks via the multiple metrics exporters that it can collect metrics from. Metrics related to energy used by the running containers and infrastructure metrics can be collected using Kepler (Kubernetes-based Efficient Power Level Exporter) [4]. Kepler Exporter exposes a variety of metrics about the energy consumption of Kubernetes components such as Pods and Nodes. Those metrics includes :

- **kepler container core joules total:** This metric measures the total energy consumption on CPU cores that a certain container has used.
- **kepler container dram joules total:** This metric represents the total energy spent in DRAM memory by a container.
- **kepler node core joules total:** This metric measures the total energy consumption on CPU cores that a certain node has used.
- **kepler node dram joules total:** This metric represents the total energy spent in DRAM memory by a node.
- **kepler node platform joules total:** This metric represents the total energy consumption of the host.

The above measures are used to provide an energy profiling of the infrastructure. For instance, **kepler_container_core_joules_total**, which provide the energy consumed while using CPU resources in Joules, combined with the container's CPU usage metrics, is used to determine the energy cost of CPU resources at a given infrastructure. So, from these metrics, we can estimate the energy that will be used by the required CPU resources of a container over a period of time. Similarly, we can profile the energy cost of memory resources for a given infrastructure.
The monitoring system collects the metrics from each infrastructure's monitoring exposure endpoints and presents them to the Application Orchestrator, the Application Profiler, and the Infrastructure Profiler. The Monitoring system also contains a Carbon Intensity collector, which collects information about the carbon intensity of the electricity used by the different infrastructures.
Carbon intensity, measured in gCO2eq/kWh, provides the equivalent amount of CO2, which would have the same warming effect on the Earth as the combination of emitted gases. Emitting one gCO2eq is the equivalent of emitting one gram of CO2 in terms of the warming effect that is caused. We use this metric to indicate the quality of energy used by the different CECC infrastructures. Hence, it is preferable to use infrastructures that use clean energy to run the most resources-intensive microservices if those nodes support the required application's QoS. Carbon intensity can be provided as metadata by the infrastructure owner when registering the clusters and nodes to be managed by the framework. If it is not provided, it can be collected from available APIs based on the geographical location of the nodes. For instance, in Europe, Electricity Map [75] provides API endpoints to retrieve the last known carbon intensity (in gCO2eq/kWh) of electricity consumed in an area. Also, National Grid ESO's Carbon Intensity API[76] provides an indicative trend of the regional carbon intensity of the electricity system in Great Britain 96+ hours ahead of real-time. It provides programmatic access to both forecast and estimated carbon intensity data. This measure allows the tracking of the carbon intensity of all infrastructures over time. Furthermore, we propose a carbon intensity collector that will collect the carbon intensity of the electricity used by the managed CECC infrastructures based on the location of the nodes. For this reason, we require the location as part of the information about the infrastructures and their nodes. The carbon intensity collector can collect metrics from different sources based on the regions managed by the application management framework.

The applications profiler is the entity responsible for creating the application based on the model described in section 6.3. It used the metrics from the monitoring system in order to analyse the load on the application and its resource utilization. Afterward, the application profiler selects the 2 levels of requirements QoS_{th} and QoS_{max} as well as the timestamps in the future at which the application will require the respective level of QoS. Indeed, for the newly deployed applications, the profiler needs time to construct the profile; during the profiling period, the requirement entered by the application owner via the application descriptor will be used at all times.
In our system, we consider two types of infrastructure: permanent and temporary. Permanent infrastructures are available for the whole duration of the application execution; we can consider public cloud clusters, sets of edge cloud clusters running on top of machines running continuously without battery or mobility limitations. On the other hand, temporary infrastructures are typically clusters deployed on mobile hosts (e.g., drones), hosts that have battery limitations (e.g., mobile devices), and edge AI single-board computers. For each infrastructure, the profiler contains its type, location, the times at which the infrastructure resources are available, and the duration of availability. The profile also includes information about the available resources at each infrastructure as well as the network latency to end devices that may communicate with the deployed applications. The profiler is also responsible for concluding the energy usage rate of the infrastructure resources and, based on the location, the carbon intensity of the infrastructure.
Moving upwards, the Application Orchestrator Manages the life cycle of the applications. It is agnostic to the CISM type and communicates with the latter via the CISM plugin. The Application Orchestrator contains the applications scheduling model, which collects information about the application requirements from the applications profiler, including trends and predicted computing and networking resources needed by the application that needs to be scheduled. Then, it uses information about the infrastructure collected from the infrastructure profiler, including the available computing and network resources, carbon intensity, availability, cost, and energy usage by its different types of resources. Then, the applications scheduling helper, which is part of the application orchestrator, uses this information to provide a plan for the deployment and migration of the application during the deployment period. That is, it will provide the best location to run the application and when the application should moved to other infrastructures using the application requirements, the carbon footprint, as well as the cost of running the overall managed applications.

6.4.2 Network solutions for seamless migration between the cloud edge and far edge

Since the framework manages the deployment of application micro-services that are deployed at the cloud edge, those application can be migrated based on their profiles representing the requirement of an application. For instance, when Far Edge locations are available near an edge location, providing lower latencies for some applications, micro-services of the application can be moved towards these Far Edge locations if low latency is required. While migrating applications, the connectivity between the different micro-services should be continuous, meaning that this migration needs to be seamless. We have investigated techniques to produce seamless migrations between Kubernetes-managed clusters. To ensure this seamless execution of the applications, we should limit the reconfiguration of the microservices after migrating other services that are needed by the application. We could achieve this by providing unchanged network names to application services. This network name is used to reach an application regardless of its location. In the Kubernetes context, this can be achieved using third-party technologies such

as Submariner[3], which allows pods in different Kubernetes clusters to communicate seamlessly using secure tunnels to provide connectivity between clusters. Skupper[4] also solves multi-cluster communication challenges by providing a layer 7 service interconnect through Virtual Application Network (VAN). The main advantage of Skupper is that it allows the usage of the same Kubernetes service name to access the application in different clusters. Finally, vxlan (layer 2) can be used in environments where there is a few number of nodes such as far edge devices with single node clusters.
Since, in our proposed framework, the applications can be moved from one infrastructure to another based on its requirements, we include the downtime incurred by the migration in the decision-making process of the CECC Application Orchestrator. We performed a test on the migration downtime between an edge cloud cluster using Kubernetes and a Far edge cluster using K3s, while the applications are deployed and migrated using the Lightweight edge slice orchestration framework presented in [9]. To produce seamless migrations, we use vxlan tunnels to provide the connectivity between the Edge and Far edge clusters. For this purpose, we reserve a sub-network of IP addresses at the CISM level to provide it to applications that require connection with other applications that can be moved between the Edge and Far Edge. Then, we create a vxlan tunnel for this sub-network between the Raspberry Pi running K3s and the Intel Tower running Kubernetes. In order to assign an IP address from the reserved pool to the applications pods, we use Container Networking Interfaces such as Multus[5] to create network attachments for pods with the selected static IP address, Calico[6] also can be used to provide static IP to pods.
The results of the migration incurred downtimes are shown in Table. 6.1.
Those values are used later in section 6.5 to approximate the downtime caused by the migration of an application and the effect on its availability requirements.

Table 6.1: Measured downtime from the application perspective during migration.

N of apps	**Graceful migration downtime (s)**	**Forced migration downtime (s)**
5	0.15	26.41
10	0.17	40.43
15	3.35	50.0
20	0.23	54.48
25	7.32	66.72
30	148.24	136.26
35	93.47	199.28
40	55.93	137.33

6.5 Application scheduling taking into account the cost and carbon footprint of the deployment

In this section, we formulate the application scheduling into the CECC infrastructure problem via an optimisation problem. The goal of the optimization model is to find the best location

[3]https://submariner.io/
[4]https://skupper.io/
[5]https://github.com/k8snetworkplumbingwg/multus-cni
[6]https://www.tigera.io/project-calico/

Table 6.2: Summary of Notations & Variables.

Notation/ Variable	**Description**
$\mathcal{C}$	Set of CECC infrastructures
$\mathcal{A}$	Set of applications
$\mathcal{T}$	Set of times
Γ	Set of resources
Δ	The total period of the application deployment planning
$\mathcal{W}_a^t$	The set of infrastructures $c \in \mathcal{C}$ that does not satisfy the latency requirements of application $a \in \mathcal{A}$ at time $t \in \mathcal{T}$ meaning $\lambda_a^t < \Lambda_{c,a}^t$
$\mathcal{D}_t$	The duration of the period starting from time $t \in \mathcal{T}$ until time $t+1$
$\mathcal{B}_c^t$	The bandwidth available for the infrastructure $c \in \mathcal{C}$ at time $t \in \mathcal{T}$.
β_a^t	The bandwidth required by application $a \in \mathcal{A}$ at time $t \in \mathcal{T}$.
$\Lambda_{c,a}^t$	The latency from infrastructure $c \in \mathcal{C}$ to the end users of application $a \in \mathcal{A}$ at time $t \in \mathcal{T}$.
λ_a^t	The latency required by application $a \in \mathcal{A}$ at time $t \in \mathcal{T}$.
$\mathcal{R}_{c,r}^t$	The amount of resources of type $r \in \Gamma$ available at CECC infrastructure $c \in \mathcal{C}$ at time $t \in \mathcal{T}$.
$\rho_{a,r}^t$	The amount of resources of type $r \in \Gamma$ required by application $a \in \mathcal{A}$ at time $t \in \mathcal{T}$.
μ_{c_1,c_2}	Migration downtime between two infrastructures $c_1, c_2 \in \mathcal{C}$.
δ_a	The availability required by the application $a \in \mathcal{A}$.
ϵ_a	The maximum allowable duration during which the Quality of Service (QoS) for application $a \in \mathcal{A}$ may be violated.
Ψ_c	The carbon Intensity of infrastructure $c \in \mathcal{C}$.
$\psi_{c,r}$	The energy consumed by a unit of resource of type $r \in \Gamma$.
$\mathcal{M}_c$	The cost of infrastructure $c \in \mathcal{C}$.
$\mathcal{Y}_{a,c}$	A Boolean constant that denotes if application $a \in \mathcal{A}$ was deployed on top of infrastructure $c \in \mathcal{C}$ before the time of placement.
α	A constant that specifies the priority between the cost and carbon footprint.
$\mathcal{X}_{a,c}^t$	A Boolean variable that denotes if the application $a \in \mathcal{A}$ is deployed on top of infrastructure $c \in \mathcal{C}$ at time $t \in \mathcal{T}$.

to execute applications, knowing that the application can be migrated between the different CECC infrastructures. Fig. 6.4 presents a visualization of the application deployment; we can see that in our model, we decide where and when the application will be deployed on top of an infrastructure. In order to do so, we introduce the time dimension. The considered timestamps are discrete values extracted from the applications' profile change and the infrastructure availability times. More concretely, the times represent the union of the timestamps from the applications profiles and the timestamps of availability of infrastructure in the case of far edge infrastructures.

For instance, in the example of Fig. 6.4, the time values are $[0, 3, 6, 11, 12, 14]$, which is the union of the timestamps from the application profile of app1, app2, app3, and app4 $[0, 6, 14] \cup [0, 3, 6, 12, 14] \cup [0, 6, 14] \cup [0, 6, 11, 14]$.

To do so, we formalise the optimization problem that models the characteristics of the system and applications in order to find the optimal placement of the applications. The used notations

are summarized in Table 6.2.
In the system, we consider a set of CECC infrastructures denoted $\mathcal{C}$ over which applications can be deployed. Each infrastructure c has available resources denoted $\mathcal{R}^t_{c,r}$ at a time $t \in \mathcal{T}$ of resources type $r \in \Gamma$, and network bandwidth $\mathcal{B}^t_c$. On the other hand, we have a set of applications $\mathcal{A}$ that needs to be deployed on top of the managed CECC infrastructures. Each application has a requirements in terms of resources $\rho^t_{a,r}$ of type $r \in \Gamma$ at a time $t \in \mathcal{T}$, bandwidth β^t_a at a time $t \in \mathcal{T}$ and latency to end users λ^t_a at a time $t \in \mathcal{T}$. For each application $a \in \mathcal{A}$, each infrastructure $c \in \mathcal{C}$ provides a latency to the end devices connected to the application represented by $\Lambda^t_{c,a}$. Additionally, each application has an availability requirement denoted δ_a. For example, five nines availability means that the application should be up and running for 99.999% of the time, while each migration of an application from infrastructure $c1$ to $c2$ produces a downtime of μ_{c_1,c_2}. Further, a QoS breach quota ϵ_a represents the percentage of acceptable time in which the QoS of the application can be breached. The last 2 elements allow the overbooking of the CECC resources without breaching the application requirements. Finally, infrastructures have a cost $\mathcal{M}_c$, a carbon intensity Ψ_c and energy usage for each type of resources $\psi_{c,r}$, which represents the amount of energy consumed by the application when a specific amount of resources is allocated to it. This metric can be provided by the infrastructure profiler based on the energy metrics collected from Kepler.
The variable of the system is $\mathcal{X}^t_{a,c}$ that represents if the application a is deployed on top of infrastructure c at time t. We also add $\mathcal{Y}_{a,c}$, which specifies if the application a was running on top of infrastructure c before computing the solution of the problem. This variable is used to keep track of the applications that are already running.
Note that the problem definition is intended to be flexible and adapt to the changes on the application profile on runtime, either due to bursts of load on the application that does not follow the application profile or adaptation of the application profile based on new monitoring information. This change accommodation is done using the values of $\mathcal{Y}_{a,c}$, which allows us to solve the problem at a time t^δ using the adapted application requirements by setting its value based on the current infrastructure on top of which the application is running, ϵ_a and δ_a which can be adapted based on the measured downtimes and QoS breach duration that occurred during the planned period.

6.5.1 Objective function

Based on the above discussion, the objective of the problem under consideration is twofold: 1) Minimize the cost of deployment, which is based on the type of infrastructure used. 2) Minimize the Carbon footprint of the deployment on top of the large CECC infrastructure, which is based on the type of energy (electricity source) used by the infrastructure. We define the problem as a multi-objective optimization problem, where the objective function is a function of two objectives. Its mathematical formulation is as follows:

$$\mathcal{F}(\mathcal{X}^t_{a,c}) \ = min \ f(\mathcal{F}_1(\mathcal{X}^t_{a,c}),\ \mathcal{F}_2(\mathcal{X}^t_{a,c})) \tag{6.1}$$

The function, s.t, constraints (6.4 - 6.8) presented bellow, represents the multi-objective optimization problem definition for efficient and carbon aware application scheduling on top of different types of CECC infrastructures. with $\mathcal{X}^t_{a,c}$ being the binary decision variable. The variable's value determines whether the application $a \in \mathcal{A}$ is deployed on top of the infrastructure $c \in \mathcal{C}$ at time $t \in \mathcal{T}$ (value 1) or not (value 0).

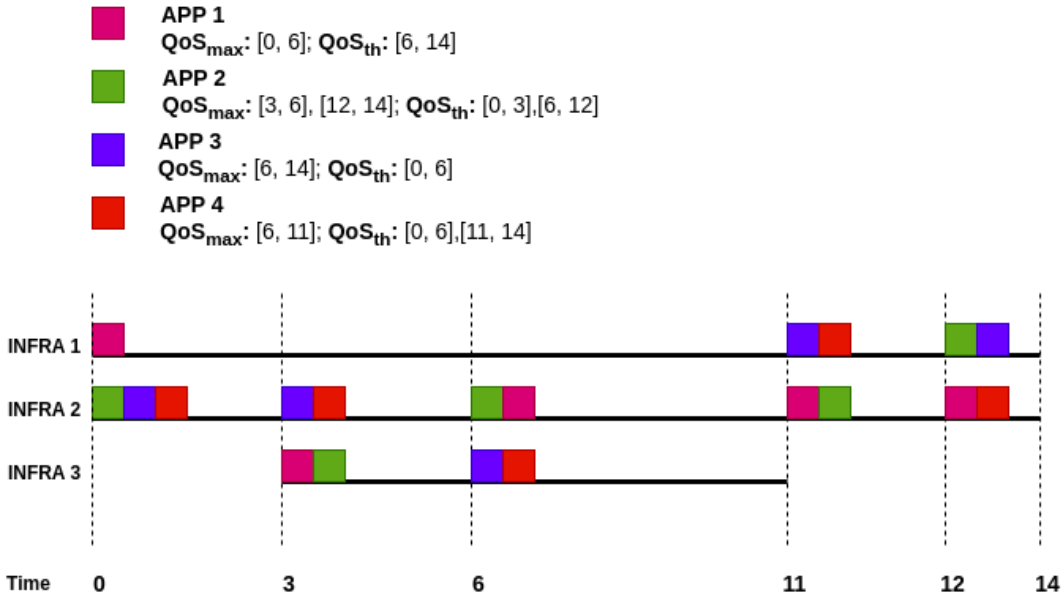


Figure 6.4: Dynamic application deployment on the CECC infrastructure over time

Cost objective function

The first objective is to minimize the cost of the deployment of the applications. Based on the used resources, each application, when deployed on top of an infrastructure, will produce a cost. In function 6.2, we calculate the cost incurred by all the applications in the system. This cost is also affected by the time spent by an application on a specific infrastructure. $\mathcal{Z}_1$ is a normalization constant.

$$\mathcal{F}_1(\mathcal{X}^t_{a,c}) = \frac{1}{\mathcal{Z}_1 \times \Delta} \sum_{a \in \mathcal{A}} \sum_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} \sum_{r \in \Gamma} \mathcal{D}_t \times \mathcal{M}_c \times \rho^t_{a,r} \times \mathcal{X}^t_{a,c} \tag{6.2}$$

Carbon footprint objective function

The second objective is to minimize the system's carbon footprint. To do so, we use the carbon intensity of the infrastructure multiplied by the resources and energy usage for the same infrastructure. Using the application's required resources, we can estimate the carbon footprint of the application deployment. Function 6.3 calculates the carbon footprint of the system based on the time spent by the applications at each infrastructure. $\mathcal{Z}_2$ is a normalization constant.

$$\mathcal{F}_2(\mathcal{X}^t_{a,c}) = \frac{1}{\mathcal{Z}_2 \times \Delta} \sum_{a \in \mathcal{A}} \sum_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} \sum_{r \in \Gamma} \mathcal{D}_t \times \Psi_c \times \psi_{c,r} \times \rho^t_{a,r} \times \mathcal{X}^t_{a,c} \tag{6.3}$$

6.5.2 Constraints

The constraints of the optimization problem are the following:

Instances constraints

Constraint 6.4 is used to make sure that at each time t, the application is running on an infrastructure.

$$\forall a \in \mathcal{A}, \forall t \in \mathcal{T} : \sum_{c \in \mathcal{C}} \mathcal{X}^t_{a,c} = 1 \tag{6.4}$$

Availability constraints

Constraint 6.5 aims to make sure that the application is available δ_a of the time. In our system, we consider only the downtime caused by the migration of the application from one infrastructure to another.

$$\begin{aligned} \forall a \in \mathcal{A} : \sum_{c_1 \in \mathcal{C}} \sum_{c_2 \in \mathcal{C}} \sum_{t>0 \in \mathcal{T}} \frac{\mu_{c_1,c_2}}{\Delta} \times \mathcal{X}^t_{a,c_1} \times \mathcal{X}^{t-1}_{a,c_2} + \\ \sum_{c1 \in \mathcal{C}} \sum_{c1 \in \mathcal{C}} \frac{\mu_{c_1,c_2}}{\Delta} \times \mathcal{Y}_{a,c_1} \times \mathcal{X}^0_{a,c_2} \leq 1 - \delta_a \end{aligned} \tag{6.5}$$

Latency constraints

Constraint 6.6 aims to assure that the application's latency requirement is respected for at least $1 - \epsilon_a$ of the time. To do so, we check that the time spent by the application running on infrastructures that do not respect its latency requirements (represented by $\mathcal{W}^t_a$) is less than ϵ_a.

$$\forall a \in \mathcal{A} : \sum_{t \in \mathcal{T}} \sum_{c \in \mathcal{W}^t_a} \frac{\mathcal{D}_t}{\Delta} \times \mathcal{X}^t_{a,c} \leq \epsilon_a \tag{6.6}$$

bandwidth constraints

We introduce constraint 6.7 in order to guarantee that each infrastructure can satisfy the bandwidth requirements of the applications running on top of it at every time t.

$$\forall c \in \mathcal{C}, \forall t \in \mathcal{T} : \sum_{a \in \mathcal{A}} (1 - \epsilon_a) \times \beta^t_a \times \mathcal{X}^t_{a,c} \leq \mathcal{B}^t_c \tag{6.7}$$

resources constraints

Lastly, constraint 6.8 is introduced to assure that each infrastructure can satisfy the resource requirements of the applications assigned to it at every time t.

$$\forall c \in \mathcal{C}, \forall r \in \Gamma, \forall t \in \mathcal{T} : \sum_{a \in \mathcal{A}} \rho^t_{a,r} \times \mathcal{X}^t_{a,c} \leq \mathcal{R}^t_{c,r} \tag{6.8}$$

6.5.3 Application scheduling algorithm

Obtaining the optimal solution to the proposed problem can be costly in terms of used resources and execution time as shown in section 6.6. Especially if the number of applications and infrastructure increases, making it unusable in real scenarios, in which the system cannot afford to wait hours and use high number of CPU cores and memory resources to schedule application. So, in order to solve the problem, we propose algorithm 1; the aim of the algorithm is to provide a heuristic solution to the MOOP 6.1 in a short time while consuming a limited amount of resources.

Algorithm 1 Application deployment algorithm

1: $\mathcal{SC} \leftarrow$ Sort$(\mathcal{C}, asc)$ {Sort infrastructure according to cost and carbon intensity using $\frac{\alpha \times \Psi_c}{MaxIntensity} + \frac{(1-\alpha)\times \mathcal{M}_c}{Maxcost}$ value}
2: **for** a **in** $\mathcal{A}$ **do**
3: $\mathcal{P}$.insert$(a.QoS_{th})$
4: $\mathcal{P}$.insert$(a.QoS_{max})$
5: **end for**
6: $\mathcal{SP} \leftarrow$ Sort$(\mathcal{P}, asc)$ {Sort application profiles based on the required latency}
7: **for** t **in** $\mathcal{T}$ **do**
8: Initialize infrastructure allocatable resources and bandwidth $\mathcal{AL}_c$
9: **for** p **in** $\mathcal{P}$ **do**
10: $a \leftarrow p.app$
11: **if** ProfileActive$(p,\ t)$ **then**
12: $allocated \leftarrow$ **false**
13: $\gamma \leftarrow Random(0, 1)$
14: **if** $\gamma \geq \frac{AllowedDowntime_a \times Random(0,1)}{\delta_a \times \Delta}$ **then**
15: **if** KeepAppOnInfra(p) **then**
16: Keep the application running on the same infrastructure it is running on
17: $allocated \leftarrow$ **true**
18: Update Infra Allocatable Resources $\mathcal{AL}_c$
19: **continue** to next application
20: **end if**
21: **end if**
22: Application can be migrated
23: **for** c **in** $\mathcal{SC}$ **do**
24: **if** DeployAppOnInfra$(p,\ c,\ \epsilon_a,\ t)$ **then**
25: The application will deployed on infrastructure c at time t
26: $allocated \leftarrow$ **true**
27: Update Infra Allocatable Resources $\mathcal{AL}_c$
28: Update $AllowedDowntime_a$
29: **continue** to next application
30: **end if**
31: **end for**
32: **end if**
33: **end for**
34: **end for**

First, in the algorithm, we sort the available infrastructures based on their carbon intensity and cost, with the variable α representing the importance of each aspect (line 1). Then as described above, the application profile provides two levels of QoS: QoS_{th} and QoS_{max}. The variable $\mathcal{SP}$ contains all the QoS levels for all applications sorted based on latency requirement (line 6). For each timestamp from the set of times, the algorithm maps each application to an infrastructure. To do so, we define a random variable γ that helps decide if the application will keep running on the current infrastructure, on top of which it was running for $t-1$ (if the infrastructure has enough resources to accommodate the application requirements). Otherwise, the ordered list of infrastructures will be explored to find the first fitting infrastructure that satisfies the application requirements at time t. After each assignment of an application to an infrastructure,

the available infrastructure resources as well as the application $AllowedDowntime_a$ are updated (lines 18, 27, 28).
The function KeepAppOnInfra(p) checks if the application a with QoS p can be kept running on top of the infrastructure over which it was running at time $t-1$. While DeployAppOnInfra(p, c, ϵ_a, t) checks if the application a can with QoS p can be deployed on top of infrastructure c at time t, if so it updates the application's ϵ_a value.

6.6 Results

To solve the optimization problem, we implemented a simulator for applications, applications' profiles and infrastructures. The simulator was used to create scenarios with a varying number of applications, infrastructure, and duration. Based on the desired configuration, which includes the number of applications, the required application availability, the number of infrastructures (including far edge infrastructures), and the deployment duration, the simulator generates several outputs. These include the application requirements for each time period, the latency between the infrastructure and the application's users, the available resources for each infrastructure, the infrastructure availability time, and the carbon intensity. Additionally, it generates the migration time between different infrastructures.
We implemented our simulation environment using Python and used Gurobi version 11.0.0 to get the optimal solutions for the test use cases. The tests are done on top of an Intel server PowerEdge T440 with 128GB of RAM and 64 Core (Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz) with hyperthreading enabled. Using Ubuntu 20.04.6 LTS an operating system.
We solve the model using α scalarization (equation 6.9)

$$min\ \alpha \times \mathcal{F}_1(\mathcal{X}^t_{a,c}) + (1-\alpha) \times \mathcal{F}_2(\mathcal{X}^t_{a,c}) \tag{6.9}$$

with:

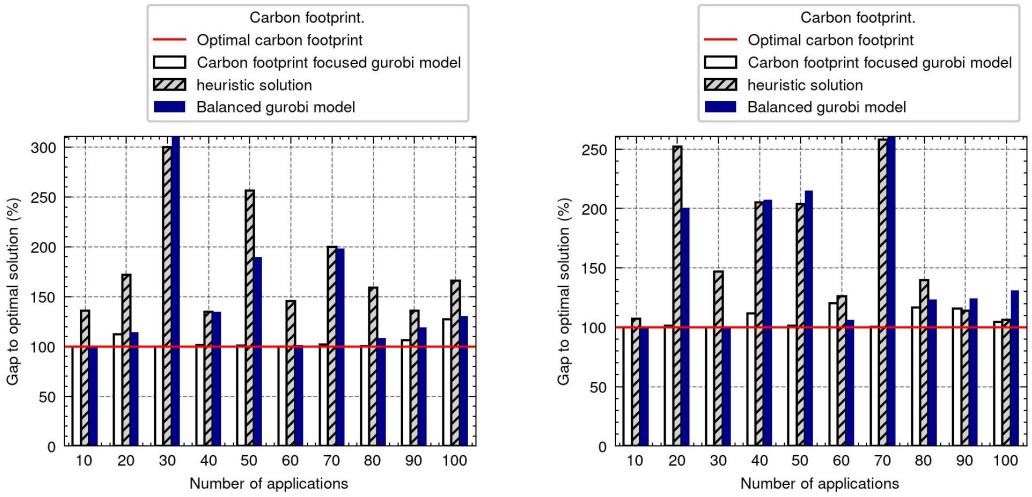
- $\alpha = 0.5$ to get the optimal solution with a trade-off between minimizing cost and carbon footprint,
- $\alpha = 0$ to get the optimal solution for carbon footprint minimization
- And $\alpha = 1$ to get the optimal solution for cost minimization.
- We also solve the problem using $\alpha = 0.1$ representing the carbon footprint-focused model.
- Wile using $\alpha = 0.9$ we solve the cost focused model.

Similarly, we test the heuristic solution using different trade-offs. The trade-offs can be configured using the values of $\frac{\alpha \times \Psi_c}{MaxIntensity} + \frac{(1-\alpha) \times \mathcal{M}_c}{Maxcost}$ where:

- we use $\alpha = 0.5$ while ordering infrastructures to get the balanced heuristic solution.
- $\alpha = 0.1$ to get the cost-focused heuristic solution.
- And $\alpha = 0.9$ to get the carbon footprint-focused heuristic solution.

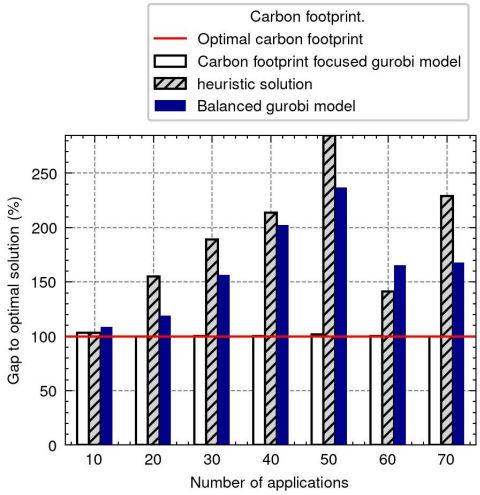
6.6.1 Test results comparison

In order to test the model's performance, we design several scenarios of deployments and infrastructure states. In each scenario, we specify the number of applications to be deployed, the number of infrastructures available, and the duration of the period over which the model plans the application LCM procedures. The scenarios range from deploying 10 applications on top of 4 infrastructures to deploying 100 applications on top of 40 infrastructures. For each scenario, we consider that 10% of infrastructure is composed of mobile far edge infrastructures; meaning that they are available only for a period of the whole duration of deployment planning. We also vary the test duration, representing the period for which we plan the deployment of the application. We start from 10 to 100 and finally, a duration of 1000. The duration represents the period over which we plan to run the application. Note that the period can be periodic, meaning that at the end of the duration, we plan the deployment of the application over the next duration again while taking into account the placement of the applications at the end of the last duration, which is essential in order to satisfy the availability constraints of the applications. In each scenario, the Infrastructure profile is represented by: the availability of the infrastructure (permanent or temporary and the time of availability), the available compute resources, the available network bandwidth, the latency from the infrastructure to the target user of each application, the carbon intensity and the energy consumption of its resources. Moreover, each application has a profile that follows the same model presented in Section 6.3 containing the required compute and network resources for each period of the duration of the deployment.
In the first results (Figs. 6.5, 6.6), We evaluate the performance gap among various models: The Carbon Footprint Gurobi Optimization Model demonstrates the optimal carbon footprint achievable in each scenario. The Cost Gurobi Optimization Model shows the optimal cost attainable in each scenario. Additionally, the Trade-off Gurobi Optimization Models (comprising balanced configuration, cost-focused, and carbon footprint-focused configurations) are analyzed. Finally, the proposed heuristic, which uses the balanced configuration ($\alpha = 0.5$). We also compare the execution time of the different models, as shown in Fig. 6.9.
Fig. 6.5 illustrates the gap between the optimal carbon footprint that can be achieved in each scenario and the carbon footprint of the carbon footprint-focused Gurobi model, the heuristic carbon footprint, and the balanced Gurobi model. We notice from the obtained results that the carbon footprint-focused Gurobi model is consistently the closest in terms of carbon efficiency to the optimal solution with a gap of less than 25%. Meanwhile, the heuristic carbon efficiency is similar to the balanced Gurobi model, as we can notice the heuristic performed better in the scenarios: (apps:30, duration: 10), (apps:40, duration: 100), (apps:50, duration: 100), (apps:70, duration: 100), (apps:10, duration: 1000) and (apps:60, duration: 1000).
Similarly, Fig. 6.6 shows the gap between the optimal cost that can be achieved in each scenario and the cost of the cost-focused Gurobi model, the heuristic cost, and the balanced Gurobi model. Again, we notice from the obtained results that the cost-focused Gurobi model is consistently the closest in terms of cost efficiency to the optimal solution with a gap of less than 50%. While the heuristic cost efficiency is similar to the balanced Gurobi model in some scenarios, it fell short in other scenarios with a gap between the 2 models being higher than 90%. However, we can notice that the heuristic performed better in the scenarios: (apps:50, duration: 10), (apps:60, duration: 10), (apps:10, duration: 100), (apps:90, duration: 100), (apps:10, duration: 1000) and (apps:70, duration: 1000).
We also examine how altering the heuristic configuration impacts the results. In Figs. 6.7, 6.8 we measure the cost and carbon efficiency of the Carbon footprint-focused heuristic solution, cost-focused heuristic solution, and the balanced heuristic solution.
From Fig. 6.7, which depicts the gap in terms of carbon efficiency between the models, we notice



(a) Duration : 10

(b) Duration : 100



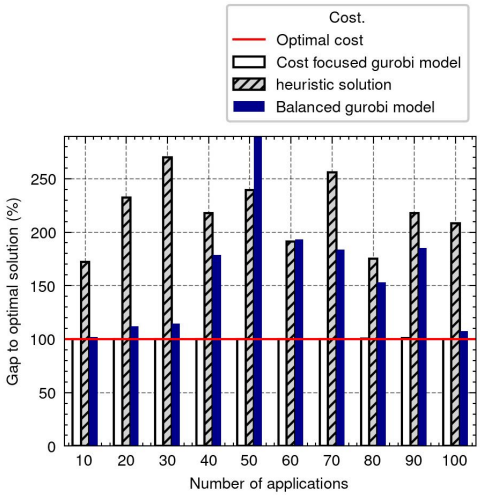
(c) Duration : 1000

Figure 6.5: The gap in Carbon Footprint (%) produced by the models for a duration of 10, 100, and 1000

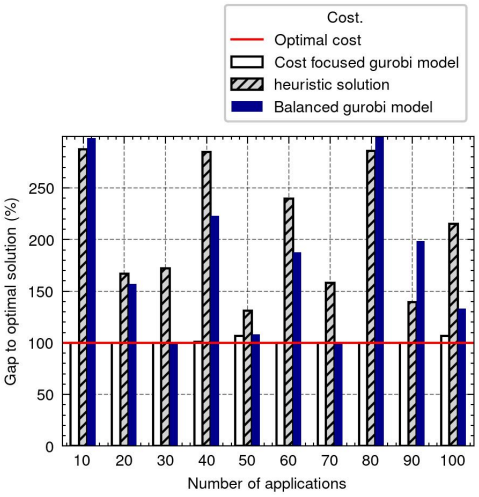
that changing the configuration of the heuristic by giving higher weight to the carbon footprint results in better carbon efficiency as we notice that the gap between carbon footprint-focused heuristic and the optimal solution is less than 40% except in 4 scenarios: (apps:20, duration: 10), (apps:20, duration: 100), (apps:40, duration: 100) and (apps:50, duration: 1000).
In Fig. 6.8, we can see that the gap in terms of cost efficiency between the cost-focused heuristic and the optimal cost solution is consistently less than 50% except in 7 scenarios: (apps:30, duration: 10), (apps:20, duration: 100), (apps:40, duration: 100), (apps:20, duration: 1000), (apps:50, duration: 1000), (apps:60, duration: 1000).

6.6.2 Discussion

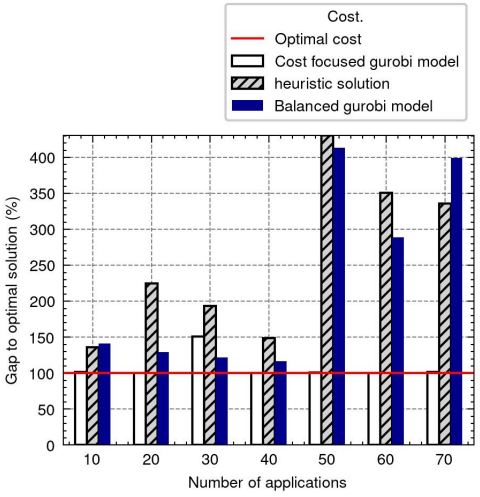
From the results obtained, we can conclude that using the heuristic method reduces the time and resources needed to plan the placement and potential migration of the deployed applications. Furthermore, it provides results that are comparable to the optimal cost or carbon efficiency that



(a) Duration : 10



(b) Duration : 100



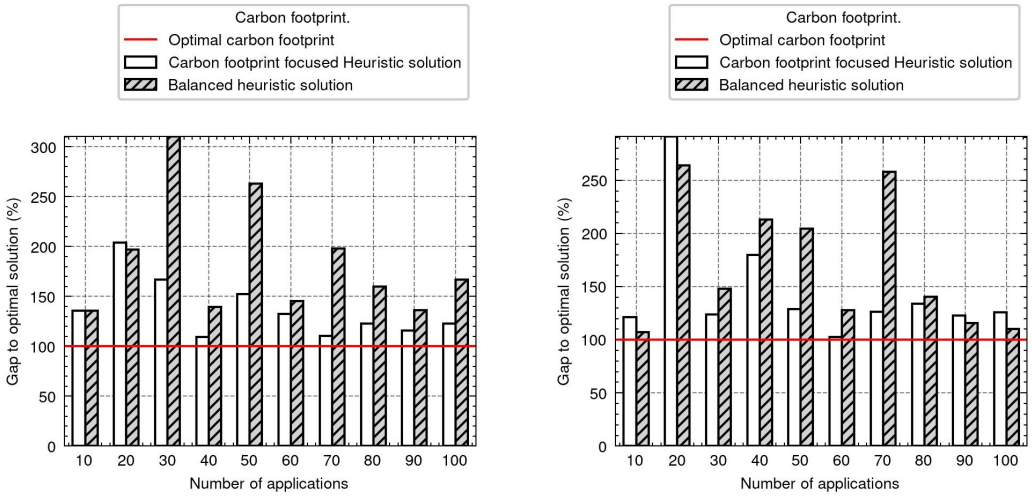
(c) Duration : 1000

Figure 6.6: The gap in the Cost (%) produced by the models for a duration of 10, 100 and 1000

can be achieved. While building the test scenarios, we did not establish a relationship between the cost of an infrastructure and its carbon intensity. This can be seen in the results, where there is a dispersion between the performance of the different models. From the obtained results, we believe that the granularity of the decision model should be configurable depending on the needs of each deployment. Indeed, for different types of applications, the deployment priority can change. From a CECCM operator point of view, the cost minimization of the workloads can have higher priority than minimizing the carbon footprint of the overall CECC deployments. The decision can also be made by the application owner, who can require that his application's carbon footprint be minimized.

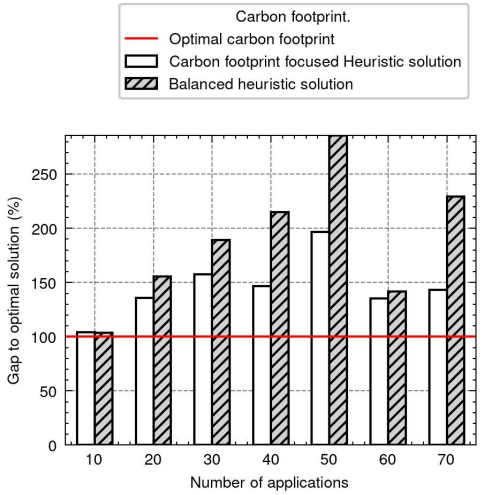
6.7 Conclusion

In this Chapter, we introduced a CECC management framework. We provided a method for defining an application profile in a way that can be used for application deployment and migra-



(a) Duration : 10

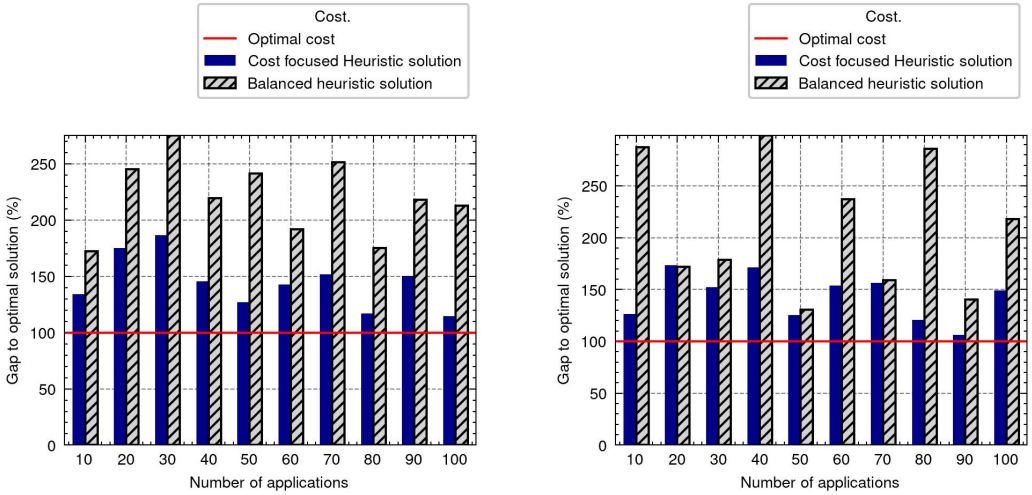
(b) Duration : 100



(c) Duration : 1000

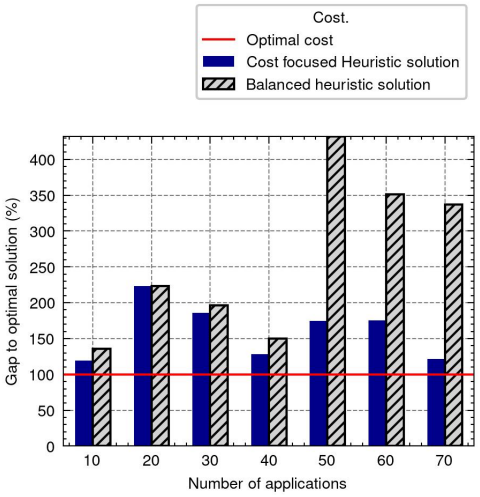
Figure 6.7: The gap in Carbon Footprint (%) produced by the different heuristic configurations for a duration of 10, 100 and 1000

tion. The CECC application orchestrator also uses the infrastructure profile, which includes the carbon intensity, energy usage of its resources, and the availability of the infrastructure. We proposed a mathematical problem definition for planning the deployment/migration of applications while satisfying their availability, compute resources and network requirements. Afterwards, we solved the problem for different deployment scenarios.



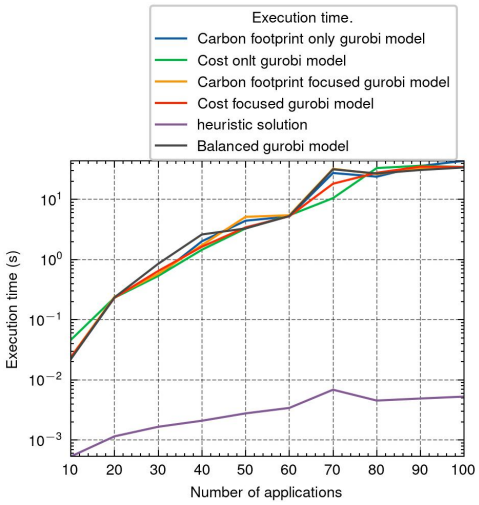
(a) Duration : 10

(b) Duration : 100

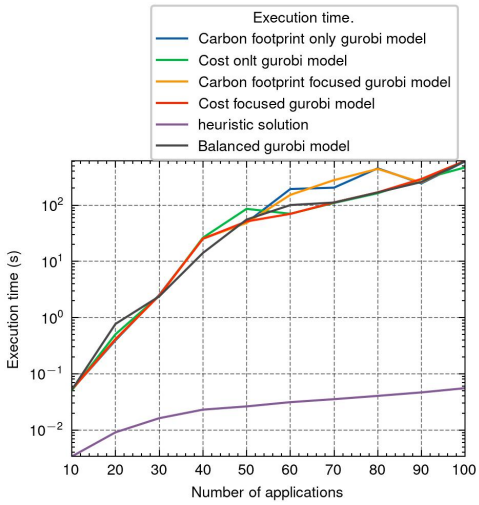


(c) Duration : 1000

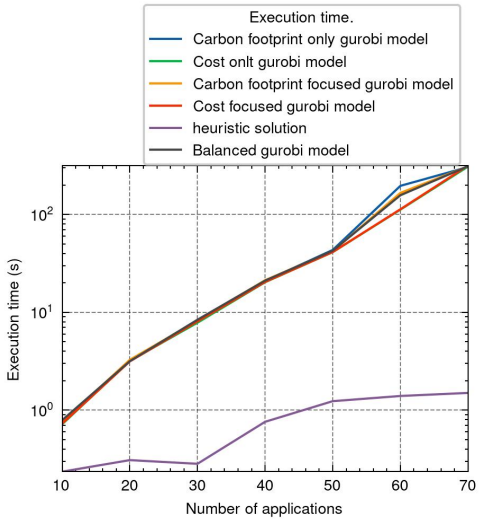
Figure 6.8: The gap in the Cost (%) produced by the different heuristic configurations for a duration of 10, 100 and 1000



(a) Duration : 10



(b) Duration : 100



(c) Duration : 1000

Figure 6.9: The execution time of the models for a duration of 10, 100 and 1000

Chapter 7

Conclusion & Perspectives

7.1 Conclusion

In conclusion, this thesis has provided solutions for automating the management of Cloud Edge Computing Continuum (CECC). The journey of this thesis commenced with the creation of an end-to-end multi-domain service monitoring framework. Such services consist of applications and network functions that span over multiple technological domains, each presenting its unique intricacies. The proposed monitoring system employs a unified structure of Key Performance Indicator (KPI), effectively abstracting away any underlying complexities. Extensive testing across different scenarios validates the framework's scalability and its ability to monitor a high number of services simultaneously. The next station of the thesis involved application profiling, in which we conducted an experimental study to explore the behavior of different types of applications in cloud-native environments. This study highlights the inability of application owners to configure the appropriate resources for their applications to operate optimally without resulting in infrastructure resource wastage. Then, we employed Artificial Intelligence (AI) and eXplainable Artificial Intelligence (XAI) techniques to build models capable of predicting performance degradation in applications, utilizing datasets generated from our study. When the model predicts a decline in application performance, the XAI module provides explanations for the model's output, facilitating the identification of the root cause of service degradation. This root cause is then addressed by the application manager. The journey of this thesis concluded with the proposal of an architecture for the Life-Cycle Management (LCM) of CECC applications. This architecture uses application and infrastructure profiles to deploy and migrate applications while considering the carbon footprint of the CECC deployment. The main challenge is the concrete representation of the application profile in a way that can be used to specify current and future application requirements. This challenge was tackled by effectively representing the application profile to facilitate deriving both current and future application requirements.

Automating the Life-Cycle Management of CECC is crucial to unlocking the full potential of emerging use cases. By providing a spectrum wherein applications can seamlessly run and migrate across various locations, manual operations (placement, scaling, migration ...) become obsolete as they fail to achieve optimal resource allocation and performance efficiency. This thesis represents incremental progress towards the paradigm of zero-touch CECC management, aiming to attain complete automation of management procedures.

7.2 Future Perspectives

7.2.1 Profile creation and management for applications through collaboration

In the second contribution of this thesis, we studied the behavior of applications across varying loads and resource configurations. It is evident that applications exhibit diverse responses to these factors, with some leaning heavily on CPU resources, others on memory, and some on both. Additionally, they may exhibit unique traffic patterns, reflecting differences in service popularity over time. This highlights the need for comprehensive data to profile applications effectively. One approach to profiling is to analyze the behavior of each deployed application and categorize them into classes based on similarities. When a new application is deployed, the system compares its behavior against existing profiles to determine the closest match. If no suitable class exists, a new one is created. Given the proliferation of management systems and platforms, this process often repeats, resulting in redundant profiling efforts for similar applications.
To address this inefficiency, there is potential for collaboration among platform providers to share application profiles. Major players like Amazon Web Services, Google Cloud Platform, and Azure possess both the resources and a diverse range of applications, making them ideal candidates for such collaboration. This partnership could encompass various aspects such as collaboration schemes, ensuring data privacy, and providing incentives to application owners for sharing profiles (e.g., monetary rewards and free access to profiles for future deployments). Such collaboration not only allows application owners to export and utilize profiles elsewhere but also enables smaller providers to access initial application profiles based on similar applications. This facilitates efficiency in profile creation and enhances interoperability across platforms.

7.2.2 Serverless computing and WebAssembly edge modules management

Serverless computing is an application deployment model where the application owner provides only the code to be executed; meanwhile, the execution environment, infrastructure, and resource allocation are managed by the serverless platform provider. A key form of this service model is Function as a Service (FaaS), where functions execute custom code in response to events, making it ideal for event-driven architectures.
Implementing such models can utilize containers, where each function runs inside a container that is terminated after the function completes. However, this approach may lead to cold-start delays if no container is available during event occurrence. An alternative approach is using WebAssembly modules. WebAssembly (WASM) is an emerging technology that, unlike containers, offers a portable binary instruction format for a stack-based virtual machine. WASM was initially designed as a compilation target for high-performance languages like C, C++, and Rust to run in web browsers. WebAssembly (WASM) can package any application and deploy it on various hardware platforms as portable modules via the WebAssembly System Interface (WASI). Its advantages include a load-time-efficient binary format and its language-agnostic nature, enabling developers to compile code from any language to WebAssembly (WASM) for execution in web browsers or server-side runtimes.
Modular cloud edge applications can benefit from the portability and fast load times provided by WASM. Thus, there is a need for an application development framework that allows the separation and migration of application modules at runtime. This facilitates compute offloading to the edge, reducing battery usage of edge devices and enabling the sharing of application modules across multiple instances to minimize overall resource usage. Such scenarios necessitate dynamic scheduling of application modules.

Dynamic scheduling of offloaded modules (as shown in Fig. 7.1), triggered by situational changes during application runtime, presents new challenges, including application state handling, minimizing development overhead, and network integration of the signaling path between applications' modules.

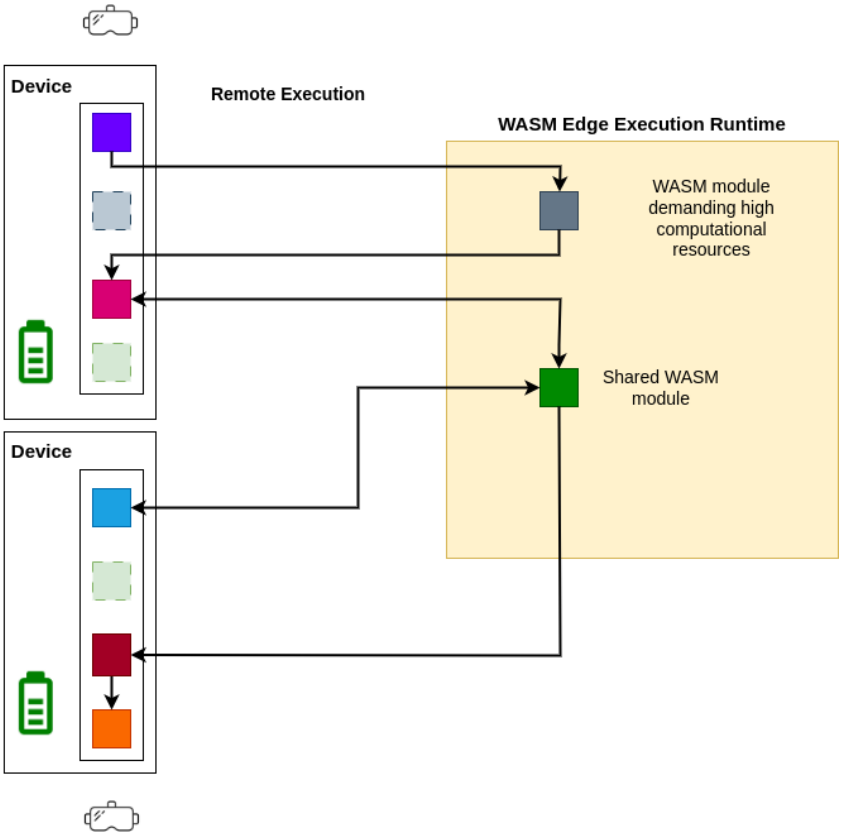


Figure 7.1: WASM application deployment on edge devices

7.2.3 Intent Driven Application Management

Considering the complex nature of CECC deployments spanning across various infrastructures and domains, application owners may lack precise knowledge of the compute and network needs for their use cases. An alternative approach is to utilize a more convenient method of describing services, relying on intent or higher-level descriptions of the operational objectives and constraints of the service provided by the application owner. Intent-based application deployment will be instrumental in enabling autonomous services. Using intent, the application owner does not need prior knowledge of the Quality of Service (QoS) requirements for their service. Instead, the user can express their service's intent in a human-readable format, such as JSON or YAML. The CECCM then translates this intent into service and network-level policies. These policies are further translated into detailed low-level configurations that can be deployed onto the CECC infrastructure. Ultimately, the CECCM should ensure the continual fulfillment of the intent throughout the service's life cycle. Efforts to facilitate intent-based management of networks fall within the scope of Intent-Based Networking (IBN). significant efforts have been devoted to defining and standardizing it, including the 3rd Generation Partnership Project (3GPP), the European Telecommunications Standards Institute (ETSI), and the TM Forum.

However, the challenge with such systems lies in constructing a knowledge base that facilitates the translation of intents into service-level policies. Intent profiling, mapping existing intents to their corresponding policies, along with large language model (LLM), can be useful for enabling intent-driven management.

Chapitre 8

Résumé

8.1 Contexte de la Thèse

Le Cloud Edge Computing Continuum (CECC) a vu le jour grâce au changement de paradigme dans le développement et le déploiement des applications, intégrant la scalabilité et la disponibilité des ressources du cloud computing avec les capacités de traitement en temps réel du edge computing. Ce continuum est rendu possible grâce à la maturation de la provision et de la gestion du cloud computing et du edge computing, offrant un spectre où les applications peuvent migrer de manière transparente entre les infrastructures cloud centralisées et cloud de bordure ou le et far edge décentralisés, en fonction de facteurs tels que les besoins en ressources de calcul, la latence et les contraintes de bande passante. À une extrémité de ce continuum se trouve le cloud computing traditionnel, caractérisé par d'énormes centres de données et des ressources de traitement centralisées, offrant une scalabilité des ressources et des capacités de stockage. En revanche, à l'extrémité opposée, le edge computing exploite des infrastructures de calcul locales composées de petits centres de données ou de dispositifs far edge, réduisant la latence et améliorant la réactivité pour les applications sensibles au temps. De plus, les dispositifs edge et far edge peuvent être mobiles ou volatils, ce qui signifie qu'ils ne sont pas disponibles en permanence en raison de restrictions de batterie ou de mobilité, comme pour les dispositifs far edge basés sur les drones ou les ordinateurs monocartes alimentés par des batteries.
Le continuum entre ces extrêmes permet une nouvelle architecture qui tire parti de la distribution dynamique des microservices entre les nœuds cloud et edge en fonction des demandes de charge de calcul et des conditions du réseau. Ce paradigme encourage le développement d'applications innovantes couvrant plusieurs domaines, tels que l'Internet of Things (IoT), les systèmes autonomes et la réalité augmentée. Dans ce cas, l'orchestration transparente des ressources à travers le continuum cloud edge est cruciale pour répondre aux exigences de performance et permet une connectivité omniprésente pour les applications déployées d'une part et utiliser efficacement les ressources d'infrastructure sous-jacentes d'autre part.

8.2 Motivation

Une infrastructure CECC s'étend sur plusieurs emplacements et domaines, incluant le cloud public, le cloud edge dédié et les appareils informatiques capables. Dans ce contexte, le Cloud Edge Computing Continuum Manager (CECCM) est un élément clé pour gérer le cycle de vie des applications et administrer les ressources d'infrastructure fédérées. Le CECCM doit être autogéré et auto-configuré pour permettre une gestion et une configuration sans intervention en prenant des décisions basées sur les données de supervision collectées concernant le comporte-

ment des applications, l'utilisation des ressources d'infrastructure ainsi que l'état du réseau. Ces décisions devraient garantir le respect des exigences des applications en fonction des capacités d'infrastructure disponibles. Pour atteindre cet objectif, le premier composant d'un CECCM est le système de supervision qui offre une visibilité sur le comportement des applications déployées et les ressources d'infrastructure. En effet, le système de surveillance devrait couvrir l'intégralité de l'infrastructure CECC.
Ensuite, pour gérer efficacement les applications, le CECCM doit comprendre le comportement et les exigences des applications. Nous désignons sous le terme de profil d'application la représentation du comportement de l'application, de ses performances, de son utilisation des ressources et de ses dépendances dans l'environnement de déploiement. En profilant les applications déployées sur le CECC, le CECCM peut optimiser le placement des applications pour améliorer leurs performances et optimiser l'utilisation des ressources d'infrastructure. Les techniques de profilage peuvent inclure des métriques du système de supervision et l'analyse des modèles de trafic du réseau. Par ailleurs, les techniques d'Artificial Intelligence (AI) peuvent être utilisées pour prédire les besoins futurs de l'application en termes de ressources de calcul et d'exigences réseau.
Les procédures de gestion du cycle de vie des applications incluent principalement le placement initial, la mise à l'échelle des ressources et la migration des applications. Étant donné que le système de gestion du CECC prend des décisions basées sur les données et les mesures collectées concernant le comportement des applications, il utilise des modèles basés sur le Machine Learning (ML) ou en les formalisant par le biais de modèles d'optimisations mathématiques. Il est important que les décisions du système puissent être fiables pour garantir le respect du Service Level Agreement (SLA) exigés par les applications.
Enfin, chaque noeud de calcul de l'infrastructure CECC consomment de l'électricité pour fonctionner et de l'eau pour le refroidissement. En fonction des sources d'énergie utilisées par les nœuds d'infrastructure, l'infrastructure CECC et les applications s'exécutant dessus produisent une empreinte carbone. En effet, de plus en plus de gouvernements prennent des mesures pour réduire les émissions de carbone ; par exemple, l'Union européenne vise à être neutre en carbone d'ici 2050 ; cet objectif est au cœur du Pacte vert européen [1]. La technologie de communication ne fait pas exception, car en 2021, une note technique d'ACM [2] estimait que le secteur des Information and Communication Technology (ICT) contribuait entre 1,8% et 3,9% des émissions mondiales de carbone. Cela motive la disponibilité de sources d'énergie verte pour alimenter les nœuds de calcul, ce qui rend nécessaire la prise de décisions tenant compte de l'empreinte carbone lors de la gestion du déploiement des applications et des infrastructures faisant partie du CECC.

8.3 Les contributions de la Thèse

Cette thèse vise à permettre la gestion automatisée des déploiements multi-domaines de CECC. Nous avons commencé par aborder le défi de la supervision unifiée de bout en bout des déploiements multi-domaines où les microservices composant l'application (ou les applications cloud-native) peuvent être déployés dans différentes infrastructures utilisant différentes technologies. Nous proposons un nouveau système de supervision utilisant des collecteurs de métriques conçus pour recueillir les métriques des sous-services d'applications dans une région de déploiement (ou infrastructure). Ces métriques sont ensuite agrégées pour offrir une vue d'ensemble de bout en bout sur les Key Performance Indicator (KPI) de service, en abstrayant les détails technologiques sous-jacents. Par la suite, nous utilisons les données collectées par le système de supervision pour comprendre le comportement des applications dans différents scénarios. Cela nous permet

de construire des ensembles de données sur les besoins en ressources de différents types d'applications sous différentes charges. Fournissant ainsi un profil de base pour l'application, qui inclut les exigences des instances de l'application en fonction de la charge de trafic arrivant vers les services. La prochaine étape de la thèse était de fermer la boucle de contrôle en produisant et en appliquant des décisions pour la gestion du cycle de vie des applications basées sur les données de surveillance. Pour cela, nous proposons un framework pour le Zero-touch Service Management (ZSM) contenant un module pour la mise à l'échelle automatique des ressources à granularité fine. Le scaler exploite l'eXplainable Artificial Intelligence (XAI) pour prendre des décisions sur quelle ressource mettre à l'échelle, basées sur la sortie d'un modèle ML qui prédit les performances des microservices ; le modèle a été entraîné sur l'ensemble de données généré par la deuxième contribution. Enfin, nous modélisons le problème du placement et de la migration des applications en utilisant un problème d'optimisation multi-objectif avec deux objectives : le premier est de réduire l'empreinte carbone du déploiement, et le second est de réduire le coût du déploiement. La figure 8.1 représente le parcours de la thèse, de la collecte des métriques à l'exécution de décisions basées sur les données, fermant ainsi la boucle de contrôle.

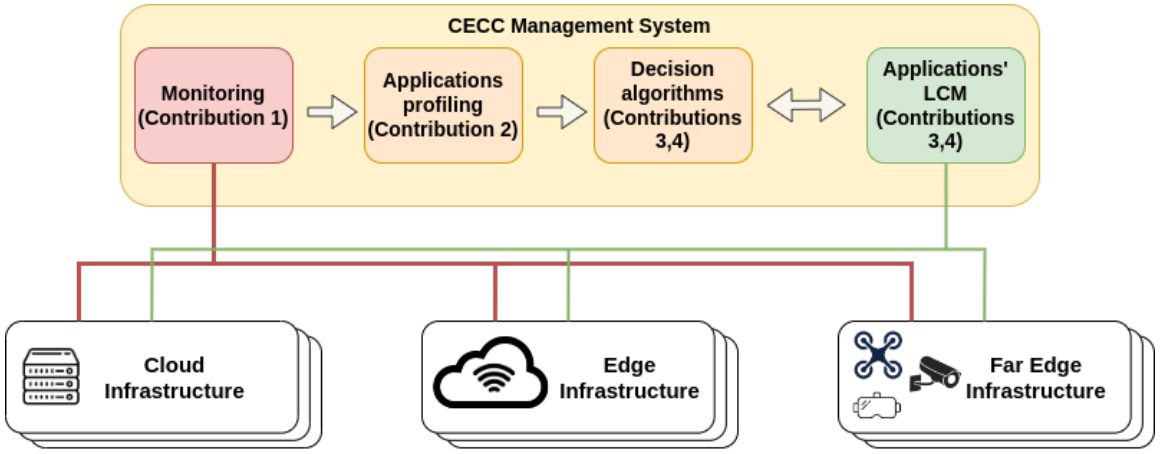


FIGURE 8.1 : Parcours de la thèse

8.3.1 Supervision de bout en bout des applications et services multi-domaines

Description du challenge

Un système de supervision efficace et adaptatif est un composant critique pour toute provision de cloud computing ou réseau afin de superviser et valider le fonctionnement des services en cours d'exécution et de l'infrastructure sous-jacente. Ceci est d'autant plus valide dans le contexte de la 5G, car elle repose sur le concept de "network slicing", ce qui ajoute de nombreux défis au système de supervision. Parmi ces défis, on trouve le fait que les tranches de réseau utilisent des ressources de différents domaines technologiques impliquant différentes entités basées sur différentes technologies, ce qui nécessite de surveiller différents types de ressources, comme le Radio Access Network (RAN), le calcul, la mémoire et le débit de données réseau. En effet, la supervision des composants RAN est totalement différente de la supervision de l'infrastructure Cloud ou de l'infrastructure de Network function virtualization (NFV). Un autre défi concerne l'évolutivité du système de supervision, car l'opérateur de réseau est censé exécuter plusieurs slices de réseau parallèles sur son infrastructure 5G. Enfin, la multi-tenancy et l'isolation entre les slices de réseau doivent être mises en œuvre ; les données liées à un slice ne doivent être visibles que par son propriétaire.

Solution proposée

Nous concevons un nouveau framework de supervision prêt pour le slicing multi-domaine du réseau pour les réseaux mobiles, y compris les déploiements CECC. Le framework est indépendant de la technologie car il fournit des mesures de manière uniforme, en utilisant une structure unifiée de KPI, abstrayant efficacement toutes les complexités technologiques sous-jacentes. Le système de supervision est évolutif et prend en charge un grand nombre de services en parallèle.

Publications

A Scalable Monitoring Framework for Network Slicing in 5G and Beyond Mobile Networks
Mohamed Mekki ; Sagar Arora ; Adlen Ksentini
IEEE Transactions on Network and Service Management (Volume : 19, Issue : 1, March 2022)

A multi-technological domains KPIs Monitoring System for Network Slicing in 5G
Mohamed Mekki ; Sagar Arora ; Adlen Ksentini
Cloud Days, 25-26 November 2021, Brest, France

8.3.2 Profilage des applications dans des environnements cloud-native

Description du challenge

L'émergence des architectures cloud-native et de la conteneurisation a changé la façon dont les applications sont développées et déployées. Les applications actuelles décomposent les fonctionnalités du service en plusieurs microservices ; chaque microservice est responsable d'un sous-ensemble de ces fonctionnalités globales. Lorsqu'ils sont encapsulés dans des conteneurs, pour être exécutés sur l'infrastructure cloud ou edge, le propriétaire de l'application doit spécifier les ressources de calcul nécessaires pour exécuter ses applications. En effet, le propriétaire del'application doit indiquer la quantité de CPU et de mémoire limite pour un conteneur exécutant un micro-service. Il arrive qu'un conteneur dépassant ces limites soit arrêté ou subisse une baisse de performances. Par conséquent, déterminer la limite à attribuer à un conteneur et configurer une ressource de service est un défi important. D'une part, le propriétaire ne comprend pas clairement l'environnement dans lequel l'application sera déployée ; d'autre part, le fournisseur de la plateforme reçoit l'application sous la forme d'un conteneur empaqueté dans lequel la charge de travail est vue comme une boîte noire. Dans de nombreuses situations, la configuration se termine par l'utilisation de configurations par défaut qui ne sont pas adaptées aux besoins de l'application. En effet, les locataires demandent naturellement une limite plus élevée que ce dont l'application a besoin, ce qui, à son tour, dans un environnement contraint (comme le edge), entraîne un gaspillage de ressources.

Solution proposée

Nous menons une étude expérimentale visant à détecter si la configuration d'un conteneur permet d'exécuter son micro-service de manière optimale. À cette fin, nous menons plusieurs expériences sur une plateforme cloud-native, en utilisant différents types d'applications avec différentes configurations de ressources. Les résultats obtenus fournissent des informations sur la manière de détecter et de corriger la dégradation des performances due à une mauvaise configuration des ressources du service.

Publications

Microservices Configurations and the Impact on the Performance in Cloud Native Environments
Mohamed Mekki ; Nassima Toumi ; Adlen Ksentini
2022 IEEE 47th Conference on Local Computer Networks (LCN), 26-29 September 2022, Edmonton, AB, Canada.

8.3.3 Mise à l'échelle automatique des ressources des applications dans des environnements cloud-native

Description du challenge

La conteneurisation des applications composées de microservices émerge comme un nouveau paradigme pour optimiser la portabilité, la flexibilité et la gestion de telles applications. Exécuter des applications cloud basées sur une architecture de microservices crée de nouveaux défis. Les applications cloud nécessitent différents Quality of Service (QoS) ainsi que des exigences en ressources variées, ce qui nécessite la conception des farameworks de mise à l'échelle (ou scaling) des resources fiables. Dans ce contexte, les algorithmes de ML, et en particulier l'apprentissage par renforcement (RL), ont largement été exploités pour concevoir des framework de mise à l'échelle intelligents et autonomes. Ils visent à déterminer les bonnes valeurs pour les différents besoins en ressources des microservices des applications et donc à répondre aux QoS des applications. Cependant, les modèles basés sur le ML deviennent de plus en plus complexes, et leurs décisions sont difficilement interprétées par les utilisateurs, notamment les administratuers des services cloud ainsi que les systèmes d'orchestration de conteneurs. Par conséquent, les utilisateurs (ou systèmes) correspondants ne peuvent ni faire confiance ni comprendre les sorties des modèles de ML, ni optimiser leurs décisions en fonction des sorties de ces modèles. La gestion fine granulaire des ressources informatiques cloud-native est l'une des principales fonctionnalités recherchées par les opérateurs cloud et edge. Elle consiste à fournir la quantité exacte de ressources nécessaire à un microservice pour éviter la sur-provisionnement des ressources, qui est, par défaut, la solution adoptée pour éviter la dégradation du service. La gestion fine granulaire des ressources garantit une meilleure utilisation des ressources informatiques, ce qui est essentiel pour réduire la consommation d'énergie et le gaspillage des ressources (vital dans le edge computing).

Solution proposée

Dans notre solution, nous proposons un nouveau framework ZSM doté d'un scaler de ressources à granularité fine dans un environnement cloud-native. L'algorithme de scaling proposé utilise des modèles d'AI/ML pour prédire les performances des microservices. Plus précisément, nous utilisons eXtreme Gradient Boosting (XGBoost) comme algorithme ML pour prédire les éventuelles violations liées aux performances de latence des applications en cours d'exécution ; si une dégradation de service est détectée, une analyse de la cause racine est alors menée à l'aide de XAI. En fonction de la sortie de l'XAI, le cadre proposé ne redimensionne que les ressources (c'est-à-dire CPU ou mémoire) nécessaires (quantité exacte) pour résoudre la dégradation du service. Le framework proposé et le gestionnaire de ressources ont été implémentés sur une plateforme cloud-native basée sur l'outil bien connu Kubernetes. Le scaler proposé avec des ressources moindres atteint la même qualité de service que le scaler par défaut de Kubernetes.

Publications

XAI-Enabled Fine Granular Vertical Resources Autoscaler
Mohamed Mekki ; Bouziane Brik ; Adlen Ksentini ; Christos Verikoukis
2023 IEEE 9th International Conference on Network Softwarization (NetSoft), 19-23 June 2023, Madrid, Spain.

8.3.4 Gestion du cycle de vie des applications prenant en compte l'énergie dans le cloud edge computing continuum

Description du challenge

Le déploiement d'applications sur une infrastructure CECC nécessite des informations détaillées sur le profil de chaque application à gérer par l'orchestrateur ou le gestionnaire d'applications CECC. Le profilage des applications désigne l'analyse systématique des caractéristiques d'une application. Ce processus vise à obtenir des informations sur le comportement, les performances, l'utilisation des ressources et les dépendances d'une application au sein de l'infrastructure cloud. En ayant accès à ces profils, le CECCM peut optimiser le placement des applications pour améliorer les performances et maximiser l'utilisation efficace des ressources d'infrastructure. Les méthodes de profilage englobent une variété de techniques, telles que la supervision des métriques système et l'analyse des modèles de trafic réseau. De plus, l'intégration de techniques d'intelligence artificielle facilite la modélisation prédictive pour anticiper les demandes futures en termes de ressources de calcul et de besoins réseau des charges de travail des applications.
De plus, l'infrastructure CECC produit une empreinte carbone, qui augmente avec l'échelle de l'infrastructure. Cependant, dans le cadre du mouvement mondial vers la durabilité, des initiatives telles que le Pacte vert européen [1] accélèrent la transition vers des sources d'énergie plus vertes. Ce changement augmente la disponibilité de l'infrastructure alimentée par des énergies vertes. Par conséquent, lors de la décision du placement optimal des microservices, les considérations de consommation d'énergie et l'utilisation de sources d'énergie verte doivent être soigneusement équilibrées tout en satisfaisant toujours le SLA de l'application. De plus, l'orchestrateur d'applications CECC (c'est-à-dire le CECCM) peut tirer parti du profil de l'application pour prédire l'utilisation des ressources et décider de migrer les microservices de l'application si une meilleure empreinte carbone peut être obtenue avec une nouvelle configuration et un nouvel emplacement d'exécution des applications.

Solution proposée

Nous proposons une architecture de l'Orchestratrice d'Applications CECC. L'architecture tire parti du profilage des applications et des infrastructures pour gérer efficacement les applications CECC. Ce profilage est réalisé en utilisant les informations de supervision, y compris les métriques énergétiques collectées à partir de systèmes de supervision de l'énergie tels que Kepler [4] et l'intensité carbone des infrastructures. Ensuite, nous définissons une méthode de modélisation des profils des applications du point de vue du CECCM. Le profil représente les besoins actuels et futurs en termes de resources de calcul et de réseau de l'application. Le profil est construit en fonction de l'utilisation historique de l'application à l'aide de méthodes statistiques. Enfin, nous modélisons le problème de sélection des meilleurs emplacements pour déployer ou migrer les applications tout en minimisant le coût du déploiement et l'empreinte carbone. Le modèle décide des emplacements actuels et futurs de chaque application en fonction du profil de l'application, des contraintes de disponibilité de l'application et des ressources disponibles

des infrastructures. Nous proposons en outre une solution heuristique pour résoudre le problème rapidement et nous comparons les différents compromis entre l'efficacité carbone et le coût.

Publications

Energy-Aware Application Life-Cycle Management in Cloud Edge Computing Continuum Using Applications Profiles
Mohamed Mekki ; Adlen Ksentini
En cours de soumission.

8.4 Conclusion

En conclusion, cette thèse a fourni des solutions pour automatiser la gestion du Cloud Edge Computing Continuum (CECC). Le parcours de cette thèse a commencé par la création d'un framework de supervision de services multi-domaines de bout en bout. De tels services consistent en des applications et des fonctions réseau qui s'étendent sur plusieurs domaines technologiques, chacun présentant ses propres intrications uniques. Le système de supervision proposé utilise une structure unifiée de Key Performance Indicator (KPI), abstrayant efficacement toutes les complexités sous-jacentes. Des tests approfondis dans différents scénarios valident la scalabilité du framework et sa capacité à superviser un grand nombre de services simultanément. La prochaine étape de la thèse impliquait le profilage des applications, dans lequel nous avons mené une étude expérimentale pour explorer le comportement de différents types d'applications dans des environnements cloud-native. Cette étude met en évidence l'incapacité des propriétaires d'applications à configurer les ressources appropriées pour leurs applications afin de fonctionner de manière optimale sans entraîner de gaspillage des ressources d'infrastructure. Ensuite, nous avons utilisé des techniques d'Artificial Intelligence (AI) et d'eXplainable Artificial Intelligence (XAI) pour construire des modèles capables de prédire la dégradation des performances des applications, en utilisant des ensembles de données générés à partir de notre étude. Lorsque le modèle prédit un déclin des performances de l'application, le module XAI fournit des explications pour la sortie du modèle, facilitant l'identification de la cause profonde de la dégradation du service. Cette cause profonde est ensuite traitée par le gestionnaire d'application. Le parcours de cette thèse s'est conclu par la proposition d'une architecture pour la gestion du cycle de vie des applications CECC. Cette architecture utilise des profils d'application et d'infrastructure pour déployer et migrer des applications tout en tenant compte de l'empreinte carbone du déploiement CECC. Le principal défi réside dans la représentation concrète du profil d'application de manière à pouvoir spécifier les exigences actuelles et futures de l'application. Ce défi a été relevé en représentant efficacement le profil d'application pour faciliter la dérivation des exigences actuelles et futures de l'application.
Automatiser la gestion du cycle de vie du CECC est crucial pour débloquer le potentiel des cas d'utilisation émergents. En fournissant un spectre où les applications peuvent s'exécuter et migrer facilement entre différents endroits, les opérations manuelles (placement, mise à l'échelle, migration ...) deviennent obsolètes, car elles ne parviennent pas à obtenir une allocation optimale des ressources et une efficacité de performance. Cette thèse représente un progrès incrémental vers le paradigme de la gestion automatique du CECC sans aide humaineh, visant à automatiser complètement les procédures de gestion.

Bibliography

[1] *The European Green Deal*. URL: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/european-green-deal_en (visited on 2024).

[2] Bran Knowles. *ACM TechBrief: Computing and climate change*. 2021.

[3] *Kubernetes*. URL: https://kubernetes.io/ (visited on 2022).

[4] *Kubernetes Efficient Power Level Exporter*. URL: https://sustainable-computing.io/ (visited on 2024).

[5] EG ZSM. "Multi-access Edge Computing (MEC); Framework and Reference Architecture". In: *ETSI: Sophia Antipolis, France* (2024).

[6] EG ZSM. "Network Functions Virtualisation (NFV) Release 4; Management and Orchestration; Architectural Framework Specification". In: *ETSI: Sophia Antipolis, France* (2022).

[7] EG ZSM. "Zero-touch network and Service Management (ZSM); Landscape". In: *ETSI: Sophia Antipolis, France* (2022).

[8] Sagar Arora, Adlen Ksentini, and Christian Bonnet. "Cloud native Lightweight Slice Orchestration (CLiSO) framework". In: *Computer Communications* 213 (2024), pp. 1–12. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2023.10.010. URL: https://www.sciencedirect.com/science/article/pii/S0140366423003742.

[9] Sagar Arora, Adlen Ksentini, and Christian Bonnet. "Lightweight edge Slice Orchestration Framework". In: *ICC 2022 - IEEE International Conference on Communications*. 2022, pp. 865–870. DOI: 10.1109/ICC45855.2022.9838854.

[10] EG ZSM. "Zero-touch network and Service Management (ZSM); Reference Architecture". In: *ETSI: Sophia Antipolis, France* (2019).

[11] Hatim Chergui et al. "Toward Zero-Touch Management and Orchestration of Massive Deployment of Network Slices in 6G". In: *IEEE Wirel. Commun.* 29.1 (2022), pp. 86–93.

[12] EG ZSM. "Zero-touch network and Service Management (ZSM); General Security Aspects". In: *ETSI: Sophia Antipolis, France* (2021).

[13] Linghe Kong et al. "Edge-computing-driven Internet of Things: A Survey". In: *ACM Comput. Surv.* 55.8 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3555308. URL: https://doi.org/10.1145/3555308.

[14] Kaniz Fatema et al. "A survey of cloud monitoring tools: Taxonomy, capabilities and objectives". In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2918–2933.

[15] Erika Puiutta and Eric Veith. "Explainable reinforcement learning: A survey". In: *International cross-domain conference for machine learning and knowledge extraction*. Springer. 2020, pp. 77–95.

[16] Scott M Lundberg and Su-In Lee. "A unified approach to interpreting model predictions". In: *Advances in neural information processing systems* 30 (2017).

[17] Kelleher. *Causal Dataframe*. 2017. URL: https://github.com/akelleh/causality.

[18] Jennifer Hill and Elizabeth Stuart. "Causal inference: overview". In: *International Encyclopedia of the Social & Behavioral Sciences: Second Edition* (2015), pp. 255–260.

[19] André Altmann et al. “Permutation importance: a corrected feature importance measure”. In: *Bioinformatics* 26.10 (2010), pp. 1340–1347.

[20] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “” Why should i trust you?” Explaining the predictions of any classifier”. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1135–1144.

[21] Korobov. *ELI5*. 2019. URL: `https://github.com/TeamHG-Memex/eli5`.

[22] Arnaud Van Looveren and Janis Klaise. “Interpretable counterfactual explanations guided by prototypes”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2021, pp. 650–665.

[23] “SMA Public Policy Position”. In: *(2020, March) 5G Spectrum* ().

[24] Adlen Ksentini and Pantelis A Frangoudis. “Toward slicing-enabled multi-access edge computing in 5G”. In: *IEEE Network* 34.2 (2020), pp. 99–105.

[25] Adlen Ksentini et al. “Providing low latency guarantees for slicing-ready 5G systems via two-level MAC scheduling”. In: *IEEE Network* 32.6 (2018), pp. 116–123.

[26] Bouziane Brik and Adlen Ksentini. “On Predicting Service-oriented Network Slices Performances in 5G: A Federated Learning Approach”. In: *2020 IEEE 45th Conference on Local Computer Networks (LCN)*. IEEE. 2020, pp. 164–171.

[27] Ramon Perez et al. “A monitoring framework for multi-site 5G platforms”. In: *2020 European Conference on Networks and Communications (EuCNC)*. IEEE. 2020, pp. 52–56.

[28] Xenofon Vasilakos et al. “Elasticsdk: A monitoring software development kit for enabling data-driven management and control in 5g”. In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2020, pp. 1–7.

[29] Xenofon Foukas et al. “FlexRAN: A flexible and programmable platform for software-defined radio access networks”. In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 2016, pp. 427–441.

[30] André Beltrami et al. “Design and Implementation of an Elastic Monitoring Architecture for Cloud Network Slices”. In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2020, pp. 1–7.

[31] Francesco Tusa, Stuart Clayman, and Alex Galis. “Dynamic Monitoring of Data Center Slices”. In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2019, pp. 286–290.

[32] Stuart Clayman, Alex Galis, and Lefteris Mamatas. “Monitoring virtual networks with lattice”. In: *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*. IEEE. 2010, pp. 239–246.

[33] Márcio Barbosa de Carvalho et al. “A cloud monitoring framework for self-configured monitoring slices based on multiple tools”. In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. IEEE. 2013, pp. 180–184.

[34] Slawomir Kukliński and Lechosław Tomaszewski. “Dasmo: A scalable approach to network slices management and orchestration”. In: *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2018, pp. 1–6.

[35] Ibrahim Afolabi et al. “Network slicing-based customization of 5G mobile services”. In: *IEEE Network* 33.5 (2019), pp. 134–141.

[36] *3rd Generation Partnership Project (3GPP), 2018b. “Study on manage-ment and orchestration of network slicing for next generation network”. 3GPP TS 28.801 version 15.1.0 Release 15.*

[37] “Generic Network Slice Template”, version 3.0”. In: *NG.116,May 2020* ().

[38] Chia-Yu Chang et al. “Slice orchestration for multi-service disaggregated ultra-dense RANs”. In: *IEEE Communications Magazine* 56.8 (2018), pp. 70–77.

[39] In: *Multi-access Edge Computing (MEC); MEC Management; “Part 1: Application lifecycle, rules and requirements management”, ETSI GS MEC 010-1 V1.1.1 (2017-10)* ().

[40] In: *ORAN Alliance, 2020b. "Operator defined next generation ran architecture and interfaces". URL:https://www.o-ran.org/* ().

[41] In: *Network Functions Virtualisation (NFV) Release 3; Virtualised Network Function; "Specification of the Classification of Cloud Native VNF implementations", ETSI GS NFV-EVE 011 V3.1.1, Oct 2018* ().

[42] Yu Gan et al. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud amp; Edge Systems". In: ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18.

[43] Pooyan Jamshidi et al. "Self-Learning Cloud Controllers: Fuzzy Q-Learning for Knowledge Evolution". In: *2015 International Conference on Cloud and Autonomic Computing*. 2015, pp. 208–211.

[44] Manuel Gotin et al. "Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 157–167.

[45] E. Casalicchio and V. Perciballi. "Auto-scaling of containers: The impact of relative and absolute metrics". In: *IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. 2017.

[46] Xuehai Tang et al. "Fisher: An efficient container load prediction model with deep neural network in clouds". In: *IEEE International Conference on Big Data and Cloud Computing (BdCloud)*. IEEE. 2018, pp. 199–206.

[47] Abeer Abdel Khaleq and Ilkyeun Ra. "Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications". In: *IEEE Access* 9 (2021).

[48] Fabiana Rossi et al. "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning". In: *IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 329–338.

[49] Krzysztof Rzadca et al. "Autopilot: Workload Autoscaling at Google". In: *Proceedings of EuroSys '20*. Heraklion, Greece: ACM, 2020.

[50] Abhishek Verma et al. "Large-Scale Cluster Management at Google with Borg". In: *Proceedings of EuroSys '15*. Bordeaux, France: Association for Computing Machinery, 2015.

[51] Duc-Hung Luong et al. "Predictive autoscaling orchestration for cloud-native telecom microservices". In: *IEEE 5G World Forum (5GWF)*. 2018, pp. 153–158.

[52] Mohamed Mekki et al. "A Scalable Monitoring Framework for Network Slicing in 5G and Beyond Mobile Networks". In: *IEEE Transactions on Network and Service Management* 19.1 (2022), pp. 413–423.

[53] *OpenAirInterface*. URL: `https://openairinterface.org/` (visited on 2022).

[54] Pantelis A. Frangoudis et al. "An architecture for on-demand service deployment over a telco CDN". In: *2016 IEEE International Conference on Communications, ICC 2016, Kuala Lumpur, Malaysia, May 22-27, 2016*. IEEE, 2016, pp. 1–6. DOI: 10.1109/ICC.2016.7510921.

[55] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 329–338. DOI: 10.1109/CLOUD.2019.00061.

[56] Qian Li et al. "RAMBO: Resource Allocation for Microservices Using Bayesian Optimization". In: *IEEE Computer Architecture Letters* 20.1 (2021), pp. 46–49. DOI: 10.1109/LCA.2021.3066142.

[57] Sreekrishnan Venkateswaran and Santonu Sarkar. "Fitness-Aware Containerization Service Leveraging Machine Learning". In: *IEEE Transactions on Services Computing* 14.6 (2021), pp. 1751–1764. DOI: 10.1109/TSC.2019.2898666.

[58] Mark Hamilton et al. "Large-Scale Intelligent Microservices". In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 298–309. DOI: 10.1109/BigData50022.2020.9378270.

[59] Imad Alawe et al. “Improving Traffic Forecasting for 5G Core Network Scalability: A Machine Learning Approach”. In: *IEEE Netw.* 32.6 (2018), pp. 42–49.

[60] Mohamed Mekki, Nassima Toumi, and Adlen Ksentini. “Microservices Configurations and the Impact on the Performance in Cloud Native Environments”. In: *2022 IEEE 47th Conference on Local Computer Networks (LCN)*. 2022, pp. 239–244. DOI: 10.1109/LCN53696.2022.9843385.

[61] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785.

[62] Krzysztof Rzadca et al. “Autopilot: Workload Autoscaling at Google”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387524.

[63] Abeer Abdel Khaleq and Ilkyeun Ra. “Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications”. In: *IEEE Access* 9 (2021), pp. 35464–35476. DOI: 10.1109/ACCESS.2021.3061890.

[64] In: *ENI Vision: Improved Network Experience using Experiential Networked Intelligence, ETSI ENI White paper* ().

[65] In: *Network Functions Virtualisation (NFV) Release 3; Architecture; “Report on the Enhancements of the NFV architecture towards Cloud-native and PaaS”, ETSI GR NFV-IFA 029 V3.3.1, Nov. 2019* ().

[66] In: *dataset: Benchmarking on Microservices Configurations and the Impact on the Performance in Cloud Native Environments* (). URL: https://zenodo.org/record/6907619#.YvDXKOxBxaR.

[67] Kuljeet Kaur et al. “KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem”. In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4228–4237. DOI: 10.1109/JIOT.2019.2939534.

[68] Walid A. Hanafy et al. “CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 7.3 (Dec. 2023). DOI: 10.1145/3626788. URL: https://doi.org/10.1145/3626788.

[69] Jungsu Han, Yujin Hong, and Jongwon Kim. “Refining Microservices Placement Employing Workload Profiling Over Multiple Kubernetes Clusters”. In: *IEEE Access* 8 (2020), pp. 192543–192556. DOI: 10.1109/ACCESS.2020.3033019.

[70] Ali E. Elgazar and Khaled A. Harras. “Enabling Seamless Container Migration in Edge Platforms”. In: *Proceedings of the 14th Workshop on Challenged Networks*. CHANTS’19. Los Cabos, Mexico: Association for Computing Machinery, 2019, pp. 1–6. ISBN: 9781450369336. DOI: 10.1145/3349625.3355438. URL: https://doi.org/10.1145/3349625.3355438.

[71] Jie Zhang et al. “Kole: Breaking the scalability barrier for managing far edge nodes in cloud”. In: *Proceedings of the 13th Symposium on Cloud Computing*. 2022, pp. 196–209.

[72] Yulai Xie et al. “Real-Time Prediction of Docker Container Resource Load Based on a Hybrid Model of ARIMA and Triple Exponential Smoothing”. In: *IEEE Transactions on Cloud Computing* 10.2 (2022), pp. 1386–1401. DOI: 10.1109/TCC.2020.2989631.

[73] In Kee Kim et al. “Forecasting Cloud Application Workloads With CloudInsight for Predictive Resource Management”. In: *IEEE Transactions on Cloud Computing* 10.3 (2022), pp. 1848–1863. DOI: 10.1109/TCC.2020.2998017.

[74] *Prometheus*. URL: https://prometheus.io/ (visited on 2024).

[75] *Electricity Maps*. URL: https://app.electricitymaps.com/map (visited on 2024).

[76] *National Grid ESO Carbon Intensity API*. URL: https://carbonintensity.org.uk/ (visited on 2024).