D.4.2 Report on the resource management procedures over CECC - Final

Document Summary Information

Project Identifier	HORIZON-CL4-2022-DATA-01. Project 101093129		
Project name	Agile and Cognitive Cloud-edge Continuum management		
Acronym	AC^3		
Start Date	January 1, 2023	End Date	December 31, 2025
Project URL	www.ac3-project.eu		
Deliverable	D4.2 Report on the resource management procedures over CECC - Final		
Work Package	WP4		
Contractual due date	30th June 2025	Actual submission date	30th June 2025
Type	R – Document, Report	Dissemination Level	PU – Public
Lead Beneficiary	UPR		
Responsible Author	Vrettos Moulos (UPR)		
Contributors	Amadou Ba (IBM), Christopher Lohse (IBM), Adlen Ksentini (EUR), Ayoub Mokhtari (EUR), Mohamed Mekki (EUR), Abd Elghani MELIANI (EUR), Shreya Chari (IQU), John Vardakas (IQU), Ben Capper (RHT), Ray Carroll (RHT), Ryan Jenkins (RHT), Christos Verikoukis (ISI/ATH), Elias Dritsas (ISI/ATH), Vasilis Avgerinos (ISI/ATH), Athanasios Kordelas (CTX), George Tsolis (CTX), John Beredimas (CTX), Alabi Rasheed (FIN), Ibrahim Afolabi (FIN), Vrettos Moulos (UPR), Dimosthenis Kyriazis (UPR) Dimitrios Amaxilatis (SPA), Nikos Tsironis (SPA), Ali Nikoukar (ION), Giannis Koulopoulos (UPR)		
Peer reviewer(s)	Ibrahim Afolabi (FIN), Souvik Sengupta(ION)		

Revision history (including peer reviewing & quality control)

Version	Issue Date	% Complete	Changes	Contributor(s)
v1.0	17/03/2025	5%	Initial Deliverable Structure	Vrettos Moulos (UPR)
V1.1	19/03/2025	15%	Drafted Section 3	Vrettos Moulos (UPR)
V1.2	28/03/2025	20%	Drafted Section 2	Amadou Ba (IBM)
V1.3	09/04/2025	30%	Drafter Section 4	Ibrahim Afolabi (FIN), Alabi Rasheed (FIN)
V1.5	22/04/2025	50%	Reviewed and Sections Updated	Giannis Koulopoulos (UPR), Ben Capper (RHT), Ray Carroll (RHT), Ryan Jenkins (RHT)
V2.0	15/05/2025	75%	Drafted Section 5, 6 and 7	Shreya Chari (IQU), John Vardakas (IQU), Ben Capper (RHT), Ray Carroll (RHT), Ryan Jenkins (RHT), Elias Dritsas (ISI/ATH), Vasilis Avgerinos (ISI/ATH), Athanasios Kordelas (CTX), George Tsolis (CTX), John Beredimas (CTX), Christopher Lohse (IBM), Amadou Ba (IBM)
V2.1	02/06/2025	90%	Conclusions & Executive Summary	Vrettos Moulos (UPR), Amadou Ba (IBM)
V3.0	08/06/2025	95%	Reviewed and quality checked (References, links, tables of contents)	Vrettos Moulos (UPR)
V4.0	09/06/2025	99%	Review	Souvik Sengupta (ION), Ibrahim Afolabi (FIN)
V5.0	20/06/2025	99%	Review Fixes	All
V6.0	28/07/2025	99%	Review Fixes (second iteration)	All
V7.0	17/08/2025	99%	Minor changes to section 8	John Vardakas (IQU), Bill Avgerinos (ISI/ATH)
V8.0	19/08/2025	99%	Review Fixes	Vrettos Moulos (UPR)
V9.0	25/08/2025	99%	Fixes to section 5.5	Ibrahim Afolabi (FIN), Alabi Rasheed (FIN)
V10	25/08/2025	100%	Final compilation	Vrettos Moulos (UPR)
V11	17/10/2025	100%	Version to submit	Vrettos Moulos (UPR)

Disclaimer

The content of this document reflects only the author's view. Neither the European Commission nor the HaDEA are responsible for any use that may be made of the information it contains.

While the information contained in the documents is believed to be accurate, the authors(s) or any other participant in the AC^3 consortium make no warranty of any kind with regard to this material including, but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Neither the AC^3 consortium nor any of its members, their officers, employees or agents shall be responsible or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein.

Without derogating from the generality of the foregoing neither the AC^3 Consortium nor any of its members, their officers, employees or agents shall be liable for any direct or indirect or consequential loss or damage caused by or arising from any information advice or inaccuracy or omission herein.

Copyright message

© AC^3 Consortium. This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.

Table of Contents

1 Executive Summary ... 15
2 Introduction ... 16
2.1 Scope ... 16
2.2 Target Audience ... 16
2.3 Mapping AC^3 Outputs ... 16
2.4 Deliverable Overview and Report Structure ... 19
3 AC^3 Architecture and WP4 Components ... 20
4 Resource Discovery and Monitoring ... 22
4.1 Scalable data collection and monitoring ... 22
4.1.1 NetScaler Advanced Monitoring in Cloud-Native Environments ... 28
4.2 Scalable monitoring solutions to reduce communications overhead ... 31
4.2.1 Data model ... 33
4.3 Resource discovery and monitoring of the federated resources ... 40
4.3.1 High-level architecture ... 40
4.3.2 Design and implementation ... 42
4.3.3 LMS Exposer: Case of Kubernetes (K3s, K8s, OpenShift) ... 42
4.3.4 Discovery ... 44
5 AI/ML Models for Prediction and Resource Management ... 46
5.1 Explainable AI ... 46
5.1.1 Explainable approaches ... 46
5.1.2 Perturbation-based methods ... 47
5.2 Gradient-based methods ... 48
5.2.1 Saliency Maps ... 48
5.2.2 Integrated Gradients ... 49
5.2.3 Gradient SHAP ... 49
5.2.4 Performance evaluation ... 49
5.3 Area Over the Permutation Curve for Regression ... 49
5.4 Experiments and results ... 50
5.5 Integration of Sustainability-Aware Scheduling Models for Federated CECC ... 56
5.5.1 Introduction ... 56
5.5.2 State of the Art in Sustainability-Aware Scheduling ... 56
5.5.3 Main Framework ... 57
5.5.4 Simulation and Results: Sustainability-Aware Scheduling in Federated CECC ... 61
5.5.5 NSGA-II-Based Multi-Objective Scheduling ... 67
5.6 CPU usage Prediction with Local Global Models ... 71
5.6.1 Global Local Time Series Prediction ... 71
5.6.2 Cloud Edge Continuum Emulator (CEC-EMULATOR) ... 75
6 Decision Enforcement with Reinforcement Learning ... 80
6.1 Knowledge models using Explainable ML to explain resource usage ... 80
6.1.1 Causal latency modelling in the context of CECC ... 80
7 Network Programmability ... 88
7.1 eBPF-based SD-WAN ... 88
7.1.1 State-of-the-Art ... 88
7.1.2 Background ... 89
7.1.3 Overview Architecture ... 90
7.1.4 Low-Latency Intelligent Network eXecution (LINX) ... 91
7.1.5 Evaluation Results ... 94
7.2 Kubernetes Network Operator ... 98

7.2.1 Network Operator Architecture 98
7.2.2 Proactive Network Lifecycle Management 99
7.2.3 Reactive Network Lifecycle Management 103
7.3 Perspectives 117
8 Power management at the network edge using Reinforcement Learning 118
8.1 Background 118
8.2 Related work 118
8.3 MEC Resource and Power Management 118
8.4 Experimental setup & Results 120
8.5 Multi-agent energy optimization using Reinforcement learning 122
8.5.1 Related work 122
8.5.2 Service-aware energy optimization 122
8.5.3 Experimental setup & Results 123
8.6 Enhancing resource optimization algorithm using Generative-AI 124
8.6.1 Related work 124
8.6.2 3.2 System model 125
8.6.3 3.3 Experimental setup & Results 125
9 Conclusions 127
10 References 128

List of Figures

Figure 1 Overall Architecture 20
Figure 2 Interfaces between WP4 components 23
Figure 3 Monitoring Design Structure 27
Figure 4 Monitoring Information Flow 28
Figure 5 Netscaler ingress controller as a sidecar with Netscaler CPX 29
Figure 6 Total number of RX and TX packets processed by the NetScaler CPX interface over time 30
Figure 7 HTTP Request and Response Rates per Second Processed by CPX 30
Figure 8 HTTP Request and Response Byte Rates 31
Figure 9 Monitoring Scalability Architectural Design 32
Figure 10 Monitoring Behavior Analysis 33
Figure 11 High-level resource discovery/exposer Architecture 41
Figure 12 Detailed technical view of the resource exposure framework. 42
Figure 13 Example of JSON format for compute resource metrics, including memory, CPU, and storage utilization. 44
Figure 14 Aggregation and abstraction of resource metrics by Discovery for the LCM 45
Figure 15 The metrics used for the latency prediction model 51
Figure 16 Feature importance across all time steps using the feature ablation method 52
Figure 17 3D Surface of feature importance over time using feature ablation 53
Figure 18 Feature importance across all time steps using Gradient Shap 53
Figure 19 3D Surface of feature importance over time using Gradient Shap 54
Figure 20 Feature importance across all time steps using Integrated Gradient 55
Figure 21 3D Surface of feature importance over time using Integrated Gradient 55
Figure 22 Number of Tasks per Node 64
Figure 23 Total Resource Demand per Node 64
Figure 24 Number of Tasks per Time Slot 65
Figure 25 Total Resource Demand per Time Slot 65
Figure 26 Histogram of Latency Violations 66
Figure 27 Histogram of Reliability Violations 66
Figure 28 Tasks Assigned in Renewable-Friendly Time Slots 67
Figure 29 Energy vs SLA Violations Pareto Front 68
Figure 30 Energy vs Carbon Footprint Pareto Front 69
Figure 31 SLA Violations vs Carbon Footprint Pareto Front 69
Figure 32 3D Pareto Front (Energy, SLA, Carbon) 70
Figure 33 Last Horizon Step Prediction Evaluation 75

Figure 34 Example of work model.json entry 76
Figure 35 Service Mesh for two applications deployed on top of the Emulated Infrastructure 77
Figure 36 Measured Latency for the app-edge application running on top of the emulated infrastructure 78
Figure 37 Collected Metrics From the Emulated Cloud Edge Continuum 78
Figure 38 Comparison between Call Graph and Latency Graph 82
Figure 39 : Overview of the proposed process 82
Figure 40 Graph illustrating the mediator assumption 83
Figure 41 Results provided by the different Causal Discovery algorithms compared to the ground truth graph. 85
Figure 42 Discovered causal graph from observational data 85
Figure 43 Comparison between the predictions of SVR with causal features and XGBoost model with all features 87
Figure 44 HELIOS Overview within AC^3 Architecture 90
Figure 45 LINX Architecture 92
Figure 46 Flow Rules Data Model 93
Figure 47 Data-Path Packet Processing 94
Figure 48 Throughput Obtained on the Downlink and Uplink per Packet Size and Number of Rx Queues 96
Figure 49 Control Signaling Latency 97
Figure 50 PLR per Packet Size and Number of Rx Queues 97
Figure 51 Network Operator Architecture 99
Figure 52 Network Operator Custom resource 100
Figure 53 Skupper Proxy annotation 101
Figure 54 Monitoring 102
Figure 55 Router Connectivity 102
Figure 56 CPU Resource allocation 104
Figure 57 Baseline Network Topology 105
Figure 58 Adaptive Routing Topology 106
Figure 59 Renewable Energy Ratio 106
Figure 60 Link Cost 107
Figure 61 Energy Link Cost 107
Figure 62 Experimental setup 108
Figure 63 Exposed Servers 109
Figure 64 Traffic flow 109
Figure 65 Connection history 110
Figure 66 Flow trace details 110
Figure 67 Skupper Proxy annotation 111

Figure 68 Baseline costs 111
Figure 69 Scheme 1 costs 111
Figure 70 Traffic distribution 112
Figure 71 Router CPU Usage 113
Figure 72 Green based Energy usage 114
Figure 73 Fossil Fuel energy usage 115
Figure 74 Server/Application CPU Usage 116
Figure 75 Total Energy 117
Figure 76. System model of the RL-based power-management system 119
Figure 77 Average delay experienced by hosted services at the MEC 120
Figure 78. Average power consumption of hosted services at the MEC 121
Figure 79. Average power consumption of service I at the MEC 121
Figure 80. Average power consumption of service II at the MEC 121
Figure 81. Average power consumption of service III at the MEC 121
Figure 82. Average power consumption of service IV at the MEC 122
Figure 83. Multi-agent RL based energy optimization 123
Figure 84. Delay performance results 124
Figure 85. Energy performance results 124
Figure 86. Latency goodness 125
Figure 87. Data rate goodness 126
Figure 88. Network resource distribution 126

List of Tables

Table 1: Adherence to AC^3 GA Deliverable & Tasks Descriptions 16

Table 2 Monitoring Architecture Requirements 25

Table 3 Comparison results between Vanilla Prometheus plus Thanos and AC^3 Solution 32

Table 4 Comparison of Exact and Evolutionary Optimization Approaches for CECC Scheduling........ 59

Table 5 Global Local Model Parameters........ 73

Table 7 Patch TST Parameters 74

Table 8 Evaluation Metrics Results........ 74

Table 9 Results provided by the different causal discovery algorithms at a microservice level (m) and an endpoint level (e) for 2 the first step of the causal discovery 84

Table 10 Comparison Results 86

Table 11 CPU Load Function of Packet Rates and Rx Queues 97

Glossary of terms and abbreviations used

Abbreviation / Term	Description
AC^3	Agile and Cognitive Cloud-edge Continuum
ADC	Application Delivery Controller
AI	Artificial Intelligence
API	Application Programming Interface
ADM	Application Delivery Management
CDF	Cumulative Distribution Function
CEC	Cloud Edge Continuum
CECC	Cloud Edge Computing Continuum
CECCM	Cloud Edge Computing Continuum Manager
CNI	Container Network Interface
CPE	Customer Platform Engineering
CPU	Central Processing Unit
CPX	Container Proxy
COE	Customer Observability Exporter
CoT	Class of Traffics
CRD	Customer Resource Descriptor
DC	Data Center
DL	Deep Learning
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q-Network
DMA	Direct Memory Access
DP	Data Plane
DPDK	Data Plane Development Kit
DRL	Deep Reinforcement Learning
DUT	Device Under Test
E2E	End-to-End
eBPF	extended Berkeley Packet Filter
FAR	Forwarding Action Rule
Gen-AI	Generative Artificial Intelligence

GNN	Graph Neural Network
GRE	Generic Routing Encapsulation
gRPC	Google Remote Procedure Calls
HardIRQ	Hardware Interrupt Request
HELIOS	High-Efficiency Layered Infrastructure with eBPF for Optimized SD-WAN
IE	Information Element
IoT	Internet of Things
IP	Internet Protocol
IRQ	Interrupt Request
ISP	Internet Service Provider
KPI	Key Performance Indicator
LAN	Local Area Network
LBR	Load Balancing Rule
LCM	Life Cycle Management
LIME	Local Interpretable Model-agnostic Explanations
LINX	Low-Latency Intelligent Network eXecution
LLM	Large Language Model
LMS	Local Management System
LSTM	Long Short-Term Memory
MCS	Monte-Carlo Simulation
ML	Machine Learning
MPLS	Multi-Protocol Label Switching
NAPI	New API
NBI	Northbound Interface
NIC	Network Interface Controller
OCM	Open Cluster Management
ONOS	Open Network Operating System
PaaS	Platform as a Service
PDI	Packet Detection Information
PDR	Packet Detection Rule
PLR	Packet Loss Rate

QoS	Quality of Service
QoE	Quality of Experience
RL	Reinforcement Learning
Rx	Receiver
SAC	Soft-Actor Critic
SBI	Southbound Interface
SDN	Software-Defined Networking
SD-WAN	Software-Defined Wide Area Network
SHAP	SHapley Additive exPlanations
SKB	Socket Kernel Buffer
SLA	Service-Level Agreement
SLO	Service Level Objectives
SNMP	Simple Network Management Protocol
SoftIRQ	Software Interrupt Request
SR	Segment Routing
SUT	System Under Test
TCP	Transmission Control Protocol
TFT	Temporal Fusion Transformer
TLS	Transport Layer Security
ToS	Type of Service
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNF	Voice over Internet Protocol
VPN	Virtual Private Network
VR	Virtual Reality
VRF	Virtual Routing and Forwarding
WAN	Wide Area Network
WPEC	Wireless Powered Edge Computing
XAI	Explainable AI

XDP	eXpress Data Path
YAML	Yet Another Markup Language

1 Executive Summary

Transitioning to the CECC model requires the development of robust frameworks and advanced resource management capabilities that can continuously optimize resources across both IT and OT environments. This need is the driving force behind the AC^3 project, which seeks to establish an innovative CECC management framework designed to enhance scalability, agility, and operational efficiency. At the heart of the AC^3 project lies the CECCM, a fundamental component facilitating the advancement of application LCM and configuration through self-management and self-configuration, all while satisfying the SLA. To realize this objective, AC^3 considers recent breakthroughs in AI and ML, particularly eXplainable AI (XAI), to deliver an efficient LCM system for CECC resources, ensuring minimal latency and SLA compliance. As demonstrated in D4.1, WP4 is responsible for developing AI/ML models for resource management (T4.2) and enforcing LCM decisions related to resource allocation (T4.3), both of which are driven by data collected through resource discovery and monitoring across the federated infrastructures (T4.1). Additionally, WP4 includes the development of Network Programmability components to enable dynamic and flexible traffic management for microservices within the CECC infrastructure. This is achieved while maintaining application SLAs through dynamic updates of routes and resources (T4.4). This deliverable outlines our efforts regarding WP4, focusing on resource management across CECC and algorithmic design of the components. To set the context, we provide the complete architecture design of the monitoring system, highlighting the importance of collecting relevant metrics to build the AI-based LCM. The primary areas of innovation for the proposed mechanisms begin with a unified monitoring data model, emphasizing the need for a comprehensive schema that captures dynamic resource states (CPU, memory, storage), classifies resources by type (edge, cloud), tracks energy sources, and determines geographical locations. Then, we architect a mechanism for collecting monitored data. Furthermore, we describe the interconnection mechanism of monitoring components between clusters and the AC^3 platform, focusing on the way we can cater to the data and how we can expose it to the services (T4.1). Upon the conclusion of data collection, efficient AI approaches for resource management (T4.2) have been developed and implemented, including ML/DL models that interpret dynamic workload information and predict resource needs in the heterogeneous CECC environment. Furthermore, we extend our work in Deliverable 4.1 by introducing methods to validate explainability techniques for ML/DL models, specifically in the context of resource usage prediction. In this situation, we designed a framework to evaluate the consistency and reliability of various explainability approaches operating in conjunction with ML/DL approaches to ensure the consistency and reliability of the explanation results. This way, the system would be able to take proactive, context-aware decisions that improve operational efficiency but also align with SLA constraints. Thus, corrective measures (T4.3) can be implemented, for example, scaling, *on-the-fly*, either CPU, memory, or both, if we are interested in vertical autoscaling or the number of pods in the context of horizontal autoscaling. The autoscaling will take place whenever performance degradation or a workload spike is detected. LCM-engine-led correction actions will ensure that these scaling procedures remain completely autonomous, transparent to the human end-user, and are conducted in an optimal manner regarding resource utilization. Along this line, we also developed knowledge models allowing us to infer the topology of microservices and determine the metrics responsible for high latencies leading to SLA non-compliance. Furthermore, we provided final deliverables and results on dynamic and flexible traffic management for Network Programmability (T4.4), particularly targeting the development of real-time route update mechanisms along with microservice-aware resource allocation. This would enable CECC applications to adapt dynamically to changing network and workload conditions with SLA commitment intact, whilst maximizing system responsiveness and efficiency.

2 Introduction

2.1 Scope

This document represents a public deliverable from AC³'s WP4, offering insights into our progress, notable achievements, and the innovative AI-driven algorithms developed for resource management, with a strong focus on explainability. This deliverable presents the validation results using specific experiments and simulation mechanisms that were designed to highlight the advantages of the approach that the project follows. Deliverable D4.2 concludes work on the design for intelligent green resource management in the Cloud–Edge Computing Continuum (CECC) and validates the mechanisms and tools developed therefor. Building upon the initial concepts and architectural foundations provided in Deliverable D4.1, this deliverable serves to extend and concretize in design, implementation, and experimental validation the work started in the former. While D4.1 set some theoretical ground and baseline models, D4.2 focuses on the full-fledged integration of the mechanisms and tools based on advanced AI and ML techniques for sustainable, autonomous, and SLA-aware management of resources federated across CECC.

More specifically, D4.2 introduces further novel contributions, mainly in the following four areas:

[Task 4.1] Develop scalable resource discovery and monitoring mechanisms consisting of a unified data model capturing heterogeneous resource states, energy profiles, and geographical information in order to achieve an observability mechanism usable against a diverse range of infrastructures.

[Task 4.2] Design and implement XAI/ML models for infrastructure-use prediction, further extending them to sustainability-aware scheduling and to engage global and local models for better forecasting.

[Task 4.3] Design green-oriented decision enforcement mechanisms based on multi-objective optimization and Reinforcement Learning, including cooperative multi-agent methods and generative AI methods for reward shaping, aimed at reducing latency, energy cost, and resource fragmentation.

Augment network programmability with innovative SD-WAN and Kubernetes-oriented orchestration techniques that allow dynamic response to routing and traffic management based on application and infrastructure needs (Task 4.4).

Such advancements are validated through a set of simulations and real-world datasets that present quantifiable improvements in energy efficiency, delay reduction, and interoperability across federated environments.

2.2 Target Audience

This deliverable is tailored for stakeholders involved in AI for resource management across hybrid, multi-cloud, and cloud-edge environments. It details the implementation of innovative AI techniques for predicting infrastructure availability and managing resource usage effectively.

2.3 Mapping AC³ Outputs

The purpose of this section is to map AC³ Grant Agreement commitments, both within the formal Deliverable and Task description, against the project's respective outputs and work performed.

Table 1: Adherence to AC³ GA Deliverable & Tasks Descriptions

AC³ GA Component Title	AC³ GA Component Outline	Respective Document Chapter(s)	Justification
DELIVERABLE			
D4.2. Report on the resource management procedures over CECC - Final			

TASKS			
Task 4.1 Resource discovery and monitoring of the federated resources	Define the necessary components and mechanisms to monitor and discover the federated CECC infrastructure resources, including the far edge. Determine the unified data model that needs to be exchanged between the different infrastructure providers. Two models will be considered, one for computing and one for networking. The data model for computing will be extended to include the available computing resources and information on the energy model used by the DC (green or brown) and the best periods suited for energy. Define scalable data collection and monitoring for the federated resources that are in use by the CECCM.	Section 4	The main building blocks present in GA and pertaining to Task 4.1 have been addressed in Section 4.
Task 4.2 AI/ML models for resource management	Develop ML models in order to predict infrastructure usage, energy usage, and resource availability, including the far edge. ML algorithms will focusing mainly on using explainable models. State-of-the-art explainable models will be tested and used.	Section 5	The main building blocks present in the GA and pertaining to Task 4.2 have been addressed in Section 5.

Task 4.3 Green-oriented LCM decisions for resource management	Devise the necessary algorithms that take as inputs the resource prediction models to guarantee the application's SLA. Devise the necessary algorithms that take as inputs the resource prediction to optimize resource usage for the federated CECC as well as the energy consumption. The envisioned solutions could be based on multi-objective optimization theory or ML- based decision-making solutions, such as Reinforcement Learning (RL). Devise algorithms that decide when to migrate microservice in order to optimize energy consumption. Build a knowledge mechanism in order to use the explainable models to evaluate the impacts of the considered decisions and derive their efficiency.	Section 6	The main building blocks present in the GA and pertaining to Task 4.3 have been addressed in Section 6.
Task 4.4 Networking programmability of CECC	Embrace two technologies, SD-WAN and CFN, for WAN and edge, respectively. Define the necessary mechanisms to allow the programmability of both networks by selecting or extending existing SDN controllers. Update the CECCM northbound API to allow the application developer	Section 7	The main building blocks present in the GA and pertaining to Task 4.4 have been addressed in Section 7.

	to request an update of the network connectivity. Improve the CECCM to allow for conflict-resolution solutions that solve contradictory requests or requests that may impact other deployed applications.		

2.4 Deliverable Overview and Report Structure

In this section, we describe the D4.2 structure, outlining the respective sections and their content.

Section 3 presents the AC^3 architecture and WP4 components with a special focus on the four tasks.

Section 4 presents the monitoring mechanism, the architecture design that it has internally, as well as the interconnections that exist.

Section 5 focuses on the AI/ML models for infrastructure usage prediction and the results that were extracted through the validation that was made. Special consideration has been given to XAI.

Section 6 provides the decision enforcement mechanism and the experiments that were made to stress the algorithm.

Section 7 is dedicated to Network Programmability.

Section 8 provides the AC3 solution for power management at the MEC servers situated at the network edge, focusing on the relation between efficient resource management and power consumption

Section 9 concludes the deliverable D4.2 and outlines the outcomes of the tests that were performed.

3 AC³ Architecture and WP4 Components

During the second phase of the design of the AC³ architecture (Figure 1), large emphasis was on refining the management plane, the central conduit for monitoring, profiling, and life-cycling applications and resources across the federated Cloud Edge Continuum (CEC) infrastructure. Extending from the initial architectural considerations, this step considers how information is dynamically collected, structured, and utilized in AI-informed decisions at a high level of automation and adaptiveness. At the center of this refinement lies the smooth integration of the monitoring components with application profiling, resource management, and AI-based orchestration mechanisms.

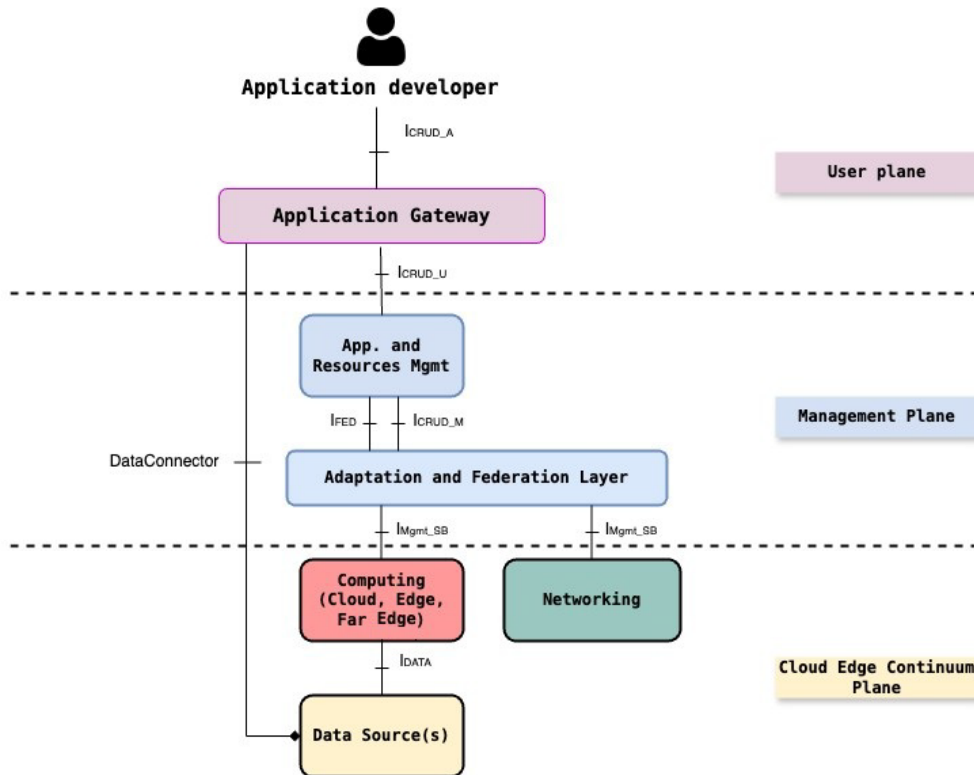


Figure 1 Overall Architecture

The management plane consists of interrelated layers responsible for monitoring application behavior and for resource availability across heterogeneous environments. It is the intake of telemetry and performance information. It is the operational "information pump" for the system. Without clean, timely, and relevant contextual data, lifecycle management (LCM) processes risk becoming ineffective when decisions have to be made on AI timeframes rather than human cycles of intervention. The value of automation in such an architecture lies not just in the mere existence of intelligent algorithms but equally in the quality and granularity of the data that feeds these algorithms.

The Application and Resource Management module serves as the central element, coordinating both deployment and runtime adaptation of microservices by means of advanced machine-learning models to maintain Service-Level Agreement (SLA) compliance while optimizing resource use, energy, and performance guarantees. The integration of AI into this module enables continuous creation of application and resource profiles through learning and adaptation. These application and resource profiles are used by the AI-based LCM engine to derive the best decisions with Reinforcement Learning techniques, among others, for placing, scaling, migrating, and network configuration of applications.

The Monitoring subsystem gathers metrics from all participating CEC infrastructures, which include measuring computing load, link status, resource availability, and energy profile, as well as geolocation parameters. These data are then harmonized and normalized so that they can feed AI models and expose KPI indicators pertinent to the developers of the Application Gateway.

The AI-based LCM then acts as the brain, marrying both app and infrastructure perspectives in its decision-making. Decisions are then implemented through the Decision Enforcement layer, which ensures uniform application of actions such as horizontal and vertical scaling, or service migration, across the infrastructure using a common data model.

4 Resource Discovery and Monitoring

Efficient and intelligent management of resources in federated cloud-edge environments demands accurate, timely, and context-driven observability. In the AC^3 architectural design, observability is provisioned through two dichotomous components: the Monitoring subsystem and the Resource Exposure framework.

The Monitoring subsystem primarily handles the collection, aggregation, and exposure of time-series telemetry data from every stratum of infrastructure, such as Central Processing Unit (CPU), memory, and network utilization, in far-edge, edge, and cloud nodes. It is oriented toward both real-time responsiveness and historical analysis. Being a core support for AI-driven lifecycle management, this component delivers dynamic metrics and key performance indicators to predictive models and decision engines aligned with SLA and sustainability targets.

On the other hand, the Resource Exposure framework undertakes structural discovery rather than continuous metric collection. It offers a snapshot abstraction of the current capabilities and state of infrastructure resources—what is there, where it is located, and the constraints applied to it. Such information includes all the types of nodes (edge, cloud, GPU-enabled, energy-aware), their geolocation, and their current states. Then the system can reason with the availability of infrastructure, perform matchmaking with application requirements, and spot deployment targets intersecting federated domains.

The distinction between these two components is thus temporal and functional:

- Monitoring addresses continuous, high-frequency telemetry and behavioral insights, optimized for AI models and reactive/proactive orchestration.
- Resource Exposure addresses event-driven, state-based infrastructure discovery, enabling the AC^3 platform to maintain an accurate registry of available resources across diverse environments.

Together, these components enable AC^3 to deliver a unified, extensible, and intelligent monitoring and discovery infrastructure that enables the AI-based LCM to make informed, context-aware decisions at scale.

4.1 Scalable data collection and monitoring

The CECCM depends heavily on monitoring as a fundamental element, specifically within its AI-based LCM. The AI-based LCM needs monitoring data from infrastructure components and applications to power multiple of its operational elements. First, the AI-based application profile uses the collected information on the applications, such as computing resources (CPU, memory, storage) consumption, energy consumed, traffic handled during a period, etc. Using this information, it is possible to build application profiles that will be employed by the AI-based LCM for application management and life-cycle optimization, such as scale down or up resources, or do migration. Second, the AI-based CEC resource profile uses monitoring information to update the profile of the CEC resource used by the CECCM. As CECCM uses resources from the federation, a profile describing in real-time the CEC resource status and profile is vital. The AI-based CEC resource profile will build on the monitoring information using the status of the CEC resource, that is, available computing resources (CPU, memory, storage), type of resource (edge, cloud, networks, far edge), type of energy used (green or brown), the covered locations, etc. All this information will also be used to predict the evolution of CEC resource usage. The AI-based LCM will also use the CEC resources for LCM decisions, such as the initial placement of applications, migration of applications for energy optimization, etc. Both profiles are combined for better resource optimization and SLA guarantee for applications. It should be noted that the monitoring components are activated once a CEC resource from the federation is selected by the CECCM (via the broker). The monitoring component is configured to consume the NBI as exposed by the LMS of the CEC resource.

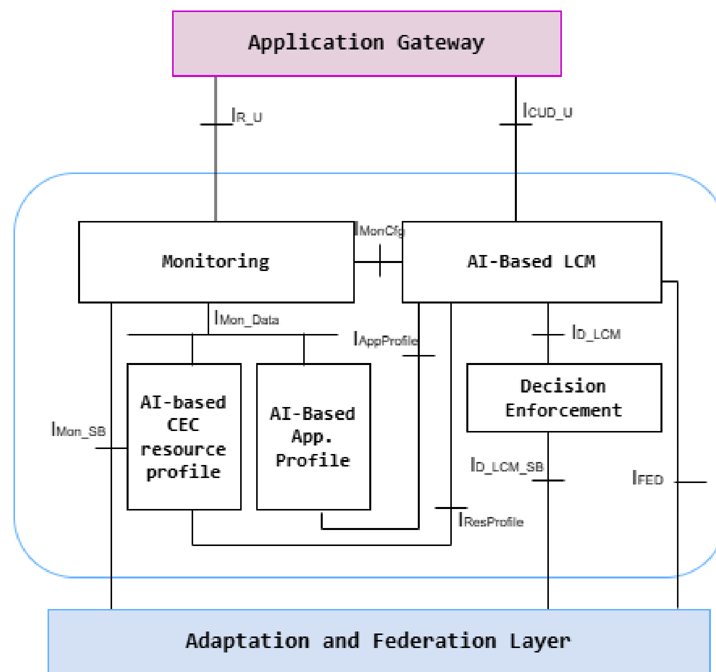


Figure 2 Interfaces between WP4 components

The monitoring architecture is designed to be generic, modular, and adaptive and hence very conducive toward the diverse needs of scalable microservice-based applications across the cloud-edge continuum. Such systems, especially those intending to have operational considerations at the edge or far edge, pose severe constraints on bandwidth, latency, and energy efficiency. In such events, monitoring cannot be considered a move side or an extra process; it has to be a core system process that is optimized for **observability with minimal overhead**. It is also true that the more granular or detailed the telemetry is for introspection purposes, the more detrimental its transmission can be in constrained environments, while, on the other hand, it becomes untenable for the system to turn invisible if the data is too sparse, thus undermining the control automation mechanisms. Hence, one critical AC^3 design tenet is the adaptive balance: that is, the monitoring system should produce **sufficiently actionable insights while avoiding the transmission of unneeded data**.

The shift toward AI-assisted lifecycle management was also influencing the monitoring strategy. Unlike conventional systems that might utilize generic or reusable models, AC^3 encourages the development of **fine-tuned, application-specific models** that faithfully represent the actual operational behavior, workload dynamics, and architectural idiosyncrasies of each deployment. This specialization is instrumental in ensuring accurate predictions, relevant anomaly detection, and sound decision support. Therefore, these environments must allow a **controlled approach to experimentation and data isolation**, where the profiling of system behavior may take place without external interference. The infrastructure for monitoring must take into consideration the performance indicators that have significance with regard to temporal depth and granularity of structure so that the unique specifications of each scenario can be modeled.

The factor of **temporal context** acquires paramount importance in system evaluation and decision-making. In fact, oftentimes, temporal correlations such as aforementioned bursts, diurnal cycles, and usage anomalies are more relevant than **time-independent metrics**. The very ability to perform time-aligned monitoring grants the system the properties of performing not only reactive adaptations but also anticipating changes based on time-learned behaviors. Maintaining such **historical traces in system behavior** allows an AI component to distinguish short-lived deviations from long-range trends. This consideration decided the monitoring architecture to support **data acquisition and storage with long-term persistence under temporal indexing**, ensuring that all information necessary for the learning and optimization processes is held.

Taking into account the variability in deployment environments, the monitoring subsystem also must ensure **extensibility and interoperability**. AC^3 caters to all sorts of applications and heterogeneous infrastructure, which are often underpinned by different monitoring stacks and architectural assumptions. To cope with such diversity, monitoring components need to abide by a **common schema-based model** that performs a decoupling of the monitoring logic from implementation details. Provided that the schema is followed, the actual monitoring tools may differ. This design enables integration of existing solutions while enforcing **uniformity for the interpretation** and subsequent consumption of data across the platform. To ensure compatibility, to provide avenues for evolution, and not to hamper the greater system, **schema validation together with versioning** is introduced.

Querying and processing of monitoring data evidently needs to be flexible and expressive, too. The system must allow complex **queries beyond the mere retrieval of metrics**; these query specifications describe temporal relations, thresholds, and aggregation logics useful for SLA enforcement, KPI validation, and orchestration feedback. These queries are of high importance both for application developers and consist of some of the major building blocks concerned with system functionalities, such as Resource Profiler, Application Profiler, and Lifecycle Management engine. Hence, the architecture provides a **query language interface** that can address all these requirements while focusing on the standards of expressiveness.

Security and data governance concerns may impact the architecture of the monitoring layer. In edge computing settings, monitoring data can bear administrative data such as geographical coordinates, topological layouts, or usage patterns that can harm privacy if exposed. AC^3 can address this by incorporating external mechanisms **for filtering and anonymizing data at the source**, so that developers and administrators may establish policies on what is being collected, stored, and shared. These settings are **configurable** and may be fine-tuned depending on the application context, legal stipulations, or organizational policy.

Regarding integration with the system, the literature records the existence of two main approaches for embedding monitoring in those microservice ecosystems. The first takes advantage of so-called sidecars in the form of containers, usually deployed in tandem with each microservice instance, collecting telemetry directly from the runtime for forwarding either to a centralized monitoring service or a dedicated processing component. This model of access indeed allows enormous visibility and macro granularity of information because it exposes metrics at the level of individual service instances. As such, it exceedingly helps with debugging or ad hoc scenarios where having a complete observability perspective is essential.

A second approach delegates monitoring and handling to the application descriptor, which prescribes how and when services should expose their pertinent metrics. Here, the microservice itself is not burdened with additional logic or infrastructure. Rather, the application is described in a way that allows the orchestration environment to centrally infer or coordinate the monitoring strategy. With AC^3, such an approach enjoys little runtime overhead and is therefore well suited for efficiency- and predictability-oriented environments that highly value privacy.

From there, this ensures that application developers may customize observability according to application logic without contaminating it with monitoring internals, while keeping the option to expose only what is considered relevant and safe. Because AC^3 does not impose a single execution model but rather supports interoperability through schema conformity, one can potentially integrate different monitoring systems whose outputs conform to the defined communication and data models. Such flexibility permits applications to be developed in any programming language, using any orchestration platform or observability tool, yet remain fully compatible with the AC^3 monitoring infrastructure. While dedicated translation or wrapper components can provide that adaptability for different monitoring sources, the applications themselves remain intact.

Being composable and open, AC^3 can thus scale across use cases and operational domains, and adapt to different monitoring needs (as depicted in Figure 2) without compromising the core design principles. Fine-grained querying, schema-based interoperability, temporal context, data minimization, and extensible integration are techniques that place monitoring not as an ancillary consideration but as a core capability of intelligent and trustworthy resource management in complex distributed environments.

Table 2 Monitoring Architecture Requirements

Requirement	Description
Adaptive Observability	Monitoring must balance actionable insights with minimal data transmission to prevent overload in constrained environments.
Application-specific Modeling	Monitoring must support profiling in isolated environments to allow the development of models tailored to specific application behaviors.
Temporal Context Support	Monitoring must capture and store time-aligned data to distinguish between short-lived anomalies and long-term trends.
Persistent Storage	Long-term retention and temporal indexing of monitoring data are essential to support learning and optimization processes.
Extensibility and Interoperability	Monitoring must support diverse tools and architectures through schema-based decoupling of logic and implementation.
Schema Conformity and Versioning	Monitoring data must conform to a common schema that includes validation and supports version control.
Flexible Query Interface	Support is required for expressive queries, including temporal logic, thresholds, and aggregation, for SLA/KPI validation.
Security and Data Governance	Mechanisms for source-side filtering and anonymization must be available to meet privacy and policy constraints.
Descriptor-based Integration	Application descriptors can centrally define monitoring strategies, reducing runtime overhead and supporting modular observability.
Customizable Observability Policies	Developers must be able to configure which monitoring data is exposed, balancing insight with security needs.
Wrapper-based Compatibility	Support for adapter components to translate third-party monitoring tools into the AC3-compliant schema.

The setting for traditional monitoring systems is suitable for static or cloud-centric cases. However, they face great challenges in a federated, multi-domain, and resource-constrained setting such as CECC.

Several open-source and commercial tools exist for monitoring the infrastructures and applications---Nagios[1], Zabbix[2], DataDog[3], Elastic Stack[4]---that are, however, usually too tightly bound to particular deployment models, do not provide flexible temporal querying, or are simply not built for consideration in distributed edge environments.

[1] https://www.nagios.org/
[2] https://www.zabbix.com/
[3] https://www.datadoghq.com/
[4] https://www.elastic.co/elastic-stack

Prometheus[5], while perhaps the most widespread and therefore increasingly cloud-native monitoring approach, is an open-source monitoring and alerting toolkit designed with reliability in mind, storing time-series data and collecting metrics in a pull manner. It offers a multidimensional data model and a powerful querying language (PromQL), with native support for the service discovery mechanism used in Kubernetes-based systems. Yet, Prometheus is naturally restricted and lacks a truly horizontal scaling approach and long-term metric storage, thus de facto rendering it inappropriate for large-scale federated environments insofar as it is left unchanged.

To tackle these issues, Thanos[6] was brought in as an extension for Prometheus. It provided global view aggregation, long-term data retention via object storage, and high availability by federating multiple Prometheus instances. It introduced components like the Receiver, the Store Gateway, the Query Frontend, and the Compactor that together set up a horizontally scalable, multi-tenant monitoring stack. Thanos turned Prometheus into a fault-tolerant and federated observability backend, suitable for hybrid and multi-cluster deployments.

Still, in vanilla Prometheus plus Thanos deployments, implementations can witness very high inter-component communication overhead when deployed at scale, especially in edge-to-cloud environments where network constraints and dynamic service mobility rule. These issues call for a more optimized context-aware architecture that can choose the domain of telemetry data and target it efficiently whenever needed by the system, so as not to overload the network unnecessarily with redundant metric propagation.

In AC^3's monitoring system, Prometheus is responsible for scraping metrics from various targets, such as application endpoints and infrastructure exporters. Prometheus is configured to remotely write the collected metrics directly to Thanos Receiver, which acts as the ingestion gateway for the Thanos ecosystem. The Receiver accepts incoming data and writes it in a block format compatible with Thanos's long-term storage model. These blocks are then uploaded to MinIO[7], which serves as the object storage backend. Thanos components, such as the Store Gateway and Query Frontend, interact with MinIO to retrieve and aggregate historical data, enabling scalable, long-term metric retention and global querying. This design structure (as shown in Figure 3) simplifies deployment by removing the need for sidecars and offers a more centralized and scalable ingestion approach. On top of Thanos (or besides it) lies a custom service that is responsible for delivering the metrics and insights collected from the various applications to all AC^3 components that may need them, such as the LCM and the KPI collection exposure.

[5] https://prometheus.io/
[6] https://thanos.io/
[7] https://min.io/

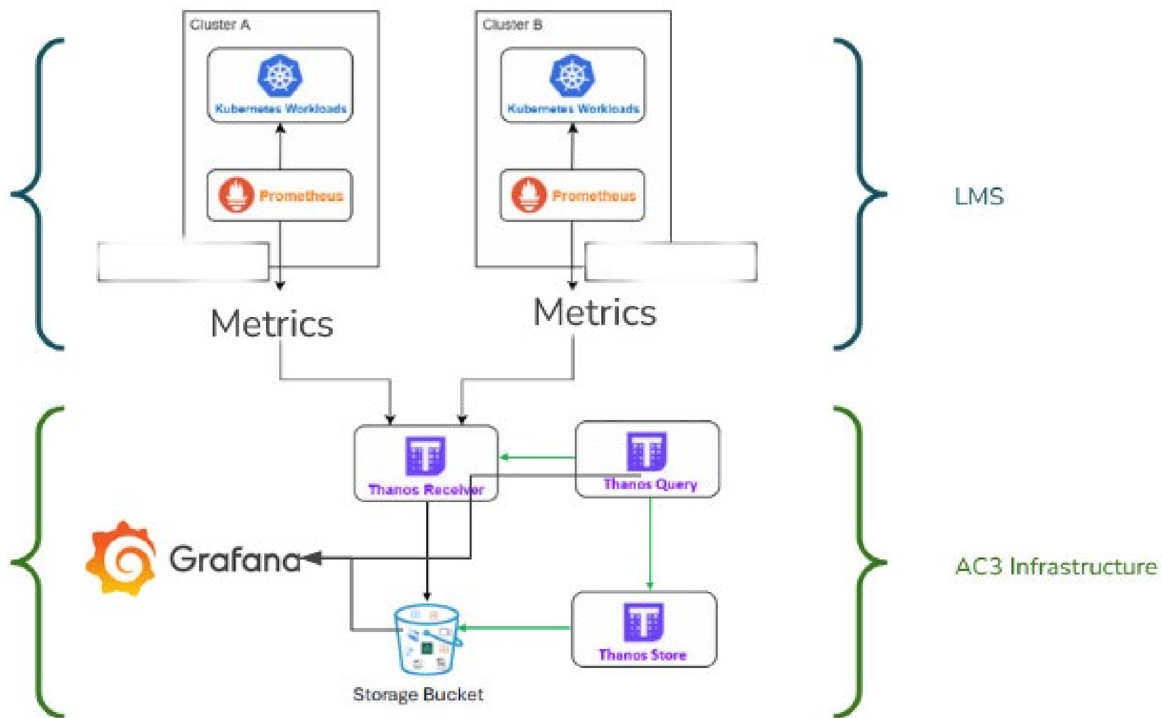


Figure 3 Monitoring Design Structure

Components that comprise the monitoring system:

Metrics collection:

Prometheus is ideal for this monitoring system due to its lightweight, modular architecture, powerful data model, and native support for service discovery and time-series metric collection. Designed for reliability and performance, Prometheus excels in dynamic, cloud-native environments by efficiently scraping metrics from targets without requiring external dependencies. Its pull-based model ensures visibility into system health even during partial outages, and its expressive PromQL query language enables complex analysis and alerting logic. When integrated with Thanos and object storage like MinIO, Prometheus becomes part of a scalable and highly available observability stack, supporting both short-term operational monitoring and long-term data retention. This makes Prometheus a robust and flexible foundation for modern infrastructure and application observability.

Metrics aggregation:

Thanos plays a critical role in this architecture by extending Prometheus into a highly available, long-term, and horizontally scalable monitoring system. While Prometheus alone is limited in storage duration and scalability, Thanos overcomes these limitations by enabling central ingestion through the Thanos Receiver and offloading data to object storage like MinIO. It allows seamless querying of both real-time and historical metrics through its Query component, deduplicating data from multiple Prometheus instances and supporting global views across clusters. Thanos also supports down-sampling and compaction, which improve query performance over large time ranges. By integrating Thanos, the architecture gains fault tolerance, global observability, and efficient long-term retention, making it suitable for enterprise-grade monitoring at scale.

Storage:

MinIO object storage offers a high-performance solution designed for modern, cloud-native environments. Its primary advantages include exceptional scalability, allowing seamless expansion from a single node to a distributed cluster, and superior performance optimized for large-scale workloads. MinIO's lightweight, container-friendly architecture makes it ideal for edge, on-premises, and hybrid cloud deployments. Additionally, its open-source model provides transparency and flexibility, while maintaining enterprise-grade features like identity and access management, object locking, versioning, and encryption—all of which contribute to a secure and resilient storage solution tailored for unstructured data.

Metrics routing:

A custom routing service is deployed alongside Thanos to enhance interoperability and data distribution across the observability ecosystem. This service consumes incoming metrics—either from the Thanos Receiver or directly from Prometheus remote write streams—and intelligently routes them to other platforms based on predefined rules or dynamic metadata. By acting as a protocol-aware, pluggable middleware, the custom service decouples data producers from consumers, allowing for flexible metric enrichment, format transformation, and access control. This ensures that critical telemetry data can be reused across all other AC^3 components

The complete monitoring information flow, integrating all the described components, is depicted in Figure 4.

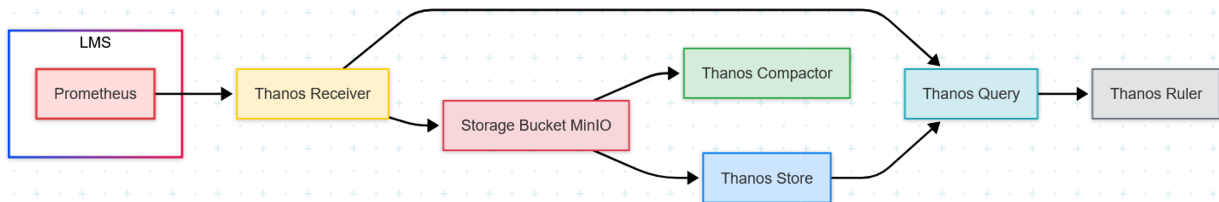


Figure 4 Monitoring Information Flow

4.1.1 NetScaler Advanced Monitoring in Cloud-Native Environments

NetScaler, as an application delivery and security platform, provides load balancing, secure remote access, traffic optimization, and protection for applications and APIs across on-premises, cloud, and hybrid environments. Ingress and traffic management capabilities can be utilized to the fullest, thereby providing great observability within the AC^3 monitoring architecture. With native Prometheus integration, NetScaler CPX delivers substantial network-layer and application-layer metrics that could flow in the AC^3 monitoring stack, providing deeper insight and intelligent management decisions for resource allocation on top of the CECC.

Modern cloud-native platforms, such as OpenShift (Red Hat's Kubernetes distribution), demand robust, scalable, and observable ingress solutions. The NetScaler Ingress Controller (NSIC)[8], particularly when deployed with CPX (the NetScaler containerized data plane), provides advanced L4-L7 ingress, load balancing, and application delivery capabilities, tightly integrated with Kubernetes-native workflows. As discussed in D4.1, the ability to programmatically manage deployed applications, expose metrics, and support automated monitoring is critical. This integration needs to be operator-driven for lifecycle automation, and observability should be seamless, ideally using de facto standards like Prometheus for metrics and Grafana for visualization. This section details the technical integration of NSIC with CPX on an AC^3 OpenShift cluster, leveraging the NetScaler Operator, and demonstrates direct metrics export to Prometheus, with Grafana dashboards as evidence of successful

[8] https://docs.netscaler.com/en-us/netscaler-k8s-ingress-controller/

integration. While application-level use case integration is ongoing, this setup forms the foundation for future, use-case-driven, AI-enabled resource management.

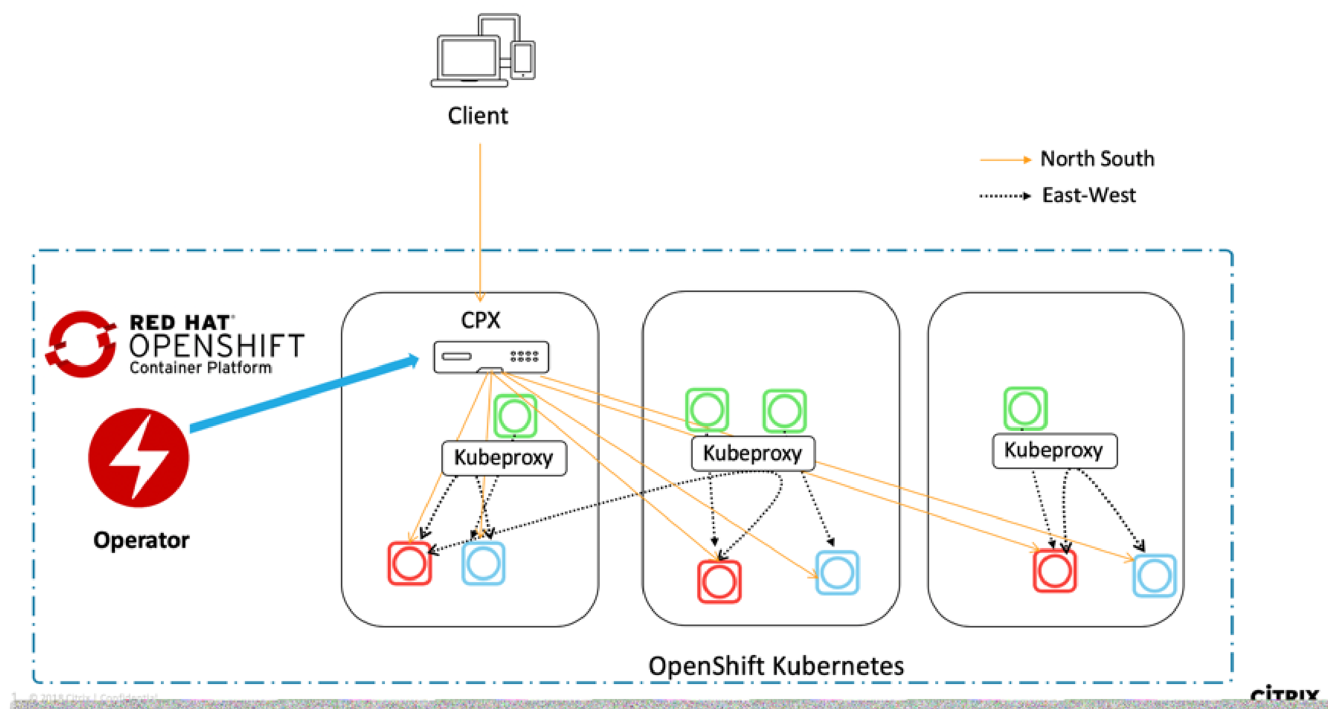


Figure 5 Netscaler ingress controller as a sidecar with Netscaler CPX

As shown in Figure 5, the deployment architecture consists of several key components. First, an OpenShift Cluster is hosted on Arsys cloud infrastructure, providing the managed Kubernetes control plane and worker nodes. In addition, the NetScaler Operator, which is a Kubernetes Operator installed in the cluster, automates the lifecycle of NSIC and CPX instances. NetScaler CPX serves as the data plane, running as a Kubernetes pod and providing ingress, load balancing, and L4-L7 application delivery for exposed services. Moreover, the NSIC is deployed as a controller pod that monitors Kubernetes resources, such as Ingress, Service, and Endpoints, and programs the CPX accordingly. Prometheus acts as the core metrics collector for the AC^3 monitoring stack within the same OpenShift environment. Additionally, Grafana is used for the visualization of Prometheus metrics. Regarding integration points, the NetScaler Operator enables declarative management of CPX and NSIC deployments, with all configuration and upgrades handled via Kubernetes CRDs. In particular, NetScaler configuration management is enhanced through a comprehensive collection of 14 CRDs that enable declarative policy management for advanced features, including Web Application Firewall, bot protection, authentication, traffic routing, and API rate limiting. These CRDs support ingress class-based multi-tenancy and service binding, enabling Infrastructure as Code workflows with version control and automated policy deployment[9].

Furthermore, NSIC with CPX supports direct export of rich metrics, including interface and HTTP statistics, to Prometheus as described in the NetScaler documentation. Prometheus scrapes metrics endpoints exposed by CPX, and Grafana dashboards are configured to visualize both L2/L3 and L7 metrics.

NetScaler CPX exposes a comprehensive set of metrics relevant for cloud-native and enterprise workloads. These include network interface metrics, such as received packets (RX), transmitted packets (TX), error counts, and byte counters. Additionally, HTTP metrics are provided, including HTTP requests per second, HTTP responses per second, request and response bytes, and status code counters. Transmission Control Protocol (TCP) and SSL metrics, while not shown in the current results, are available for more advanced deployments. Furthermore,

[9] https://github.com/netscaler/netscaler-k8s-ingress-controller/tree/master/crd

application health indicators are exposed, including backend health and server up or down state. The integration leverages NetScaler's native Prometheus exporter capability, which is supported in CPX 13.1 and newer releases. To enable metrics export, when deploying NSIC with CPX via the NetScaler Operator, a configuration flag is set to activate Prometheus metrics. Following this, a Kubernetes Service is created to expose the CPX metrics endpoint within the cluster. The Prometheus instance is then configured, either through static configuration or service discovery, to scrape the CPX endpoint at regular intervals. This direct export approach is preferred over legacy methods, such as using a sidecar metrics exporter, because it reduces operational complexity and enables richer, lower-latency metrics. Within the monitoring stack, the Prometheus Operator automates the management of Prometheus and ServiceMonitor CRDs. Additionally, custom Grafana dashboards are configured to visualize metrics relevant to both the interface and application layers. All necessary metrics are being collected and are available for any downstream analytics or AI/ML pipeline. Although full integration with AC^3's unified monitoring model is pending use-case alignment, the current configuration demonstrates how NetScaler's metrics map to typical fields in a cloud-edge resource observability schema.

Figure 6 Total number of RX and TX packets processed by the NetScaler CPX interface over time

The Grafana dashboards shown demonstrate the success of the NetScaler integration in the AC^3 monitoring platform. For this purpose, an Apache HTTP server was deployed, serving as a test workload for ingress and monitoring. As illustrated in Figure 6, the panel shows the cumulative count of RX and TX packets on the NetScaler CPX interface over time. Both RX (green) and TX (red) counters increase steadily during periods of HTTP activity, then plateau when traffic stops. This confirms that CPX is actively proxying traffic for the Apache example app, and metrics are being accurately collected and exported. The continuous increase and stabilization of counters correlate with HTTP request/response activity observed in the following panels.

Figure 7 HTTP Request and Response Rates per Second Processed by CPX

The panel presented in Figure 7 displays the rate of HTTP requests and responses per second processed by CPX. When the Apache app is accessed, the request and response rates rise and stabilize at ~2/sec, then fall to zero when idle. Multiple activity periods are visible, matching periods of increased packet counts in the interface panel. Demonstrates that HTTP-layer metrics are accurately tracked and aligned with user/application activity.

Figure 8 HTTP Request and Response Byte Rates

Finally, the panel in Figure 8 tracks the rate of bytes processed for HTTP requests and responses. The spikes and sustained plateaus in request and response byte rates directly correspond to periods of HTTP activity. The drop to zero at the end of each burst indicates that the metrics are reset or that the application is idle.

The integration of NetScaler Ingress Controller with CPX in AC³'s OpenShift environment, managed by the NetScaler Operator and exporting metrics directly to Prometheus, has been successfully demonstrated. Grafana dashboards confirm the correct collection and visualization of both interface and application-level metrics. While integration with advanced AC³ use cases is ongoing, this setup provides a robust, operator-driven, cloud-native ingress and observability foundation for future work in federated resource management and AI-driven automation.

4.2 Scalable monitoring solutions to reduce communications overhead

In a cloud solution where multiple applications are running, the volume of monitoring metrics can easily pile up to hundreds of TBs/sec. Given that, it was paramount for us to reduce the communication overhead between the central monitoring system and the applications. To achieve that, we opted for an architecture where every local Prometheus would send metrics to the central monitoring only when they were available (push mechanics). Prometheus knows the central monitoring system through a discovery service, which serves twofold: we avoid unnecessary calls from Thanos (central) to Prometheus (local) in order to probe for new metrics, and as a bonus, Thanos doesn't need to know where and when a new LMS is being deployed.

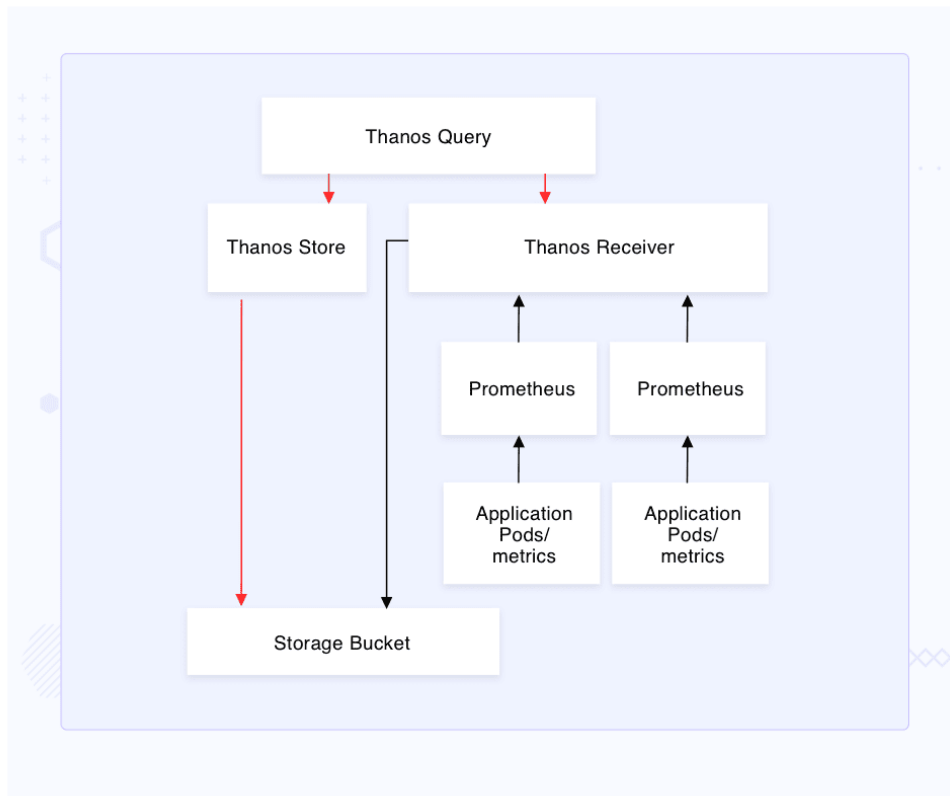


Figure 9 Monitoring Scalability Architectural Design

Moreover, we reduced the communication overhead between the monitoring component and all other components that may require data from it (LCM, front-end, application profiler). We achieved that by creating a service where each component designates through a single REST API the metrics it needs, and from that point forward, without further communication with the component (a comparison result can be shown in Table 3). This has the added benefit of resilience, since the queue is persistent and can replay messages that a component might have lost, due to network or application failure. The service also exposes a REST API from which all AC^3 components can obtain metrics. This API can serve both live and historical monitoring data. Having the API in front of the monitoring system facilitates the interoperability between monitoring components. At the moment, we are using the Prometheus stack, but we could replace it with anything without impact on the other AC^3 components.

Table 3 Comparison results between Vanilla Prometheus plus Thanos and AC^3 Solution

Metric	Vanilla Prometheus + Thanos	AC3 Monitoring
Ingestion Latency (sec)	12	3
Number of API Calls to Central System	1200/hour	90/hour
CPU Load on Local Node (%)	25	9
Missed Metrics During Network Failure	High	None (due to persistent queue)

Finally, we leveraged the power of PromQL, the Prometheus Query Engine, to facilitate custom metrics based on the existing ones. The issue we tried to address is that many times the user is interested in either compound metrics or metrics that span a different timeframe than the ones given by default. For example, you might need to see the memory consumption of a RabbitMQ application for the last six hours, when your workload mattered most. This metric is not available by default, but we can request it like this:

```
rate(rabbitmq_process_resident_memory_bytes[6h]) * 3600 * 6
```

which returns all the data points that the user/application needs in order to further analyze a certain behavior or make a decision. Furthermore, the end-user can then visualize this result using Grafana, as shown in Figure 10.

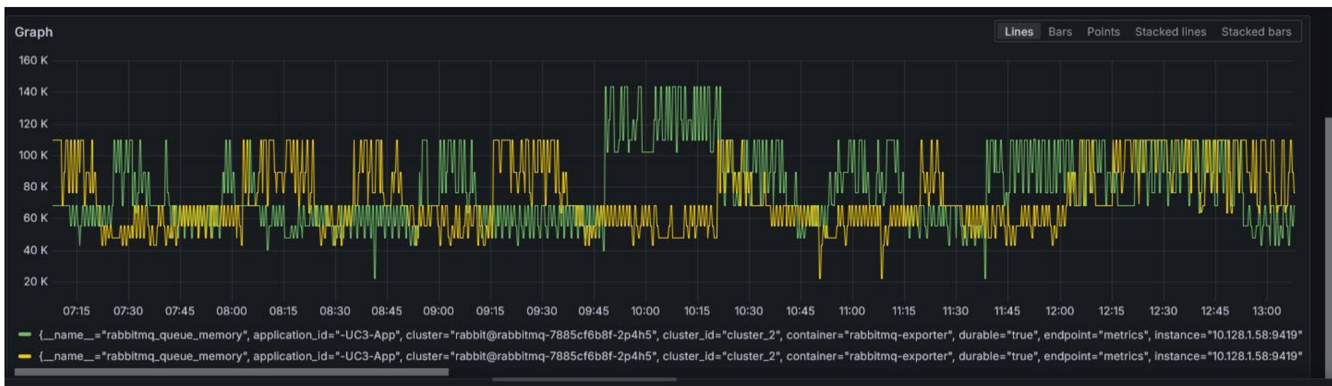


Figure 10 Monitoring Behavior Analysis

4.2.1 Data model

Interoperability between system components is not just a design preference—it is a structural necessity when aiming to exchange monitoring information reliably and to guarantee consistent interpretation and handling of data across services. Within AC^3, the ambition extends beyond internal consistency. One of its long-term objectives is to broaden the applicability of its components, making them reusable by the broader technical community and infrastructure providers. This kind of openness demands that the underlying data model be both structurally abstract and semantically expressive: abstract enough to remain adaptable across deployments, and precise enough to capture the full context and meaning of each metric.

To this end, each metric must be represented with ontological clarity, allowing any monitoring component to understand not only the value but the role and intent behind it. The objective is to minimize the need for translation layers or ad hoc wrappers at integration points. A uniform structure for exposing metric characteristics is thus not a convenience but a strategic move: it reduces integration friction and encourages adoption across varied environments.

4.2.1.1 For computing

The model we propose—and which is illustrated in the figure below—takes inspiration from well-established practices. The formatting approach used by Prometheus was deliberately adopted due to its widespread recognition and formal endorsement by institutions such as the Cloud Native Computing Foundation[10]. This choice aligns our data model with existing observability ecosystems while keeping it extensible. In parallel, OpenTelemetry was evaluated as a candidate, but its design leans toward log and trace aggregation, making it less suitable for AC^3's primary focus: real-time metric collection and monitoring. That said, our review of relevant literature revealed two consistently referenced standards: OpenTelemetry and OpenMetrics. Rather than create

---

[10] https://www.cncf.io/

a new standard, we made the conscious decision to build on an existing one. OpenMetrics aligned more directly with the architectural and design decisions already made within AC[3], especially in terms of semantic modeling and native compatibility with the components in use.

For our use case—anchored in quantitative metric tracking—OpenMetrics provides the right level of fidelity. Based on this foundation, we developed a lightweight wrapper that extends its baseline schema. This wrapper enables grouping of multiple metrics into a single API request, simplifies source identification, and takes advantage of OpenMetrics' labeling convention to introduce contextual fields such as inter-cluster relations.

From an operational perspective, the model does not merely function as a backend to internal AI systems (described in Section 3); it instead becomes a crucial tool for system developers, DevOps engineers, and application architects. It illuminates resource dynamics and bottlenecks often overlooked during development. Further, this could serve as a backend feeding visualization tools, thus allowing for quick generation of real-time graphs and performance analytics that eventually feed into monitoring dashboards or administrative UI.

Given the inherently distributed and service-oriented nature of AC[3]-supported applications, extra parameters were added to the schema to organize and give context to monitoring data. This organization makes sure that services know not just what they are seeing but also where and why. So, every metric is wrapped along with identifiers that place it within a definitive organizational and topological context.

These additions include:

```
cluster_id: a unique reference for identifying the cluster of origin
application_id: a reference to the application emitting the metric
metrics: a structured list containing all metric entries associated with the given
cluster and application
```

Being able to prioritize what needs to be monitored, assess what metrics are valid for a certain segment of the infrastructure, and relate those measurements to their logical or physical views, release such properties. This fosters decision-making based on precision and contextual information, whether it is done at the orchestration level, by life cycle managers powered by AI, or by the system health operators.

As a result of this work, the model represents no prescriptive artifact toward a certain framework; instead, we see it as a design tool: pragmatic, extensible, and ready to confront scalable multi-tenant infrastructures where observability needs to be suited for insight both at local and global levels.

The data model we ended up with is the following:

```
{
  "description": "Schema for monitoring data",
  "type": "object",
  "properties": {
    "cluster_id": {
      "description": "The cluster identifier",
      "type": "string"
    },
    "application_id": {
      "description": "The application identifier",
      "type": "string"
    },
    "metrics": {
      "type": "array",
```

```
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string",
            "description": "Name of the metric (e.g., 'http_requests_total')"
          },
          "type": {
            "type": "string",
            "description": "Metric type",
            "enum": [
              "counter",
              "gauge",
              "histogram",
              "summary"
            ]
          },
          "value": {
            "type": [
              "number",
              "string"
            ],
            "description": "Metric value"
          },
          "timestamp": {
            "type": "string",
            "format": "date-time",
            "description": "When this specific metric was collected"
          },
          "labels": {
            "type": "object",
            "additionalProperties": {
              "type": "string"
            },
            "description": "Key-value pairs of metric labels"
          },
          "quantiles": {
            "type": "object",
            "additionalProperties": {
              "type": "number"
            },
            "description": "For summary/histogram metrics, the quantile values"
          },
          "metadata": {
            "type": "object",
            "properties": {
              "help": {
                "type": "string",
                "description": "Description of what the metric represents"
              }
            }
          }
```

```
        },
        "required": [
          "name",
          "type",
          "value"
        ]
      }
    }
  }
}
```

Our main contributions to the original standard were three-fold: we created a wrapper that can encapsulate multiple metrics on a single API call, we added the info needed to identify the origin of the metrics, and we leveraged the labeling system of OpenMetrics to be able to identify inter-cluster metrics.

The schema defines the following properties:

- cluster_id -> The id of the specific cluster where the metric originated from,
- application_id -> The id of the application where the metric originated from,
- metrics -> a list with all the available metrics for the specific cluster and application,

4.2.1.2 For networking

In a multi-cloud architecture, network performance directly impacts application reliability, user experience, and operational costs. Monitoring metrics like latency, throughput, and packet loss is critical because distributed workloads depend on seamless communication across cloud providers, regions, and clusters. When a data model had to be defined for networking in a multi-cloud / multi-cluster environment like AC^3, the biggest challenge was to account for metrics between clusters. AC^3 opted for a simple model where all the metrics that concern communication between clusters are under **interClusterCommunication,** and the only thing they need to denote is the destination cluster ID to determine which two clusters the metrics are about, since the origin cluster ID is baked in the data model.

Networking data model:

```
{
  "networkMetrics": {
    "type": "object",
    "properties": {
      "bandwidth": {
        "type": "object",
        "properties": {
          "inbound": {
            "type": "number",
            "description": "Inbound bandwidth in bits per second"
          },
          "outbound": {
            "type": "number",
            "description": "Outbound bandwidth in bits per second"
          }
        }
      },
      "throughput": {
        "type": "object",
```

```json
      "properties": {
        "inbound": {
          "type": "number",
          "description": "Inbound throughput in bytes per second"
        },
        "outbound": {
          "type": "number",
          "description": "Outbound throughput in bytes per second"
        }
      }
    },
    "latency": {
      "type": "object",
      "properties": {
        "internal": {
          "type": "number",
          "description": "Internal network latency in milliseconds"
        },
        "external": {
          "type": "number",
          "description": "External network latency in milliseconds"
        }
      }
    },
    "packetLoss": {
      "type": "number",
      "description": "Percentage of packets lost"
    },
    "errorRate": {
      "type": "number",
      "description": "Percentage of network errors"
    },
    "connectionCount": {
      "type": "integer",
      "description": "Number of active network connections"
    },
    "interClusterCommunication": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "targetCluster_id": {
            "type": "string",
            "description": "Identifier of the target cluster"
          },
          "latency": {
            "type": "number",
            "description": "Latency between clusters in milliseconds"
          },
          "throughput": {
            "type": "number",
            "description": "Throughput between clusters in bytes per second"
```

```
          },
          "packetLoss": {
            "type": "number",
            "description": "Packet loss percentage between clusters"
          }
        },
        "required": [
          "targetCluster_id"
        ]
      }
    }
  }
}
}
```

Example of a networking metric:

```
{
  "networkMetrics": {
    "bandwidth": {
      "inbound": 1250000000, // 1.25 Gbps incoming traffic
      "outbound": 750000000 // 750 Mbps outgoing traffic
    },
    "throughput": {
      "inbound": 92000000, // 92 MB/s actual data received
      "outbound": 58000000 // 58 MB/s actual data sent
    },
    "latency": {
      "internal": 2.4, // 2.4ms within cluster
      "external": 34.7 // 34.7ms to external services
    },
    "packetLoss": 0.02, // 0.02% packets lost
    "errorRate": 0.001, // 0.1% network errors
    "connectionCount": 1428, // Active connections
    "interClusterCommunication": [
      {
        "targetCluster_id": "europe-prod-01",
        "latency": 28.5, // 28.5ms to European cluster
        "throughput": 42000000, // 42 MB/s cross-cluster throughput
        "packetLoss": 0.15 // 0.15% packet loss between clusters
      },
      {
        "targetCluster_id": "asia-edge-02",
        "latency": 152.8, // Higher latency to Asia cluster
        "throughput": 18000000,
        "packetLoss": 0.32 // Higher packet loss over long distance
      }
    ]
  }
}
```

4.2.1.3 For energy

For defining the energy consumption data model, exporters like Kepler can be used, which is the industry standard for measuring energy-related metrics. Kepler directly correlates CPU, memory, GPU, and other component usage with power draw, enabling accurate tracking of energy efficiency. It integrates seamlessly with Prometheus and Grafana, making it easy to visualize trends and set alerts for abnormal consumption, which has the added benefit of making it easier to define energy-related KPIs / SLAs.

Energy consumption data model

```
{
  "title": "AC3 Energy Consumption Metrics",
  "type": "object",
  "properties": {
    "timestamp": {
      "description": "The time at which the metrics were collected",
      "type": "string",
      "format": "date-time"
    },
    "cluster_id": {
      "description": "The cluster identifier",
      "type": "string"
    },
    "application_id": {
      "description": "The application identifier",
      "type": "string"
    },
    "node": {
      "description": "Node-level energy metrics",
      "type": "object",
      "properties": {
        "name": {
          "description": "Name of the Kubernetes node",
          "type": "string"
        },
        "cpu_energy_joules": {
          "description": "CPU energy consumption in joules",
          "type": "number"
        },
        "memory_energy_joules": {
          "description": "Memory energy consumption in joules",
          "type": "number"
        },
        "gpu_energy_joules": {
          "description": "GPU energy consumption in joules (if applicable)",
          "type": "number"
        },
        "other_components_energy_joules": {
          "description": "Energy consumed by other components (e.g., disk, 
network)",
          "type": "number"
```

```
        },
        "total_energy_joules": {
          "description": "Total energy consumed by the node in joules",
          "type": "number"
        }
      },
      "required": [
        "name",
        "cpu_energy_joules",
        "memory_energy_joules",
        "total_energy_joules"
      ]
    }
  },
  "required": [
    "timestamp",
    "application_id",
    "cluster_id"
  ]
}
```

The metrics that extend the data model that we define are the following:

- **gpu_energy_joules:** This measures the total energy consumption on the GPUs that a certain node has used. Currently, Kepler only supports NVIDIA GPUs
- **memory_energy_joules:** his metric describes the total energy spent in DRAM by a node.
- **cpu_energy_joules:** This measures the total energy consumption on CPU cores that a certain node has used.
- **other_components_energy_joules:** This measures the cumulative energy consumption on other host components besides the CPU and DRAM.
- **total_energy_joules:** This metric represents the total energy consumption of the host.

An example of energy consumption metric is provided below:

```
{
  "timestamp": "2025-05-18T14:30:45Z",
  "cluster_id": "ac3-prod-cluster-1",
  "application_id": "ai-inference-service",
  "node": {
    "name": "gpu-node-1",
    "cpu_energy_joules": 320.75,
    "memory_energy_joules": 95.20,
    "gpu_energy_joules": 210.50,
    "other_components_energy_joules": 45.30,
    "total_energy_joules": 671.75
  }
}
```

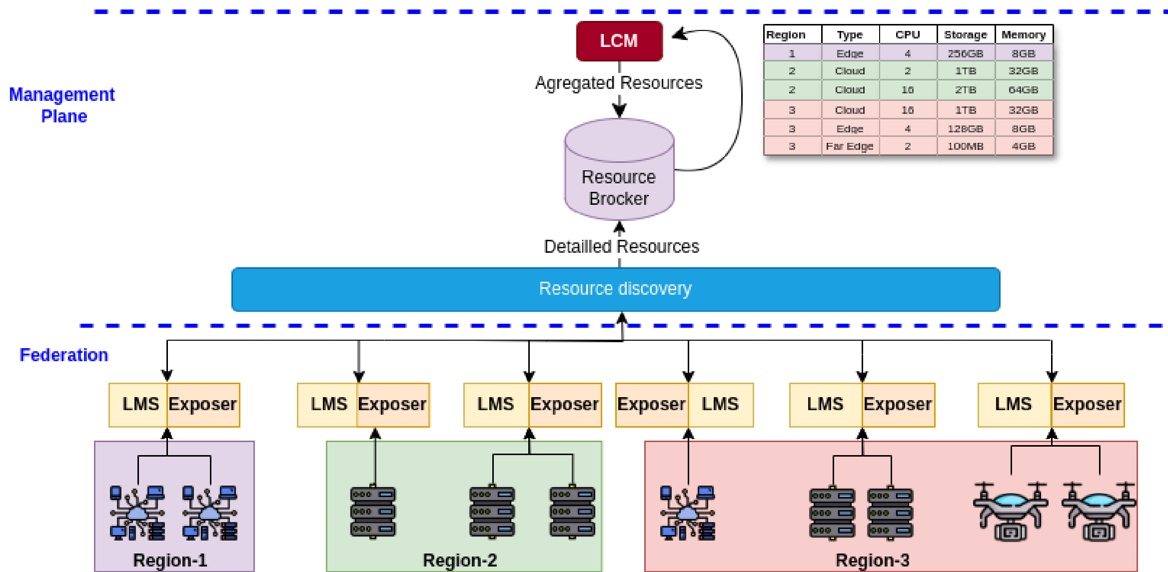
4.3 Resource discovery and monitoring of the federated resources

4.3.1 High-level architecture

Resource exposure and discovery are a critical part of federated infrastructures. It provides visibility into the available computing and networking resources that an AI-based LCM can leverage for microservice deployment.

In this section, we propose a novel approach to define an abstracted resource exposure mechanism that gathers various metrics from the underlying federated infrastructure. This allows the LCM to obtain up-to-date information about the resources managed by the local management systems (LMSs), including their current status.

The envisioned solution and its integration within the AC^3 architecture are illustrated in. It highlights the AC^3 approach to decouple service orchestration—handled by the LCM—from resource management, which is managed by the Exposer and the Discovery components.



Region	Type	CPU	Storage	Memory
1	Edge	4	256GB	8GB
2	Cloud	2	1TB	32GB
2	Cloud	16	2TB	64GB
3	Cloud	16	1TB	32GB
3	Edge	4	128GB	8GB
3	Far Edge	2	100MB	4GB

Figure 11 High-level resource discovery/exposer Architecture

On the federation side, the key component is the Resource Exposer, which is deployed within each infrastructure. The LCM acts as a consumer of the Resource Discovery module through the Broker. Through this interaction, the LCM obtains a global, aggregated view of the available resources across all infrastructures, organized by regions, as illustrated in the table in Figure 11. A key innovation of the proposed system lies in its abstraction of resources for the LCM, enabling it to handle the heterogeneity and dynamic nature of the federated infrastructure, particularly far-edge resources that may appear or disappear dynamically due to the mobility of edge nodes such as autonomous vehicles, cars, and other mobile platforms.

The Resource Discovery component interfaces with the underlying computing resources through the Exposers. In the example shown in Figure 1, we focus on computing resources exposed by various LMS Exposers, each responsible for a specific technological domain. The Discovery component is responsible for identifying these LMS Exposers and consuming their exposed APIs to retrieve information about available computing resources, including their availability and type (e.g., edge, central cloud, or far-edge).

All retrieved information is stored in a shared broker by the LCM consumers, enabling aggregation and use by the LMS decision-making modules. In the example illustrated in the figure, the Discovery component collects data from LMS Exposers assigned to three different regions. Each LMS reports its node type (edge, central cloud, or far-edge), along with details about available resources. In this case, the exposed resources pertain to computing, specifically CPU, memory, and storage. However, the system is flexible and can also support the exposure of other resource types such as GPUs, DPUs, or FPGAs.

The Discovery component periodically collects and refreshes this data in the broker. In parallel, the LCM Consumer aggregates this information by region and resource type, abstracting away low-level LMS-specific details.

4.3.2 Design and implementation

In this subsection, we present the technical design and implementation of the proposed Exposure and discovery framework, which is composed of 2 main software components: the discovery and the exposer modules. The detailed architecture is presented in Figure 12.

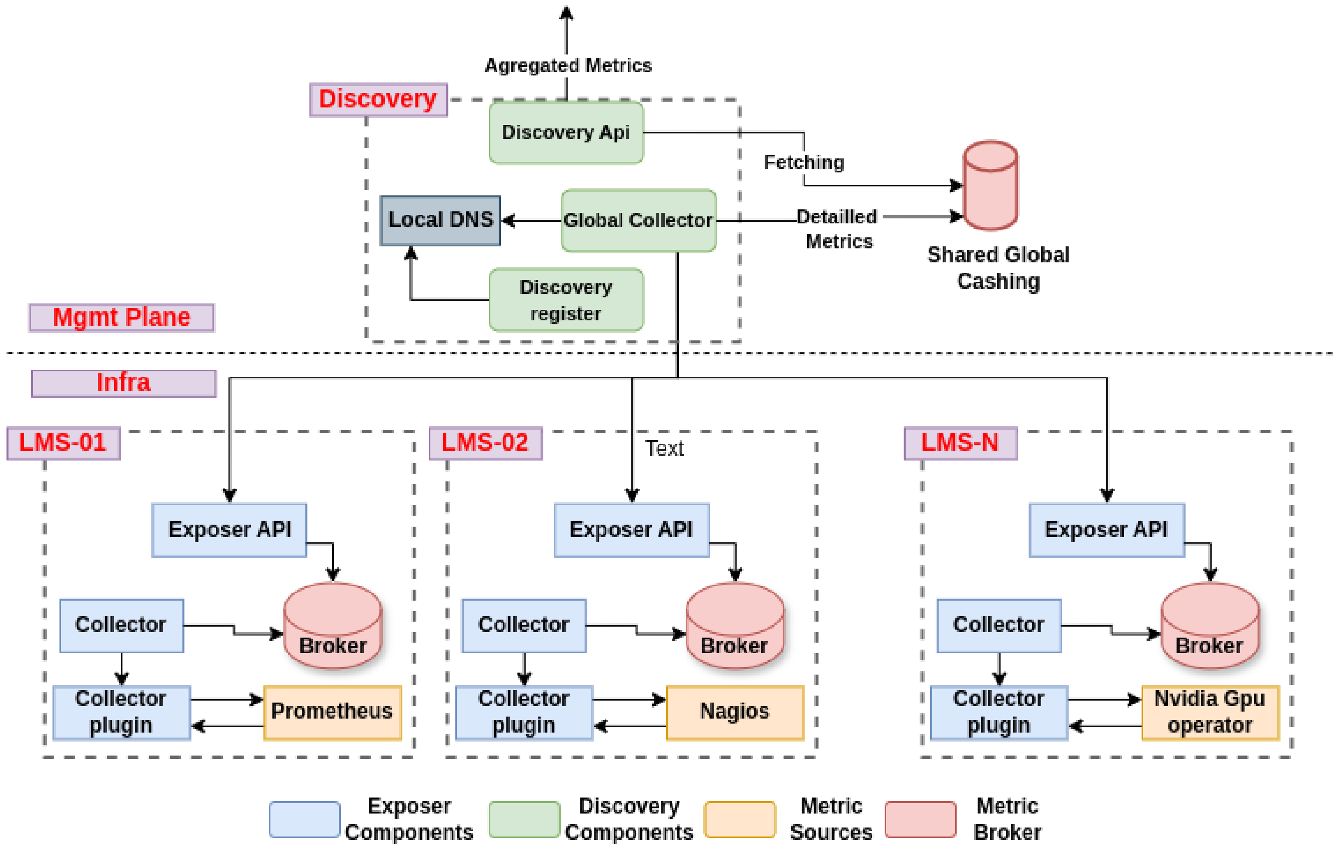


Figure 12 Detailed technical view of the resource exposure framework.

4.3.3 LMS Exposer: Case of Kubernetes (K3s, K8s, OpenShift)

The LMS Exposer is deployed at the infrastructure level and represents the lowest layer of the resource exposure framework. Its main function is to supply the Discovery component with detailed information about the computing resources available in the environment where it operates.

Designed as a containerizable component, the LMS Exposer can be easily deployed across a wide range of modern container orchestration platforms, including Kubernetes and its lightweight variants such as K3s, KubeEdge, and OpenShift.

Due to the heterogeneity of cloud-edge infrastructures, environments often rely on a variety of monitoring tools. Some use standard solutions like Prometheus or Nagios, while others rely on platform-specific tools such as Tegrastats (commonly used on NVIDIA devices) or htop. To accommodate this diversity, the LMS Exposer uses a pluggable architecture for metric collection. Each plugin serves as an adapter between the LMS Exposer and the specific monitoring tool used in the target infrastructure, ensuring compatibility and modularity.

Collected metrics are temporarily stored in a message broker, enabling decoupled and efficient communication between system components. This architectural choice addresses two key challenges:

Scalability and Responsiveness: It supports concurrent metric requests from multiple entities—especially relevant when a single computing node belongs to multiple local systems—while maintaining low response latency under high load.

Freshness and Lightweight Operation: By avoiding long-term metric storage and retaining only recent data, the LMS Exposer ensures that the Resource Federation and Access Control (RFAC) component always receives up-to-date resource information. To integrate smoothly with various deployment contexts, the framework supports multiple message broker options, including RabbitMQ, Apache Kafka, and Redis. To ensure platform-agnostic operation for higher-level components such as the Discovery, the LMS Exposer presents all collected metrics using a unified data format (see Figure 11). Since metric sources vary widely in how they expose data—each with its own structure, semantics, and granularity—the unified format standardizes these differences, offering a consistent and structured representation of metrics regardless of the underlying platform. This abstraction layer simplifies downstream processing and enables seamless integration across heterogeneous infrastructures.

Figure 13 illustrates the unified formats used for compute-related metrics captured by the LMS Exposer.

The compute metrics encompass information on memory, CPU, and storage:

- Memory metrics report the total, available, and used memory at both the cluster and individual node levels.
- CPU metrics include the number of cores, per-core usage, average utilization, and free computational capacity.
- Storage metrics provide details on total, used, and available storage, along with I/O statistics such as read/write operation counts and bandwidth usage over time.

This unified format facilitates consistent metric interpretation, enabling informed orchestration and decision-making processes at higher layers of the resource exposure framework.

```
"2024-07-19 16:26:22.980972": {
        "numberOfNodes": 7,
         "memory": {
            "provisioned_memory": {
                "totalProvisionedMemory": "value",
                "detailedValues": {
                   "machine01": "value"}},
            "memory_available": {
                "totalAvailableMemory": "value",
                "detailedValues": {
                    "machine01":"value"}},
            "memory_usage": {
                "totalUsedMemory": "value",
                "detailedValues": {
                   "machine01": "value"}}},
        "cpu": {
            "totalProvisionedCpuCores": {
                "totalCpuCores": "value",
                "numberOfCpuCoresPerNode": {
                   "machine01": "value"}},
            "usedCpu": {
                "detailedCpuUsageValues": {
                   "machine01-cpu01":"value"},
                "averageCpuUsageValuesPerNode": {
                  "machine01":"value"}},
            "freeCpu": {
                "detailedFreeCpuValues": {
                  "machine01-cpu01":"value"},
                "averageFreeCpuValuesPerNode": {
                  "machine01":"value"}}},
        "disk": {
            "freeDisk": {
                "totalAvailableDsikSpace": "value",
                "detailedValuesPerNode": {
                "machine01":"value"}},
            "usedDisk": {
                "totalAvailableDsikSpace": "value",
                "detailedValuesPerNode": {
                 "machine01":"value"}},
            "diskThroughput": {
                "read": {
                    "detailedValuesPerStorageUnit": {
                        "machine01-sda": "value",}},
                "write": {
                    "detailedValuesPerStorageUnit": {
                    "machine01-sda":"value"}}}}
}
```

Figure 13 Example of JSON format for compute resource metrics, including memory, CPU, and storage utilization.

4.3.4 Discovery

The Discovery component plays a central role in managing LMS Exposer instances and providing the LCM with up-to-date resource information across the infrastructure. Its primary responsibilities include:

- Managing the lifecycle of multiple Exposer instances,
- Facilitating seamless communication via local DNS,
- Ensuring reliable and timely delivery of resource metrics to upper-layer components.

When a new LMS Exposer becomes available within the infrastructure, it initiates a self-registration process through the Discovery's southbound registration API. This registration signals the Exposer's operational readiness. Upon successful registration, the Discovery records the Exposer's Internet Protocol (IP) address and domain name in a local DNS system, which enables efficient and transparent communication between components.

This decentralized, self-registration mechanism enhances the framework's scalability and flexibility by eliminating the need for the Discovery component (at the management plane) to proactively scan for new infrastructures. Each Exposer independently registers itself, streamlining the integration of new computing nodes or clusters and supporting dynamic changes without requiring manual reconfiguration.

To maintain a reliable and current resource view, the Discovery module continuously monitors all registered Exposers using a heartbeat mechanism. If an Exposer fails to send periodic heartbeats or becomes unreachable, the discovery automatically unregisters it and removes its DNS entry. This ensures that higher-level components always operate based on an accurate view of available resources.

Beyond registration and monitoring, the discovery includes a global collector that periodically queries all active Exposers for compute resource metrics. These metrics are then published to a global message broker, using a decoupled architecture similar to that employed by the LMS Exposer. This approach supports concurrent access, maintains performance under high load, and avoids the overhead of long-term metric storage. The discovery northbound API aggregates those resources to present them to the LCM to prevent overloading the LCM with excessive detail. This model contains only high-level information about resource availability across the infrastructure, as shown in Figure 14.

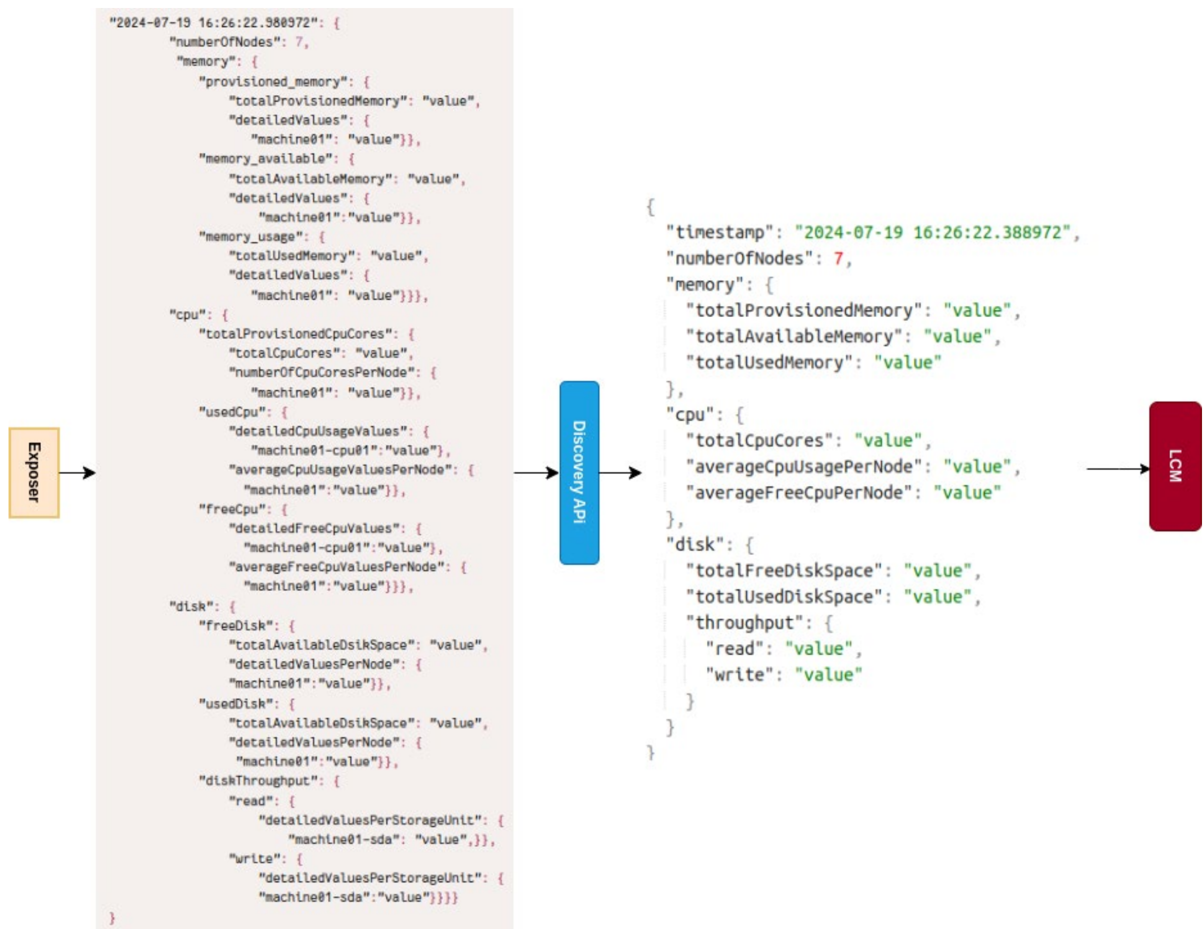


Figure 14 Aggregation and abstraction of resource metrics by Discovery for the LCM.

5 AI/ML Models for Prediction and Resource Management

This work develops an explainable AI mechanism for the Resource Management of the AC3 project. To predict resource usage, we developed several machine learning models, including XGBoost, and deep learning methods such as the Temporal Fusion Transformer (TFT), as detailed in Deliverable 4.1. To interpret the predictions generated by these models, we employed SHAP in conjunction with XGBoost and leveraged the inherent explainability features of TFT, such as its variable selection mechanism and multi-head attention. In this deliverable, we extend our work by introducing methods to validate explainability techniques for deep learning models, specifically in the context of resource usage prediction. For example, we designed a framework to evaluate the consistency and reliability of the various explainability approaches used in the context of explaining deep learning models. By cross-validating these techniques, we were able to assess the robustness of the model's interpretability and ensure that the explanations aligned with domain knowledge and actual system behavior. This validation process is critical for building trust in deep learning models used in operational decision-making for resource management.

5.1 Explainable AI

As mentioned earlier, to explain the decisions made by the developed AI models, we employ a range of explainability techniques and compare their outputs to identify commonalities and assess consistency. This approach is driven by two main objectives. First, by comparing multiple explainability methods, we can validate their reliability—if different methods consistently highlight the same features as influential, it strengthens confidence in their conclusions. Second, this comparison enables us to qualitatively assess the performance of each method and identify the most effective one for downstream tasks such as automated cloud resource scaling. To this end, we incorporate both global and local interpretability methods. Global methods provide insights into the model's overall behavior, while local methods focus on explaining individual predictions. Local explanations can also be aggregated across multiple instances to infer consistent patterns of feature influence at a global level. In general, interpretation techniques aim to quantify the contribution of each input feature to the model's output. However, evaluating the accuracy of these importance scores is challenging due to the absence of ground truth for explanations. A common strategy is to perturb the most important features—those identified by the interpretability method—and measure the resulting changes in model output. This provides a quantitative estimate of the explanation's fidelity and helps assess the trustworthiness of the interpretability method in practical scenarios.

5.1.1 Explainable approaches

In this section, we present the interpretation methods used in our study and outline the strategies employed to evaluate their performance across different approaches. Our interpretability work is based on deep learning methods, as detailed in Deliverable 4.1. Interpretation methods for time series data have gained significant traction in recent years [1]. In particular, attribution methods—which assign an importance or contribution score to each input feature with respect to the model's output—are widely used to explain the behavior of deep learning models. These methods help quantify how much each feature influences a model's prediction. Generally, attribution methods can be grouped into three main categories: a) **Gradient-based methods**, b) **Perturbation-based methods**, and c) **Backpropagation-based methods**. Given the nature of our application, we focus on the first two categories: gradient-based and perturbation-based attribution. Let $f: \mathbb{R}^d \longrightarrow \mathbb{R}$ be a predictive model, where $f \in \mathrm{F}$ and $X \in \mathbb{R}^d$is an input vector with d features. An attribution method is defined as a function ϕ: $\mathrm{F} \times \mathbb{R}^d \rightarrow \mathbb{R}^d$where each component $\phi_i(f, X)$ represents the estimated contribution of the $i-th$ input feature. In many attribution frameworks, a reference input $G \in \mathbb{R}^d$ that acts as a baseline is also required for computing relative feature contributions. We focus our analysis on two widely adopted attribution techniques:

- **Gradient-based attribution** computes the gradient of the output with respect to each input feature, and

- **Perturbation-based attribution** measures the change in model output when individual input features are altered or masked.

These methods form the main aspect of our explainability framework for explaining the deep learning models experimented for predicting resource usage.

5.1.2 Perturbation-based methods

Perturbation-based methods are especially valuable when working with black-box models such as deep neural networks, where internal model parameters and structures are not easily interpretable. These methods estimate feature importance by systematically modifying the input data and observing how such perturbations affect the model's predictions. The core idea is that if altering a specific feature leads to a significant change in the output, that feature is likely important to the model’s decision-making process. Common perturbation strategies include removing a feature (e.g., setting it to zero or a baseline value), masking it (e.g., replacing it with noise), or replacing it with values drawn from a reference distribution. The model is then reevaluated with each perturbed input, and the resulting change in the output prediction is used to compute an attribution or importance score for the altered feature. Because they do not rely on gradients or internal model mechanics, perturbation-based methods are model-agnostic. This makes them highly flexible and broadly applicable across various types of models, including those where gradient information is not available or reliable. In this work, we consider feature ablation, feature permutation importance, and occlusion sensitivity.

5.1.2.1 Features ablation

Feature Ablation is a perturbation-based explainability technique that estimates feature importance by systematically removing input features and observing the effect on the model's output or performance. The method works by iteratively ablating one feature at a time, typically by setting it to a neutral value such as zero, the mean, or a baseline reference, and then measuring how much the model’s prediction or overall performance changes as a result. This change serves as an indication of the feature's contribution to the model’s decision. Feature Ablation can be applied as both a global and local interpretation method [2]. In the global context, the technique is used across the entire dataset to evaluate the average importance of each feature to the model's predictions. In the local context, it focuses on a single instance and assesses how the prediction for that specific input changes when individual features are ablated. One of the main advantages of Feature Ablation is its simplicity and intuitive rationale. It provides a direct and interpretable measure of feature influence: if removing a feature leads to a significant degradation in model performance or a substantial change in output, the feature is deemed important. Conversely, if the model output remains largely unaffected, the feature is likely less relevant.

5.1.2.2 Feature permutation importance

An alternative to filtering or removing features to assess their importance is the Feature Permutation Importance method. This approach explains model decisions by evaluating the impact of randomly shuffling (or permuting) the values of each input feature across the dataset and measuring how this perturbation affects the model’s performance [3]. The underlying intuition is straightforward: if permuting a feature’s values disrupts the relationship between that feature and the target variable, and consequently leads to a significant drop in model accuracy or another performance metric, then that feature is considered important. The process is composed of the evaluation of the baseline performance of the model on a validation or test dataset. For each feature in the dataset, a) randomly shuffle its values across all data instances, breaking the association between the feature and the output, b) recalculate the model’s performance using the permuted dataset, and c) compute the difference in performance compared to the baseline, finally d) rank features based on the magnitude of performance degradation. This method is typically used as a global interpretability tool, providing an aggregate view of feature importance across the entire dataset rather than for individual predictions. It is particularly useful

for identifying which features the model relies on most during inference and is widely used in conjunction with ensemble methods such as random forests and gradient boosting machines, although it can be applied to any model. This approach has several merits, such as being model agnostic, as it does not rely on internal model parameters or gradients, making it applicable to any type of machine learning model. It is intuitive as the method provides a direct measure of a feature's contribution to model performance, and comprehensive by evaluating features one by one, it generates a ranked list of their relative importance.

5.1.2.3 Occlusion sensitivity

It is a perturbation-based, local interpretability method that explains a model's prediction by systematically masking or occluding parts of the input and observing the resulting changes in the model's output [4]. The key idea behind occlusion sensitivity is that if a particular portion of the input is important to the model's prediction, then removing or masking that part should significantly change the model's output. Conversely, if masking a region has little or no effect, then that region is likely less important for that specific prediction. Its mechanism is based on a *Baseline Output,* which computes the model's original prediction for a given input instance. *Iterative Occlusion*: Slide a fixed-size window over the input. *Masking*: At each step, the portion of the input under the window is occluded—commonly by zeroing out the values, replacing them with a constant, or using a reference baseline. *Re-Evaluation:* After each occlusion, the model is re-evaluated on the modified input, and the change in prediction is recorded. Occlusion sensitivity is considered a local method because it explains the prediction for a single input instance, rather than providing a global view of the model's behavior across all data. The approach is model agnostic.

5.1.2.4 Counterfactual Explanations

Given a particular input instance X and its prediction $f(X)$, a counterfactual explanation finds an alternative input X' as close as possible to X according to some distance metric, such that the model prediction on X' changes to a desired target outcome $f(X') / f(X)$. This often involves solving an optimization problem to minimize the distance between X and X' subject to the constraint that the prediction changes. Counterfactual explanations are inherently local because they explain individual predictions by focusing on the minimal perturbations needed for that specific case. By highlighting which features must change and by how much to alter the outcome, counterfactuals provide practical guidance, for example, in decision-making scenarios autoscaling cloud resources [5].

5.2 Gradient-based methods

Gradient-based methods are a class of model-specific explainability techniques that leverage gradient information from neural networks to estimate the contribution of each input feature to the model's output. They are especially well-suited for models where gradient computation is straightforward, such as deep learning models. The fundamental idea is to compute the partial derivative of the model's output with respect to each input feature. This derivative measures how sensitive the model output is to small changes in the input, in other words, how much the output would change if we slightly adjusted each input feature. They start with a forward pass to compute the model's output for a specific input. Then, they compute the gradient of the output with respect to each input feature. They interpret these gradients as importance scores, where the higher the gradient, the more influence the input feature has on the prediction.

5.2.1 Saliency Maps

Saliency maps explain which time steps or input variables contributed most to the model's decision for a specific input sequence. It highlights the important time steps, for example, which moments in the sequence mattered most. It determines important variables/features: If the input has multiple features (multivariate time series), it highlights which features at which times had the most influence. Saliency is a simple gradient-based approach; it uses the gradient of the model at the input level as the corresponding feature attribution. This method takes

a first-order approximation of the function, in which the gradients serve as the coefficients of each feature in the model. It uses, therefore, the gradients to determine important features [6].

5.2.2 Integrated Gradients

Integrated Gradients (IG) is a gradient-based feature attribution method that aims to address limitations of raw gradients, such as noise and saturation, by computing a more stable and theoretically grounded importance score for each input feature. The core idea is to average the gradients of the model's output as the input varies along a straight path from a baseline input (e.g., all zeros) to the actual input. This integrated gradient provides a smoother and more reliable estimate of how each input feature contributed to the final prediction. This approach satisfies *sensitivity,* meaning if two inputs differ in only one feature and have different predictions, the differing feature must get non-zero attribution, and *implementation invariance,* which means that equivalent models produce identical attributions [7].

5.2.3 Gradient SHAP

Gradient SHAP is a method that combines ideas from SHAP values, Integrated Gradients, and SmoothGrad. It is a gradient-based feature attribution method that approximates SHAP values by using stochastic sampling of baseline inputs and integrating gradients. It provides a principled way to estimate how much each input feature contributes to the prediction, in a manner that is more stable than raw gradients and more efficient than exact SHAP in the context of deep learning models [8]. Gradient SHAP approximates SHAP values by sampling a set of random baselines, drawing random interpolations between the baselines and the input, computing gradients along these interpolated paths, and averaging the results. The outcome is generally a more efficient and scalable method for explaining deep learning models than exact SHAP.

5.2.4 Performance evaluation

The lack of ground truth for interpretation represents a major challenge in evaluating the effectiveness of explainability methods. A wide range of techniques exists for interpreting the predictions made by deep learning models, which creates the need for robust evaluation methods to assess their explanatory power. Such evaluation is essential for building trust in these explainability approaches. In this work, we consider two approaches for assessing the performance of interpretability methods.

5.3 Area Over the Permutation Curve for Regression

The *comprehensiveness* and *sufficiency* metrics have been proposed to evaluate the faithfulness of model interpretations. Comprehensiveness measures whether the features identified as important truly contribute to the prediction. It is computed by removing (masking) the most important features and observing the change in the model's output. A significant drop in confidence indicates high comprehensiveness. In contrast, sufficiency assesses whether the identified features alone are enough to make the prediction. This is evaluated by masking all features except the most important ones and measuring the change in the model's output. A small change indicates high sufficiency. Ideally, a good interpretation should have high comprehensiveness loss (large drop in confidence when key features are removed) and low sufficiency loss (small change in confidence when only key features are kept).

This allows to determine the area over the perturbation curve for regression (AOPCR). We define the top $k\%$ most relevant features of the $i-th$ input X_i as selected by the interpretation method as $X_{i,1:k}$. The input after removing these top features is denoted as $X_{i,\setminus 1:k}$. Then, for a model $f(\cdot)$ comprehensiveness and sufficiency are defined as follows

$$C = \left| f(X_i) - f\left(X_{i,\setminus 1:k}\right) \right|$$

$$S = \left| f(X_i) - f(X_{i,\,1:k}) \right|$$

The aggregated comprehensiveness score is referred to as the Area Over the Perturbation Curve for Regression, AOPCR

$$AOPCR_c = \frac{1}{K \times \tau_{max}} \sum_{\tau}^{\tau_{max}} \sum_{k}^{K} \left| f(X_i)_\tau - f(X_{i,\,\backslash 1:k})_\tau \right|$$

$$AOPCR_s = \frac{1}{K \times \tau_{max}} \sum_{\tau}^{\tau_{max}} \sum_{k}^{K} \left| f(X_i)_\tau - f(X_{i,\,1:k})_\tau \right|$$

We consider this evaluation metric as it is generally applied to time series problems [9].

5.4 Experiments and results

In Deliverable 4.1, we focused on vertical resources, referring to computational attributes such as CPU and memory allocated to each microservice. In this work, we shift our emphasis to horizontal resources (particularly inspired by use case 3), which are characterized by the number of pods used in each microservice that participates in a transaction. To evaluate the performance of the proposed approaches, we consider a complex architecture composed of microservices and multiple transactions, where each transaction is associated with a specific workload. Our objective is to determine how the latency of a given transaction is influenced by both the workload and the availability of horizontal resources (i.e., the number of pods) in the microservices involved. Figure 15 presents the calls at the transaction level, the pods deployed at the service level for the given transaction, and the latency, which serves as the target variable for prediction. As an example, consider the task of predicting the latency at the front-end for transaction_1. In this case, pod_service_1 and pod_service_2 represent the pods used in two microservices that are part of transaction_1, and are considered to have a direct impact on the transaction's latency. To train our predictive model, the input features consist of the number of calls across the four transactions and the number of pods deployed for each service involved in the transaction for which we aim to predict the latency. In Deliverable 4.1, we extensively explored prediction methods, particularly those based on deep learning approaches such as the Transformer, where we arrived at a very good performance in terms of *R2,* generally > 0.9*.* In this work, our focus shifts to explainability. After predicting the latency, we aim to identify the most influential features contributing to the prediction, in order to gain insights into the adjustments needed in case of SLO (Service Level Objective) non-compliance and bring the system back to the desired working condition.

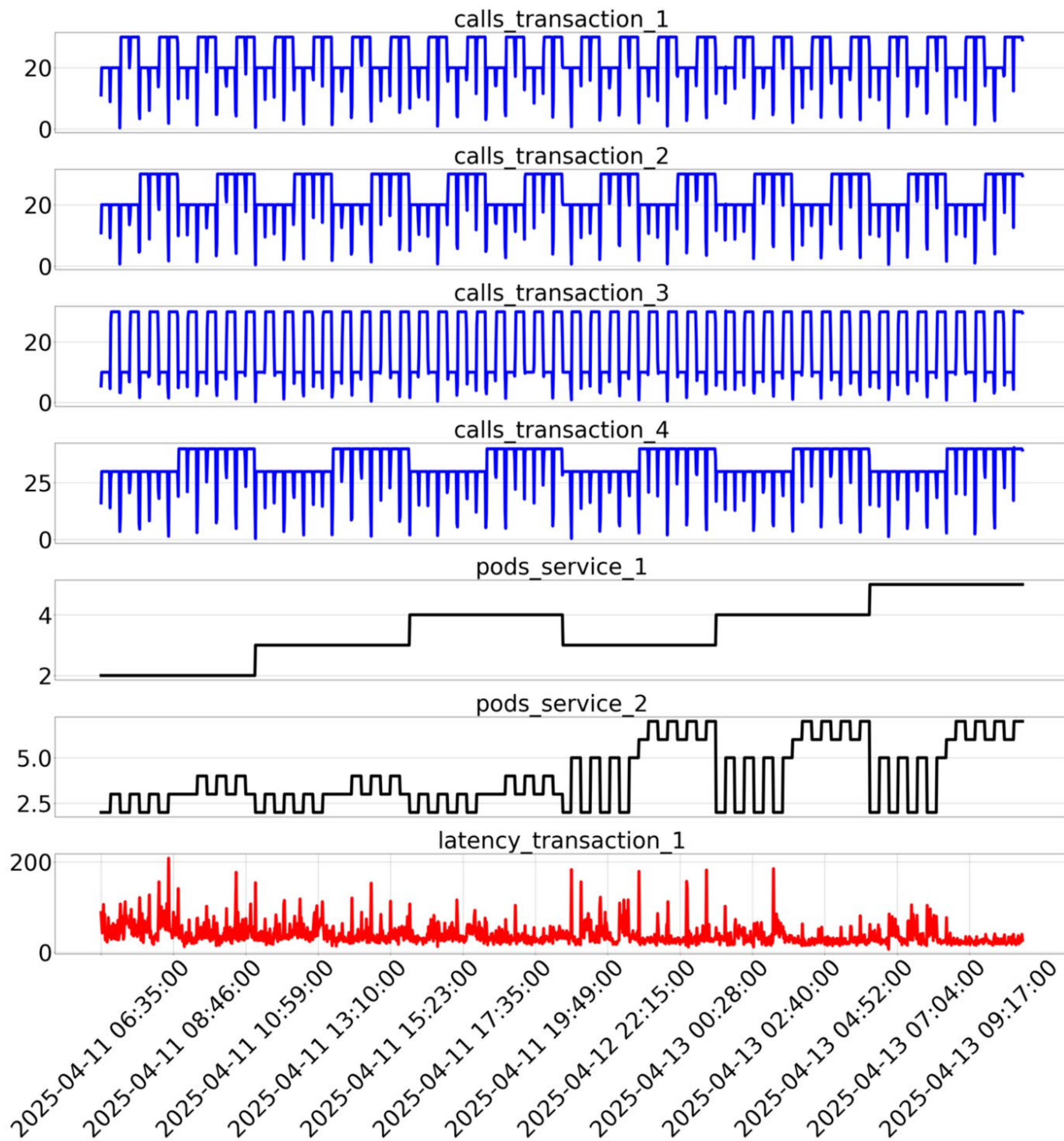


Figure 15 The metrics used for the latency prediction model

The application of the Feature Ablation method to the predicted latency results in the analyses shown in Figure 16 and Figure 17. Figure 16 displays both positive and negative feature importance scores. A positive ablation score indicates that removing the feature worsened the model's performance, meaning the model relies on that feature to make accurate predictions. In contrast, a negative ablation score suggests that removing the feature improved the model's performance, implying that the feature may be irrelevant, noisy, or even harmful to the prediction. From the results, we observe that among the actionable insights, the most influential feature is pod_service_2. This suggests that, in the event of an SLO (Service Level Objective) violation, this feature should be the primary candidate for investigation and adjustment. And the least influential feature importance is pod_service_1.

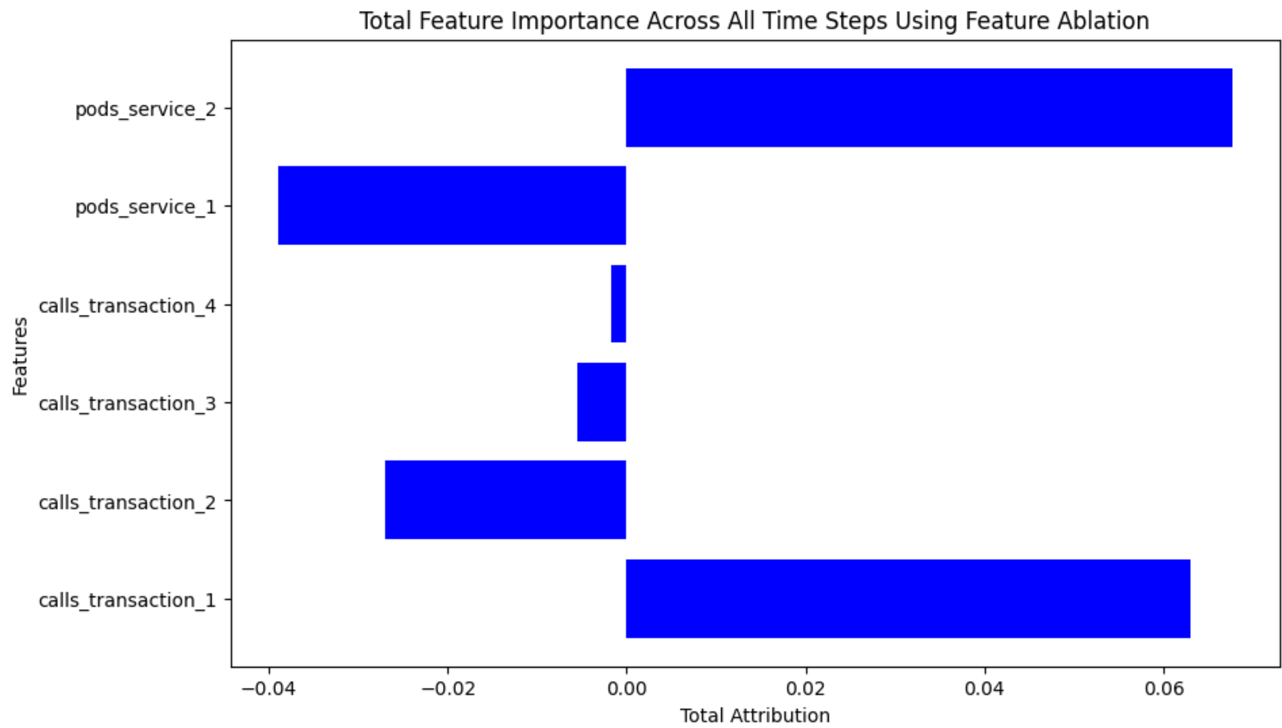


Figure 16 Feature importance across all time steps using the feature ablation method

Figure 17 presents a 3D surface of Feature importance obtained by the Feature Ablation method. It presents the advantage of giving a visual explanation of how several features jointly influence the model's output by revealing nonlinear dependencies and synergistic effects.

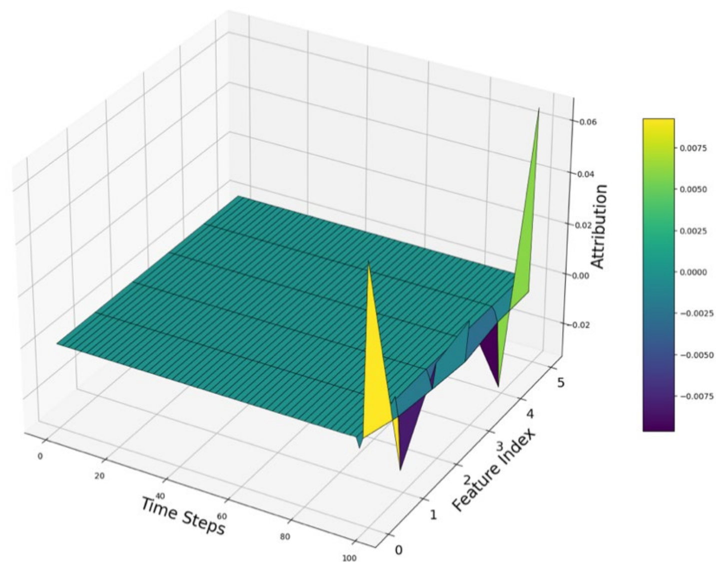


Figure 17 3D Surface of feature importance over time using feature ablation

Figure 18 confirms the results obtained in Figure 16 , where the most influential feature is given by the pod_service_2. And the least influential by pod_service_1.

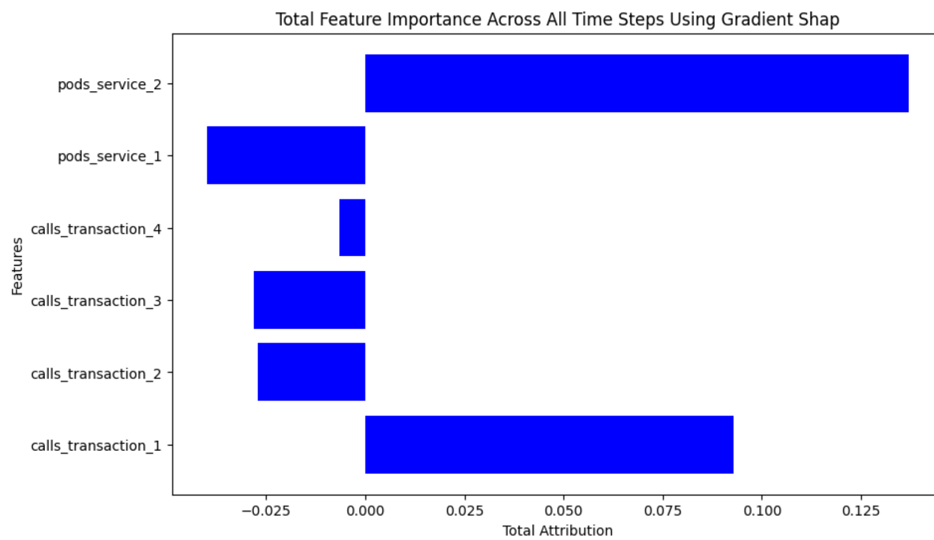


Figure 18 Feature importance across all time steps using Gradient Shap

Figure 19 presents a 3D visualization of the Gradient SHAP results, showing how multiple features contribute to the model's predictions. This plot visualizes explanation values (Gradient Shap), helping to identify how different combinations of feature values impact the importance or influence each feature has on the predicted latency. By observing the joint distribution of Gradient SHAP values across two or more features, we can gain insights into feature interactions, detect regions of high sensitivity, and better understand the model's decision logic.

This enhances explainability by going beyond simple rankings and uncovering how and when certain features matter most.

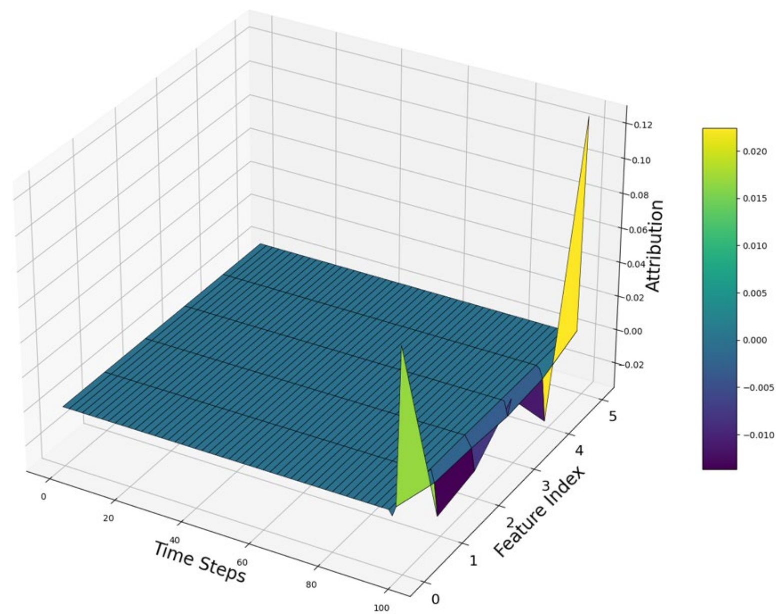


Figure 19 3D Surface of feature importance over time using Gradient Shap

Figure 20 corroborates the results observed in Figure 16 and Figure 18, identifying pod_service_2 as the most influential feature and pod_service_1 as the least influential in the latency prediction task. This agreement across multiple interpretability techniques, including Feature Ablation, Gradient SHAP, and Integrated Gradients, as well as other evaluated methods, demonstrates the consistency and robustness of the feature importance rankings. The convergence of these distinct attribution methods validates the reliability of the identified key drivers impacting the model predictions. Moreover, this multi-method explainability framework enables both quantitative explanation and cross-validation of feature contributions, thereby strengthening model interpretability and supporting informed optimization strategies for SLOs.

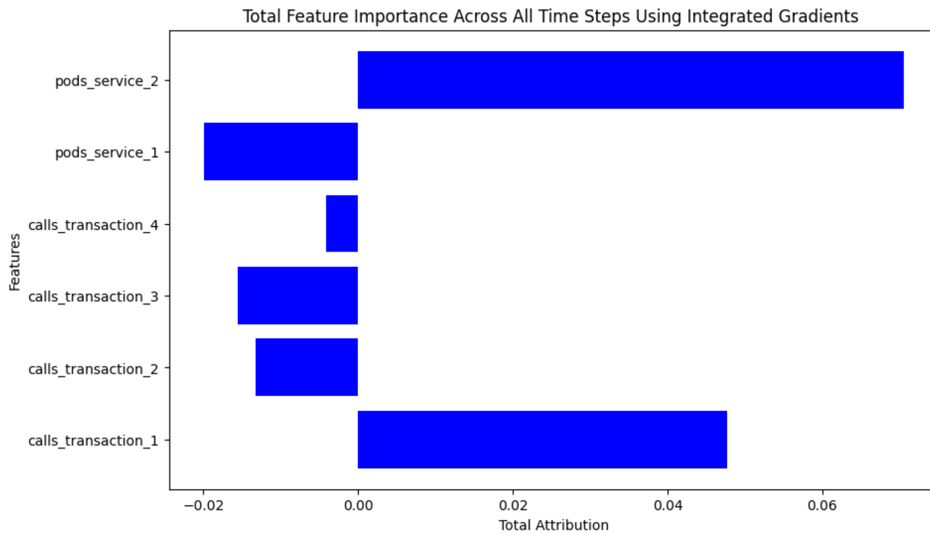


Figure 20 Feature importance across all time steps using Integrated Gradient

Similarly, Figure 21 demonstrates the consistency of the 3D explanation visualizations across the different explanation methods evaluated in this study. The figure highlights how multiple attribution techniques produce comparable patterns of feature importance and interaction effects in three-dimensional space, reinforcing the reliability of the explanations. This alignment across methods confirms that the joint influence of features on latency predictions is robust and not an artifact of a single approach, thereby providing deeper insight into the model's behavior.

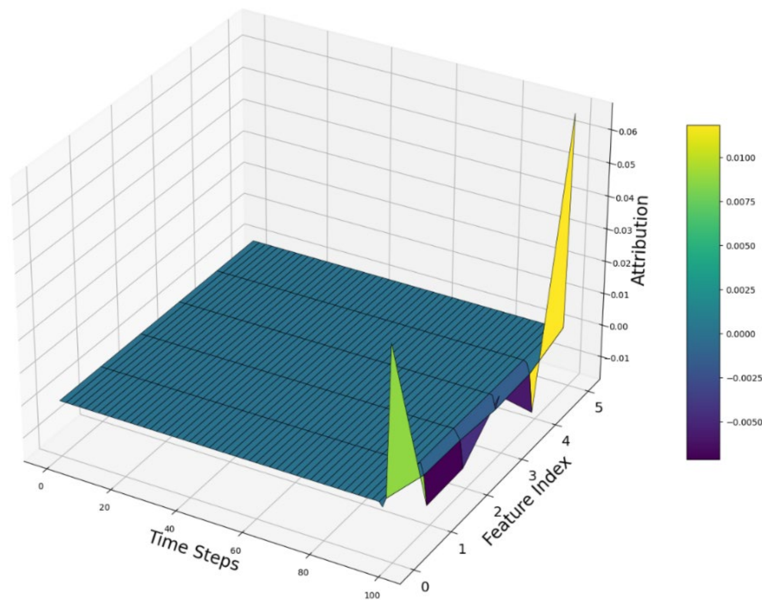


Figure 21 3D Surface of feature importance over time using Integrated Gradient

Using the AOPCR, we observe that a larger area corresponds to a steeper increase in RMSE as important features are removed, which indicates better explanations. We compared the performance of different explanation methods based on Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) metrics. The results indicate that the Integrated Gradients method consistently yields the lowest RMSE and MAE values, suggesting superior

fidelity in capturing feature importance. As discussed earlier, these metrics are computed by systematically removing the most influential features identified by each explanation method and evaluating the resulting impact on model predictions. Specifically, this approach assesses whether the model's performance deteriorates (higher error) or improves (lower error) after feature removal, directly relating to the concepts of Comprehensiveness and Sufficiency. A more comprehensive explanation implies that removing important features significantly degrades prediction accuracy, while sufficiency ensures that the selected features alone contain enough information to maintain performance. Thus, the lower error metrics associated with Integrated Gradients highlight its effectiveness in accurately identifying critical features that govern the model's behavior.

5.5 Integration of Sustainability-Aware Scheduling Models for Federated CECC

5.5.1 Introduction

The evolution of Cloud–Edge Computing Continuum (CECC) brings transformative capabilities for modern digital infrastructures, enabling support for latency-sensitive applications such as smart city services, autonomous vehicles, and real-time analytics. These applications require ultra-low response times, high availability, and efficient resource utilization. However, this technological progress is accompanied by significant challenges, particularly concerning energy efficiency and environmental sustainability. With data centers and edge devices projected to contribute over 5% of global carbon emissions by 2025, the need for energy-conscious resource management becomes critical.

Federated CECC environments add further complexity due to their heterogeneous nature, integrating diverse computing nodes with varying capacities, energy profiles, and intermittent renewable energy availability. Maintaining strict Service Level Agreements (SLAs) in such dynamic and distributed systems is increasingly difficult. Traditional scheduling approaches often fall short, lacking the flexibility to consider energy fluctuations, connectivity limitations, and sustainability targets.

To address these challenges, the integration of multi-objective optimization frameworks into resource management procedures has gained attention. These frameworks aim to balance energy consumption, SLA compliance, and carbon footprint reduction. Building on this approach, a sustainability-aware scheduling model has been developed and integrated into the AC^3 CECCM, enabling dynamic, green-oriented resource allocation while maintaining service quality across federated CECC infrastructures.

5.5.2 State of the Art in Sustainability-Aware Scheduling

Sustainability-aware scheduling has emerged as a critical research domain in response to the increasing energy consumption and environmental footprint of cloud and edge infrastructures. The federated Cloud-Edge Continuum (CECC) introduces additional challenges due to the heterogeneity of resources, diverse service-level requirements, and the need to account for renewable energy availability.

Early work addressed energy reduction in mobile edge computing (MEC) and collaborative cloud-edge systems. For example, Gao et al. [10] leveraged MIMO techniques to lower energy consumption and task offloading delays, while Wang et al. [11] optimized resource allocation in cloud–edge elastic optical networks to reduce energy demand and blocking probability. Similarly, Chen et al. [12] and Liu and Liu [13] proposed scheduling and offloading strategies to support energy-constrained or resource-intensive workloads. While these studies achieved improved energy efficiency, they largely overlooked SLA compliance and renewable energy integration.

Several works employed Reinforcement Learning (RL) to adapt scheduling policies dynamically. Hu et al. [14] introduced a double deep Q-learning approach for water conservancy edge networks, while Zheng et al. [15] and Tang et al. [16] applied RL for workload and caching optimization in MEC systems. Ding et al. [17] presented a Q-learning framework for cloud computing task scheduling. These methods showed reduced energy use and improved CPU utilization, but lacked adaptability to heterogeneous federated infrastructures and did not explicitly enforce SLA requirements.

To maintain service quality, researchers have proposed SLA-aware optimization techniques. Jiang et al. [18] used game theory to balance resource utilization and SLA adherence in MEC environments, while Kumar et al. [19] introduced SLA-aware scheduling in cloud systems using optimization models. Vehicular edge-cloud frameworks, such as Li et al. [20], ensured strict latency guarantees but remained specific to narrow use cases. Overall, many of these models did not incorporate sustainability factors such as renewable energy or carbon footprint.

A smaller body of work integrates renewable energy awareness into scheduling. Chen et al. [21] proposed an energy-harvesting framework that leverages renewable sources, while Mills et al. [22] combined AI with microgrid optimization to achieve near net-zero emissions. Wang et al. [23] explored sustainable energy scheduling in edge and cloud infrastructures. Although these solutions highlight the benefits of renewables, they often lack SLA management or are limited to domain-specific scenarios (e.g., microgrids or power systems).

Recent studies have investigated multi-objective optimization methods that jointly address energy, SLA, and sustainability goals. Liu and Liu [13] proposed evolutionary computation-based offloading algorithms, while Chen et al. [21] provided a comprehensive review of collaborative scheduling approaches. These frameworks highlight the promise of evolutionary algorithms such as NSGA-II for exploring trade-offs across competing objectives. However, they remain underexplored in large-scale federated CECC contexts, where scalability and real-time adaptability are critical.

Despite these advancements, existing approaches share several limitations:

- **Lack of holistic integration:** Most focus on either energy efficiency, SLA adherence, or renewable energy, but not all simultaneously.
- **Limited scalability:** Many models are restricted to small-scale or domain-specific environments, making them less practical for federated CECC systems.
- **Weak handling of trade-offs:** Approaches often optimize a single weighted objective, providing limited flexibility to operators who must balance carbon footprint, SLA, and energy consumption.

We directly address these gaps through a hybrid sustainability-aware scheduling framework. By combining MILP-based optimization for globally optimal offline planning with NSGA-II-based evolutionary methods for scalable runtime adaptation, to ensure:

- SLA guarantees alongside sustainability objectives,
- explicit integration of renewable energy availability and carbon footprint penalties,
- flexibility through Pareto-optimal trade-offs that can be exploited by the CECCM.

This approach represents a step forward from the state of the art, enabling federated CECC environments to achieve both green operation and SLA compliance under dynamic real-world conditions.

5.5.3 Main Framework

The proposed framework introduces an integrated scheduling model that addresses three critical objectives in federated Cloud–Edge environments: energy efficiency, SLA adherence, and carbon footprint management. Unlike traditional resource allocation approaches, which often treat these objectives in isolation, this framework consolidates them within a unified optimization strategy. By doing so, it enables intelligent, adaptive scheduling decisions that balance performance guarantees with sustainability goals.

At the core of the framework is a detailed modeling of both tasks and infrastructure nodes. Applications are decomposed into tasks or microservices, each characterized by specific resource demands, such as CPU and memory, along with stringent service-level requirements including latency, reliability, and throughput. These tasks are further constrained by temporal factors, such as defined arrival and deadline times, which must be respected in the scheduling process.

The computing infrastructure is modeled as a heterogeneous collection of nodes spanning edge, fog, and cloud layers. Each node is defined by its baseline power consumption when active, maximum resource capacities, and indicators for renewable energy availability. Furthermore, nodes are equipped with battery storage capabilities that allow for local buffering of renewable energy, adding an additional layer of complexity and opportunity for green energy exploitation. This detailed modeling ensures that the system reflects the real-world diversity and constraints typical of federated Cloud–Edge deployments.

Service Level Agreements (SLAs) are incorporated into the framework through a dual mechanism. Nodes that cannot meet critical SLA requirements, such as latency thresholds for time-sensitive tasks, are excluded from consideration for those tasks. However, the model also allows for flexibility through the use of penalty functions. In cases where minor SLA deviations occur—such as slight latency overshoots or marginal reductions in throughput—these are quantified and penalized, increasing the overall scheduling cost but preserving operational feasibility. This approach avoids infeasibility traps common in hard-constraint-only models, while still enforcing quality of service commitments. A distinguishing feature of the framework is its explicit treatment of renewable energy and carbon footprint considerations. Each node is evaluated per time slot for renewable energy availability, encouraging the scheduler to assign tasks to green-powered nodes when possible. Non-renewable energy consumption is penalized based on the carbon intensity of the power source, modeled through emission factors. This design ensures that energy consumption is not treated as a flat cost but is contextually weighted to reflect environmental impact, driving the system towards more sustainable operation.

The model also incorporates task migration capabilities, allowing for dynamic workload reallocation across nodes and time slots. Migration decisions are made with full consideration of the associated energy overhead for data transfers and the potential SLA impacts from migration-induced delays or downtime. By modeling these trade-offs explicitly, the framework supports adaptive scheduling while avoiding hidden operational costs.

The optimization process is approached through a hybrid algorithmic strategy that combines the strengths of exact and evolutionary optimization methods.

5.5.3.1 Mixed Integer Linear Programming (MILP):

- Captures the scheduling problem in a deterministic mathematical form, with binary and continuous variables representing task assignments, node activations, migrations, and battery states.
- Objective functions minimize energy, carbon footprint, and SLA penalties under strict capacity, latency, reliability, and renewable energy constraints.
- Solved using the Gurobi optimizer, MILP produces globally optimal schedules under fixed inputs, making it ideal for design-time planning and benchmarking.

Its limitation lies in scalability, as computational complexity grows rapidly with larger federated deployments.

5.5.3.2 Non-dominated Sorting Genetic Algorithm II (NSGA-II):

- A multi-objective evolutionary algorithm that evolves a population of candidate schedules using crossover, mutation, and non-dominated sorting.
- Simultaneously minimizes three objectives: energy consumption, SLA violations, and carbon footprint.
- Constraint violations (e.g., infeasible task placement or SLA breaches) are handled through penalty functions in the fitness evaluation.
- Returns a Pareto front of diverse non-dominated solutions, offering operators flexibility to select trade-offs depending on real-time conditions.

Configured in AC3 experiments with a population of 100, 50 generations, crossover probability of 0.9, and mutation probability of 0.1, NSGA-II demonstrated scalability and adaptability in dynamic CECC environments.

The two algorithms play complementary roles within the AC3 framework:

Table 4 Comparison of Exact and Evolutionary Optimization Approaches for CECC Scheduling

Aspect	MILP (Exact)	NSGA-II (Evolutionary)
Optimization type	Deterministic, globally optimal	Stochastic, Pareto front exploration
Objective handling	Weighted sum (single solution)	Multi-objective (diverse solutions)
SLA enforcement	Hard constraints with strict guarantees	Penalty-based, soft feasibility
Renewable/carbon integration	Explicit constraints and penalties	Explicit in objectives
Scalability	Limited to small/medium problem sizes	Scales to large, dynamic environments
Use case	Offline planning, benchmarking	Runtime adaptation, real-time decisions
Optimization type	Deterministic, globally optimal	Stochastic, Pareto front exploration

Thus, MILP establishes baseline optimality for controlled scenarios, while NSGA-II provides runtime adaptability and scalability. Together, they form a robust hybrid solution that ensures the CECCM can operate sustainably while maintaining SLA guarantees across federated Cloud–Edge infrastructures.

To formalize these objectives, the resource management problem is framed as a multi-objective optimization task. The central goal is to minimize total energy consumption, reduce SLA violations, and lower the overall carbon footprint through smart workload placement and resource activation strategies. This is captured in the following combined objective function:

$$Z = w_1 * E_{total} + (1 - w_1) * SLA_{violation}$$

where $\mathbf{E_{total}}$ represents the total energy consumption and carbon-weighted cost, $\mathbf{SLA_{violation}}$ aggregates penalties for service-level breaches, and $\mathbf{w_1}$ serves as a weighting factor balancing sustainability and performance priorities. This formulation allows for adjustable prioritization depending on operational goals, making the framework adaptable to various deployment scenarios.

The optimization process follows a hybrid method that combines an exact MILP model for design-time planning with an evolutionary NSGA-II procedure for runtime adaptation. Below are the core decision variables, objectives, and constraints used in both, aligned with our article's notation.

- **Decision variables (per task i, node j, time slot t)**

$$X_{ijt} \in \{0,1\} \text{ (task i runs on node j at t)}$$

$$Y_{jt} \in \{0,1\} \text{ (node j active at t)}$$

$$M_{ijkt} \in \{0,1\} \text{ (task i migrates j } \rightarrow \text{ k at t)}$$

$$S_{jt} \text{ (battery level)}, B_{jt}^{ch}, B_{jt}^{dis} \text{ (charge/discharge)}$$

- **Power / energy model**

Total power of node jj at time tt = baseline + task load – battery offset:

$$P_{jt} = P_j^0 Y_{jt} + \sum_i \partial_{it} X_{ijt} - B_{jt}^{dis}$$

Migration energy cost = tasks moved × distance × per-unit migration overhead:

$$E_{migrate} = \sum_t \sum_j \sum_{k \neq j} \sum_i M_{ijkt} \; d(j,k) \; E_{unit}^{mig}$$

- **Carbon footprint and renewables**

Non-renewable energy use weighted by emission factor (kg CO_2/unit):

$$CF_{total} = \sum_t \sum_j (1 - U_{jt}) (P_{jt} - B_{jt}^{dis}) EF_j$$

- **Battery dynamics and bounds**

Battery evolves each slot by charging and discharging, bounded by capacity:

$$S_{j,t+1} = S_{jt} + B_{jt}^{ch} - B_{jt}^{dis}, \quad 0 \leq S_{jt} \leq S_j^{max}$$

Discharge limited by current state and charge limited by excess renewable availability:

$$0 \leq B_{jt}^{dis} \leq \delta S_{j,}, \quad 0 \leq B_{jt}^{ch} \leq \max \{0, \gamma RE_{jt} - P_{jt}\}$$

- **Assignment, capacity, and SLA constraints**

Each task is assigned to at most one node at a time and Node resource capacity (CPU, memory) cannot be exceeded:

$$\sum_j X_{ijt} \leq 1 \quad (\forall i, t), \quad \sum_i Re_i X_{ijt} \leq C_j \quad (\forall j, t)$$

Task latency ≤ SLA threshold, reliability ≥ SLA, throughput ≥ SLA:

$$L_{ij} X_{ijt} \leq L_i^{SLA}, \quad R_{ij} X_{ijt} \geq R_i^{SLA}, \quad T_{ij} X_{ijt} \geq T_i^{SLA}$$

- **SLA penalty aggregation**

Quantifies how much tasks exceed latency, fall short on reliability, or throughput:

$$SLA_{violation} = \sum_{t,j,i} (\max \{ 0, L_{ij} - L_i^{SLA} \} + \max \{ 0, R_i^{SLA} - R_{ij} \} + \max \{ 0, T_i^{SLA} - T_{ij} \})$$

- **Migration consistency**

Migration occurs if a task runs on j at t and on k at t+1:

$$M_{ijkt} \geq X_{ijt} + X_{ik,t+1} - 1 \qquad (\forall i, j = k, t < T)$$

MILP:

The MILP minimizes a weighted energy–SLA objective while accounting for carbon and migration costs and allowing price-aware energy:

$$\min Z = w_1 * E_{total} + (1 - w_1) * SLA_{violation}$$

$$E_{total} = \sum_{j,t} (P_{jt} \ \ Price_{jt}) - \alpha \ \sum_{j,t} RE_{jt} + \beta CF_{total} + \lambda E_{migrate}$$

subject to all constraints above (assignment/capacity, SLA, battery/renewables, migration).

NSGA-II:

For online scheduling under workload and energy variability, we evolve candidate schedules (chromosomes encoding complete task–node–time maps) to simultaneously minimize:

$$\min (E_{total} , SLA_{violation} , CF_{total})$$

This framework brings together advanced task and resource modeling, SLA enforcement, sustainability-aware energy management, and multi-objective optimization techniques into a cohesive scheduling solution for federated Cloud–Edge environments. It directly enhances the CECCM's capabilities, aligning with the broader goals of WP4 in enabling efficient, green, and SLA-compliant resource management.

Within the AC3 architecture, the proposed scheduling algorithms are embedded in the CECCM, which forms the core of WP4. The MILP solver operates in the Decision Module at design time, producing baseline schedules that quantify trade-offs between SLA compliance, energy efficiency, and carbon footprint. These optimal schedules act as references and benchmarks for system-wide planning. The NSGA-II implementation is used within the runtime scheduling component of the CECCM, where it leverages monitoring data on workload arrivals, renewable availability, and SLA performance to generate adaptive schedules in real time. The scheduling outputs are passed to the deployment orchestrator, which enforces them across federated edge, fog, and cloud nodes, ensuring that task placements and possible migrations are consistently executed.

Through this link, the hybrid optimization framework becomes an integral part of AC3's operational loop: monitoring data informs the decision logic, the scheduling algorithms generate optimized policies, and the orchestrator enforces these policies on the infrastructure. This connection ensures that the proposed scheduling solutions are not stand-alone models but key enablers of AC3's sustainability-aware and SLA-compliant resource management objectives.

5.5.4 Simulation and Results: Sustainability-Aware Scheduling in Federated CECC

This section presents the simulation setup and results of the sustainability-aware scheduling model integrated into the CECCM resource management procedures. The objective is to evaluate the effectiveness of multi-objective optimization in balancing energy efficiency, carbon footprint reduction, and SLA compliance in federated Cloud–Edge environments.

5.5.4.1 Simulation Setup

The experimental scenario involves scheduling 100 tasks across 10 heterogeneous computing nodes, representing a mix of edge, fog, and cloud resources. The time horizon is divided into 8 discrete time slots. Each task is defined by its required execution window (arrival and deadline), CPU capacity demand, and service-level constraints, specifically latency and reliability requirements.

The infrastructure nodes are characterized by their compute capacity, baseline power consumption, and emission factors representing carbon intensity. Additionally, certain time slots offer renewable energy

availability for specific nodes, while battery storage dynamics are considered to buffer renewable energy and offset non-renewable usage.

The scheduling problem is formulated as a multi-objective optimization, aiming to minimize total energy consumption, reduce SLA violations, and lower the carbon footprint. These objectives are combined in a weighted objective function expressed as:

$$Z = w_1 * E_{total} + (1 - w_1) * SLA_{violation}$$

where E_{total} represents total energy usage and carbon-weighted costs, $SLA_{violation}$ aggregates penalties for service-level breaches, and w_1 balances sustainability and performance priorities.

A Mixed-Integer Linear Programming (MILP) solver is used to compute an optimal scheduling plan for this scenario.

5.5.4.2 Optimization Results and Analysis

To evaluate the effectiveness of the proposed scheduling model, we considered two internal baselines. The first is the MILP formulation itself, which provides an exact globally optimal solution for small-scale instances and therefore establishes the theoretical lower bound for energy consumption, carbon footprint, and SLA violations. The second is a greedy scheduling strategy, in which tasks are allocated to the first available node that satisfies SLA constraints without any awareness of energy consumption or carbon intensity. This baseline reflects the behavior of sustainability-agnostic schedulers and highlights the additional value of explicitly integrating sustainability into the decision process.

Against these baselines, the MILP solver produced an optimal solution with a final objective value of 512.95. This result comprises three key cost components: total energy consumption of 955.00 Wh, SLA violation penalties of 70.91, and a carbon footprint of 226.38 CO_2 units. The energy consumption includes 292.86 Wh attributed to baseline node activity, reflecting the overhead of keeping nodes powered on across multiple time slots. These results are summarized in Table 5.

Table 5 Key Metrics from the Optimized Solution

Metric	Value	Notes
Objective Value	512.95	Weighted sum of energy, SLA, and carbon components
Total Energy	955.00	Includes baseline and task-specific CPU load
Carbon Footprint	226.38	Reflects non-renewable energy consumption
SLA Violation	70.91	Aggregated latency and reliability penalties
Baseline Energy	292.86	Idle consumption of powered-on nodes

A sample of task assignments is presented in Table 4, showing how the first 10 tasks are allocated across nodes and time slots. Node 7 emerges as the primary computing node due to its high capacity and reliability, despite its relatively high latency, which contributes to SLA penalties.

Table 6 Sample of Task Assignments (Tasks 1–10)

Task	Node ID	Time Slot
1	7	2
2	7	6
3	7	0
4	7	2
5	7	4
6	5	2
7	7	1
8	7	1
9	7	0
10	7	3

These experiments confirm both the validity and the effectiveness of our approach. The MILP optimum establishes the correctness of the formulation by providing globally optimal trade-offs under fixed conditions. In subsequent experiments, the NSGA-II algorithm is evaluated against this optimum to demonstrate that it converges close to the lower bound while scaling to larger problem sizes where MILP becomes infeasible. Moreover, when compared with the greedy baseline, the sustainability-aware framework consistently reduces both energy consumption and carbon emissions while preserving SLA compliance. This dual comparison—against an exact optimal reference and a naive sustainability-agnostic scheduler provides a strong validation of the proposed hybrid scheduling model.

Figure 22 illustrates the overall distribution of tasks across nodes. Node 7 clearly dominates the allocation due to its large capacity, while Nodes 3, 5, and 10 handle a smaller share of tasks. Other nodes remain underutilized because of resource limitations, energy constraints, or the inability to meet SLA requirements.

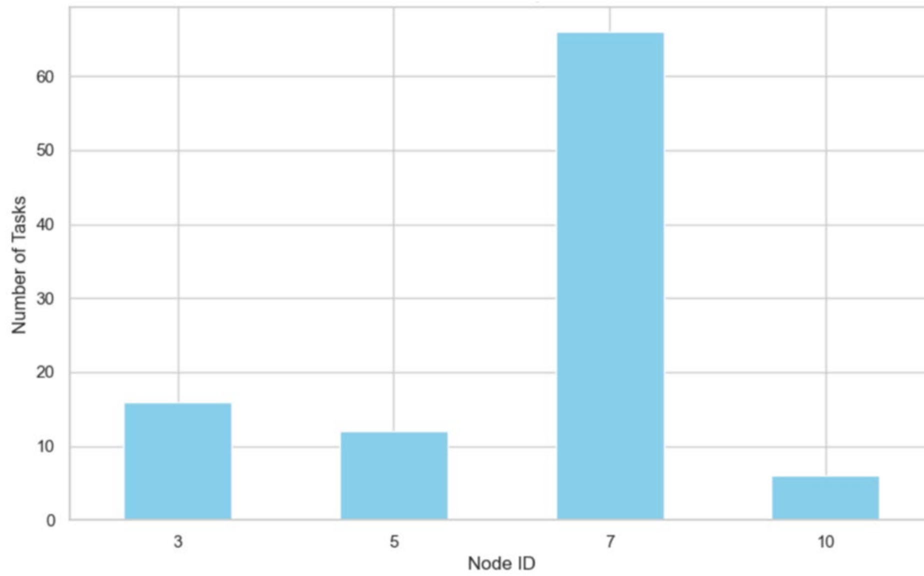


Figure 22 Number of Tasks per Node

Similarly, Figure 23 shows the total CPU resource demand served by each node. This visualization reinforces the trend observed in task assignments, with Node 7 being the primary consumer of compute resources.

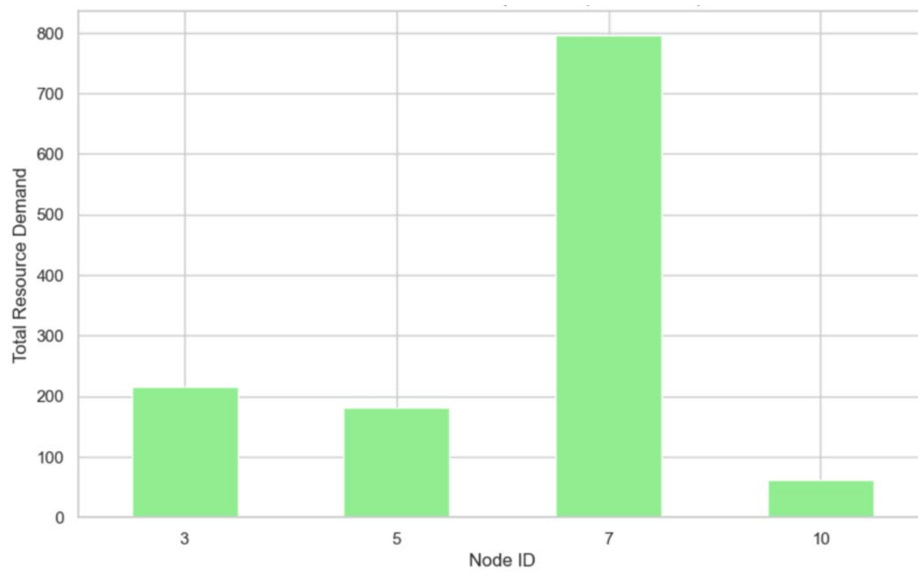


Figure 23 Total Resource Demand per Node

The temporal distribution of task assignments is shown in Figure 24. Tasks are spread across time slots, with higher concentrations in certain slots, possibly aligning with periods of renewable energy availability or optimal scheduling windows.

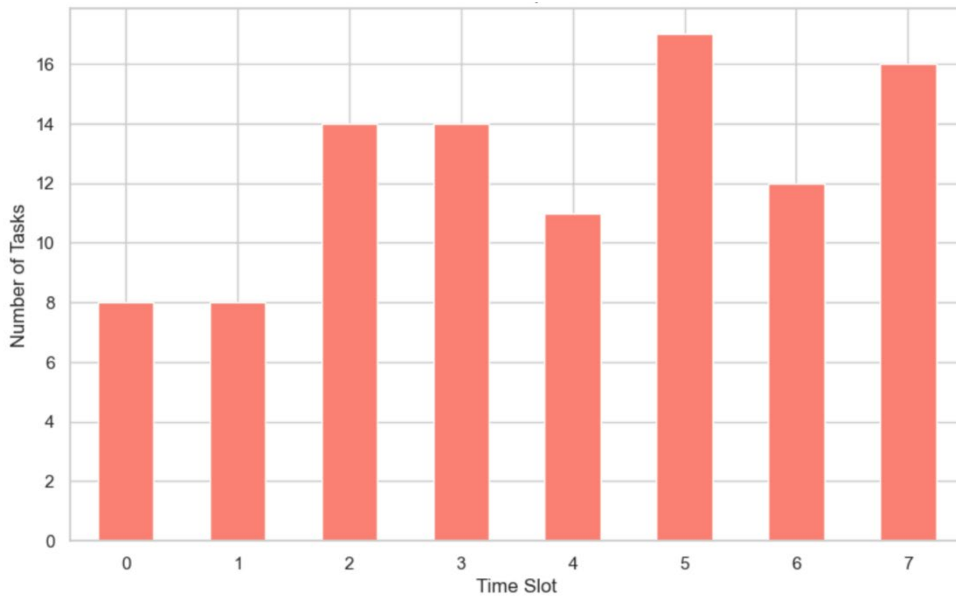


Figure 24 Number of Tasks per Time Slot

Figure 25 complements this view by displaying the total resource demand per time slot, highlighting how workload is distributed to optimize both performance and energy efficiency.

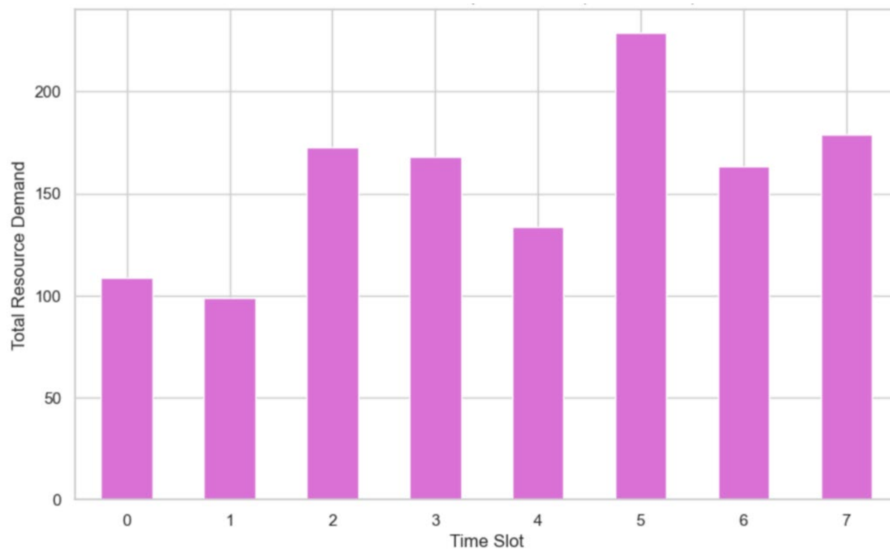


Figure 25 Total Resource Demand per Time Slot

5.5.4.3 *SLA Performance and Carbon-Aware Scheduling*

The effectiveness of the scheduling model in maintaining SLA compliance is analyzed through Figure 26 and Figure 27. Figure 26 presents the distribution of latency violations, revealing that most tasks experience no or

minimal SLA breaches. However, the reliance on Node 7 introduces higher-latency paths, contributing to a long-tail distribution of minor violations.

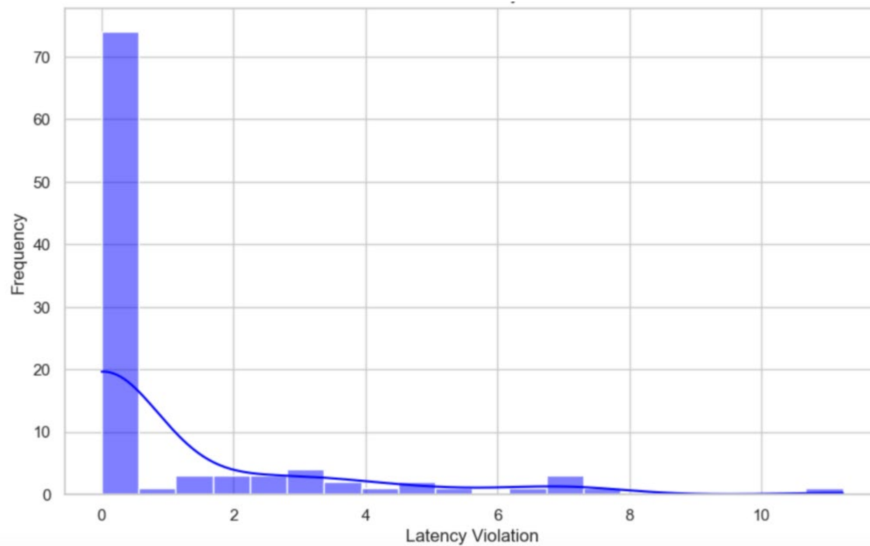


Figure 26 Histogram of Latency Violations

Figure 27 shows the distribution of reliability violations. Overall, reliability penalties remain low due to the high baseline reliability of most nodes, indicating that the primary SLA challenges stem from latency rather than reliability.

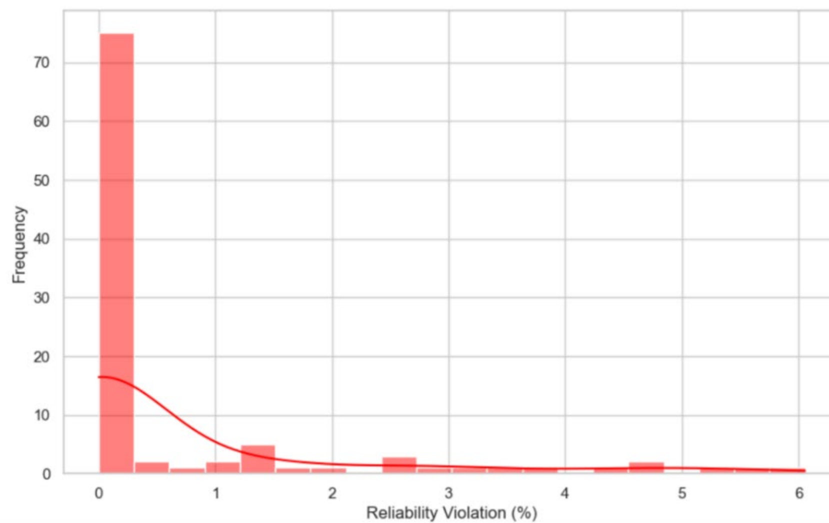


Figure 27 Histogram of Reliability Violations

An important aspect of the model is its ability to leverage renewable energy sources to reduce the carbon footprint. Figure 28 illustrates the number of tasks scheduled in time slots where renewable energy is available.

Approximately 30% of tasks benefit from renewable alignment, contributing significantly to the reduction of non-renewable energy consumption and associated emissions.

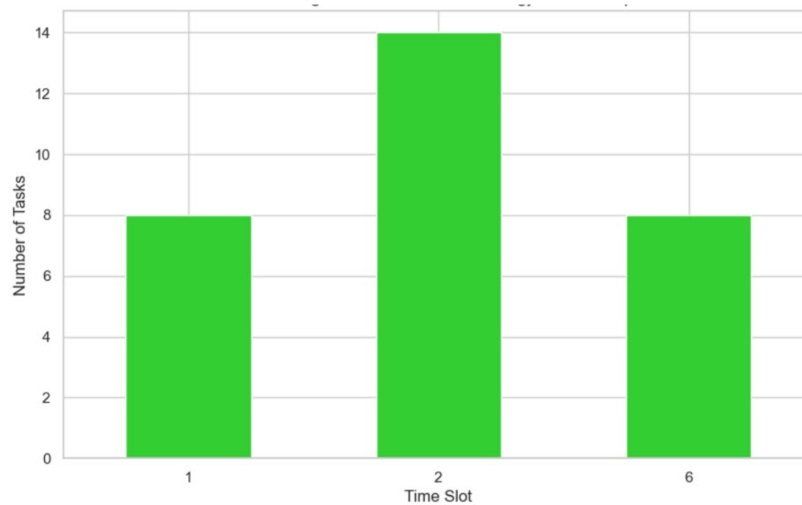


Figure 28 Tasks Assigned in Renewable-Friendly Time Slots

The simulation demonstrates that the proposed scheduling model effectively balances energy efficiency, SLA adherence, and carbon footprint reduction. While task consolidation on high-capacity cloud nodes improves resource utilization and energy efficiency, it introduces SLA trade-offs, primarily in latency performance. The model manages these trade-offs through penalty functions, maintaining service quality without sacrificing environmental objectives.

The alignment of task scheduling with renewable energy availability has a direct and measurable impact on carbon footprint reduction. Although baseline power consumption remains significant, reflecting the operational need to keep nodes active, the model's intelligent task placement strategies mitigate its impact.

5.5.5 NSGA-II-Based Multi-Objective Scheduling

In addition to the Mixed-Integer Linear Programming (MILP) approach previously discussed, we evaluated a metaheuristic optimization method based on the Non-Dominated Sorting Genetic Algorithm II (NSGA-II). This method is well-suited for addressing multi-objective problems where conflicting goals—such as energy efficiency, SLA compliance, and carbon footprint reduction—must be balanced simultaneously. Unlike MILP, NSGA-II does not rely on scalarizing objectives into a single weighted sum but rather explores a Pareto front of solutions, enabling more flexible and adaptable decision-making.

5.5.5.1 Algorithm Setup

The NSGA-II simulation retained the same scenario configuration used for the MILP evaluation. Specifically, 100 tasks were scheduled across 10 heterogeneous computing nodes over 8 discrete time slots. Each task was constrained by latency and reliability requirements, while the nodes exhibited varying compute capacities, baseline power consumption, and renewable energy availability.

The optimization objectives remained focused on minimizing three key metrics: total energy consumption, SLA violations, and carbon footprint. To ensure feasibility, task placement constraints and SLA violations were managed through penalty functions integrated into the fitness evaluation. The algorithm was configured with a

population size of 100 and executed over 50 generations, allowing it to gradually converge towards a diverse and feasible Pareto front of solutions.

5.5.5.2 Evolutionary Results

At the conclusion of the NSGA-II run, the best observed solution achieved a total energy consumption of 2344.61 Wh, with SLA violations amounting to 116.20, and a carbon footprint of 371.19 CO_2 units. These results, summarized in Table 5, reflect the trade-offs inherent to the NSGA-II approach. While energy consumption and SLA violations are higher compared to the MILP solution, NSGA-II provides a broader exploration of feasible trade-offs, offering alternative configurations where decision-makers can prioritize different objectives.

Table 7 NSGA-II Best Observed Values

Metric	Value
Total Energy	2344.61
SLA Violations	116.20
Carbon Footprint	371.19

5.5.5.3 Visualization of Trade-Offs

The performance of NSGA-II across the three objectives is visualized through a series of Pareto front plots. Figure 29 illustrates the trade-off between energy consumption and SLA violations. As expected, reducing energy consumption tends to increase SLA penalties, reflecting the need to consolidate tasks on high-capacity but high-latency nodes, similar to observations made in the MILP scenario.

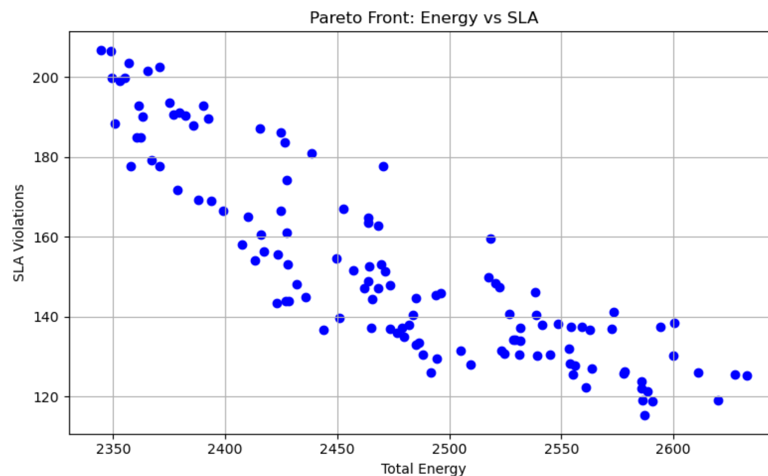


Figure 29 Energy vs SLA Violations Pareto Front

In Figure 30, the relationship between energy consumption and carbon footprint is presented. The plot shows that while lower energy consumption generally correlates with a reduced carbon footprint, the relationship is not strictly linear. This is due to the variability of renewable energy availability across time slots and nodes, which can influence carbon intensity independent of total energy usage.

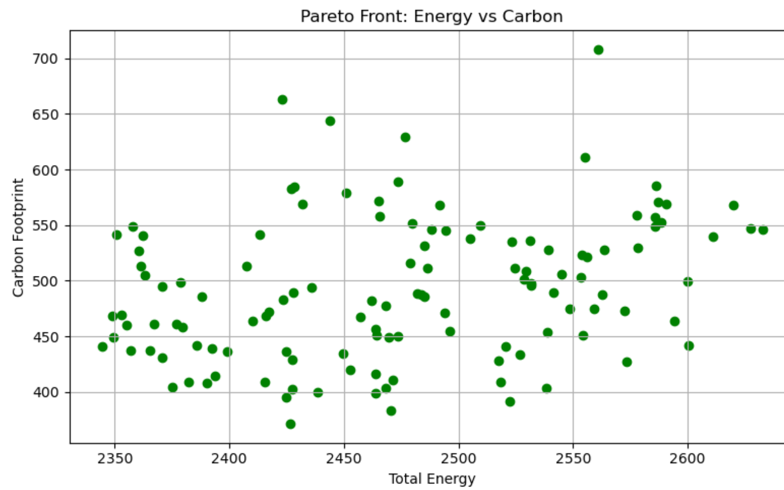


Figure 30 Energy vs Carbon Footprint Pareto Front

Figure 31 explores the trade-off between SLA violations and carbon footprint. It highlights clusters of feasible solutions where moderate SLA compliance is achieved without incurring excessive carbon impact. The figure underscores the advantage of NSGA-II in identifying configurations that exploit renewable availability while maintaining acceptable service levels.

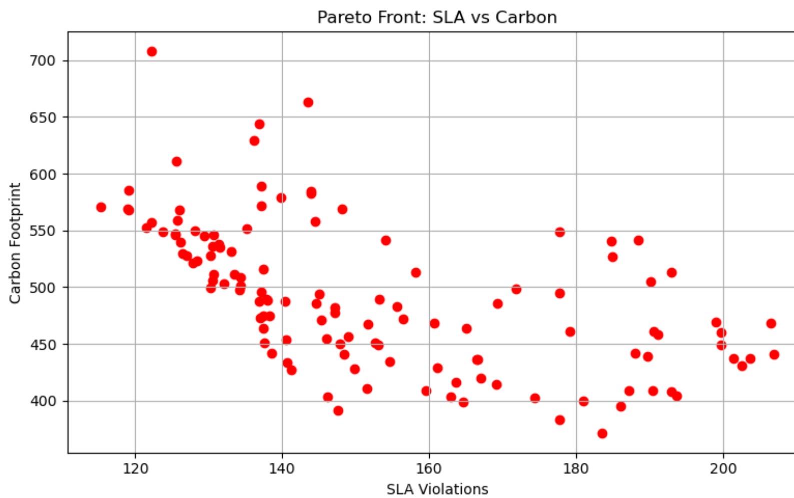


Figure 31 SLA Violations vs Carbon Footprint Pareto Front

To provide a comprehensive view of the solution space, the complete three-dimensional Pareto front is shown in Figure 32. This visualization captures the multidimensional trade-offs across energy consumption, SLA violations, and carbon footprint, offering a holistic perspective on the possible optimization outcomes.

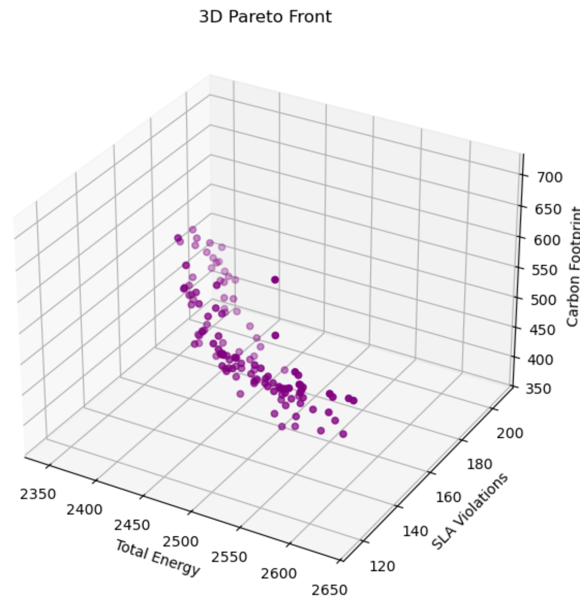


Figure 32 3D Pareto Front (Energy, SLA, Carbon)

5.5.5.4 Comparative Analysis with MILP

A detailed quantitative comparison between MILP and NSGA-II is presented in Table 8. The MILP solver, acting as an exact optimizer, achieved a total energy consumption of 955.00 Wh, consisting of 292.86 Wh of baseline idle power and 662.14 Wh of dynamic CPU energy. This consolidation strategy also allowed approximately 30% of tasks to be aligned with renewable-friendly slots, thereby reducing the overall carbon footprint to 226.38 CO_2 units. SLA penalties were limited to 70.91, primarily due to Node 7's higher latency profile.

In contrast, the best observed NSGA-II solution consumed 2344.61 Wh, with SLA penalties of 116.20 and a carbon footprint of 371.19 CO_2 units. The lower renewable utilization compared with MILP contributed to the higher emissions. However, NSGA-II offers scalability and flexibility, producing an entire Pareto front that illustrates explicit trade-offs between objectives. Moreover, the evolutionary approach runs in seconds to minutes for the tested configuration (100 tasks, 10 nodes, 8 slots), while the MILP formulation requires hours of solver time to converge.

Table 8 Comparative Evaluation: MILP vs. NSGA-II Approaches

Criteria	MILP (Exact Optimization)	NSGA-II (Metaheuristic Optimization)
Total Energy	955 Wh	2345 Wh
SLA Violations	70.91	116.20
Carbon Footprint	226 CO_2	371 CO_2
Baseline Energy	292.86 Wh	600 Wh
CPU Utilization	72%	58%
Active Nodes	4	7

Renewable Usage Ratio	30% tasks in green slots	Lower (less alignment observed)
Solver Runtime	Hours (Gurobi)	Seconds–minutes (50 gens × pop 100)
Solution Diversity	Single optimal point	Multiple Pareto-optimal solutions

5.6 CPU usage Prediction with Local Global Models

To improve the accuracy of resource consumption prediction across dynamic and heterogeneous cloud–edge environments, we propose a novel modeling approach based on **Global Local Models for Time Series Prediction**. These models combine shared global trends with localized node-specific behaviors, enabling more precise CPU usage forecasts and smarter orchestration decisions. This prediction capability directly advances the goals of AC3 by strengthening the application and resource management layer of the architecture. It enables proactive scaling, optimal placement, and efficient scheduling decisions through accurate and context-aware CPU usage forecasts. In practice, the algorithm will be integrated into the AI-based CEC resource profile component, where the described CPU prediction mechanism extends the resource profile with forward-looking availability information, allowing more intelligent and adaptive orchestration. Moreover, the Local Global modeling approach is extensible to other resource and application-level metrics commonly collected in cloud–edge environments, such as memory usage, latency, and network throughput, further broadening its impact on intelligent orchestration and system optimization.

A key challenge remains: training such models in the lack of real-world datasets from edge deployments (expressing local behavior) is extremely difficult. To address this, we developed the **Cloud Edge Continuum Emulator (CEC-EMULATOR)**—an emulation platform that generates realistic time-series and topological data by simulating multi-zone deployments, heterogeneous node capabilities, and spatially distributed traffic. This provides a practical and scalable solution for training and benchmarking predictive models in the absence of large-scale production data. For the reasons described above, we test and train the proposed models only on this synthetically generated dataset, validating its efficacy and its potential to be integrated in the AC3 platform as a core CPU prediction component. Furthermore, the **CEC-EMULATOR** can also be used as a digital twin sandbox where application developers can test their applications and create more suitable configuration mappings with the LMS. This functionality can be enhanced even further by incorporating real data-driven traces from the AC3 platform inside the emulator. In the following sections, we describe:

- First, the global local prediction paradigm, showing forecasting results from CPU metrics collected from the emulator platform.
- Second, we give an overview of the emulator platform and describe its capabilities.

5.6.1 Global Local Time Series Prediction

Building upon the CEC-EMULATOR's rich dataset of CPU utilization across distributed microservice meshes, we developed a Global-Local Model that leverages both node-specific characteristics and global temporal patterns to predict resource consumption. This hybrid approach addresses the heterogeneous nature of the cloud-edge continuum by learning localized node behaviors while benefiting from shared knowledge across the entire system.

5.6.1.1 Model Inputs

The Global-Local Model consumes time-series data extracted from CEC-EMULATOR's multi-zone deployments, specifically designed to capture the complex spatio-temporal dynamics of CPU usage across heterogeneous nodes. The input data structure consists of:

- **Context Window**: Historical CPU usage values spanning a configurable window
- **Node Identification**: String identifiers for each node (e.g., "cloud-node-01", "edge-site-1-node-02")
- **Temporal Resolution**: Per-minute CPU core utilization measurements normalized to [0,1] range
- **Data Format**: Sliding window sequences where each sample contains:
    - Input: (batch size, context length) tensor of historical CPU values
    - Target: (batch size, prediction length) tensor of future CPU values to predict
    - Node Names: List of corresponding node identifiers for each batch sample

The dataset preprocessing pipeline implements global normalization across all nodes to ensure consistent scaling, temporal ordering by node and timestamp, and configurable train/validation/test splits with chronological boundaries. The `NodeCPUDataset` class handles the sliding window generation, creating overlapping sequences that preserve temporal continuity while maximizing training data utilization.

Unlike traditional time-series models that treat each node independently, our approach explicitly models the node identity as a first-class input feature. This design choice enables the model to learn node-specific behaviors (local component) while sharing temporal patterns across the infrastructure (global component).

5.6.1.2 Global-Local Model Architecture

The Global-Local Model implements a hybrid architecture that combines learnable node embeddings with sophisticated time-series processing to capture both spatial and temporal dependencies in CPU usage patterns.

Local Component: Each node in the infrastructure is represented by a learnable embedding vector of configurable dimension. These embeddings serve as the "local" component, capturing node-specific characteristics such as:

- Hardware capabilities and resource constraints
- Typical workload patterns and usage profiles
- Spatial location effects (cloud vs. edge deployment)
- Service placement preferences and affinity rules

The node embeddings are implemented as a standard embedding layer:

$$e_i = \text{Embedding}(i) \in R^{d_e}$$

where $i \in \{1,2, \dots , \mathrm{N}\}$ is the node index, N is the total number of nodes, and d_e is the embedding dimension.

Time Series Encoder: The temporal processing component transforms raw CPU usage sequences into rich feature representations. Two architectural variants are supported:

- **Basic Model**: Utilizes fully-connected layers with configurable activation functions (ReLU, GELU, Tanh) and dropout for regularization:
- **Advanced Model**: Employs transformer-based architecture with multi-head attention, positional encoding, and layer normalization for enhanced temporal modeling capabilities:

Global Fusion Network: The global component combines the encoded time-series features with node embeddings through concatenation, followed by a multi-layer prediction head. This fusion enables the model to make node-aware predictions that leverage both historical patterns and spatial context. For the advanced model, cross-attention mechanisms further enhance the interaction between temporal and spatial representations before final prediction generation.

Model Variants: Blending the above components together we can get the following family of global local models:

- **Basic Global-Local Model**: ~26K parameters with simple feedforward architecture
- **Advanced Global-Local Model**: ~109K parameters with transformer-based temporal processing

- **Regularization**: Optional KL-divergence regularization on node embeddings to prevent overfitting

5.6.1.3 Training & Evaluation

The training methodology employs PyTorch Lightning for scalable and reproducible experimentation, with careful attention to temporal validation and hyperparameter optimization. The parameters of our best performing model can be seen in Table 5.

Table 5 Global Local Model Parameters

Data Parameters:	***Data Parameters:***
Window	*60*
Horizon	*30*
Batch Size	*32*
Model Parameters	
Node Embedding Dimension	*32*
Time Series Encoder Dimension	*64*
Global Hidden Dimension	*64*
Number of Global Layers	*2*
Activation Function	*RELU*
Training Parameters	
Learning Rate	*1e-3*
Weight Decay	*1e-5*
Optimizer	*Adam*
KL Regularization Weight	*0.1*

Loss Function and Regularization: The training objective combines Mean Squared Error (MSE) for prediction accuracy with optional KL-divergence regularization on node embeddings:

$$L_{total} = L_{MSE}(\hat{y}, y) + \lambda \cdot L_{KL}$$

Mean Squared Error (MSE) Loss: This measures the average squared difference between the predicted values and the actual target values across all samples. It encourages the model to make accurate predictions by penalizing larger errors more heavily.

Kullback-Leibler (KL) Divergence Regularization: This term regularizes the node embeddings by encouraging their distribution to be close to a standard normal distribution (mean zero, unit variance). For each node embedding, the KL divergence is computed between its learned distribution (parameterized by its mean and variance for each embedding dimension) and the standard normal. This helps prevent overfitting by discouraging the embeddings from drifting too far from a regularized space.

The total loss is a weighted sum of these two components, where the KL regularization term is scaled by a hyperparameter lambda. The MSE ensures prediction accuracy, while the KL divergence promotes generalization by regularizing the node embeddings.

5.6.1.4 Evaluation Metrics and Results

To validate the effectiveness of the Global-Local architecture, we compare against a state-of-the-art time-series forecasting baseline: PatchTST (Patch Time Series Transformer). PatchTST represents the current standard for transformer-based time-series prediction, using patch-based tokenization and attention mechanisms.

PatchTST Configuration:

Table 6 Patch TST Parameters

Context Window	***60***
Prediction Horizon	30
Patch Length	3
Model Dimension	64
Attention Heads	4
Hidden Layers	2
FFN Dimension	256
Parameters	~106K

Comparative Results:

Table 7 Evaluation Metrics Results

Model	Test MSE	Test MAE	Test MAPE	Parameters	Architecture
Global-Local (Advanced)	0.00174	0.02696	0.14966	109K	Hybrid spatial-temporal
PatchTST Baseline	0.00248	0.03316	0.18658	106K	Pure temporal transformer

Performance Analysis: As can be seen in Table 7, the Global-Local model significantly outperforms the PatchTST (parameters can be seen in Table 6) baseline across all metrics while maintaining comparable parameter count. This demonstrates that explicitly modeling node heterogeneity through learnable embeddings provides substantial benefits over treating all nodes as homogeneous time series. The hybrid architecture's ability to capture both local node characteristics and global temporal patterns proves more effective than pure temporal modeling for CPU usage prediction in heterogeneous cloud-edge environments. Specifically:

- **MSE Improvement**: 29.8% reduction in prediction variance
- **MAE Improvement**: 18.7% reduction in absolute error
- **MAPE Improvement**: 19.8% reduction in percentage error

- **Parameter Efficiency**: Similar model complexity (~109K vs 106K parameters)

Performance Analysis: Visualization analysis (Figure 33) reveals that prediction accuracy remains stable across the 30-step forecast horizon, with only gradual degradation in longer-term predictions. The model successfully captures both short-term fluctuations and longer-term trends in CPU utilization patterns.

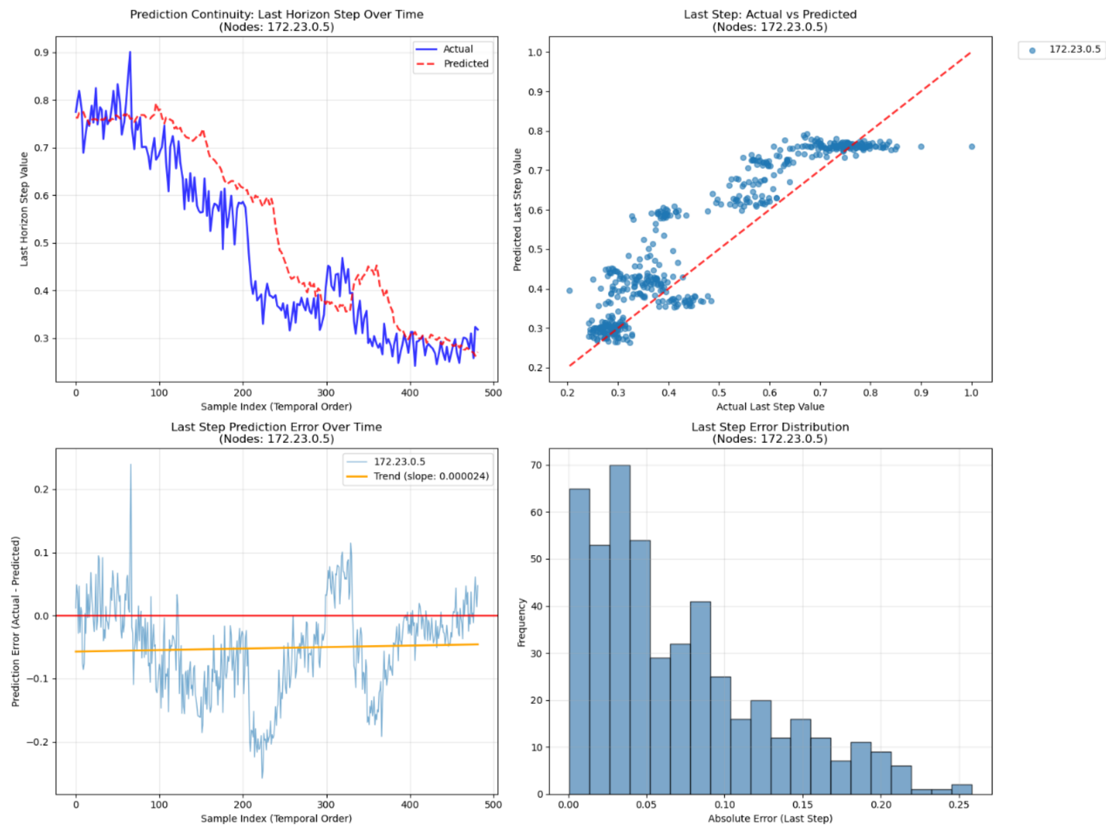


Figure 33 Last Horizon Step Prediction Evaluation

5.6.2 Cloud Edge Continuum Emulator (CEC-EMULATOR)

Modern distributed applications increasingly span both centralized cloud data centres and geographically dispersed edge sites to meet low-latency, high-throughput, and resilience requirements. However, evaluating performance, placement strategies, and orchestration policies across this cloud–edge continuum is challenging due to heterogeneous hardware, dynamic workloads, and network variability.

CEC-EMULATOR addresses these challenges by extending the µBench benchmarking framework to support:

- Multi-site topologies: Define and deploy microservice meshes across cloud and multiple edge zones.
- Node heterogeneity: Simulate real-world hardware constraints (CPU, memory, bandwidth) per node.
- Spatially-aware traffic: Generate site-specific request patterns to capture local usage profiles.
- Dynamic failure and throttling: Introduce node volatility to test resilience and migration strategies.

To that extent, CEC-EMULATOR can be used not only for providing synthetic data for ML model training but also as a digital twin onboarding framework for validating intended behaviour for applications by the application owner.

5.6.2.1 μBench

μBench is an open-source Python framework that automates the creation and execution of synthetic microservice workloads. By decoupling topology generation, workload modeling, and deployment, it simplifies benchmarking distributed systems. Its core architecture consists of the following main components:

1. **ServiceGraphGenerator**: Generates a service-graph topology based on user preferences (e.g., random, Barabási–Albert, hierarchical).
2. **WorkModelGenerator**: Transforms the service graph into a "workmodel.json", embedding per-service resource-stress parameters (CPU, memory, disk I/O, sleep) and call probabilities.
3. **K8sDeployer**: Renders Kubernetes manifests for each service-cell and a gateway proxy, then applies them to a target cluster.

Each service in the generated model is instantiated using a microservice-cell—a container that executes two distinct roles: the internal function applies system resource stress (CPU, memory, disk I/O), and the external function makes outbound REST (or Google Remote Procedure Calls (gRPC)) calls to other microservices as described in the model. This design pattern ensures that the emulator not only generates synthetic traffic but also simulates realistic processing loads per service. Specifically:

- Internal Function: Reads its internal_service.loader configuration from workmodel.json and applies CPU/memory/disk stress workloads.
- External Function: Sends HTTP/gRPC requests to downstream services according to the external_services sequences and probabilities.

The workmodel.json file describes each service (s0, s1, …) with the following fields:

- **internal service**: Configuration for resource stress (CPU stress, memory stress, etc.) and functional identifiers.
- **external services**: Ordered lists of call sequences, each with a **sequence length** and target service list.
- **request method, workers, threads**: Runtime parameters for request handling.

In Figure 34 you can see how the above settings are specified for service s0.

```
{
  "s0": {
    "internal_service": { /* CPU-intensive load */ },
    "external_services": [ { "seq_len": 100, "services": ["s1"] } ],
  }
}
```

Figure 34 Example of work model.json entry

μBench was originally developed to support **single-application deployments**, offering a great foundation but limited in terms of scalability, multi-tenancy, and placement flexibility. CEC-Emulator extends and generalizes over those areas.

5.6.2.2 CEC-EMULATOR

The CEC-Emulator builds on μBench's foundation to offer a highly extensible platform tailored for research and experimentation in cloud-edge continuum environments. It introduces a number of critical capabilities that make it suitable for evaluating modern distributed systems:

Infrastructure Simulation with Cloud-Edge Zoning

CEC-Emulator constructs a virtual infrastructure by initializing a K3d-based Kubernetes cluster, where each node is logically labeled as either “cloud” or an “edge site” (e.g., edge-site-1, edge-site-2). These labels act as affinity zones, enabling fine-grained control over microservice placement. This infrastructure abstraction allows users to replicate deployment patterns that are common in edge computing, such as pushing latency-sensitive services to the edge while keeping data-intensive processing in the cloud. For example, a user may choose to deploy the s0 service—the API gateway or user entry-point—at multiple edge sites, while placing backend services like s2 in the cloud zone to minimize resource strain on edge devices. Such configurations are easily defined in the workmodel.json by specifying node affinities per service, and CPU, RAM, and I/O stress factors for each service accordingly. An example of two deployed services, one with edge affinity policies and one without can be seen in Figure 35.

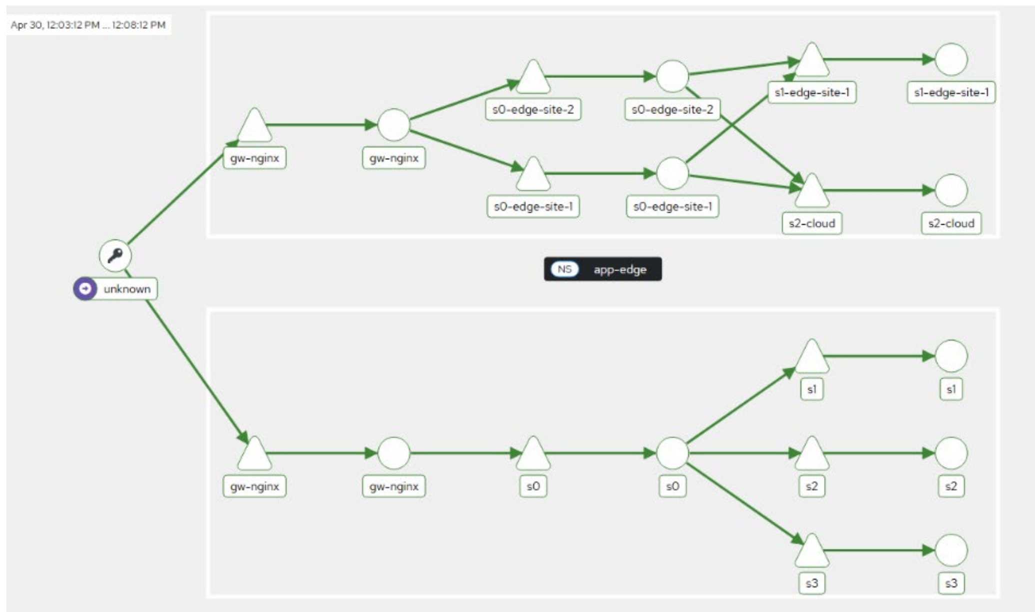


Figure 35 Service Mesh for two applications deployed on top of the Emulated Infrastructure

Multi-Application Deployment

One of the extensions introduced in CEC-Emulator is the support for concurrent deployment of multiple independent applications, each with its own microservice mesh. While µBench restricted experimentation to a single service graph at a time, CEC-Emulator allows parallel microservice applications running on top of the same infrastructure, with isolation between their components and configurations. This capability enables side-by-side performance evaluation of competing designs, stress-testing of shared infrastructure, or simulating multi-tenant environments where different applications compete for the same underlying resources. A typical use case might involve deploying one application to cloud-only nodes and another distributed across cloud and edge sites, then observing their relative performance under simulated traffic (Figure 36).

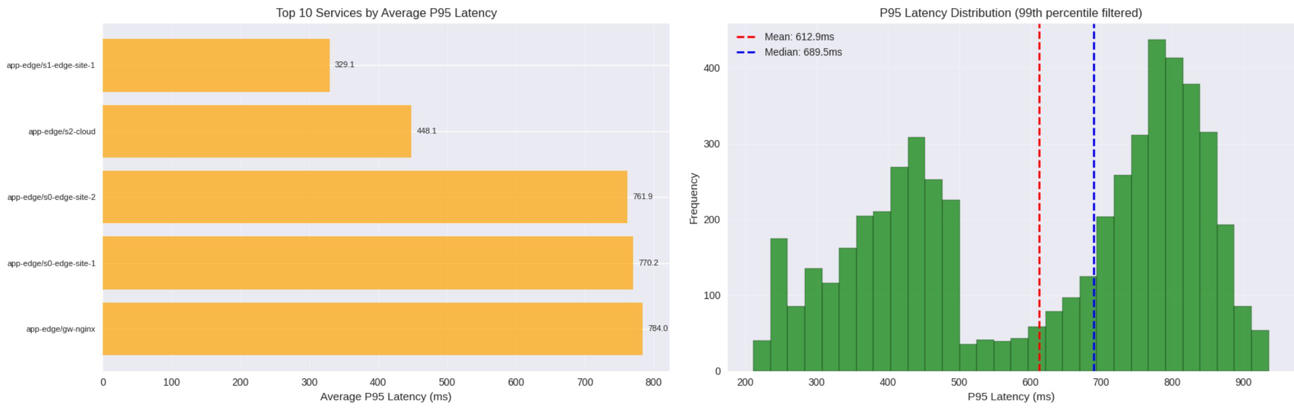


Figure 36 Measured Latency for the app-edge application running on top of the emulated infrastructure

Per-Site Traffic Generation and Spatial Behavior Simulation

Another feature is the emulator's ability to simulate localized traffic patterns by generating external requests within specific zones. For instance, s0 services deployed at edge-site-1 and edge-site-2 can each receive traffic shaped by unique statistical patterns, mimicking real-world user access in different geographies or time zones. This spatial diversity results in distinct CPU usage profiles per node, which is critical for understanding how edge load balancing, autoscaling, and migration strategies respond to actual usage variation. The ability to inject traffic with temporal and spatial locality offers a better approximation of actual edge systems. The diagrams in Figure 37 showcases the difference in CPU usage per node based on the differences in the generated traffic.

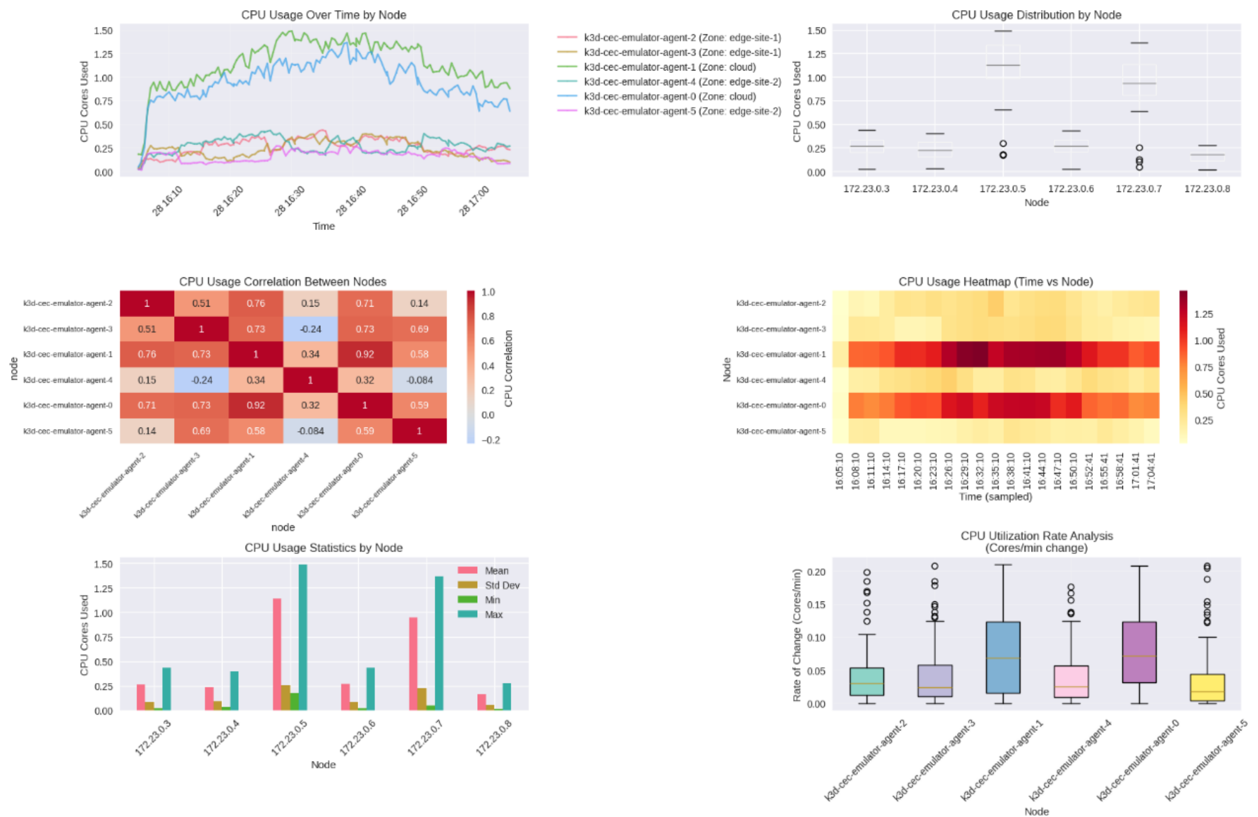


Figure 37 Collected Metrics From the Emulated Cloud Edge Continuum

Flexible Deployment and Reusability

Application definitions, deployment templates, and stress profiles are all decoupled to enable fast prototyping of new experiments. Moreover, the k8parameters.json file allows per-application customization of service configurations—such as replica counts, namespace isolation, or resource limits—while the core workload logic remains unchanged. In the next subsection, we extensively use a dataset extracted from the above emulator to benchmark global local models that blend local small learned features for each node in combination with a global model learning from all nodes together.

6 Decision Enforcement with Reinforcement Learning

6.1 Knowledge models using Explainable ML to explain resource usage

This work addresses the knowledge model of AC^3 as indicated in the decision enforcement module of the AC^3 architecture. In this section, we focus on using a causal latency model based on a knowledge graph to explain the causes of high latencies resulting from unavailable resources.

6.1.1 Causal latency modelling in the context of CECC

The use of microservices-based architectures is becoming more prominent due to their advantageous characteristics, such as manageability, scalability, and flexibility. However, their management can be complex, and their performance can be affected by high latencies, which can alter the Service Level Objective (SLO). In order to identify the causes of high latencies, we develop a causal modelling framework that is capable of analysing and reconstructing latencies within microservice-based architectures. To this end, we employ causal discovery to identify the causes of latencies. Our model integrates domain knowledge to impose constraints on the causal graph, ensuring the accuracy of the discovered relationships as well as accelerating the causal discovery. To validate our approach, we reconstruct latency metrics using machine learning techniques and demonstrate the effectiveness of our approach by accurately capturing the interrelationships between microservice resources. Our framework provides a better understanding of the causes of latencies that lead to SLO violations and paves the way for sophisticated mechanisms that enable proactive management of resources in CECC.

6.1.1.1 Objective of causal latency modelling in the context of CECC

Devising mechanisms for predicting and mitigating the causes of latency in microservices-based architectures in the context of CECC is essential for maintaining system performance and reliability. To address this challenge, causal mechanisms are being employed in the prediction of end-to-end latency. This is exemplified in [24], where the authors develop a data-driven cluster manager for interactive cloud microservices that is Quality of Service (QoS)-aware. In a similar way, [25] presents a causal modelling framework for estimating end-to-end latency distributions in microservice-based web applications. [26] uses the Program Evaluation and Review Technique (PERT) to inform the design of their Graph Neural Network (GNN). Their approach determines the causal interaction between microservices through a graph. Furthermore, [27] develops an approach called GRAF, a GNN-based proactive resource allocation framework for minimizing total CPU resources while satisfying the SLO. The existing approaches to latency prediction based on causal mechanisms generally have a similar structure. However, they do not consider constrained causal discovery mechanisms with domain knowledge to enhance the causal discovery process.

We develop a causal discovery framework that can be used to analyse latency within microservices-based architectures. Causal discovery is the process of identifying and recovering causal relationships between variables in multivariate systems. Two distinct approaches to the recovery of causal relationships can be distinguished: an interventional setting, in which the system can be interacted with and changes forced; and an observational setting, in which the data is based on previous recordings. The modular setup of contemporary microservice architectures presents a significant challenge in identifying relationships between different parameters of the services, such as CPU, memory, and the number of replicas. Prior research has employed causal discovery on a combination of observational and interventional data to find out the causes of failures or high latency outliers in complex microservice settings [28], [29], [30], [31], [32]. Other works use reinforcement learning with prior causal knowledge in order to enhance the automatic scaling of services to accommodate varying demands [32]. And the authors in [33] propose a Granger causality framework to detect causal dependencies based on error log timestamps. However, these approaches are inadequate for identifying the primary drivers of latency from observational metric data in a real-world deployed cloud microservice architecture without introducing faults or intervening in the system. Our approach employs causal discovery methods to uncover the underlying causal structure that contributes to high latency. By integrating domain

knowledge into our model, we constrain the causal graph, ensuring that the discovered relationships are accurate, as well as accelerating the causal discovery process.

Here, we make the following contributions:

1. We propose a comprehensive end-to-end framework that infers the topological structure of microservices through the analysis of latency data.
2. We incorporate domain knowledge into the causal discovery process, thereby ensuring the relevance of the inferred relations between the latencies of different microservices.
3. We evaluate the performance of several causal discovery approaches using constraints in practical settings with microservices-based architectures and a real dataset.
4. We validate our framework by reconstructing latencies using machine learning approaches combined with our causal discovery framework. To the best of our knowledge, this is the first work to demonstrate how constrained causal discovery methods can be used to discover factors contributing to latency of microservices from observational data.

6.1.1.2 Background on Causal Discovery

Causality is the study of cause and effect relationships. In this context, a variable X is said to cause another variable Y if changes in X lead to changes in Y, denoted as $X \rightarrow Y$ [34]. For multivariate datasets comprising multiple random variables, the causal relationships among variables can be modelled using a Structural Causal Model (SCM). In an SCM, each variable X is assigned a value based on a function of a subset of its respective causal parents P_{a_X}, such that $X = f_X(P_{a_X}, \eta_X)$, where η_X is an independent noise term [35]. The SCM can be represented graphically as a Directed Acyclic Graph (DAG) G. Causal discovery aims to infer the structure of G from observational or interventional data [16]. For non-time series data, with no autocorrelation or lagged dependencies, $G = (V, D)$ is a DAG. Where V represents the set of vertices and D represents the set of directed edges. The PC-Algorithm [36] is a prominent causal discovery method for such data. It reconstructs G by performing conditional independence tests to determine the graph's skeleton, starting from a fully connected graph and iteratively removing edges. Subsequently, edges are oriented using a set of rules outlined in [36]. If the direction of an edge cannot be determined, the algorithm outputs a bidirectional edge, which might be directed using additional background knowledge.

For time series data, causal discovery focuses on identifying either the full time series graph G or a summary graph G_{sum} is defined as a directed graph $G = (V \times Z, D)$, where $V = \{1, \dots, d\}$, with edges $\big((i, t-k), (j, t)\big)$ that are invariant under time translation. Usually the existence of a finite maximum time lag. $\tau = \max_{i,j \in V}\big\{k \big| \big((i, t-k), (j, t)\big) \in D \big|\big\} < \infty$ is assumed, and that the contemporaneous component of G is acyclic. The summary graph G_{sum} is a directed, potentially cyclic graph over V which contains a directed edge $(i, t-k) \rightarrow (j, t) \in D$ for some lag k. A notable algorithm for causal discovery in time series data is PCMCI+ [18], which extends the PC algorithm to account for both contemporaneous and time-lagged dependencies while considering autocorrelations. Similar to the PC algorithm, PCMCI+ utilizes conditional independence tests and starts with a nearly fully connected graph as the dependencies can only propagate forward in time. The algorithm applies orientation rules and outputs a bidirectional edge when the direction is not clearly determined [37].

6.1.1.3 Constrained Causal Discovery in Microservice-Based Architectures

We first lay out some preliminaries about microservices-based architecture, then present our assumptions. Subsequently, we formalize our proposed causal latency model and present our causal discovery framework.

A. Assumed Background Knowledge

Based on domain-specific knowledge, we formulate the following assumptions regarding the causal relationships between resources and latency in a microservice application:

A1) Any two endpoints within the same microservice never call each other.

A2) The number of client requests does not directly affect application latency.

A3) High infrastructure usage degrades application performance, not the other way around.

A4) Endpoints of the same microservice are deployed to the same host.

A5) Any metric x_i recorded at any microservice v_i only has a direct influence on the latency l_i of the same microservice, and not on any other latency l_y .

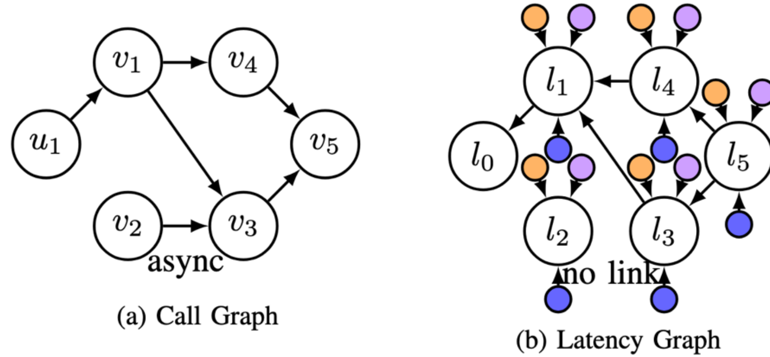


Figure 38 Comparison between Call Graph and Latency Graph

A common way to visualise the dependencies in a microservice based architecture is through a call graph as shown in Figure 38a, where the microservices are connected via edges D. The edge $(i, j) \in D$ indicates that the microservice v_i calls the microservice v_j. A more insightful representation of the microservice-based architecture is a latency graph, which can be approximated by the reversed call graph as shown in Figure 38b. The difference between the reversed call graph and a potential latency graph is that asynchronous calls represented in the call graph are not present in the latency graph.

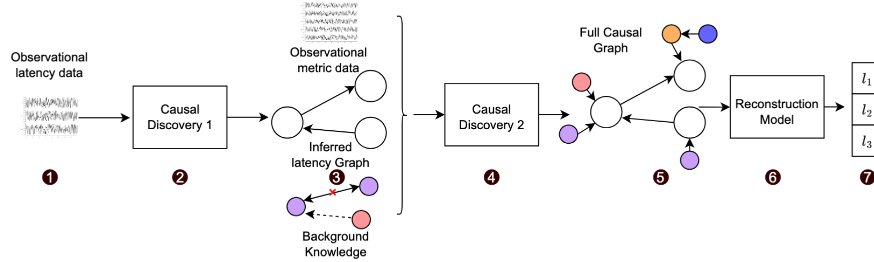


Figure 39 : Overview of the proposed process

As we assume that the latency of each microservice does solely depend on its own resources and other latencies we further make the following assumption which is illustrated in:

Assumption 1: Any resource r_{iz}, $z \in R_i$ where $|R_i|$ is the number of resources, can only influence the latency for the endpoint l_i directly (see Figure 40). It never directly influences l_j, meaning l_i acts as mediator between any resource $r_{iz} \in R_i$ contributing to l_i. However, we can never intervene on latencies only on resources. This means if we want to intervene on any latency l_i we can only intervene on it by proxy by intervening on its resources R_i.

Since we do not expect cyclical call or latency relationships, we assume that the latency graph takes the form of a DAG. For the latency DAG in Figure 38b, the structured causal model can be defined as:

$$l_0 := f_{u_1}(l_1, \eta_{l_{u1}}), \quad l_1 := f_{l_1}(l_4, l_3, \eta_{l_1}),$$
$$l_3 := f_{l_3}(l_5, \eta_{l_3}), \quad l_4 := f_{l_4}(l_5, \eta_{l_4}), \quad l_5 := f_{l_5}(\eta_{l_5}),$$

where η_x represents the noise term for each latency. We further know that η_x is not a random noise because the latency of each microservice depends on external factors such as CPU, memory limits and the number of pods. We can further assume that the relationship between any latency l_i and another latency l_j is approximately linear. This means we can rewrite $l_1 = f_{l_1}(l_{p1}, \eta_{v1})$, where $l_{p1} = c_3 l_3 + c_4 l_4$ are the latency values of the causal parents l_1, and c_3, c_4 are constants. η_{l1} can then be described by the latency limiting resources $r_{1i} \in R_1$ of v_1 plus a remaining random noise term η'_{l1} thus $\eta_{l1} = f_{n1}(R_1, \eta'_{l1})$ where f_{n1} is the noise function for l_1.

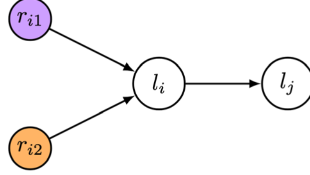


Figure 40 Graph illustrating the mediator assumption

Thus, we further make the following proposition:

Proposition 1: For the general case the latency of any microservice v_i can be represented as a function of the latency of its causal parents P_{ali} its resources R_i and the noise term η'_{vi} (e.g. influenced by different number of calls) by the structural equation f_i with

$$l_i = f_i(Pa_{li}, R_i, \eta'_{vi})$$

Considering these assumptions and the approach to modelling latency, we propose a framework that utilizes causal discovery to efficiently build a causal representation of microservice based architectures. An overview of the methods is given in Figure 40. We denote the available observational data as $X \in \mathbb{R}^{N \times v}$, where N is the number of observations and v is the number of variables. In step (2) we first select only the subset of all latency measurements $L \subset X$ (1) this is used to determine the latency Graph G_L. We use a linear conditional independence test as we assume that the relationship between two latency measurements $l_i, l_j \in L$ given a third latency l_k can be regressed out by a linear function; meaning l_i and l_j are conditional independent given l_k. Then, we determine G_L using a causal discovery algorithm fulfilling the assumptions about the data. Depending on the frequency of the recorded data it might be necessary to use a time series causal discovery algorithm where the maximum time lag τ has to be determined by using data analysis methods, or if the recorded frequency of the values is too low we might also use non time series causal discovery algorithms. If we assume latent confounding, meaning that two latency values are caused by the same unknown factor e.g. an external API, then, we also need to consider this information in this step. The graph G_L is combined with the measurements of the performance metrics and resources $R \subset X = X/L$ (observations in X without the latency) and the background knowledge based on assumptions *A1) - A4),* which rules out some possible edges (e.g., calls and latency cannot be directly connected based on *A2)* or helps orient edges (e.g., the CPU cannot be caused by latency but the opposite is possible). In (3) causal discovery is performed on a microservice level retrieving *n* sub graphs G_{R_i} which contain information about latency drivers for each service (4). In (4) these sub graphs can be combined to the overall causal graph G as they all have the latency nodes $l_i \in G_L$ as a common element with G_L. Thus, $G = G_L \cup \left(\cup_{i=1}^{m} G_{R_i}\right)$. This graph (5) gives us information of the latency driving factors in the whole system. To test if the determined factors are correct and thus the proposed causal latency model holds we select the causal features X_c to reconstruct the latency of a given endpoint v_i by $X_c = a(m_i, G, d)$ where a returns the causal ancestors of v_i up to a specified depth d. These features are then fed into a reconstruction model which estimates the structural equation of l_i. If the reconstruction is successful based on an error metric, we can assume that the causally determined features are useful in reconstructing the latency. This does not guarantee that the features are indeed causal, but it shows that these features are empirically contributing factors to the latency.

6.1.1.4 Dataset

We consider a microservice-based application, where each of its components is represented by a distinct microservice. The metrics such as CPU utilisation, calls per second, and memory utilisation are reported at an endpoint or microservice level. This architectural approach demonstrates the practical implementation of microservices and provides a robust platform for testing various resource provisioning mechanisms specific to a microservice environment. For our experiments, we used a total of 1,563 values in a sample interval of 1 minute (roughly 26 hours of observational data). We replace the few missing values with previous value imputation, and remove variables with fewer than two unique values across all observations.

6.1.1.5 Experiments and Results

The following section outlines the experimental setup employed for step (2), the causal discovery of the full causal Graph (4), and the reconstruction of the latency at the endpoint level (6). Prior to executing the two causal discovery algorithms and the prediction model, a train-test split is conducted, with 10% of the dataset aside for evaluation of the reconstruction step of the latency per endpoint. This is done to guarantee that data known in the causal discovery step is not used for the assessment of the reconstruction. To validate the suitability of the data for the causal discovery algorithms, we test each observed variable for stationarity. Therefore, we perform the augmented Dick-Fuller test and reject the null hypothesis of a unit root with $p-value < 0.01$ for each time series. Thus, the data fulfils the stationarity assumption.

A. Causal Latency Graph Discovery

a) Setup: For the causal latency graph discovery of G_L, we compare the performance of three causal discovery algorithms: PCMCI+, PC, and FCI, using accuracy, recall, F1, and structural Hamming distance (SHD) as validation metrics. As the ground truth, we set the reversed call graph based on the known topology of our microservice-based application deployment as an approximation of the latency graph. This approximation may not represent the true latency graph since some function calls could be asynchronous. For PCMCI+, we set the maximum time lag $\tau = 3$, determined from autocorrelation plots showing that correlation degrades at this lag. The significance level for the conditional independence test is set to $p_\alpha = 0.01$. We use the linear partial correlation (ParCorr) test due to expected linear dependencies between latency values. The metrics are calculated based on the discovered summary graph, as ground-truth data for time lagged dependencies is unavailable. We compare PCMCI+ with the PC algorithm to assess whether lagged time series information improves edge discovery and orientation. Additionally, we use the FCI algorithm to examine if accounting for latent confounders, like an external API in some cloud deployments, enhances causal discovery. For both PC and FCI, we use the linear Fisher-Z conditional independence test with $\alpha = 0.01$. All algorithms use *A1)* as background knowledge, removing any connections between latency values of endpoints of the same host.

Table 8 Results provided by the different causal discovery algorithms at a microservice level (m) and an endpoint level (e) for 2 the first step of the causal discovery

Algorithm	Accuracy		Precision		Recall		F1		SHD	
	m	e	m	e	m	e	m	e	m	e
PCMCI$^+$	**0.83**	0.91	0.55	0.42	**1.00**	**0.83**	**0.71**	**0.56**	**5**	8
PC	0.80	0.91	0.50	0.42	**0.83**	**0.83**	0.63	**0.56**	6	8
FCI	**0.83**	**0.93**	**0.60**	**0.50**	0.50	0.50	0.54	0.50	**5**	**6**

b) Results: Table 8 shows the accuracy, precision, recall, and F1 scores for the three causal discovery algorithms at both the endpoint and microservice levels. At the microservice level, connections are aggregated: if an endpoint in a microservice v_i connects to an endpoint v_j, an edge (m_i, m_j) is added to the graph. For PCMCI+, the discovered summary graph G_{sum} is derived from the results. At the endpoint level, the results are compared without post-processing, except for removing edges where endpoints from the same microservice are connected. Recall is a key metric here, as it measures the proportion of accurately inferred edges. PCMCI+ recovers all edges at the microservice level with reasonable precision. As shown in Figure 41, incorrect edges are

mainly due to uncertain orientation. This is consistent across endpoint graphs, where no spurious influences are found; only the direction of the causes may be misidentified.

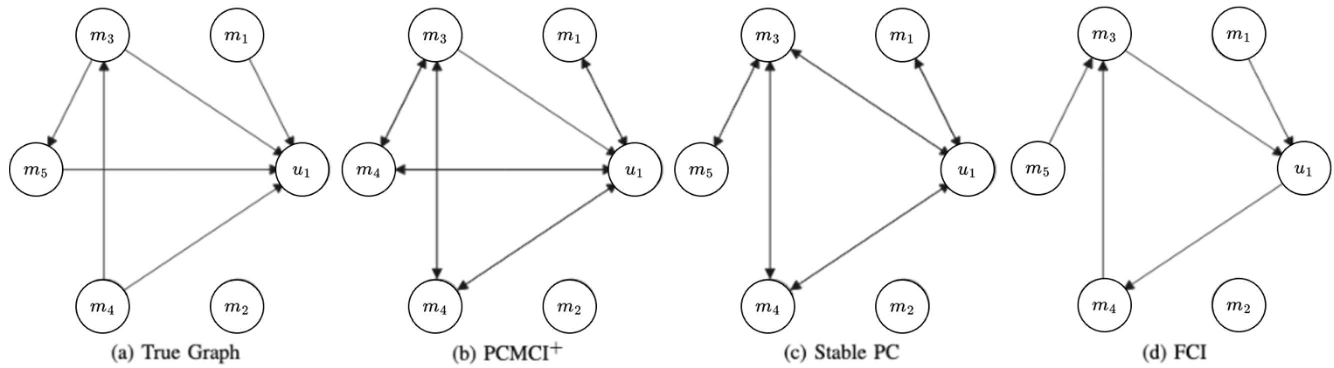


Figure 41 Results provided by the different Causal Discovery algorithms compared to the ground truth graph.

The time series data in PCMCI+ appears to help in more accurately orienting edges. At the endpoint level, PC and PCMCI+ have similar recall, while the FCI algorithm has a lower structural Hamming distance (SHD), suggesting fewer edge modifications are needed to achieve the ground truth graph. For step 3, the graph from PCMCI+ is selected as it includes time-dependent information useful for understanding system dynamics, making it the preferred input for 3.

B. Full Causal Graph Discovery

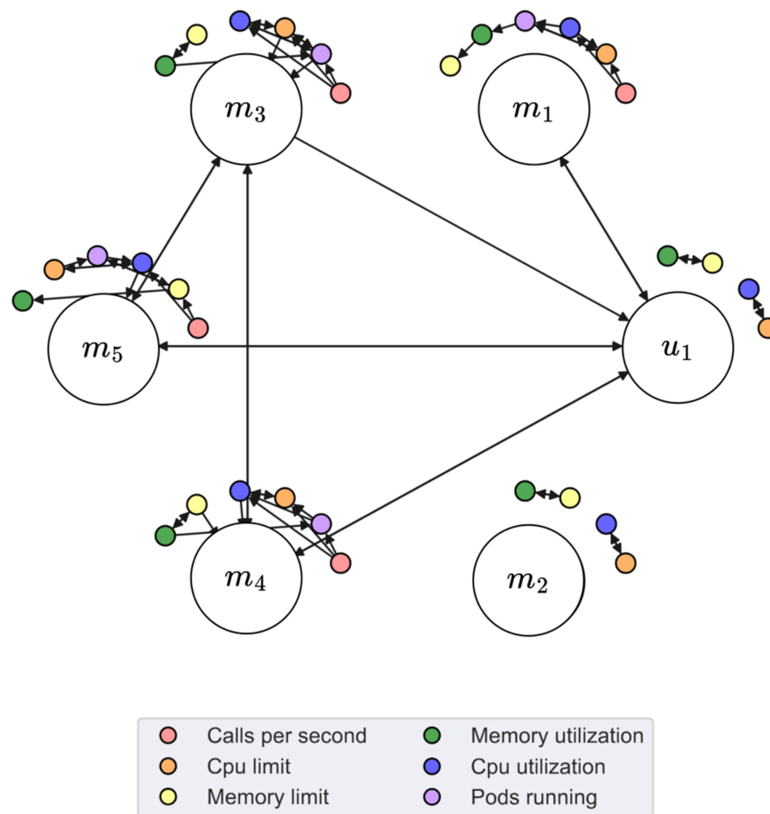


Figure 42 Discovered causal graph from observational data

a) Setup: To construct the complete causal graph, we perform causal discovery independently for each service endpoint to obtain the corresponding subgraphs. These subgraphs are then merged into the global causal graph. We employ the PCMCI+ algorithm for this purpose. Based on lagged mutual information and autocorrelation analyses, we observe significant temporal dependencies up to three time steps. Accordingly, the maximum time lag is set to $\tau = 3$. The significance level is set to $p = 0.01$, and we utilize the Regression Conditional Independence test [19], which is appropriate for handling categorical variables—such as the number of running pods—present in our dataset.

To improve efficiency and reduce the number of conditional independence tests, we incorporate background knowledge (assumptions *A1–A5)* as constraints on the search space. This ensures that not all variable combinations are tested, thereby enhancing the scalability and tractability of the algorithm.

b) Results: Figure 42 displays the resulting full causal graph at the microservice level, offering a holistic view of service dependencies. This visualization highlights the principal resources driving service latencies across the system.

The main latency drivers identified are the m3, m1, and m4 microservices. Specifically: For the m3 service, the number of running pods and the CPU limit are key latency factors. For m5, CPU usage is the dominant resource. For m4, both memory limit and CPU usage significantly influence latency.

Interestingly, the m1 and u1 services—despite being directly involved in handling user interactions—show no resource-driven impact on their latencies within the observed system. The u1 service, which functions as the system's entry point, is also unaffected by internal resource constraints.

C. Latency Reconstruction Model

Table 9 Comparison Results

	SVR				SVR (ground truth)				Lasso				XGBoost			
	τ	d	R^2	MSE	τ	d	R^2	MSE	τ	d	R^2	MSE	τ	d	R^2	MSE
u_{1,e_1}	3	1	0.88	31.66	1	1	0.88	**31.3**	3	0	0.82	7579.87	1	-	**0.96**	10.28
u_{1,e_2}	2	1	0.61	580.27	0	1	**0.99**	**1.01**	3	1	-82	4522.09	1	-	0.98	32.41
u_{1,e_3}	3	1	0.82	21.31	3	1	**0.83**	20.74	3	2	0.57	150.15	1	-	0.84	**18.64**
u_{1,e_4}	3	1	0.92	280.48	2	1	**0.93**	**257.03**	1	0	-3.52	8.11	0	-	0.95	190.94
m_{1,e_1}	3	1	0.92	2.18	0	-	-	-	3	0	0.6	346.57	1	-	**0.98**	**0.67**
m_{5,e_1}	0	1	0.79	314.82	0	1	**0.98**	**33.01**	3	1	0.77	11.06	1	-	0.47	788.62
m_{4,e_1}	3	2	**0.82**	**1.85**	3	2	-0.19	12.13	3	2	0.2	16222.78	1	-	0.86	1.46
m_{4,e_2}	3	1	**0.96**	27.92	0	-	-	-	1	2	0.8	50.53	1	-	**0.96**	**26.71**
m_{3,e_1}	0	1	**0.94**	**85.84**	1	1	-0.1	1638.4	0	0	-2.01	123382.31	1	-	0.17	1240.63
m_{3,e_3}	3	1	0.92	122.25	0	1	0.92	**121.37**	3	1	-3.7	46.61	3	-	**0.97**	51

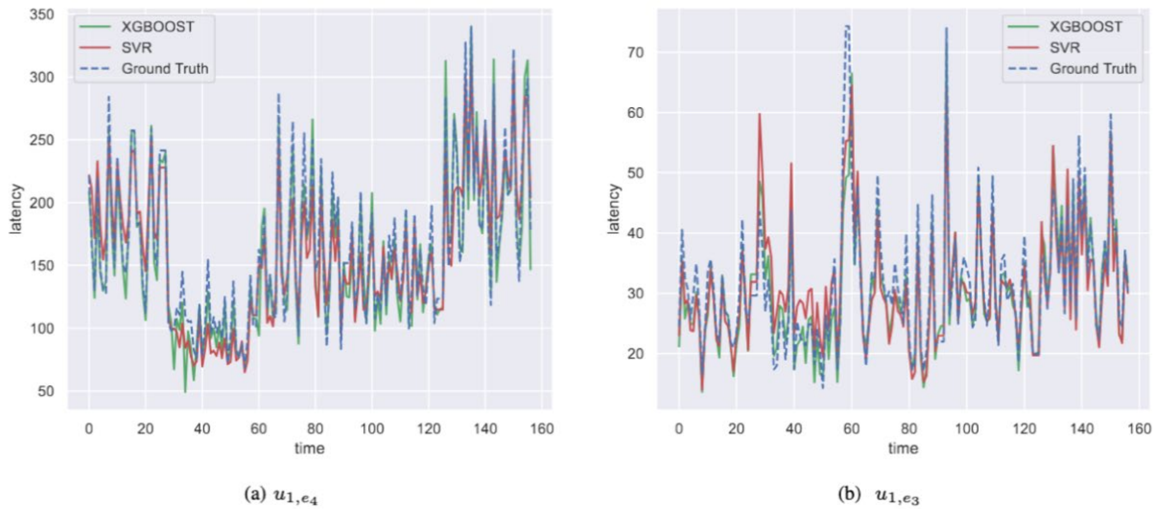
a) Setup: The latency reconstruction model serves two purposes: (1) It acts as a proxy for evaluating the second causal discovery step in the absence of ground truth data. (2) It helps identify which variables impact the latency, offering insights into which service or resource may be increasing latency at a specific endpoint. To select causal features, we consider depths $d \in \{1,2,3\}$ where depth 1 includes second-degree predecessors as features, and depth 2 includes third-degree predecessors of an endpoint v_i in the full causal graph. The optimal depth and maximum time lag (τ) are determined for each endpoint and model, with τ ranging from 0 to 3.

We use a Lasso model [38], a linear model with feature selection, and a Support Vector Regression (SVR) model [39] with a radial basis function kernel, thus making it a non-linear model. An XGBoost model [40], known for strong feature selection and capturing non-linear relationships, is used as a benchmark with access to all features except the endpoint under reconstruction. The optimal time lag for XGBoost is also selected. The models are evaluated using the *R2* score (with 1 being the best) and mean squared error (MSE), where a lower value indicates

better performance. For the SVR, the best hyperparameters are determined over a grid of 1100 different models using 10-fold cross-validation. The Lasso parameters are also selected using 10-fold cross-validation.

b) Results: Table 9 shows the metrics for latency reconstruction using either features from the causal graph, and in the case of the XGBoost baseline model, using all features except the endpoint's own latency. The SVR model, using only causal features, is the only one with an *R2* value over 0.5 for all endpoints. Except for m3 m3_e1 and m5_e1, the XGBoost benchmark generally outperforms other models. The Lasso model struggles with complex routes, indicating linear models are insufficient for these cases.

Overall, predictions using the inferred latency graph are more consistent than those using the ground truth latency graph, suggesting that the reversed call graph may not fully capture the underlying dynamics. As illustrated in Figure 43, both the SVR with causal features and the XGBoost model are able to reconstruct the latency at the selected endpoint, which also generalises to other endpoints. While the SVR model generally underperforms compared to XGBoost, the focus is not on performance alone but on demonstrating that latency reconstruction is feasible using causal features.



(a) u_{1,e_4} (b) u_{1,e_3}

Figure 43 Comparison between the predictions of SVR with causal features and XGBoost model with all features

The good result for the SVR with causal features shows that we can estimate latency values based on causal predecessors, suggesting that the proposed latency model in (1) holds empirically. We further investigate the contributing features to the prediction using permutation importance. For example, the u1, e4 endpoint has meaningful features contributing to the SVR predictions, such as cpulimit@m3, cpu-utilisation@m4, and latency@m3.

For the model without causal features, only non-intervenable features—such as latency and calls per second—have high permutation scores. Additionally, some latency values for endpoints that, according to the ground truth, should not influence the latency at u1, e4 are incorrectly determined as important features using permutation importance. We observe similar behaviour across all other endpoints.

7 Network Programmability

Network Programmability is a critical aspect for achieving the AC^3 project's primary goal of intelligently deploying and managing applications across a federated infrastructure. The ability to adapt the network dynamically to changing deployment configurations and conditions is a vital component. As presented in D4.1, we are exploring 2 distinct (but inter-related) technologies to provide network programmability capabilities within the AC^3 project, specifically Software-Defined Wide Area Network (SD-WAN) and Kubernetes-based networking.

In this section, we extend and expand upon the work described in D4.1 for both approaches. Specifically for SD-WAN we describe an extension to include eBPF, providing more optimized performance. For Kubernetes, we describe the implementation of the architecture described in 4.1 and extend this with additional performance-enhancing behaviors.

7.1 eBPF-based SD-WAN

In Deliverable D4.1, we introduced an SD-WAN architecture designed for the CECC framework. We also described the proposed SD-WAN solution overlay management, hybrid Wide Area Network (WAN) support and application-aware routing using Linux IPtables and IPRules and compared performances with the VRF based approach used in the leveraged open-source solution. Results showed that our proposed IPTables and IPRules-based approach is more RAM and CPU efficient compared to the VRF approach. However, this solution relied on the Linux networking stack that can be less optimized for high-performance networking due to additional overhead and frequent context switching, which negatively impacts throughput and latency.

The CECC presents a federated topology composed of different cloud, edge, and far-edge domains, generating dynamic and QoS-sensitive traffic that must be managed with the SD-WAN. Unlike traditional enterprise SD-WAN scenarios, which connect predictable endpoints like headquarters and branch offices, CECC edge gateways face complex workloads requiring both performance and programmability. Addressing this challenge necessitates a shift to lower-level packet processing to reduce latency and improve throughput.

For that, we proposed the use of eBPF[11] (extended Berkeley Packet Filter) technology to enhance the SD-WAN data plane. eBPF enables dynamic and safe packet processing programs such as eXpress Data Path (XDP) to be executed within the Linux kernel. These programs can be attached to various interfaces for network packet reception, enabling in-kernel packet classification, routing, and QoS enforcement, eliminating context switches between user and kernel space, and thus reducing latency and improving the throughput.

Figure 44 shows our High-Efficiency Layered Infrastructure with eBPF for Optimized SD-WAN (HELIOS)within the AC^3 architecture. HELIOS introduces a high-efficiency layered infrastructure that utilizes eBPF to implement optimized SD-WAN edge nodes for CECC. Inspired by the Metro Ethernet Forum (MEF) [41], it is structured into two main layers: the Control Layer and the Infrastructure Layer.

7.1.1 State-of-the-Art

CEC is undoubtedly the next evolution of cloud computing, where workload is executed over the continuum to gain the low-latency capabilities of edge and far-edge computing nodes, while centralized cloud resources handle high-load, and delay-tolerant tasks. The CEC will build on the advance and democratization of edge computing, with nodes positioned close to end users and an increasing array of end-user devices—such as Internet of Things (IoT) gateways, Unmanned Aerial Vehicle (UAVs), and smartphones—capable of running workloads. Meanwhile, the shift toward microservice-based applications will further drive CEC adoption, enabling application components to be distributed across the continuum to meet SLA requirements. However, interconnecting cloud, edge, and far-edge nodes presents challenges, as multiple providers must collaborate to build a cohesive

---

[11] https://ebpf.io/

continuum and ensure seamless application deployment per SLA standards [42]. This cooperation is essential, as edge nodes have limited computing capacity and require support from cloud and other edge providers to manage sudden computation demands.

Interconnecting CEC nodes that carry the Data Plane (DP) of deployed microservices is essential, requiring (i) programmability to enable network orchestrators to specify QoS levels and resiliency for each service, (ii) QoS enforcement to ensure SLA—such as low latency, high bandwidth, and minimal packet loss [43]—and (iii) resiliency by enabling route selection flexibility for services [44]. Two primary interconnection solutions are considered: relying on the underlying network (e.g., Segment Routing or Multi-Protocol Label Switching (MPLS) tunnels) or using overlay networks over existing Internet links (e.g., SD-WAN). While the first approach efficiently supports QoS, it assumes that all CEC nodes can control the underlying network to define QoS levels and paths for application microservices, a capability that cloud providers typically possess when connecting their own data centers with MPLS tunnels. However, this approach is limited since cloud and edge providers often rely on network links operated by third-tier Internet Service Provider (ISP). Additionally, CEC infrastructures often include edge and far-edge computing resources connected via standard network connectivity, with no control over the underlying network.

On the other hand, SD-WAN is a key technology for interconnecting CEC nodes as it meets the three main requirements of programmability, QoS, and resiliency. Leveraging Software-Defined Networking (SDN) principles, SD-WAN simplifies network management by decoupling hardware from control programs and using software and open APIs to abstract infrastructure. It creates overlay networks on top of heterogeneous underlay networks, including those from different ISP, while maintaining consistent addressing. SD-WAN supports multiple concurrent WAN connections and can interconnect sites in various topologies, such as mesh, to build a full overlay where SD-WAN edge nodes manage routing for SLA, resiliency, and scalability without relying on underlying network resources. However, most current SD-WAN solutions are proprietary, limiting innovation. This work addresses that gap by proposing an SD-WAN framework that leverages eBPF in the Linux kernel to design SD-WAN edge nodes, enabling overlays between CEC nodes. These nodes are managed by Open Network Operating System (ONOS) controller to ensure QoS across links that connect microservices across the continuum.

7.1.2 Background

7.1.2.1 eBPF

eBPF is a Linux-based Virtual Machine (VM) that runs sandboxed programs in a privileged mode to safely and efficiently extend the capabilities of the kernel with custom code that can be injected at run-time without requiring changes in the kernel source code or load kernel modules. The eBPF is an event-driven program triggered when the kernel or application passes a certain hook point. eBPF has some predefined hook points that include system calls, every kernel function, kernel trace points, and network events, to name few. If a predefined hook does not exists for particular need, it is possible to create one at kernel probe or user probe, almost anywhere.

7.1.2.2 XDP

Is a network type of eBPF programs, designed for high-performance packet processing. Identified by its hook point within Linux kernel's networking stack, specifically in the reception chain of the network device driver, prior to Socket Kernel Buffer (SKB) allocation.

The XDP program is triggered immediately on the ingress path in response to network events, typically upon packet reception. The hook point varies according to how the XDP program is attached. We distinguish the generic, native, and offloaded XDP. Generic XDP is loaded into the kernel as part of the regular network path, making it suitable for testing. Native XDP is loaded within the network card driver, providing better performance but requiring driver support. Offloaded XDP runs directly within (smart) embedded Network Interface Controllers (NICs) and requires specific device support. Network device drivers may not support XDP hooks, in which case the generic model is utilized. In Linux 4.18 and later, supported drivers include Veth, Virtio, Tap, Tun, Lxgbe, I40e,

Mlx5, and MLX4, to name few. XDP has become a popular mechanism for accelerating and offloading packet processing from user-space applications for high-performance networking applications.

7.1.3 Overview Architecture

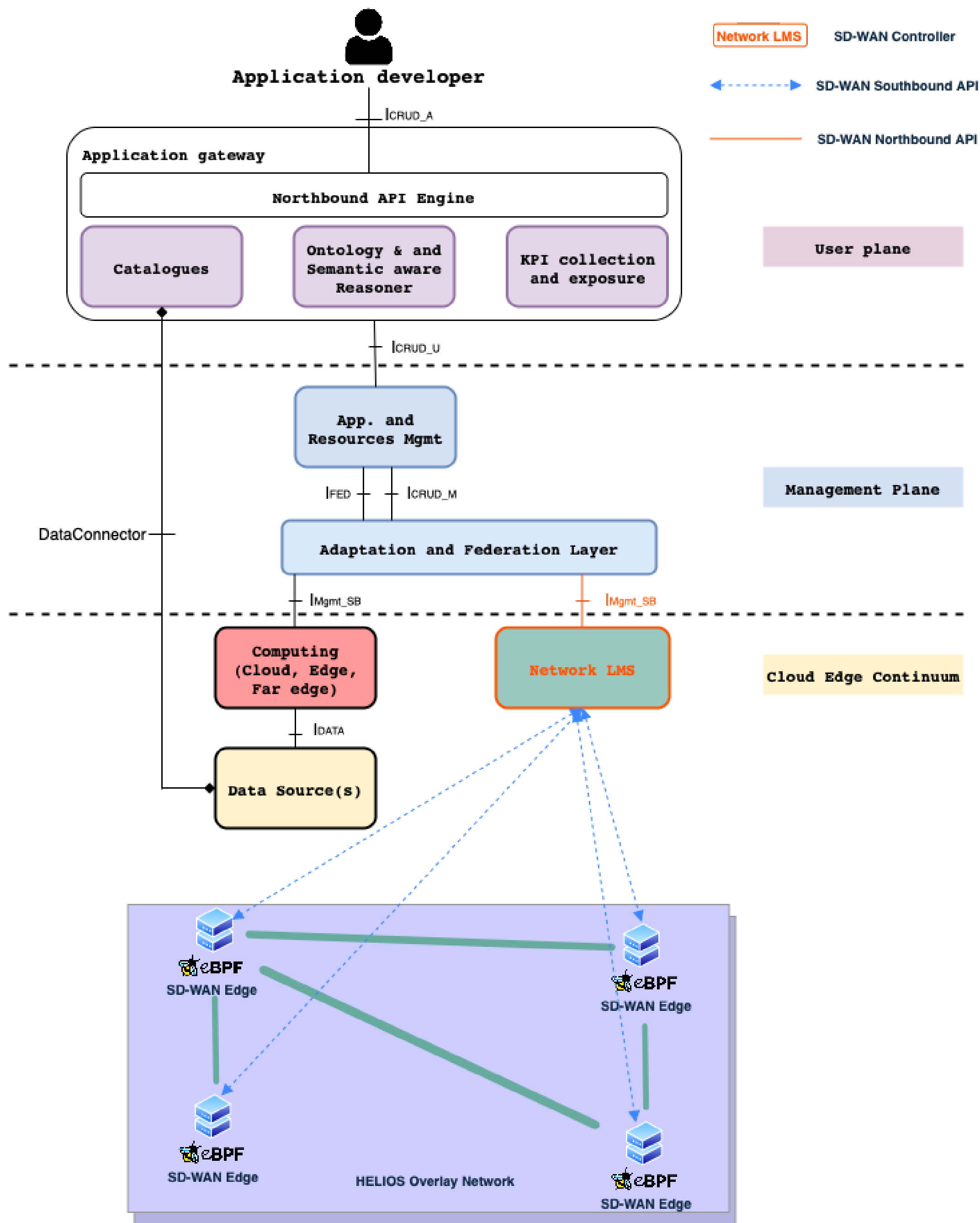


Figure 44 HELIOS Overview within AC³ Architecture

The Control Layer hosts the ONOS SDN controller, which acts as both a server and a client for respectively, the above and below layers. As a server, it receives requests from the application logic via its Northbound Interfaces (NBIs), extracts the flow rules and pushes them to the SD-WAN edges at the infrastructure layer using gRPC protocol. The controller can dynamically create, update, and delete eBPF maps in the edge gateways, perform monitoring, and retrieve network telemetry.

The Infrastructure Layer comprises SD-WAN edge gateways where the eBPF programs operate. These gateways serve as termination points for SD-WAN tunnels and leverage eBPF to manage traffic flows efficiently. Unlike traditional SD-WAN implementations that rely on Linux networking stack or user-space Data Plane Development Kit (DPDK) with high resource usage, this design employs a kernel-based approach that enhances throughput, reduces latency, and enables hardware resource sharing across multiple applications.

7.1.4 Low-Latency Intelligent Network eXecution (LINX)

Each SD-WAN edge gateway runs a system called LINX (Low-Latency Intelligent Network eXecution) that consists of a management layer and a data path layer Figure 45.

7.1.4.1 Management Layer

The management layer is implemented in user space and is responsible for orchestrating the configuration and control interfaces. It begins with a Yet Another Markup Language (YAML) configuration validator, which initializes system parameters including control and data plane interfaces and port settings. A gRPC server listens on port 50051, enabling the reception and processing of control messages from the ONOS controller. These messages include instructions for creating, modifying, or deleting flow rules. The Generic Routing Encapsulation (GRE) overlay manager is tasked with establishing and managing tunnel lifecycles, ensuring reliable encapsulation and routing between CECC domains. Additionally, the eBPF program manager controls the deployment, update, and lifecycle management of XDP-based eBPF programs. This component ensures that eBPF logic is correctly instantiated in the kernel and that the relevant program and maps are available for runtime packet processing.

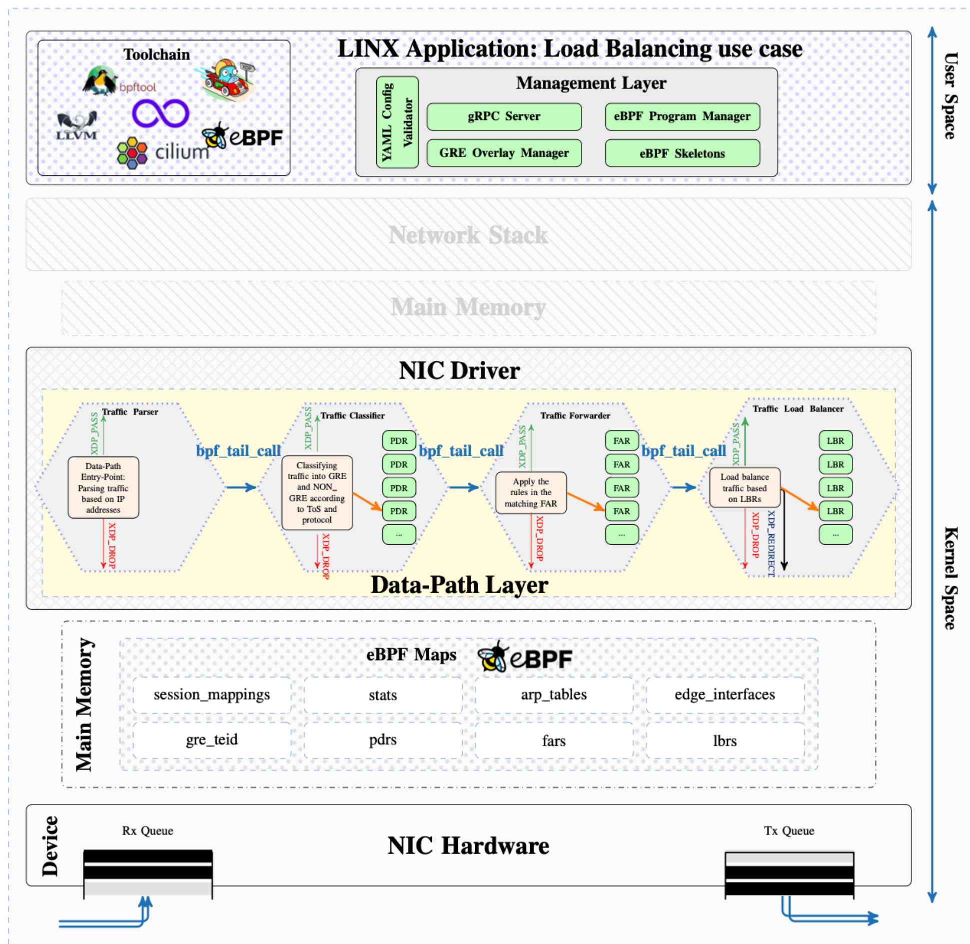


Figure 45 LINX Architecture

7.1.4.2 Data-Path Layer

The data path layer is responsible for processing End-User/Data Center (DC) traffic, implementing a pipeline where decisions are made about the fate of each packet, whether it is passed, dropped, or redirected. The pipeline is divided into four key components as follows: (i) The Traffic Parser is considered the entry-point to the Data-Path, it parses incoming traffic and matches fields in Packet Detection Rules (PDRs). (ii) Traffic Classifier categorizes traffic by uplink (e.g., GRE) and downlink (e.g., NON-GRE) flows, as well as other criteria such as Classes of Traffics (CoTs) and protocols, using PDRs and matching Packet Detection Information (PDI). (iii) Traffic Forwarder, following classification, it directs traffic to its destination: either to the Local Area Network (LAN) (i.e., End-User) by removing GRE headers or to the WAN (i.e., DC) with appropriate GRE encapsulation, in alignment with the Forwarding Action Rules (FARs). Finally, (iv) Traffic Load Balancer, ensures traffic adheres to predefined Load Balancing Rules (LBRs) updates, regarding GRE overlay selection, monitoring and dynamically adjusting flows to optimize network Key Performance Indicators (KPIs) and enhance resource efficiency.

7.1.4.3 Control Signaling

In this work, we define the following key concepts and terms to better understand our data model in Schema 1 (Figure 46):

- Flow Rule: A set of instructions dictating how traffic is handled within SDN architectures. They are crucial for efficient, secure traffic management, as they define specific criteria and attributes (such as source and

destination IP addresses, protocols, and ports) against which traffic is evaluated for redirection, modification, or dropping.
- Match Field: An attribute within the PDR that is used to identify and categorize incoming packets based on predefined criteria. It plays a crucial role in determining how packets are processed by the network, serving as a basic element of a flow rule. In addition to the standard criteria, match fields can also include a Virtual Local Area Network (VLAN) tag, an MPLS Label, or any other header field such as Type of Service (ToS).
- Information Element (IE): A structured data field used to convey specific types of information within control gRPC signaling. Each IE encapsulates essential data required for the management and control of packet flows, represented as a tuple (type, length, value), where type denotes the information kind (e.g., IP address or protocol), length specifies the data size, and value contains the actual content.

Schema 1 Flow Rules' Data-Model/Grammar

```
1: request: operand rules
2: operand: create | update | delete
3: rules: rule [,rules]
4: rule: PDR | FAR | LBR | [IEs]
5: IEs: IE [,IEs]
6: IE: (type, length, value)
7: length: u8 | u16 | u32 | u64
8: value: CONST (decimal)
9: typ: IP address | MAC address | protocol |
   ToS | action | interface | interface index
10: PDR: (IP address, u32, src_ip), (IP
   address, u32, dst_ip), (protocol, u8,
   TCP|UDP), (ToS, u8, CONST)
11: FAR: (action, u8, REDIRECT|PASS|DROP),
   (interface, u32, GRE-WAN1|GRE-WAN2|..|
   GRE-WANi|NON-GRE), (ifIndex,u32, 1|2|..|i+1)
12: LBR: FAR, (ToS, u8, CONST)
```

Figure 46 Flow Rules Data Model

- PDR: A set of rules or criteria used to identify and handle packets in a specific manner within the Data Path. Each PDR defines conditions (i.e., Match Fields) and actions (i.e., FAR and LBR) for packet processing.
- FAR: Defines the actions to be taken on packets matching a PDR, such as forwarding, buffering, or duplicating packets.

- LBR: A rule designed to distribute network traffic across multiple paths or endpoints efficiently, often for load balancing, redundancy, or failover purposes.

7.1.4.4 Data-Path Traffic Processing

The in-kernel packet processing journey begins at the NIC Receiver (Rx) queue where the NIC's Direct Memory Access (DMA) triggers a Hardware Interrupt Request (IRQ) (HardIRQ), invoking the NIC Driver's IRQ handler. This handler then initiates the New API (NAPI) subsystem via a Software IRQ (SoftIRQ), starting packet processing via the driver's registered poll function, which implements the XDP hook for eBPF XDP program. To ensure uninterrupted processing, the NIC driver disables further IRQs. The eBPF XDP program, executed by the XDP hook, marks the entry-point for the created XDP pipeline. Starting execution, the XDP program accesses a context object metadata, encapsulated within the optimized xdp_md struct . Following data parsing, control may transfer to other XDP programs via bpf_tail_call function . After parsing, metadata fields can be read from the context object, which also allows attachment of custom metadata . In this regard, XDP programs access persistent data

structures (eBPF maps) via functions such as bpf_map_lookup_elem or bpf_map_update_elem. A final verdict is issued, determining packet handling, such as dropping, retransmission, kernel processing, or redirection. Once packet processing is completed, the NAPI subsystem is deactivated, and IRQs from the NIC device are re-enabled. This treatment is summarized in Algorithm 1 (Figure 47).

Algorithm 1 Data-Path Packet Processing

```
1: Receive Packet on Rx queue
2: DMA Triggers HardIRQ
3: Invoke NIC Driver's IRQ Handler
4: Initiate NAPI Subsystem (SoftIRQ)
5: Start Packet Processing via Driver Poll
6: Trigger XDP pipeline
7: Packet enters XDP_Parser
8: bfp_tail_call XDP_Classifier
9: bfp_tail_call XDP_Forwarder
10: bfp_tail_call XDP_Load_Balancer
11: Redirect packet to TX queue
```

Figure 47 Data-Path Packet Processing

7.1.5 Evaluation Results

7.1.5.1 Experimental Setup

We tested our approach using RFC 2544-like tests. These tests are designed to evaluate the throughput, latency, and overall performance of the solution under various conditions. The System Under Test (SUT) consists of a loopback between two devices: one hosting a traffic generator such as TRex, and the other, the Device Under Test (DUT), hosting the solution to test, in our case LINX. TRex is an open-source realistic traffic generator powered by DPDK, used to measure the maximum sustainable throughput of a DUT.

7.1.5.2 Experimental Platform

To evaluate the performance of our HELIOS framework, we utilized two identical hosts for the testing environment. One host was configured with TRex traffic generator, while the LINX solution was deployed on the second host. On both devices, we have installed an Ubuntu 22.04 operating system, with kernel version 5.15 to ensure optimal compatibility with eBPF and DPDK. Each device is powered with the 12th Generation of the Intel i7, embedding 12 Cores, clocking at 4.9 GHz, and using 25 MiB of L3 cache memory. Regarding the network interfaces, both hosts were outfitted with Dual-port Intel Ethernet X550T adapters, supporting 10 Gbps on each port, ensuring reliable high-speed data transmission.

7.1.5.3 Test Cases

We scripted Python test cases to mimic data traffic on both uplink and downlink scenarios, leveraging TRex APIs.

- Uplink: We generate User Datagram Protocol (UDP)/TCP traffic in TRex, simulating a client sending data via one of the dual-port X550T adapters to the DUT. The LINX gateway, acting as the edge node, processes the traffic, encapsulates it in a GRE header, and forwards it to the appropriate WAN interface, as determined by the load balancer based on SDN controller rules. The traffic is looped back over the GRE overlay to TRex on the second port, simulating a remote-region node.
- Downlink: In the downlink scenario, the simulated remote-region node, represented by TRex, generates UDP/TCP traffic that is encapsulated in a GRE tunnel and sent to the DUT (i.e., LINX gateway) via the

WAN interface. The DUT decapsulates the traffic and forwards the resulting packets to the end-user (i.e., TRex) over the LAN through the other port on the Intel X550T adapter, completing the End-to-End (E2E) delivery process.

- Control Signaling: The ONOS SDN Controller communicates with LINX gateways to dynamically manage and distribute the flow rules. We simulate situations where the SDN needs to update the flow rules within the remote regions and estimate the delay.

7.1.5.4 Results

Across all tests, we analyze the effect of packet injection rate (f_i), packet size (S_p) $(\{100, \ldots, 1400\} \cup \{50, 1450\})$, and the number of Rx queues (c_j), where $(i, j, p) \in \{1, 2, \ldots, 10\} \times \{1, 2, \ldots, 12\} \times \{1, 2, \ldots, 16\}$. We used the 'irqbalance' service to distribute incoming traffic evenly across the active Rx Queues, with each Rx Queue pinned to a single CPU core to optimize performance. Each test configuration was repeated $50\times$ to ensure accuracy and consistency.

a. Throughput

Figure 48 shows the peak throughput (in MPackets/s) obtained for downlink (Figure 48a) and uplink (Figure 48b) scenarios function of (S_p) and (c_j) . All curves exhibit a similar pattern, decreasing smoothly from 10 MPackets/s and converging toward 1 MPackets/s. This trend resembles an exponential decay function, with throughput gradually declining and approaching an asymptote. The similarity between the uplink and downlink curves suggests that direction has minimal impact on throughput behavior; however, packet size (S_p) and, to a lesser extent, the number of Rx Queues c_j , appear to influence the results. Indeed, the more (S_p) increases, the more throughput decreases, which is intuitive given that throughput is measured in MPackets/s. With a fixed 10 Gbps bandwidth, the maximum achievable throughput is approximately 10 MPackets/s (according to TRex performance) for 50-Byte packets; as packet size grows, the packet rate declines, even though the bandwidth remains fully utilized. The impact of (c_j) appears minimal with the current SUT, but we anticipate that with a more powerful SUT, this impact will become evident. Transmitting hundreds of millions of packets would more clearly demonstrate the importance of multiple Rx Queues in handling high packet volumes efficiently.

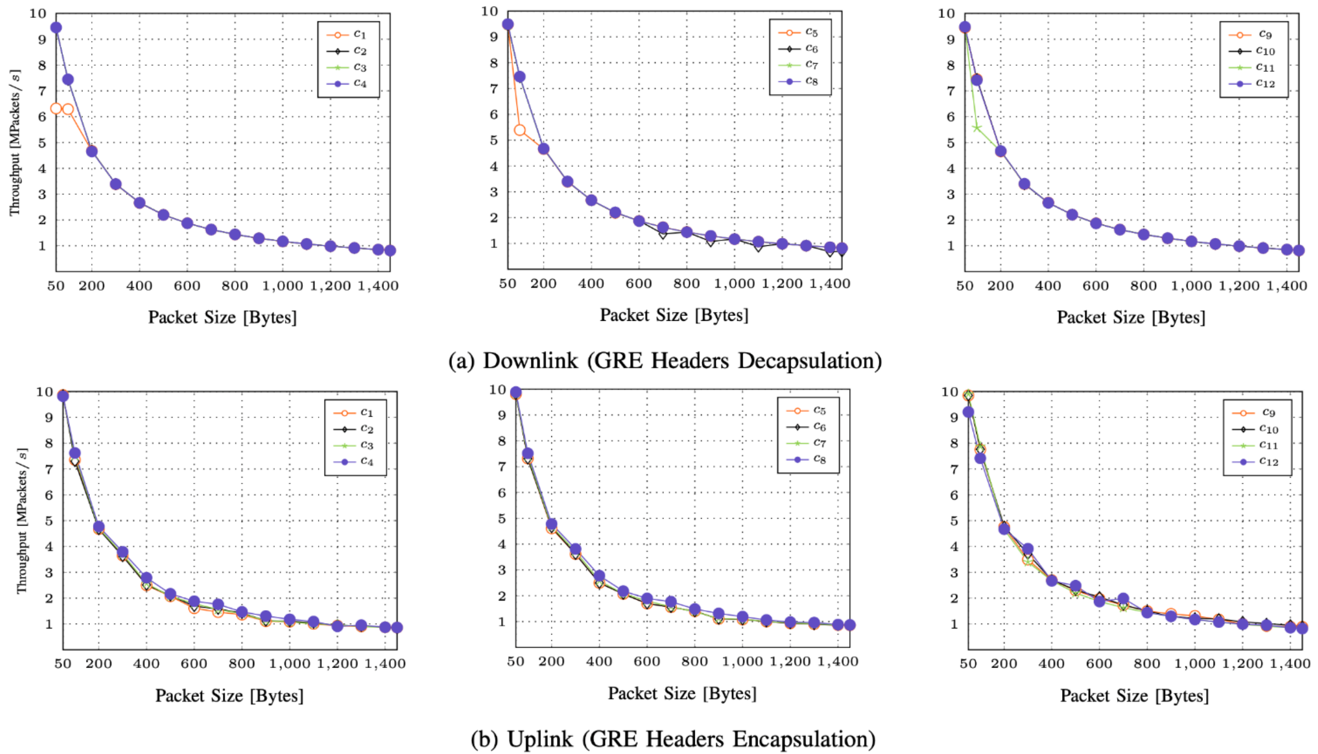


Figure 48 Throughput Obtained on the Downlink and Uplink per Packet Size and Number of Rx Queues

b. CPU Load

Table 10 displays the average CPU usage in percentage as a function of f_i and c_j. The results reveal a trend where the higher f_i, the greater CPU load will be across all configurations. At higher packet rates, particularly beyond f_4 (4 MPackets/s), CPU usage nears saturation with a single Rx Queue (i.e., c_1), indicating inefficiency under high packet rates. In contrast, increasing the c_j value significantly improves CPU efficiency. For example, in configuration (f_4, c_1), CPU load is 99.23%, reaching 100% in (f_7, c_1) and higher. However, this load drops to 19.19% with only two Rx Queues. With $c_j \geq 6$, CPU load remains below 30% and increases more slowly with further packet injections, showing better efficiency and reduced risk of saturation. Configurations c_8 to c_{12} exhibit the lowest CPU usage, with c_8 at 20.06% and c_{12} at 7.84% for 10 MPackets/s, highlighting the effectiveness of multiple Rx Queues in reducing CPU load even at high packet rates.

c. Control Signaling Latency

Figure 49 depicts the Cumulative Distribution Function (CDF) obtained for the control signaling scenario, derived from a dataset of 1, 000 values representing the delay ratio achieved in less than x ms. The results show a low-latency distribution characterized by minimal variation with the following statistical measures: a mean of 0.8155 ms, variance of 0.3539 ms^2, standard deviation of 0.5949 ms, and a coefficient of variation of 0.7295. We find that approximately 50% of control plane operations complete within 0.7 ms, illustrating that half of all operations complete under 1 ms, while the 10th percentile is as low as 0.146 ms, indicating that a significant portion of operations are nearly instantaneous. The curve progresses smoothly, with 90% of operations achieving a latency below 1.6 ms. The upper tail of the CDF reaches 2.6 ms, indicating the maximum observed latency, which likely represents rare worst-case delays. Overall, these results indicate reliable low-latency performance ideal for real-time applications requiring immediate rerouting or policy adjustments.

Table 10 CPU Load Function of Packet Rates and Rx Queues

Injections [MPackets/s]	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
c_1	7.92	10.79	31.41	99.23	99.33	99.73	100	100	100	100
c_2	2.88	5.56	9.35	19.19	65.15	68.34	98.41	98.87	98.95	99.98
c_3	1.86	3.52	5.74	10.63	15.21	23.22	68.38	76.85	93.11	93.29
c_4	1.82	2.75	5.67	7.47	9.39	12.29	23.45	30.76	83.85	84.49
c_5	1.46	2.48	4.98	4.39	7.78	15.67	24.30	27.51	45.09	45.67
c_6	1.19	2.36	3.44	3.62	6.34	6.35	17.51	18.32	29.56	29.86
c_7	1.12	1.48	2.84	4.68	4.69	5.65	16.14	17.10	24.88	25.17
c_8	1.01	1.84	3.18	3.46	5.38	5.80	15.43	16.88	19.78	20.06
c_9	1.09	1.46	2.45	3.18	3.59	4.07	9.48	10.01	15.98	16.69
c_{10}	1.01	1.28	2.45	3.35	3.64	4.28	8.32	9.62	11.34	11.69
c_{11}	1.20	1.24	2.33	3.16	3.23	5.34	5.98	6.57	8.7	8.9
c_{12}	1.10	1.18	2.12	2.91	3.11	3.27	3.89	4.48	7.28	7.84

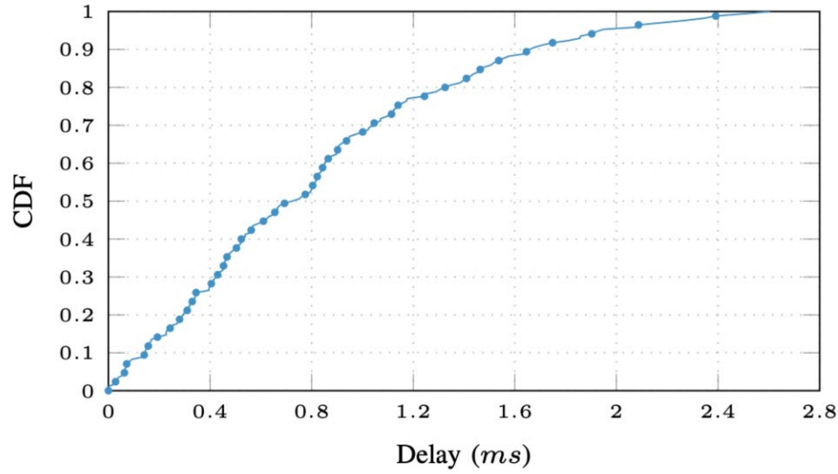


Figure 49 Control Signaling Latency

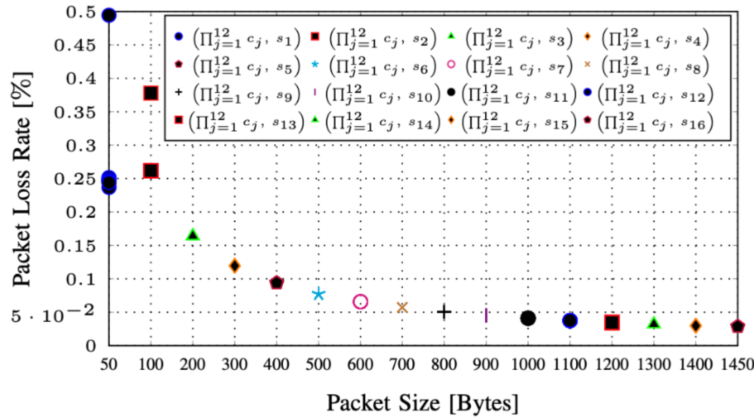


Figure 50 PLR per Packet Size and Number of Rx Queues

d. Packet Loss Rate

Figure 50 presents the PLR (in percentage) plotted against packet size (s_p) and Rx Queue count (c_j) for the uplink scenario, where we fully utilize the system's available bandwidth (i.e., 10 Gbps). For each s_p, we have represented 12 PLR average values corresponding to the values of c_j, we thus obtained s_p-dependent c_j-dependent global

measures. We have made several observations from the graph: (i) - The observed PLR trend resembles an exponential decay function, where the rate of decrease is rapid for smaller packet sizes and gradually levels off for larger packet sizes, approaching an asymptote rather than continuing to decrease linearly. (ii) - The c_j-dependent PLR values generally converge except for c_1 and c_2. In these cases, the high traffic volume- approximately 10 and 8 MPackets/s, for packet sizes of 50 and 100 Bytes, respectively- overwhelms a single Rx Queue, resulting in increased PLR. (iii) - With a sufficient number of Rx queues ($c_j \geq 3$), the PLR remains consistently below 0.15% and eventually stabilizes around 0.03%. This value is negligible in comparison to the total number of transmitted packets, indicating a highly efficient packet delivery. (iv) - The PLR is impacted by the packet volume, not the size. For instance, with 50-Byte packets, we can generate in practice around 10, MPackets/s (based on TRex experience). In contrast, larger packets like 1450 Bytes result in fewer packets per second, reducing the load on each RX queue, since this later can only handle a certain packet rate before overflow. To sum up our findings in this work; we first found that eBPF XDP-based solutions are primarily influenced by the volume of injected traffic and the number of Rx Queues, while packet size affects throughput only at the injection phase.

7.2 Kubernetes Network Operator

In D4.1, we presented our initial concept and design for achieving Kubernetes-based network programmability through the Kubernetes Network Operator. The Network Operator creates an abstraction layer on top of pre-existing Kubernetes-based network technologies (e.g., Skupper/Submariners/etc), by using a common API and Controller logic, which allows the CECCM to create both intra and inter-cluster overlay networks using the most appropriate technology. We divided the 2 core roles of the Operator into Proactive and Reactive orchestration and presented the core design features of each. In this deliverable, we follow this work with a detailed description of the technical design and implementation of the Operator, outlining the core features and benefits achieved, as well as detailing some experimental validation carried out to validate the behavior of the system. Specifically, we present how the Operator proactively creates and configures a network overlay on demand, leveraging existing Kubernetes network technologies (in this work specifically, the Skupper Virtual Application Network), and then reactively configures and adapts this overlay to improve the performance of key network requirements, such as green energy usage.

7.2.1 Network Operator Architecture

In Figure 51 below, we present the architecture of the Network Operator. The core components are the NB API, the Controller, and the Reactive Control module, as well as the specific network technology plugins.

NB API: The Northbound API is the entry point to the system. It is intended to be a central interface to capture the network overlay requirements used to create connectivity. The API is designed to be abstract in order to capture the core network requirements across multiple underlying technologies.

Network Controller: The Network Controller contains the core system logic required to carry out the connectivity requests delivered via the NB API. Primarily, this consists of CRUD operations for the network links, as well as all supporting logic required to establish links across multiple clusters.

Reactive Control: The reactive control module deals specifically with monitoring the behavior of the network and performing various configuration changes to improve the overall operation, in line with some specific requirements.

Plugins: The plugins represent the specific network technologies that are used by the controller to implement the network overlay links.

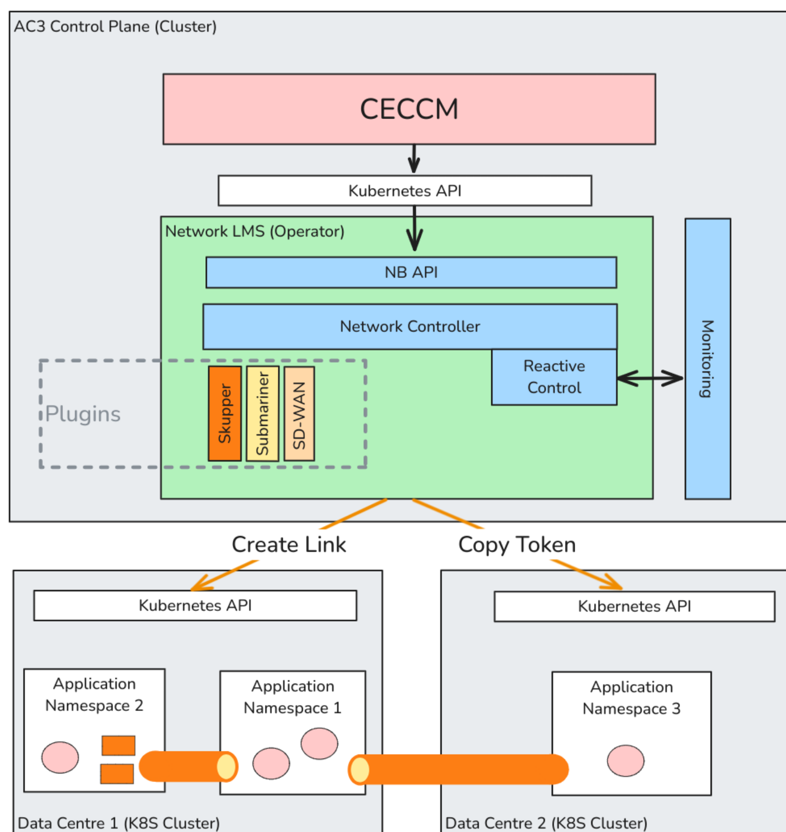


Figure 51 Network Operator Architecture

7.2.2 Proactive Network Lifecycle Management

To enable dynamic and federated networking, the AC^3 Network Operator provides proactive lifecycle management through a technology agnostic approach. At the core of this is a Custom Resource Definition (CRD) that abstracts the desired state of cross-cluster network connectivity. This allows us to create, update and delete network links at the application level declaratively, reducing manual configuration and increasing reliability.

The operator also supports cluster context switching and multi-cluster management, streamlining operations across Kubernetes environments. We enable fine-grained control over how services are exposed and routed, adapting to application-specific requirements. Additionally, we have a flow-collector is place that feeds real-time metrics into the system, this helps us keep track of adjustments that might need to be made on the network that will increase performance.

The AC^3 Network Operator handles network configuration requests by selecting the appropriate network technology through a plugin-based architecture – currently implemented with Skupper.

7.2.2.1 Network Operator API

The AC^3 Network Operator lets you define and manage how applications connect across multiple clusters using a simple, structured Kubernetes resource using Kubernetes Custom Resource Definition (CRD). This CRD supports source-to-target link creation, allowing precise definitions of network intent across environments in the form of Cluster → Application → Service. Within each AC^3 Network resource, the sourceCluster and targetCluster fields specify which clusters are involved in the connection — with the source cluster initiating the link and the target cluster(s) receiving it. The sourceNamespace and targetNamespace fields localize this operation, identifying where applications reside. The support for multiple targetNamespace entries allows fan-out connectivity, enabling a single source to link to multiple destinations. The applications field lists the specific services (e.g.,

nginx, rabbitmq) to be exposed across clusters, guiding the operator in automating service exposure, proxy configuration, and routing. To secure the setup, the operator handles token-based authentication by referencing secretNamespace, secretName, and secretName2 — ensuring certificate-based tokens are generated, distributed, and trusted between clusters. This CRD-driven workflow abstracts complex, error-prone networking tasks and forms the foundation for consistent, scalable network programmability across heterogeneous Kubernetes environments.

```yaml
apiVersion: ac3.redhat.com/v1alpha1
kind: AC3Network
metadata:
  name: ray-ac3network
  namespace: ac3no
spec:
  link:
    sourceCluster: "ac3-cluster-2"
    targetCluster: "ac3-cluster-1"
    sourceNamespace: "sk1"
    targetNamespace:
      - "sk3"
      - "sk4"
    applications:
      - "nginx"
      - "rabbitmq"
    secretNamespace: "sk1"
    secretName: "sk1-token"
    secretName2: "sk1-token"
    port: 5672

```

Figure 52 Network Operator Custom resource

7.2.2.2 Network Controller

As stated previously, the network controller implements all the core logic for overlay network lifecycle management, including creation, validation, deletion, and updating of the network, as well as the supplemental logic for creating and managing links across multiple clusters.

7.2.2.2.1 Link Management

The Network Operator is designed to simplify and automate the process of building a dynamic, federated application network across Kubernetes clusters. At its core, the operator handles the provisioning of Skupper resources by creating ConfigMaps tailored to each namespace and deploying the Skupper router to initialize them as Skupper sites. When a new AC^3Network custom resource is applied, the operator automates the creation of secure application links by generating certificate-based Skupper tokens in the source namespace and securely copying them to the target namespaces. This process manages the secure exchange of credentials and ensures that cross-cluster communication is established seamlessly, all behind a simple declarative interface. As new namespaces are added to the network, links are dynamically created, allowing for scalable, multi-site topologies to form without requiring manual configuration. The system is resilient and adaptable, capable of responding to topology updates or site changes with minimal disruption. Beyond connectivity, the operator is also responsible for exposing services: it continuously monitors Kubernetes Deployments and StatefulSets, injecting the necessary Skupper proxy annotations—such as those shown in Figure 53—that enable secure, service-level routing across the network. This eliminates the need for complex ingress setup or external DNS management. By orchestrating configuration, link creation, token distribution, and service exposure, the Network Operator abstracts the

complexity of multi-cluster networking and delivers a powerful, declarative tool for platform engineers and developers. The result is a robust, energy-aware networking layer that can scale with applications, adapt to changes in infrastructure, and provide consistent connectivity and observability across environments.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  # annotations:
  #   skupper.io/proxy: tcp
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginxinc/nginx-unprivileged:stable-alpine
        imagePullPolicy: IfNotPresent
        name: nginx
        ports:
        - containerPort: 8080
          name: web
          protocol: TCP
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        terminationGracePeriodSeconds: 30
```

Figure 53 Skupper Proxy annotation

7.2.2.2.2 Cluster management

To support operations across multiple Kubernetes clusters, the operator includes built-in Cluster Management and Context Switching. It can interface with multiple kubeconfigs and switch contexts on the fly, applying configurations or fetching resources from the appropriate cluster. This abstraction layer removes the operational burden of manual context switching and makes automation workflows cluster-agnostic.

7.2.2.2.3 Monitoring

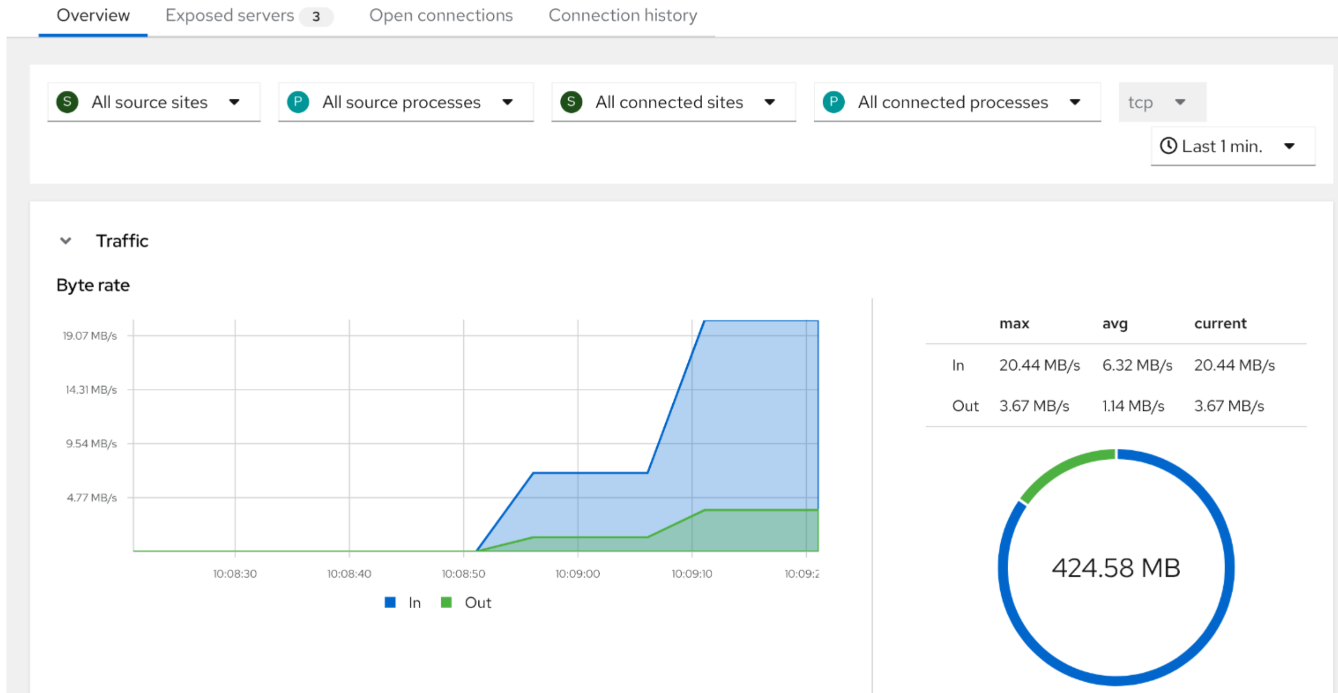


Figure 54 Monitoring

The Flow Collector is a tool provided by Skupper that provides a detailed, real-time view of traffic flowing through the Skupper network. When running a test, we can inspect how traffic behaves by selecting an exposed service within the Flow Collector interface. This reveals several informative tabs, including Overview, Exposed Servers, Open Connections, and Connection History. The Overview tab presents a high-level visualization of traffic flow, including byte rate and throughput trends, offering insight into how the service is performing during the test. This visibility is essential for understanding routing behavior, identifying potential bottlenecks, and evaluating the impact of configuration changes such as CPU scaling or link cost adjustments.

Skupper Router Topology:

Router ID	Next Hop	Link	Ver	Cost	Neighbors	Valid Origins
ns1-skupper-router-59f5c657f-v89ff	(self)	-	1	-	ns2-skupper-router-7c49448f59-jzr9v, ns3-skupper-router-779cb95d9-qm8b8, ns4-skupper-router-647776bf55-wv6vk	[]
ns2-skupper-router-7c49448f59-jzr9v	-	0	1	177	ns1-skupper-router-59f5c657f-v89ff, ns4-skupper-router-647776bf55-wv6vk, ns3-skupper-router-779cb95d9-qm8b8	[]
ns3-skupper-router-779cb95d9-qm8b8	ns2-skupper-router-7c49448f59-jzr9v	-	1	384	ns1-skupper-router-59f5c657f-v89ff, ns2-skupper-router-7c49448f59-jzr9v	[]
ns4-skupper-router-647776bf55-wv6vk	-	2	1	113	ns1-skupper-router-59f5c657f-v89ff, ns2-skupper-router-7c49448f59-jzr9v	[]

Figure 55 Router Connectivity

Once inside the router, we can inspect the status of the routing network as seen in Figure 55, thus showing all routing instances. For each router, it lists key information such as the Router ID (the pod name), the Next Hop (used for indirect routing paths when a direct link is unavailable), the Link index (used internally to track router links), the Version of the Skupper protocol (usually 1), the Cost to reach the router (with lower values indicating more preferred paths), and a list of Neighbors—routers that are directly connected. Additionally, the Valid Origins field is shown, though it typically remains empty unless you're working with advanced routing

configurations. This output is essential for diagnosing network issues, understanding traffic flow, and verifying the overall mesh connectivity. So, in this example, we were inside the ns1 router.

7.2.3 Reactive Network Lifecycle Management

While proactive configuration enables planned and predictable connectivity, real-world networks often face unexpected changes and performance shifts. Reactive Network Lifecycle Management focuses on how the system responds to these dynamic conditions in real time. The AC^3 Network Operator is being extended to observe, detect, and respond to issues such as application load changes, network latency, and link failures. By integrating monitoring data and implementing responsive logic, the operator can make adjustments that maintain connectivity, performance, and resilience without manual intervention. This approach lays the groundwork for smarter, self-adjusting networks that evolve based on actual usage and conditions.

7.2.3.1 Adaptive Network Resource Allocation

In a multi-site service network powered by Skupper, inter-cluster communication is mediated through Skupper routers deployed in each namespace. These routers are responsible for managing traffic flow, connection establishment, and message forwarding between sites. However, under high traffic volumes—especially in scenarios with a lot of workloads, such as connection surges- we observed a significant decrease in router performance. The underlying issue is CPU saturation. As routers approach their CPU limits, their ability to process incoming connections and forward messages deteriorates. This results in increased latency, slower connection setup times, queuing delays, and reduced throughput—all of which negatively impact the responsiveness and reliability of the network. Testing confirmed that router performance is tightly linked to available compute resources. Manually increasing CPU allocations led to improved connection responsiveness and higher throughput, demonstrating that additional CPU headroom can directly mitigate the bottleneck.

Dynamic Router Auto-scaling

Our goal is to create a reactive and self-adjusting network environment by enabling automated scaling of Skupper router resources in response to runtime conditions. By dynamically adjusting CPU and memory allocations, we aim to maintain optimal router performance even as workloads fluctuate. This capability would help ensure consistently low connection latency and high throughput, regardless of traffic patterns. Automating resource provisioning also reduces operational overhead and increases network reliability. Ultimately, this work contributes to a broader vision of a responsive, programmable network infrastructure—one that can intelligently adapt to changing conditions without human intervention.

To realize this goal, we implemented a simple, yet effective threshold-based scaling mechanism focused on CPU utilization. The strategy is grounded in the principle: CPU util > X%, increase, meaning that when CPU usage crosses a predefined threshold, the system responds by increasing resource allocations. This approach involves three core components: monitoring, triggering, and scaling. First, we monitor the CPU usage of Skupper router pods in real time using Prometheus, which collects metrics from across the cluster. This monitoring layer provides the visibility needed to detect when a router is approaching saturation. Next, we define CPU utilization thresholds that serve as triggers for action. For example, if CPU usage exceeds 75% for a sustained period, it indicates that the router is under load and requires additional compute resources. This threshold acts as a decision point: CPU util > 75% → scale up. Optionally, a lower threshold can be set to scale down during low-demand periods, promoting efficient resource usage. When the upper threshold is breached, the system initiates a scaling action. This involves patching the router's deployment or StatefulSet in Kubernetes to increase the CPU (and potentially RAM) resource requests and limits. Updates are applied using rolling strategies to ensure service continuity and avoid downtime. By starting with a clear, rule-based system, we establish a reactive foundation for dynamic resource allocation. This lays the groundwork for future enhancements, such as predictive scaling, integration with traffic metrics, or multi-router coordination for more holistic optimization.

7.2.3.1.1 Experimental Validation: Impact of CPU on Router Latency

To validate our hypothesis that CPU allocation directly impacts router performance, we conducted a series of HTTP/2 benchmark tests. Our experimental setup involved directing 10,000 concurrent connections to a single Skupper router. We systematically varied the CPU allocated to the router, starting from 0.5 CPU and increasing it stepwise. The results, as depicted in the "Latency vs. CPU" graph, clearly demonstrate a strong inverse relationship between allocated CPU resources and router latency. Initially, at lower CPU allocations, latency was significantly high. However, as we progressively increased the CPU, we observed a dramatic and consistent reduction in latency, confirming that boosting CPU resources is an effective strategy for improving router responsiveness and throughput under heavy connection loads. This empirical evidence solidified our understanding of the problem and underscored the necessity of a dynamic resource allocation mechanism.

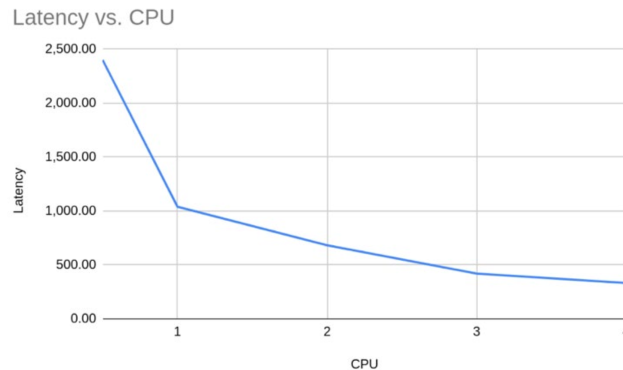


Figure 56 CPU Resource allocation

7.2.3.2 Adaptive Routing

One of the most impactful aspects of our reactive work is the implementation of a cost-based link system. This mechanism allows the operator to influence how traffic is routed across the network by assigning relative costs to each available path. In systems like Skupper, a lower cost means a more preferred path, enabling us to control traffic distribution in real time. This is particularly valuable when workloads spike, as it lets the operator shift traffic dynamically to less loaded or more energy-efficient sites. In the context of supporting green energy initiatives, this capability allows traffic to be routed based on sustainability criteria — for example, preferring data centers powered by renewable energy sources. By adjusting link costs programmatically, we create the foundation for a responsive and energy-aware network fabric, capable of optimizing not just performance, but also environmental impact.

7.2.3.2.1 Skupper Link Cost and Routing Overview

At its core, Skupper's routing mechanism assigns an integer weight, or "cost", to every site link. By default, this cost is set to 1. The fundamental principle is that Skupper routes traffic along the path with the lowest cost, which is calculated as the sum of link costs across all hops in a multi-hop route. Consequently, a lower cost makes a path more preferred for traffic. This cost mechanism acts as both a link weight and a threshold, influencing how paths are chosen and when alternative links are considered.

Skupper employs Dijkstra's algorithm for path selection. For each site in the network, Skupper determines the shortest path (i.e., the lowest cost path) to every other site. This path calculation is dynamic and automatically recalibrates if links are added or removed from the network topology. In terms of routing, Skupper prioritizes the lowest-cost links. However, there's a consideration for link capacity: a link's capacity is defined by its base cost plus the number of current connections. If the combined cost and current connections exceed this defined capacity (threshold), Skupper will then select other, less preferred links to route traffic.

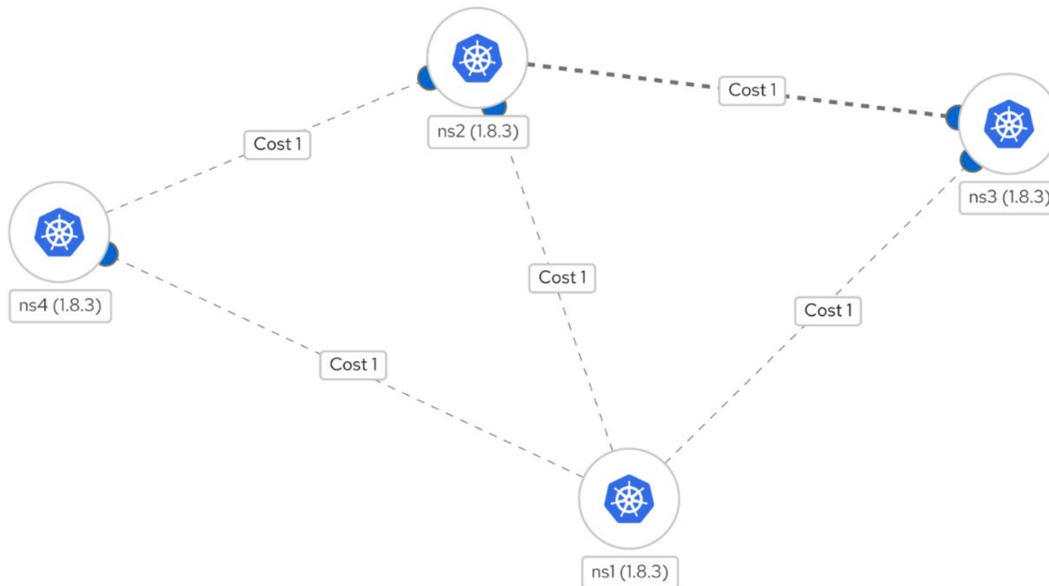


Figure 57 Baseline Network Topology

In the baseline cost topology Figure 57, all links between network sites are assigned a default, static cost of 1. This configuration reflects a simple and uniform routing strategy where no path is favoured over another, resulting in equal weighting across all available routes. This model is ideal for demonstrating default behavior in a network where routing decisions are made solely based on connectivity rather than any performance or efficiency considerations. The uniformity of link costs ensures deterministic behavior, with all routes treated equally by the routing algorithm regardless of traffic load, latency, or resource consumption.

7.2.3.2.2 Adaptive Routing Framework

In our work, we aim to strategically utilize Skupper's cost-based routing to achieve fine-tuned traffic distribution without resorting to physical topology changes. Our primary goal is to fine-tune traffic distribution by adjusting the cost of inter-site links instead of altering the network topology. This approach offers significant flexibility because the link costs are configurable, allowing us to dynamically adapt them based on various non-functional requirements.

For instance, we can adjust link costs to prioritize traffic flow based on cluster sustainable energy ratios. A link to a site with a higher proportion of renewable energy could be assigned at a lower cost, encouraging more traffic to flow through it. Similarly, we could factor in node energy efficiency, assigning lower costs to links connected to more power-efficient nodes. Other critical non-functional requirements, such as latency, could also be incorporated; links with lower anticipated latency could be given lower costs. By dynamically adjusting these costs, we can ensure that Skupper's routing algorithm inherently favors paths that align with our specific objectives, ultimately leading to a more intelligently distributed and optimized network. Skupper cost allocation is left to the user and is arbitrarily set to their needs and is done on a link-by-link basis.

7.2.3.2.3 Case Study: Energy-Aware Routing Cost Model

In this section, we explore a specific adaptive routing model to promote sustainable energy usage within the network. The core aim is to essentially redirect ‘load’ (both networking and computational) throughout the network based by prioritizing sites with better sustainable energy profiles.

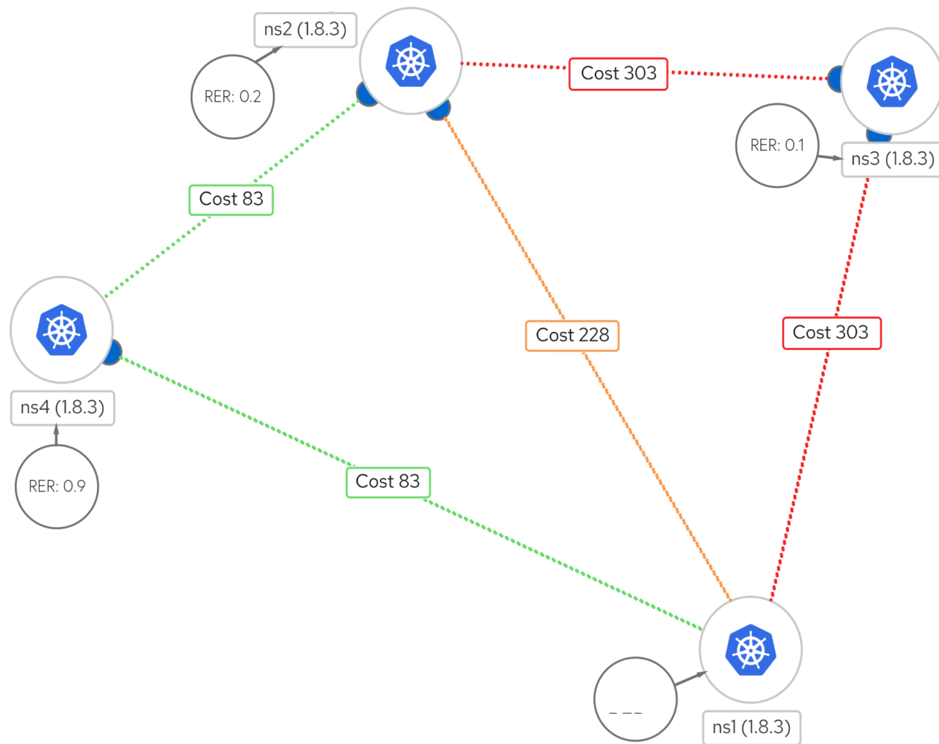


Figure 58 Adaptive Routing Topology

The energy-based route cost allocation topology introduces dynamic, differentiated costs based on improving sustainable energy usage or resource impact of the available paths. In this model, links are assigned variable costs to reflect their relative energy consumption or sustainability footprint. For example, the links between ns1 and ns4, and between ns4 and ns2, are assigned a relatively low cost of 83, indicating they are more energy-efficient paths. In contrast, the links from ns2 to ns3 and from ns3 back to ns1 have much higher costs of 303, suggesting higher energy usage or environmental impact. The direct link between ns1 and ns2 is assigned a moderate cost of 228. This allocation allows the network to prioritize greener or more efficient routes dynamically, supporting sustainability goals while still maintaining connectivity.

To help make networks more environmentally friendly, we use an energy-aware routing model that chooses paths based on how much renewable energy each site uses. Most routing systems focus only on speed or efficiency, without thinking about the energy behind them. Our model changes that include energy usage—especially renewable energy—as part of how we decide which paths to use. This helps reduce the use of non-renewable energy and guides network traffic through "greener" routes.

To rank the energy sustainability of each location, we use a metric called the **Renewable Energy Ratio (RER)** Figure 59**,** as presented there. This is a number that shows how much of a site's energy comes from renewable sources. It's calculated by dividing the renewable energy used by the total energy the site uses. The result is a value between 0 and 1—where 0 means the site uses only non-renewable energy, and 1 means it's fully powered by renewables. We use this value to measure how environmentally friendly each site is.

$$RER = \frac{RenewableEnergyConsumption}{TotalEnergyConsumption}$$

Figure 59 Renewable Energy Ratio

Once we know the RER for each site connected by a network link, we calculate the **Link Cost (LC)** Figure 60. This is done by taking the average of the inverse (1 divided by the RER) of each site's value. Sites that rely more on non-renewable energy will make the link cost higher. This means traffic is less likely to go through those less

sustainable sites. On the other hand, links between greener sites (with high RER values) will have lower costs and are more likely to be used.

$$LC_i = \frac{\frac{1}{RER_a} + \frac{1}{RER_b}}{2}$$

Figure 60 Link Cost

Finally, as Skupper effectively uses the cost as a threshold for the number of connections on a link, we must set the cost relative to activity (connections) in the network, or risk the cost threshold being significantly too low or too high. To do this we use a value called the **Energy Link Cost (ELC)** (Figure 61)**,** which divides the network's total request load across the links in a smart way. The ELC is found by scaling each link's cost against the total cost of all links and multiplying it by the total traffic. In simple terms, links with lower costs (greener paths) get more traffic, and high-cost (less green) links get less. This makes the whole network more energy-aware and helps it use renewable energy wherever possible.

$$ELC_i = \frac{LC_i}{\sum_{i=1}^{n} LC_i} * RequestRate$$

Figure 61 Energy Link Cost

7.2.3.2.4 Experimentation

To assess the impact of our energy-based routing strategy, we conducted a series of tests comparing two routing models, a "Baseline" setup where all the links have a default cost of 1, and then our adaptive routing energy-based routing scheme "Scheme 1". In the energy-aware setup, the links costs were set based on renewable energy ratios at each site. Our goal was to measure the effect of these cost allocations on four key tests at the router level across 3 sites ns2, ns3, and ns4. These tests included overall energy consumption, green energy usage, fossil fuel usage, and CPU usage. Each test was recorded for both routing schemes and compared them side by side to evaluate their performance.

Client-Server Distribution:

Our test setup designates ns1 as the client site, originating all traffic. This traffic is then directed towards ns2, ns3, and ns4, which host the application servers. Consequently, the routing decisions illustrated in the graphs reflect how Skupper routes these client requests from ns1 to the respective application sites, driven by our configured link costs. As seen in Figure 62

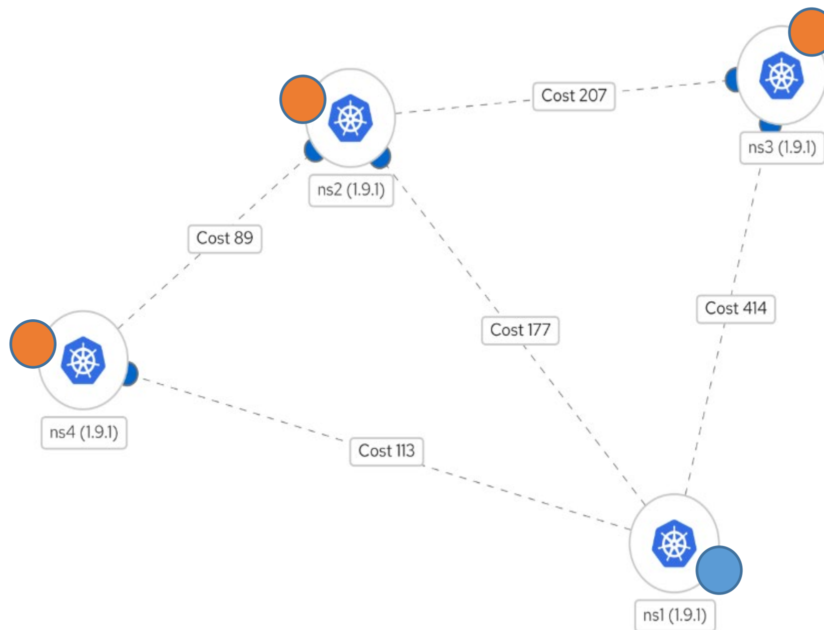


Figure 62 Experimental setup

Request Rates:

While request rates can vary in a real-world scenario, for these specific tests, we maintained a fixed request rate of 1000 connections. This standardization is vital as it ensures that any observed differences in CPU utilization or distribution results between "Baseline" and "Scheme 1" are directly attributable to the routing logic and not to fluctuations in client demand. It provides a consistent load against which to evaluate the routing schemes.

Benchmark tests:

To validate the performance and behavior of our Skupper-based network, we designed a comprehensive benchmarking setup that simulates a federated multi-namespace, multi-cluster topology. We created four isolated namespaces (ns1 through ns4), with ns1 acting as the central client and the others hosting instances of the h2load-server application. Skupper was initialized in each namespace, with ns1 configured with the web console and flow collector for visibility into traffic flow and latency. Applications in ns2, ns3, and ns4 were exposed over Skupper, and inter-namespace links were established using token-based authentication. Crucially, each link was assigned a custom cost value, allowing us to simulate dynamic routing decisions based on link preference — a key feature for traffic optimization. For example, links with lower costs (e.g., cost 1) were prioritized for routing over higher-cost links (e.g., cost 60), enabling us to control and test how workloads would flow across the network. With this environment in place, we ran benchmarking tests using the benchdog-h2load-client image from ns1, simulating high-load HTTP/2 traffic scenarios. This allowed us to analyze network performance, latency, and the effectiveness of Skupper's routing behavior in a controlled, observable setup. The tests confirmed how link costs and resource provisioning (like CPU) at the router level impact overall responsiveness and workload distribution.

Exposed services:

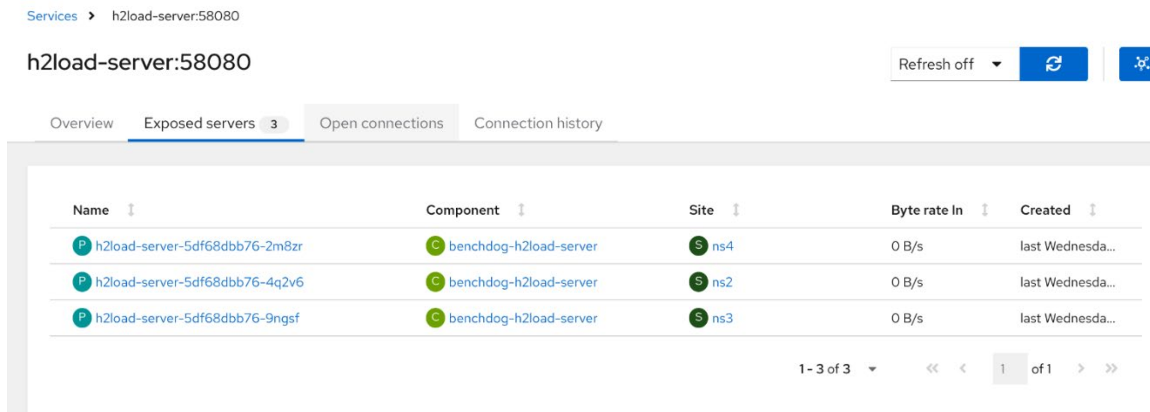


Figure 63 Exposed Servers

We can see in Figure 63 that services, such as h2load-server, are successfully exposed across the network. This confirms that each application is discoverable and registered within the network fabric, ready to accept traffic. It's a key validation step to ensure the benchmark environment is properly initialized.

Process and Site Pairs

Figure 64 Traffic flow

Figure 64 shows which sites are actively connected and exchanging data. We can clearly observe the communication paths forming between the client in ns1 and each of the target namespaces (ns2, ns3, and ns4). This helps verify that the topology is behaving as intended, especially when working with multiple link costs and routing strategies.

Connection History:

Client | Search by client

Closed	Client	Client Site	Port	Data Out	Latency Out	Server	Server site	Data In	Latency In	Duration
today at 10:25 AM	h2load-client	ns1	35194	485.72 KB	28.95 ms	h2load... 6-2m8zr	ns4	2.64 MB	487 µs	64.97 sec
today at 10:25 AM	h2load-client	ns1	35208	405.08 KB	17.37 ms	h2load... 6-4q2v6	ns2	2.2 MB	963 µs	64.96 sec
today at 10:25 AM	h2load-client	ns1	35220	533.78 KB	33.91 ms	h2load-... 6-9ngsf	ns3	2.9 MB	2.85 ms	64.95 sec

Figure 65 Connection history

In Figure 65, a timeline of how connections are formed, maintained, and closed between services. We can track things like latency, data volume, and how long sessions persist. This is particularly valuable during benchmarking, as we can observe how traffic shifts over time—such as whether the network routes traffic through the lowest-cost link (ns4) under normal load, and when it starts to favour higher-cost links due to saturation or CPU limits.

Details

Connection Closed

Trace
ns1 -> ns4 -> ns2
Duration
65.05 sec

Client	Server
Process	**Process**
h2load-client	h2load-server-5df68dbb76-4q2v6
Host	**Host**
10.128.1.148	10.128.1.240
Port	**Port**
56816	46714
Bytes transferred	**Bytes transferred**
401.04 KB	2.18 MB
Byte unacked	**Byte unacked**
44.49 KB	93.79 KB
Window Size	**Window Size**
1.39 MB	1.39 MB
Latency	**Latency**
12.75 ms	1.36 ms

Figure 66 Flow trace details

In this detailed view, we're able to break down traffic flows by source and destination, looking at metrics like total bytes sent, latency per flow, and which route was taken. This provides a clear picture of how traffic is distributed in practice and how performance is impacted by factors like CPU scaling or link configuration. It gives us the data needed to tune the network and validate that changes (like increased CPU at the router or adjusted link costs) are having the desired effect.

Energy Calculation Assumptions

Our work makes specific assumptions regarding the calculation of energy usage at the site level. We do not dynamically measure real-time energy consumption at the sites, but rather use approximations based on CPU usage and a fixed CPU energy profile. Specifically, we define this as CPU rated at 150 watts maximum and employing a simple linear utilization model for energy consumption. This provides a baseline power consumption figure for the computing resources at each site, allowing us to quantify the energy impact of CPU utilization changes.

Test Pre-requisites:

Namespace	RER Value
ns1	0.2
ns2	0.5
ns3	0.1
ns4	0.9

Figure 67 Skupper Proxy annotation

Scheme 1 Costs:

Link	Cost
ns1 - ns2	177
ns1 - ns3	414
ns1 - ns4	113
ns4 - ns2	89
ns2 - ns3	207

Figure 69 Scheme 1 costs

Baseline costs:

Link	Cost
ns1-ns2	1
ns1-ns3	1
ns1-ns4	1
ns4-ns2	1
ns2-ns3	1

Figure 68 Baseline costs

7.2.3.2.5 Results

The core aim of this work was essentially to redirect 'load' (both networking and computational) throughout the network based on the sustainable energy profile of the sites in the network. Load in this context can be directly interpreted as CPU utilisation. In Figure 68 and Figure 69, we present both the network path distribution and CPU utilisation profiles for both Scheme1 and our Baseline, in order to demonstrate how our approach redistributes traffic and processing workloads toward sites with greater renewable energy availability.

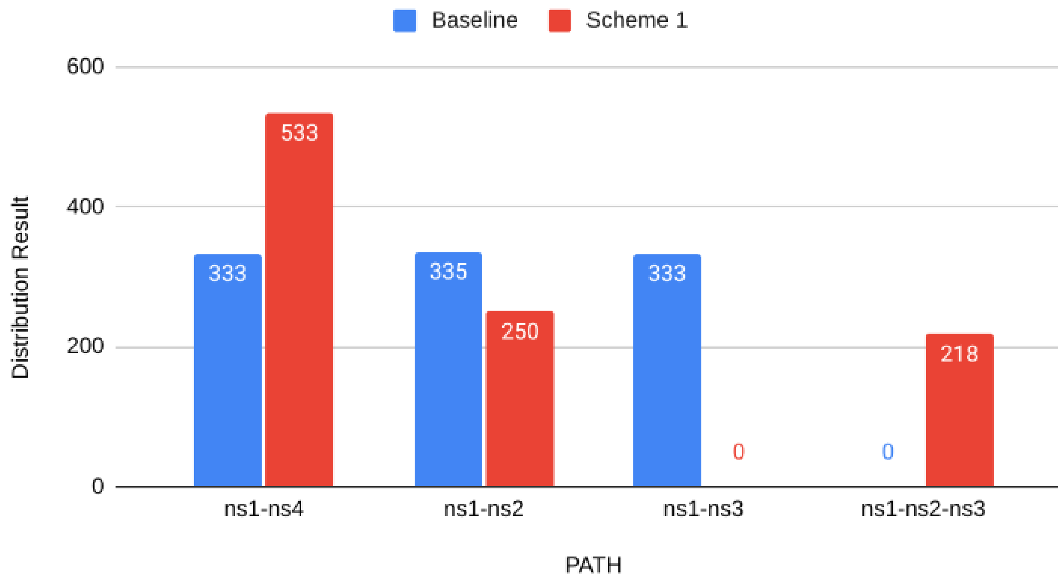


Figure 70 Traffic distribution

Figure 70 illustrates how "Scheme 1" significantly alters traffic distribution compared to the "Baseline" across different network paths. For the ns1-ns4 path, the "Baseline" shows a distribution result of 333, which dramatically increases to 533 under "Scheme 1." This substantial rise indicates that "Scheme 1" heavily prioritizes this path. Given previous information about ns4 having a lower assigned cost (113) and a high Renewable Energy Ratio (0.9), this surge confirms that traffic is being strongly redirected towards this more energy-efficient and reliable route. Conversely, for the ns1-ns2 path, the "Baseline" has a distribution result of 335, which decreases to 250 in "Scheme 1." This reduction suggests that while ns2 remains a viable option, "Scheme 1" routes less traffic through it compared to the baseline, possibly due to a higher relative cost (177) or to further optimize green energy usage by favoring ns4.The ns1-ns3 path shows the most drastic change, with a "Baseline" distribution result of 333 dropping to 0 under "Scheme 1." This complete cessation of traffic for this direct path is a direct consequence of ns3's very high assigned link cost (414) and low Renewable Energy Ratio (0.1) in "Scheme 1." This clearly demonstrates "Scheme 1's" effectiveness in avoiding less energy-efficient routes. Finally, a new path, ns1-ns2-ns3, appears in "Scheme 1" with a distribution result of 218, while it had no traffic in the "Baseline." This indicates that when the direct ns1-ns3 path is effectively blocked due to its high cost, "Scheme 1" intelligently finds an alternative multi-hop route through ns2 to reach ns3. This demonstrates the routing scheme's ability to maintain connectivity while still attempting to optimize path selection, potentially leveraging ns2's relative efficiency even when the ultimate destination (ns3) is less desirable from an energy perspective. In summary, this graph strongly confirms that "Scheme 1" effectively manipulates traffic distribution by dynamically adjusting path costs. It successfully steers traffic towards energy-preferred routes (like ns1-ns4), reduces reliance on less efficient direct paths (like ns1-ns3), and intelligently utilizes multi-hop alternatives, when necessary, thereby aligning network behavior with energy-aware objectives.

Correlating with this altered traffic distribution, we see corresponding changes to the CPU utilisation profiles across the sites.

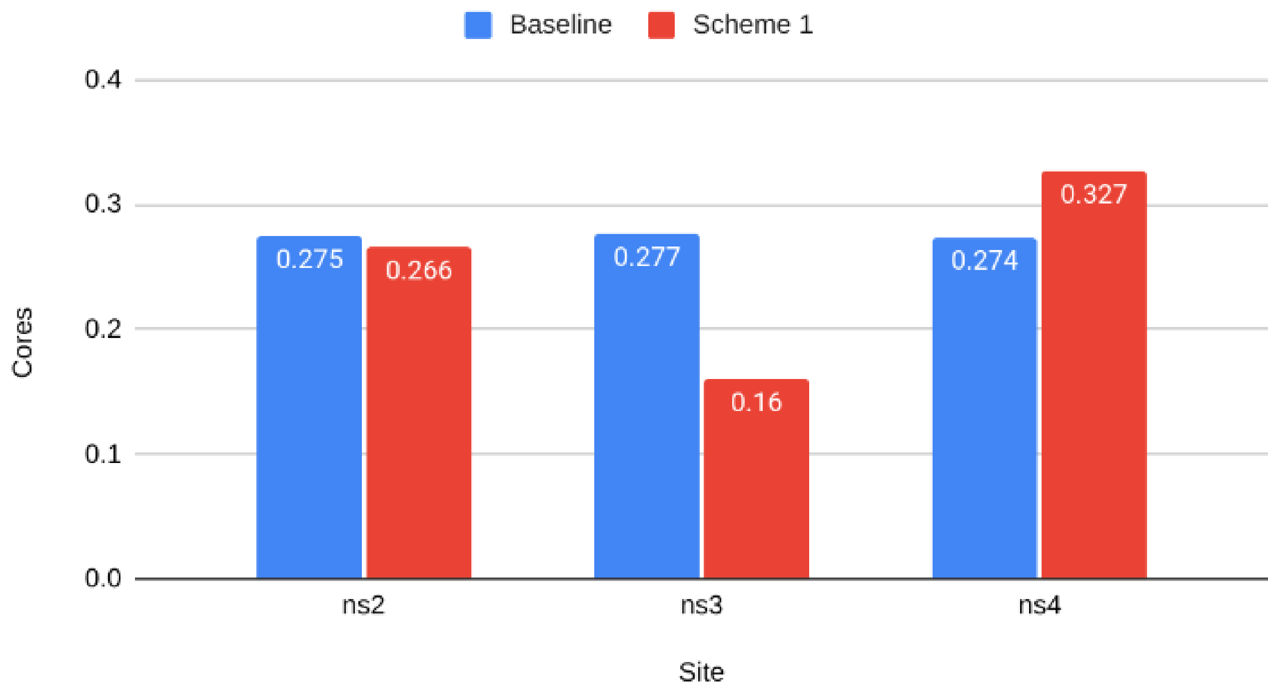


Figure 71 Router CPU Usage

At ns2, CPU usage sees a minor reduction, indicating that while its role in handling traffic remains significant, some workload was subtly redirected, aligning with its increased link cost in Scheme 1. The most pronounced change occurs at ns3, where CPU usage plummets due to Scheme 1's strategic routing, directly attributable to its very high assigned link cost and low Renewable Energy Ratio (RER). This significantly reduced CPU utilization at ns3's router demonstrates that it is processing considerably less data, contributing to the objective of minimizing fossil fuel consumption at less energy-efficient nodes. Conversely, ns4 experiences an increase in CPU usage. This surge is consistent with the energy-aware routing strategy, as its lower assigned link cost and high RER led to it being actively favored by Scheme 1 to handle more traffic.

The heightened CPU activity at ns4's router signifies that it is now processing a greater volume of data, effectively leveraging its higher renewable energy availability and supporting the overarching goal of promoting green energy usage within the network. In summary, the "Router CPU" graph powerfully illustrates the efficacy of "Scheme 1" in directing traffic flows to achieve energy-aware objectives. The observed shifts in CPU usage—a decrease at ns3 and an increase at ns4—directly correlate with the strategic redirection of traffic away from less energy-efficient nodes and towards those with a greater reliance on renewable energy. This dynamic redistribution of workload, reflected in CPU utilization, is a crucial mechanism through which the scheme optimizes overall energy consumption.

Following on from the expected shift in CPU utilisation, Figure 72 and Figure 73 below show the corresponding changes in both Sustainable and Fossil-based energy usage, respectively.

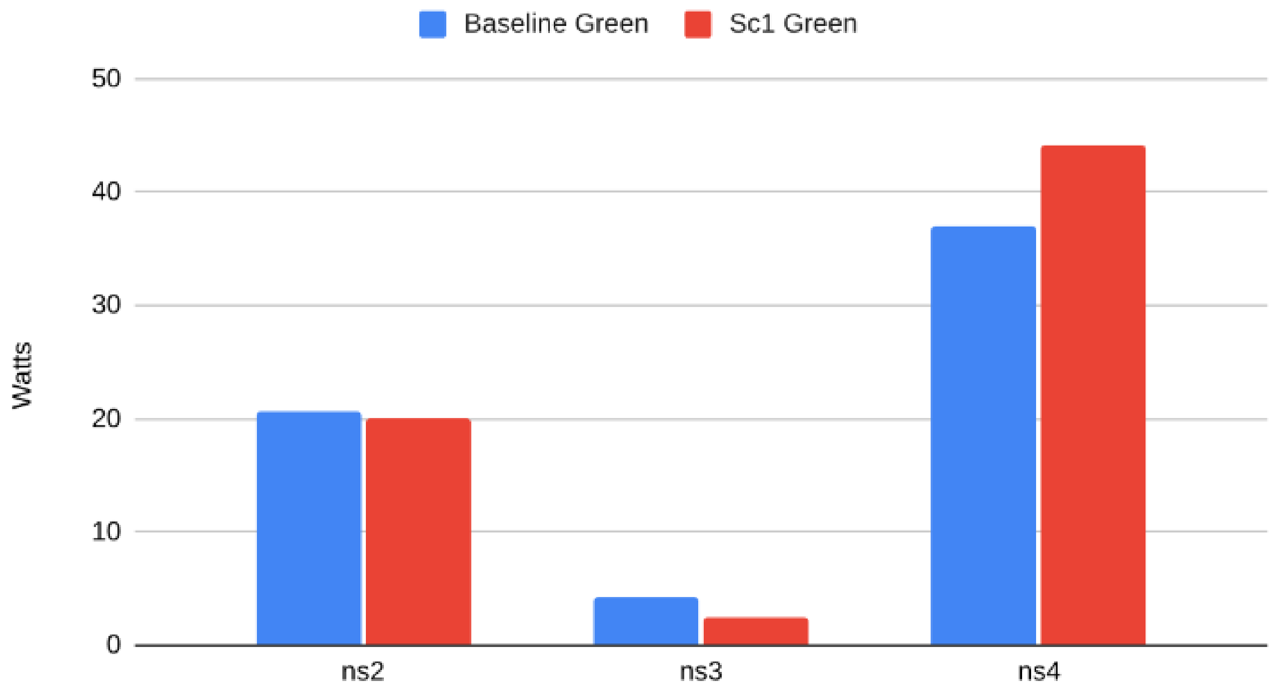


Figure 72 Green based Energy usage

Namespace ns4, which had the highest RER (Renewable Energy Ratio) value of 0.9, showed a noticeable increase in green energy usage under Scheme 1 compared to baseline. This uptick indicates that the routing algorithm effectively prioritized traffic through ns4, leveraging its high renewable energy availability. This outcome aligns with the goal of directing traffic toward sites with a greater capacity for sustainable energy consumption.

Namespace ns2, with a moderate RER value, maintained relatively consistent green energy usage between the baseline and Scheme 1, reflecting its continued but balanced role in traffic handling. Meanwhile, ns3, which had the lowest RER value of 0.1, demonstrated minimal green energy usage in both scenarios. This confirms that the system avoided routing traffic through ns3 due to its limited renewable energy contribution, reinforcing the strategy's alignment with energy-aware principles.

Overall, the data illustrate Scheme 1's effectiveness in steering network traffic toward nodes with better green energy profiles. By emphasizing routes that optimize for both reliability and sustainability, the system ensures that greener nodes like ns4 play a more central role in network operations, thus supporting environmentally conscious network programmability.

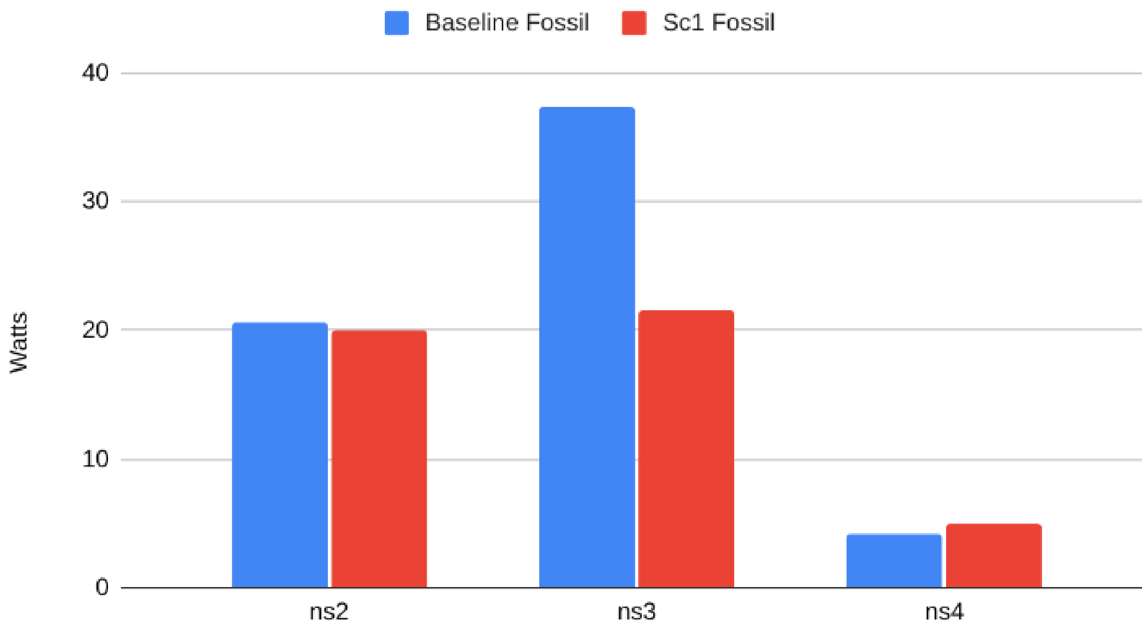


Figure 73 Fossil Fuel energy usage

Figure 73 clearly demonstrates the positive impact of Scheme 1's adaptive, cost-aware routing on sustainable energy across the network. In the baseline scenario, where all link costs were equal, namespace ns3 consumed the highest amount of fossil fuel energy, indicating that it handled a significant portion of the traffic. However, under Scheme 1, where a much higher link cost was assigned to ns3 (cost = 414), its fossil fuel usage dropped substantially. This suggests that traffic was intentionally and effectively routed away from ns3, resulting in a notable reduction in energy consumption at that site.

Namespace ns2 saw only a slight decrease in fossil fuel usage under Scheme 1, despite its cost being increased to 177. This indicates that it remains a relatively efficient and viable routing option. Meanwhile, namespace ns4, which had a lower assigned cost (113) and a high RER value (0.9), experienced a minor increase in fossil energy usage. This shift implies that Scheme 1 redirected some traffic toward ns4, capitalizing on its high reliability and lower relative energy impact. Overall, these results validate the effectiveness of Scheme 1's routing strategy in not only optimizing end-to-end message delivery but also aligning network behavior with energy-aware goals. By leveraging cost adjustments based on reliability and energy profiles, the scheme significantly reduced fossil fuel consumption of less efficient nodes.

Finally, as well as a change in the router energy profiles, we also expect to see a shift in the energy usage of the services themselves, based on reduced traffic arriving at these sites as a result of targeted link costs. This can be seen in Figure 74 below.

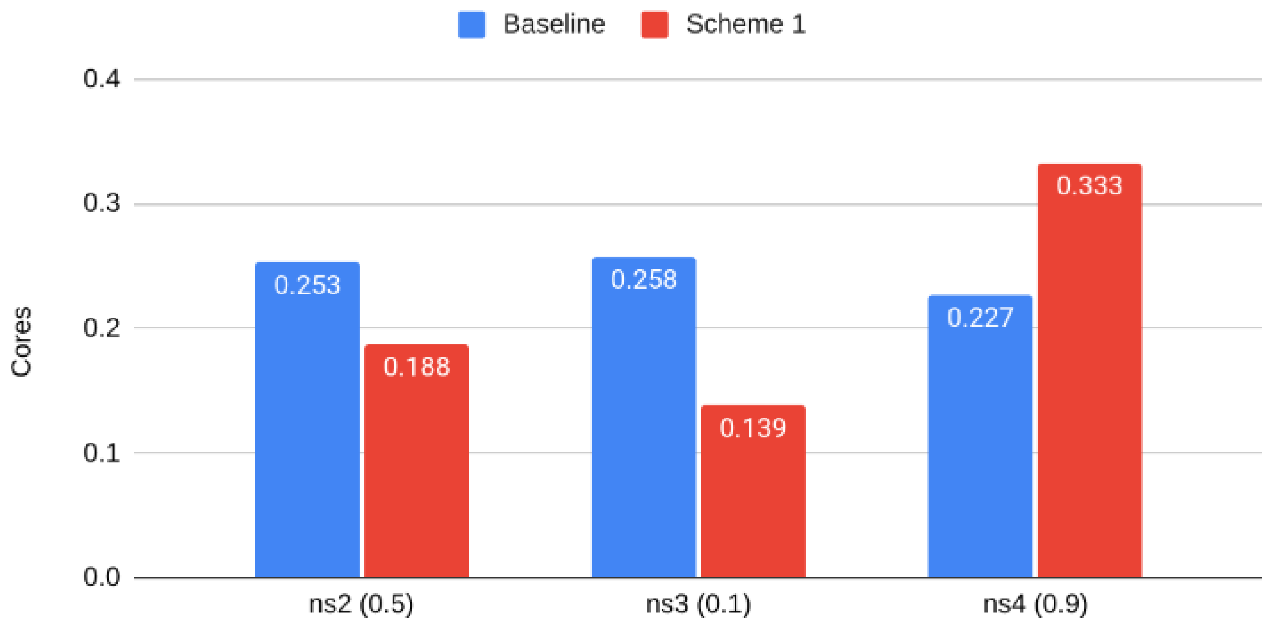


Figure 74 Server/Application CPU Usage

The Server CPU chart, which depicts server CPU utilization measured in cores, clearly illustrates the impact of "Scheme 1" on application workload distribution. The server at ns2 shows a decreased CPU usage (from 0.253 to 0.188 cores), indicating a reduction in application load as some traffic was subtly redirected. Most notably, ns3's server CPU usage drastically drops (from 0.258 to 0.139 cores), reflecting the routing scheme's deliberate redirection of traffic away from this less energy-efficient site. Conversely, ns4's server CPU usage significantly increases (from 0.227 to 0.333 cores), demonstrating that "Scheme 1" effectively prioritized this more energy-efficient site, resulting in its application server handling a substantially higher volume of requests. This data directly supports that the routing scheme successfully shifts computational load across application servers to align with energy optimization goals.

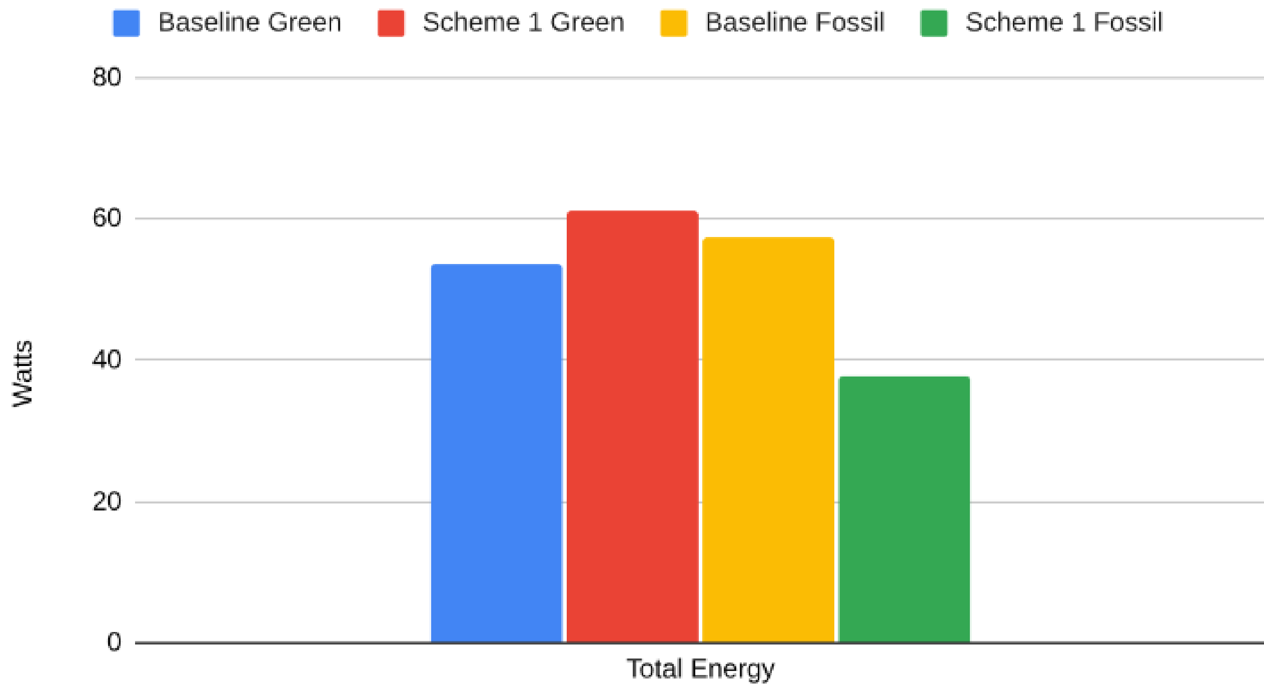


Figure 75 Total Energy

Figure 75 provides an overarching view of the network's total energy consumption, consolidating data from all individual locations. It clearly shows a significant positive shift in energy sourcing: under "Scheme 1," the overall use of green, environmentally friendly energy increases notably compared to the "Baseline." This indicates that our smart routing is successfully directing more of the network's operations to areas powered by clean sources. Simultaneously, the consumption of fossil fuels across the entire network experiences a substantial decrease. This reduction confirms that the routing system is effectively minimizing reliance on less sustainable energy, even across diverse sites. Ultimately, this aggregated data demonstrates that "Scheme 1" effectively optimizes the network's collective energy footprint, leading to a more environmentally conscious operation.

7.3 Perspectives

While the current implementation of the AC^3 Network Operator is tightly integrated with Skupper as the underlying network technology, the architecture has been designed to support a plugin-based model. This opens the door for future expansion to other technologies like Submariner, Istio, or other service meshes and connectivity tools. Our goal is to enable the operator to intelligently select the most appropriate network backend based on the deployment environment, performance requirements, or security constraints. For example, low-latency, service-mesh-aware environments may benefit from Istio, while flat, L3-based connectivity could be handled by Submariner. By generalizing the logic and abstracting provider-specific details behind a common CRD, we aim to make multi-cluster networking more adaptive, technology-agnostic, and policy-driven.

8 Power management at the network edge using Reinforcement Learning

The computationally intensive and latency-critical nature of tasks at the network edge places significant strain on the resource capacity of MEC servers. Continuing from D4.1, we present a Reinforcement Learning (RL)-based solution for power management at the MEC servers situated at the network edge. Further, a relation between efficient resource management and power consumption is established. Thus, the contribution presented in this section is in line with the AC3 objectives as it addresses the resource management and energy efficiency problems [45].

8.1 Background

For maintaining quality of service in resource-constrained environments, Multi-access Edge Computing (MEC) poses as a promising solution. Some of the problems that can be addressed using MEC capabilities are resource orchestration, energy management, network adaptability and security. However, computational requirements from heterogeneous services associated with the end users can put immense pressure on the MEC servers if not handled efficiently. To this end, we identify an application-aware resource optimisation technique using RL that aids in creating an intelligent tool for the existing sustainable resource allocation problem. This strategy accommodates resource constraints from different network services without degrading overall network performance. Further, within our approach, we introduce a connection between power consumed at the MEC due to different services and the resources consumed by them. Thus, we contribute to the green-oriented AI-based solutions for the management of CECC infrastructure as targeted by the AC3 project.

8.2 Related work

In this section we mention few of the eminent state-of-the-art literature which is relevant to our objective. Energy minimization by classifying task requests at the MEC is studied by the authors of [46]. Industrial Internet-of-Things (IIoT) networks is one of the use-cases under edge network management and the authors of [47] have proposed a multi-constrained optimization model for energy efficiency in their work specifically for IIoT. In the research work attempted in [48], task execution delay and energy consumption is optimized using Lagrangian-constrained optimization. With limitations on the power capacity, authors of [49] and [50] propose efficient edge resource allocation schemes. There are many works that have utilized the potential of RL as well. The resources for dense multimedia broadband traffic is efficiently managed by using deep RL in [51]. The authors of [52] focus on offloaded computations using Deep Deterministic Policy Gradient (DDPG). Energy optimized minimization of task queues using actor-critic learning is considered in [53]. Finally, in [54] a Deep Q-Network (DQN) optimizing user offloading data ratio while encouraging green computing is presented.

8.3 MEC Resource and Power Management

We consider a network framework that at a system level connects the MEC to the end user services. As a result, the system model in Figure 76 is considered for realizing the experiments described in later sections. A single MEC server M hosts integral offloaded network service s that belongs to a set of broader network services S and supplies computational resources to applications such as Virtual Reality (VR), smart homes, AI/ML models at the end devices and remotely controlled robots. The RL agent is hosted at the MEC server *M.* The agent interacts with the services $s \epsilon S$ to learns the network demands patterns. A service $s \epsilon S$ places its resource demands in terms of *CPU* requirements to the MEC M. The *CPU* requirements for $s \epsilon S$ *is* denoted as c, while C is the maximum available computational capacity of the server. Computational resources c *a*t MEC server M is expected to vary dynamically due to the dynamic behaviour of the tasks offloaded by the end users. We focus only on resource management problem and not on resource admission. For convenience, it assumed that all the services in $s \epsilon S$ are admitted into the MEC M. Resources provisioned to each of the individual service $s \epsilon S$ is based on the decisions made by the RL agent A.

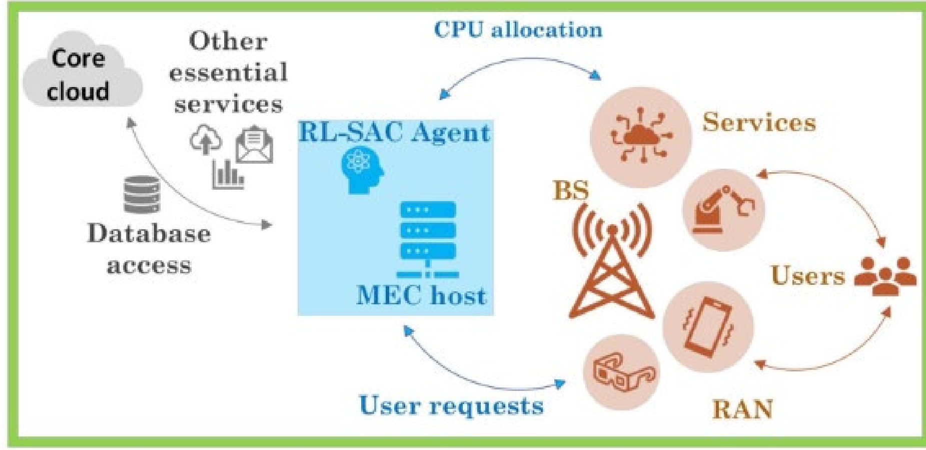


Figure 76. System model of the RL-based power-management system

The resource and power management at the MEC *M* is posed as an optimization function *O*(*t*), which is expressed as below with constraints:

$$O(t) = max \sum_{s=1}^{S} w_s(R_s - P_s)(t) \tag{8.1}$$

Here, w_s is the preference value which helps in prioritizing different services. The objective is to maximize the resource allocation while minimizing the power consumption. The maximum capacity is monitored using $\sum_{s=1}^{S} c_a \leq C$. The resources are restricted by using a satisfaction parameter α. The incoming resource demand L_s should be within a permissible range in comparison to the allocated resources R_s that is, $L_s \leq \alpha R_s$. Additional constraint for the latency is also added to meet the objectives. Finally, the power consumption is controlled by allowing resources to consume power proportional to the resources consumed by them that is, $\sum_{s=1}^{S} p_a \leq \frac{R_s}{C} P$. Task execution delay and power consumed by the services is used as the KPI(s) for this work. The delay is calculated using the equation, $d = g \frac{c_s}{c_a}$. Here *g* is the gain factor which controls the impact of resource demand for a service c_s and allocated resource c_a. Power consumption is modelled as $P = P_{idle} + (P_{max} - P_{idle}) \frac{c_s}{c_a}$ [55]. The state space, action space and reward function used for this work are defined below:

- State space: The state space at time *t* comprises of the maximum *CPU* capacity of the server *C*. It further contains *CPU* demand c_s, *CPU* allocation c_a, the delay incurred d_s and the power consumed p_s by each service $s \epsilon S$.
- Action space: The action space comprises of the random *CPU* allocations for each service $s \epsilon S$ that the RL agent makes given the state space S_t and the transition probability *T*(*t*).
- Reward: The reward function r_t guides the RL agent's policy learning, as presented in (8.2). The first term of (8.2) rewards the agent every time it takes the correct action. The function $f(c_d)$ is dependent on the difference between the allocated resources c_a and the resource demand c_s for the services $s \epsilon S$. Within the function *f* the difference $c_d = c_a - c_s$ is multiplied with appropriate preference value w_s as introduced in the problem formulation. The second term of (8.2) which is $\sum \Omega(c_s, d_s, p_s)$ is used for punishing the agent every time it violates any of the constraints mentioned in the problem formulation. $\Omega(c_s, d_s, p_s)$ is a function dependent on the resource demand c_s, delay d_s and power consumed p_s as per the set constraints.

$$r_t = \frac{f(c_d)}{S} - \sum \Omega(c_s, d_s, p_s) \tag{8.2}$$

8.4 Experimental setup & Results

The experiments were simulated using Python 3.11 and Tensorflow 2.15. Google cluster usage-traces are used for training the RL agent [56]. Google usage-traces provide information classified into different tasks with different priorities along with their CPU utilization values. The CPU utilization and task classification is of particular interest to this work to emulate different services being hosted by MEC *M*. Further, a Soft-Actor Critic (SAC) algorithm is used for the agent. SAC proves to be advantageous as it brings stability while training by exploiting two Q-networks $Q(s_{t,}a_t)$ approximators. Additionally, the well-known Knapsack approach and the proportional distributor (allocation of resources proportionately based on the service preference) approach are used as the baselines for comparison. For the simulations, 4 network services are hosted at the MEC *M*. Each of these services have diverse delay requirements. Some of the training parameters used are mentioned in the table below.

Table I Training parameters

Learning rate	**Discount factor**	**Trade-off coefficient**	**Episodes**	**Steps**
0.0001	0.9	0.9	20	1024

Average delay and average power consumption at the MEC is used as the main KPIs for evaluating the effectiveness of the proposed idea in this work. In Figure 77, the average delay experienced by each of the 4 services being hosted at MEC *M* is calculated using the considered KPI formula. The RL-SAC algorithm incurs an extremely small delay compared to the Knapsack and proportional distributor algorithms. The sophisticated resource allocation scheme while using RL-SAC algorithm is confirmed by lower delay values. The average power consumption by each service *sϵS* is being shown in Figure 78. Similarly, lower power consumption values are observed using our approach. Further, in Figure 79-Figure 82 the power consumption by each individual service *sϵS* is shown respectively. The *CPU* demands in the form of utilization ratio are divided into different bins for representation purposes. For each demand bin the proposed RL-SAC approach gives a lower power consumption value in comparison to the baselines. Idle server consumption is also showcased by adding a bin with value of zero. Both the overall power consumption and the consumption at a service level is therefore reduced by our method.

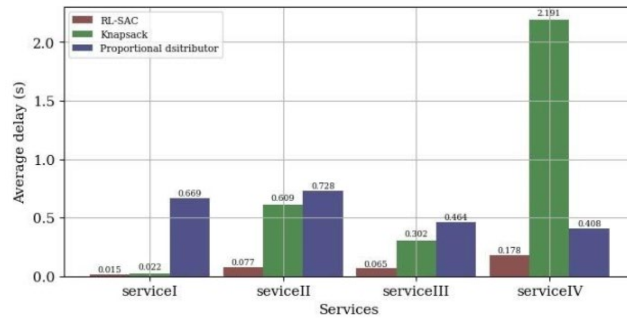


Figure 77 Average delay experienced by hosted services at the MEC

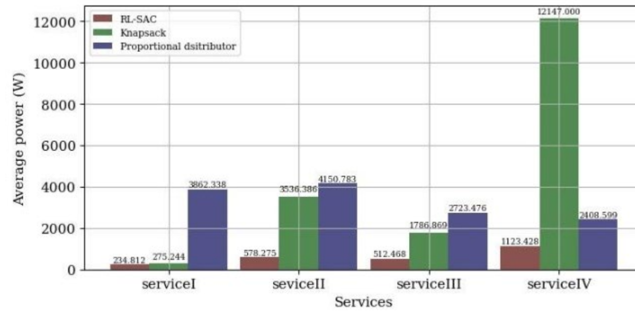


Figure 78. Average power consumption of hosted services at the MEC

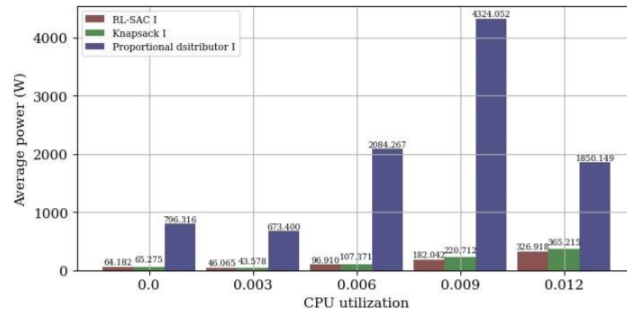


Figure 79. Average power consumption of service I at the MEC

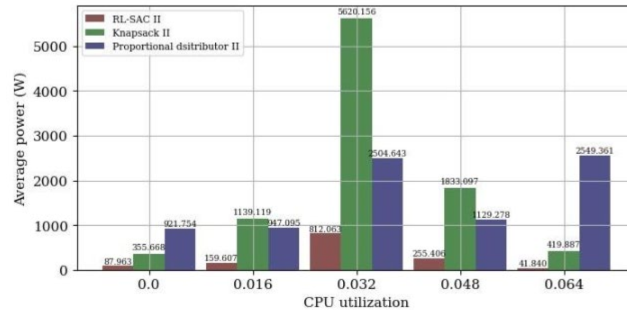


Figure 80. Average power consumption of service II at the MEC

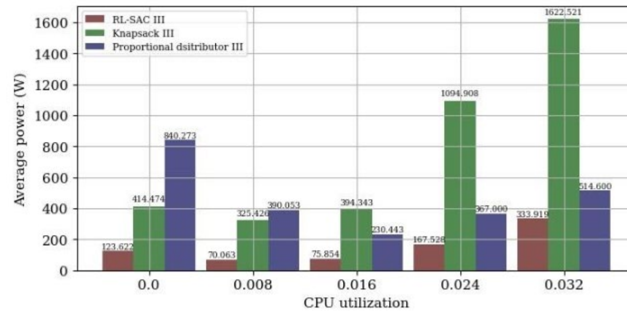


Figure 81. Average power consumption of service III at the MEC

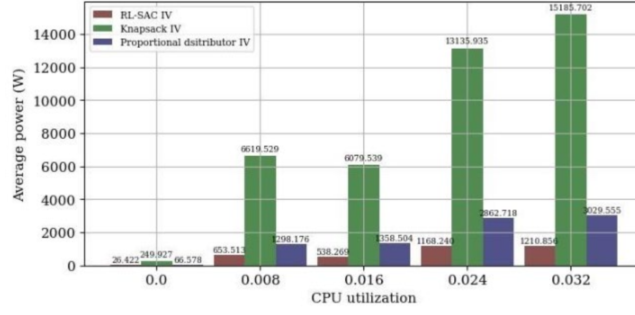


Figure 82. Average power consumption of service IV at the MEC

8.5 Multi-agent energy optimization using Reinforcement learning

The above presented results are further extended using a multi-agent setup which utilizes a decentralized decision mechanism [57]. This work expands the distributed service-aware idea from before to a cooperative environment alleviating the energy dissipation issues which is beneficial for the CECC infrastructure management.

8.5.1 Related work

Several notable works can be found within the literature attempting to solve problems of a similar nature. However, resource-dependent energy management is yet to be explored. A few of the accomplished works within the literature are introduced here on. The authors of [58] and [54] achieve lower energy consumption while optimizing resource allocation. Wireless Powered Edge Computing (WPEC) networks which improve the energy efficiency, is used in [52] utilizing DDPG. In [59], an edge pricing model for cloud resources is proposed for efficient resource allocation. The work in [60] addresses integrating MEC with Unmanned Aerial Vehicles (UAVs) for energy efficiency. Similarly, for an IIoT setup, in [61] energy trade-off for offloading tasks to the edge is studied.

8.5.2 Service-aware energy optimization

The system model from Figure 76 is extended to a multi-agent solution as shown in Figure 83. We expand the initial system model to include multiple MEC servers holding on to computational and memory resources. We use a complex version of the delay and power formula used above to fit the requirements of the multi-agent system to handle multiple resource types. The delays are measured using G/G/1 queuing model while the power consumption calculations were performed using (8.3) and (8.4). ρ here is the demand-allocation ratio, $\beta$and μ are used to control the linear or non-linear behaviour of power consumption. The energy is further calculated as $E_T = \int_{t_0}^{T} P_{CPU} + P_{mem}\, dt$. All the other constraints from the previous subsection are extended to include the memory resource aspects as well. Additionally, the services are assumed to be assigned to one server at a time in a round-robin system. Similarly, the state and action spaces are expanded to include energy consumption and memory resources. However, the reward term is modified to $R_{global} = \frac{R_{local}^{MEC_i}}{\sum R_{local}^{MEC_i}}$ to accommodate individual server performance and the overall multi-agent system performance while minimizing energy and delay values.

$$P_{CPU} = \begin{cases} P_{max}^{CPU} + \left(P_{max}^{CPU} - P_{idle}^{CPU}\right)\beta_{CPU}\rho^{\mu} & if\, c_a < c_d \\ P_{max}^{CPU} & if\ c_a \geq c_d \end{cases} \tag{8.3}$$

$$P_{Mem} = \begin{cases} P_{max}^{Mem} + \left(P_{max}^{Mem} - P_{idle}^{Mem}\right)\beta_{Mem}\rho & if\, m_a < m_d \\ P_{max}^{Mem} & if\ m_a \geq m_d \end{cases} \tag{8.4}$$

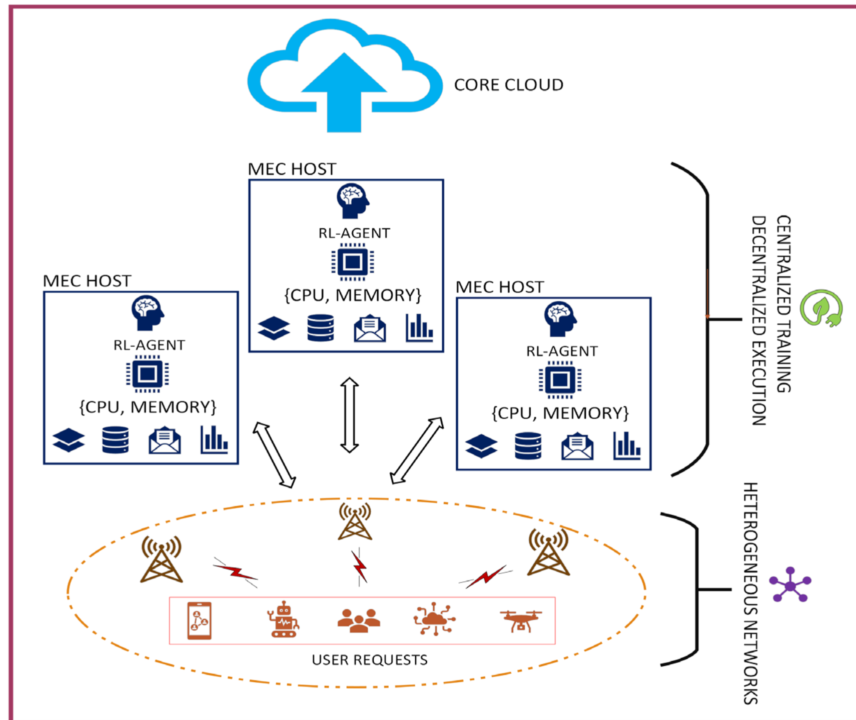
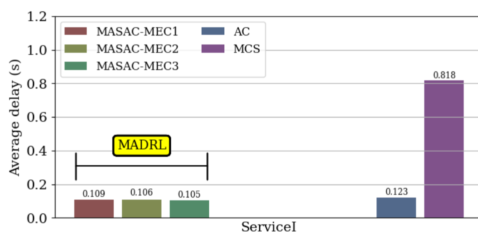


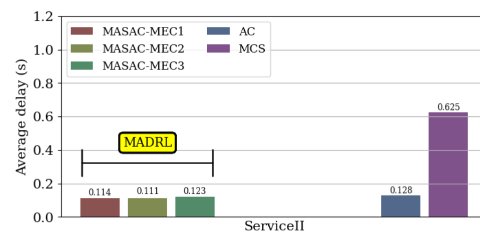
Figure 83. Multi-agent RL based energy optimization

8.5.3 Experimental setup & Results

A multi-agent cooperative SAC-based RL setup is used for the experiments instead of a single agent. Along with the CPU resources, memory resources are now further included. The training is performed on the information received from the Google-cluster traces. We again consider four services as provided by these traces. The baseline comparison is performed against single-agent RL algorithm using actor actor-critic method and Monte-Carlo Simulations (MCS). The multi-agent solution incurs the least delay for all the services in comparison to the single agent RL and MCS, as can be seen in Figure 84. We conclude that both diversity within services and within the network. The feedback from the cooperative training improves the performance of the multi-agent system where more servers are in use. Similar results are observed for energy consumption in Figure 85, where our approach consumes less energy in comparison to the baselines. Efficient resource allocation by the RL agent helps in alleviating energy consumption further. Since CPU and memory both were used for this work, the interoperability of both resources is also considered by the multi-agent setup. Further, the cooperative training mechanism enables scalability.



(a)



(b)

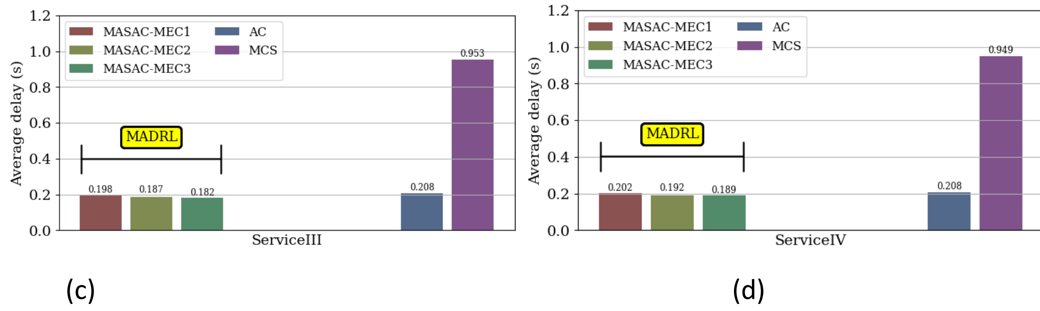


Figure 84. Delay performance results

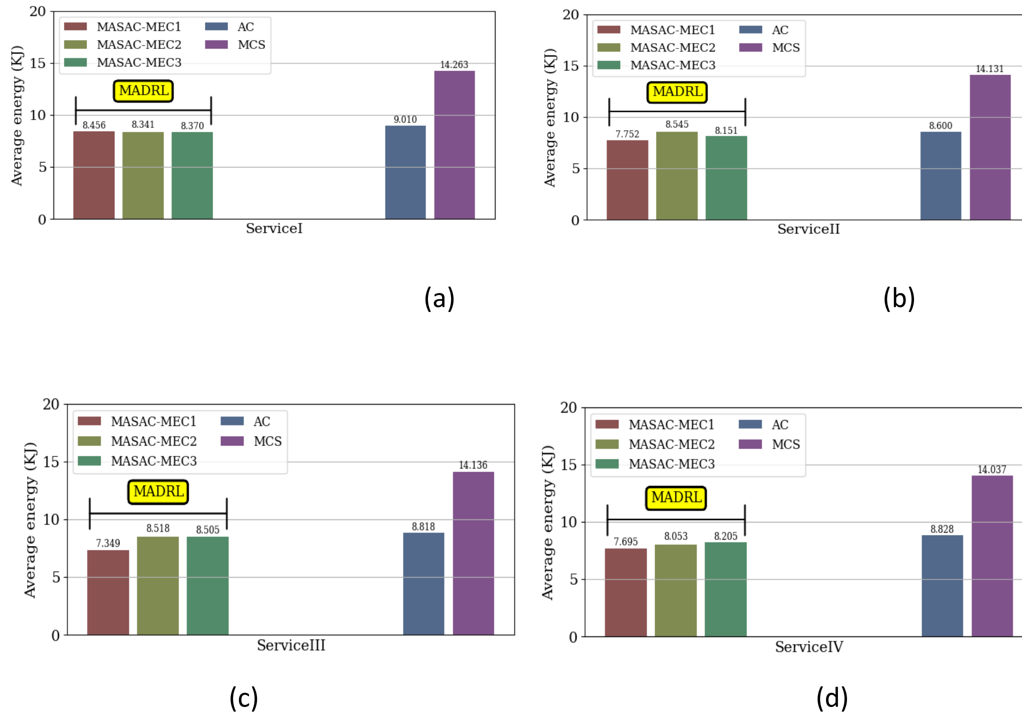


Figure 85. Energy performance results

8.6 Enhancing resource optimization algorithm using Generative-AI

We use resource allocation strategy from section 1 and enhance it using Generative-AI (Gen-AI). Reward shaping within the RL domain is extremely time-consuming and not easily generalisable. Hence, Large Language Models (LLMs) which are pre-trained on billions of parameters can be utilized instead of a customized reward function. The underlying assumption is that in the future, wireless networks will co-exist with LLMs in the CECC and they can be used to optimize model training. For this work, Math Σ tral model along with Chain-of-Thought (CoT) prompting is used instead of a reward function to see any improvement in the results [62].

8.6.1 Related work

Edge inferencing using LLM is targeted by the authors of [63]. In [64], the impact on energy while using diffusion models image generation at the edge is studied. An LLM inference model for optimizing task offloading to the edge is considered in [65]. The authors of [66] suggest a rewardless algorithm with active inferencing. Edge device

service optimization is implemented in [67] using prompt engineering. Attention-based diffusion SAC is used by [68] to address user requests generated by AI generated content.

8.6.2 3.2 System model

We use the same environment from Figure 76 but with more network resources. For this work, CPU, memory and bandwidth resources are considered. Network latency and data rate is used as metrics for measuring the performance of the RL agent. Shannon formula $Data\ rate = Blog_2(1 + SNR)$, where *B* is the bandwidth is for calculating both the metrics. The objective is to maximize the resource allocation and observe the difference while using the Math Σ tral. The state space now includes the resource allocations, demands, data rates and latency while the action space being the predicted allocations by the RL agent. The rewards change for this work by completely eliminating a custom function and relying on the Math Σ tral model.

8.6.3 3.3 Experimental setup & Results

Microsoft Azure's traces for packing are pre-processed to get resource requests in the form of CPU, memory and bandwidth. The experiments were run using NVIDIA GH200 resources. The results are compared against a single RL agent with a custom reward model, a Bayesian optimizer and random allocation. The results are reported in terms of latency and data rate goodness that is effectiveness of the measured metrics for an allocated unit of resource. if the latency or the data rate value is high when there is resource over-allocation that is the goodness value is low indicating high resource expenditure and vice versa. Figure 86 and Figure 87 summarize the results attained. We can observe that the RL algorithm combined with the LLM model in the place of a reward function performs better in both the cases than the baselines. We further state that the improvement can be attributed to LLMs context-aware scoring system. Further, the resource allocation distribution is also studied. Our approach distributes the resources in a balanced way compared to the others as shown in Figure 88.

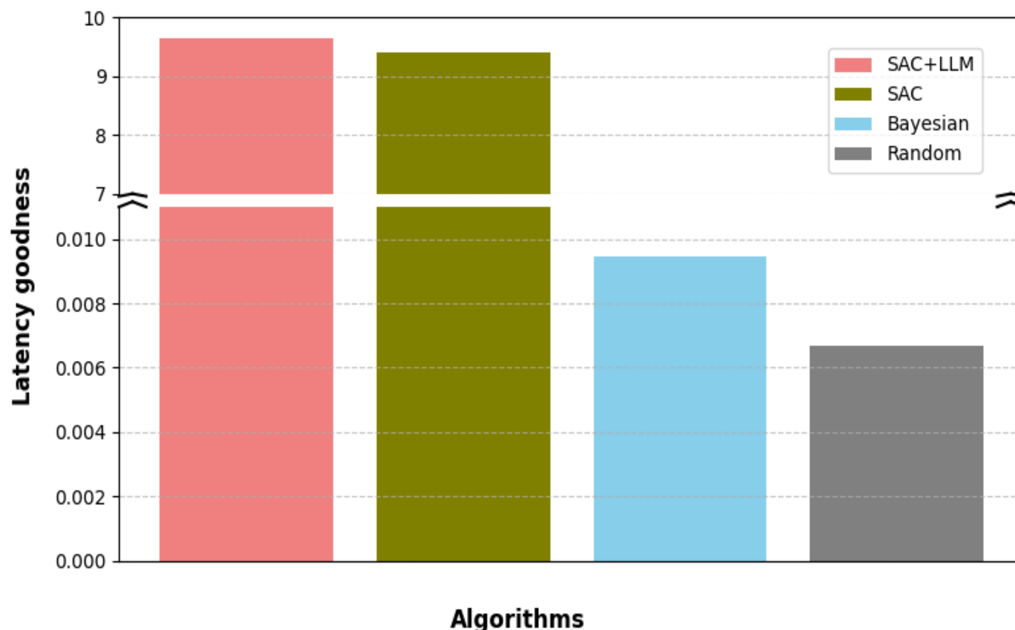


Figure 86. Latency goodness

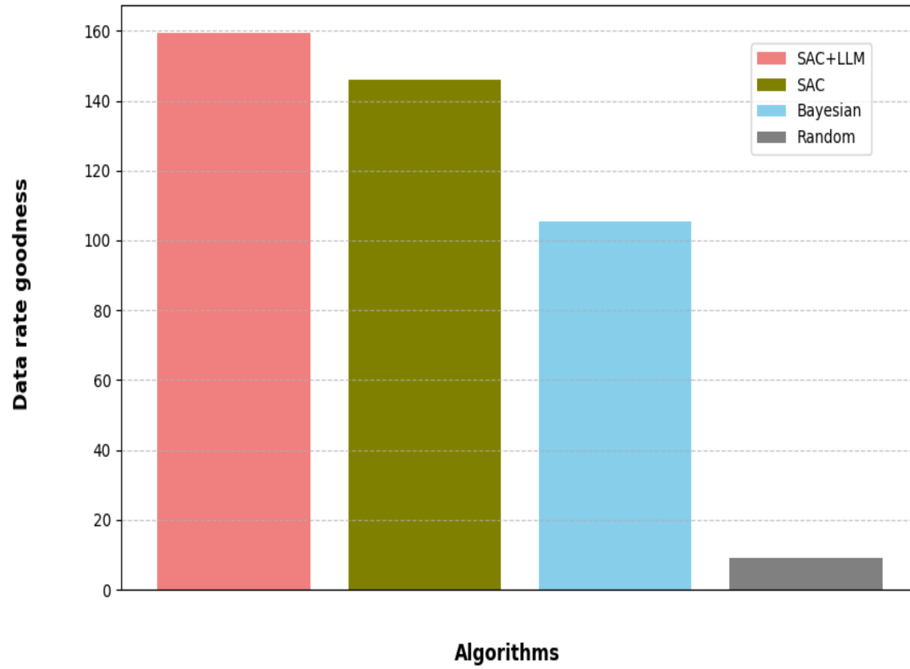


Figure 87. Data rate goodness

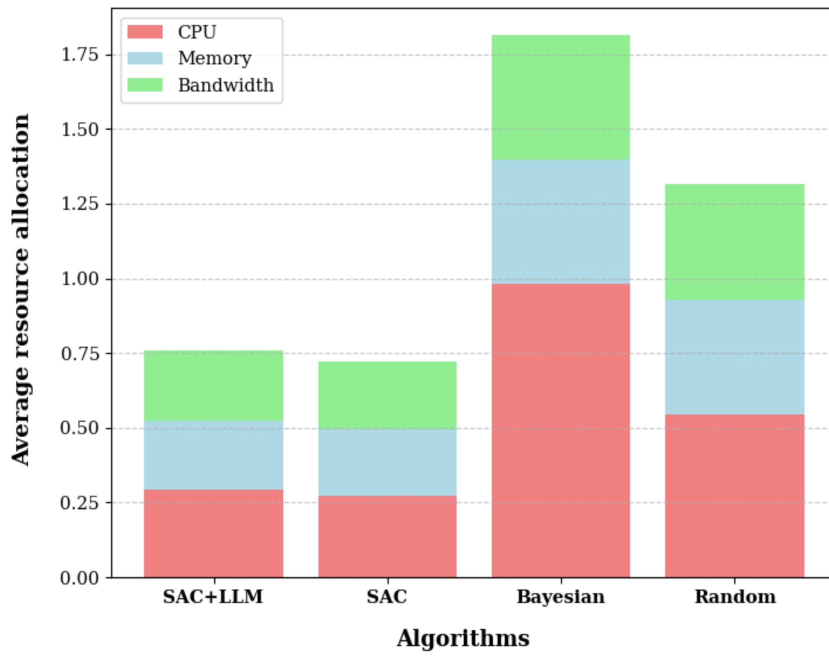


Figure 88. Network resource distribution

9 Conclusions

A summary of the technological advances implemented in AC^3, focusing on the overarching domain of resource management amidst the Cloud-Edge Computing Continuum (CECC) applied to AI-assisted monitoring, prediction, and enforcement. The results presented here constitute a powerful step toward intelligent infrastructure management that is SLA-aware and sustainable across heterogeneous and federated environments.

The developed monitoring framework supports scalable and low-overhead observability, reporting fine-grained, temporally aligned measures through a unified and extensible data model. It may be trusted to deliver real-time and historical data about compute, network, and energy dimensions for AI-driven decisions on application lifecycle management.

Explainable AI (XAI) models are used to project the consumption of resources to accurately predict workloads and allocate them in due time. Thus, interpretability validation approaches have been introduced to help build confidence in the AI models, thereby rendering the decision process transparent and enabling the acceptance of automated resource management decisions.

To ensure the maximum energy utilization while respecting application SLA and energy constraints, sustainability-aware scheduling algorithms interlace with reinforcement learning-based adaptive decision enforcement at the core. These constitute the proactive, self-optimizing lifecycle management engine reacting in real time to workload and infrastructure conditions that continuously change on the fly.

The other side of this work on network programmability is bringing advanced SD-WAN and eBPF-based capabilities towards intelligent microservice-aware traffic management across CECC, thus reliably promoting agile network decisions that are bound by application performance needs and operational constraints.

RL-based approaches were also presented where can optimize resource allocation and power management near the edge while tackling MEC environment challenges. The single-agent and multi-agent RL methods proposed considerably reduce delays and energy consumption when compared with conventional baselines, thus providing an appealing approach for the scalable and green-oriented CECC infrastructure management. These findings point to a promising result toward the realization of intelligent, adaptive, and energy-efficient edge networks in unison with the objectives of the AC3 project.

The mechanisms and prototypes that have been developed and validated within this deliverable represent a very good foundation to carry the AC^3 integration and validation phases forward. Future work (in WP5) will give further refinement and look to expand the real, deployed footprint of these mechanisms and more tightly coordinate them with CECCM.

10 References

[1] A. A. Ismail, M. K. Gunady, H. C. Bravo, and S. Feizi, 'Benchmarking Deep Learning Interpretability in Time Series Predictions', *CoRR*, vol. abs/2010.13924, 2020, Accessed: June 09, 2025. [Online]. Available: https://arxiv.org/abs/2010.13924

[2] H. Suresh, N. Hunt, A. E. W. Johnson, L. Celi, P. Szolovits, and M. Ghassemi, 'Clinical Intervention Prediction and Understanding using Deep Networks', *ArXiv*, May 2017, Accessed: June 09, 2025. [Online]. Available: https://www.semanticscholar.org/paper/Clinical-Intervention-Prediction-and-Understanding-Suresh-Hunt/2d86b0b1937faa78c0111321dc29629f80f30f38

[3] T. Chen, 'Investigating the mental health of university students during the COVID-19 pandemic in a UK university: a machine learning approach using feature permutation importance', *Brain Inform.*, vol. 10, no. 1, p. 27, Dec. 2023, doi: 10.1186/s40708-023-00205-8.

[4] M. D. Zeiler and R. Fergus, 'Visualizing and Understanding Convolutional Networks', Nov. 28, 2013, *arXiv*: arXiv:1311.2901. doi: 10.48550/arXiv.1311.2901.

[5] M. Pawelczyk, S. Bielawski, J. V. D. Heuvel, T. Richter, and G. Kasneci, 'CARLA: A Python Library to Benchmark Algorithmic Recourse and Counterfactual Explanation Algorithms', *ArXiv*, Aug. 2021, Accessed: June 09, 2025. [Online]. Available: https://www.semanticscholar.org/paper/CARLA%3A-A-Python-Library-to-Benchmark-Algorithmic-Pawelczyk-Bielawski/7b700dbbdc38b714916a93065c1d11ea16728e7c

[6] K. Simonyan, A. Vedaldi, and A. Zisserman, 'Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps', *CoRR*, Dec. 2013, Accessed: June 09, 2025. [Online]. Available: https://www.semanticscholar.org/paper/Deep-Inside-Convolutional-Networks%3A-Visualising-and-Simonyan-Vedaldi/dc6ac3437f0a6e64e4404b1b9d188394f8a3bf71

[7] M. Sundararajan, A. Taly, and Q. Yan, 'Axiomatic Attribution for Deep Networks', presented at the International Conference on Machine Learning, Mar. 2017. Accessed: June 09, 2025. [Online]. Available: https://www.semanticscholar.org/paper/Axiomatic-Attribution-for-Deep-Networks-Sundararajan-Taly/f302e136c41db5de1d624412f68c9174cf7ae8be

[8] G. Erion, J. D. Janizek, P. Sturmfels, S. M. Lundberg, and S.-I. Lee, 'Learning Explainable Models Using Attribution Priors', *ArXiv*, June 2019, Accessed: June 09, 2025. [Online]. Available: https://www.semanticscholar.org/paper/Learning-Explainable-Models-Using-Attribution-Erion-Janizek/ab3bafd5647b501c8f569498844368a07c659f3a

[9] O. Ozyegen, I. Ilic, and M. Cevik, 'Evaluation of interpretability methods for multivariate time series forecasting', *Appl. Intell.*, vol. 52, no. 5, pp. 4727–4743, Mar. 2022, doi: 10.1007/s10489-021-02662-2.

[10] X. Gao, D. Zhang, W. Bao, X. Zhu, and H. Yan, 'Energy-efficient Cooperative Storage Scheduling for Mobile Edge Cloud under Unstable Communication Conditions', in *2020 IEEE 10th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, Beijing, China: IEEE, July 2020, pp. 39–42. doi: 10.1109/ICEIEC49280.2020.9152366.

[11] L. Gu, W. Zhang, Z. Wang, D. Zeng, and H. Jin, 'Service Management and Energy Scheduling Toward Low-Carbon Edge Computing', *IEEE Trans. Sustain. Comput.*, vol. 8, no. 1, pp. 109–119, Jan. 2023, doi: 10.1109/TSUSC.2022.3210564.

[12] S. Chen, H. Tang, M. Zhao, Y. Chen, X. Yang, and K. Hu, 'Efficient Scheduling of Energy-Constrained Tasks in Internet of Things Edge Computing Networks':, *Int. J. Swarm Intell. Res.*, vol. 15, no. 1, pp. 1–17, Aug. 2024, doi: 10.4018/IJSIR.350221.

[13] M. P. J. Mahenge, C. Li, and C. A. Sanga, 'Energy-efficient task offloading strategy in mobile edge computing for resource-intensive mobile applications', *Digit. Commun. Netw.*, vol. 8, no. 6, pp. 1048–1058, Dec. 2022, doi: 10.1016/j.dcan.2022.04.001.

[14] J. Hu, Y. Hu, and Z. Tong, 'An energy efficient resource allocation method for intelligent water conservancy edge network based on DDQL', in *2022 7th Asia Conference on Power and Electrical Engineering (ACPEE)*, Hangzhou, China: IEEE, Apr. 2022, pp. 130–134. doi: 10.1109/ACPEE53904.2022.9783898.

[15] T. Zheng, J. Wan, J. Zhang, and C. Jiang, 'Deep Reinforcement Learning-Based Workload Scheduling for Edge Computing', *J. Cloud Comput.*, vol. 11, no. 1, p. 3, Dec. 2022, doi: 10.1186/s13677-021-00276-0.

[16] Y. Tang, 'Minimizing Energy for Caching Resource Allocation in Information-Centric Networking with Mobile Edge Computing', in *2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, Fukuoka, Japan: IEEE, Aug. 2019, pp. 301–304. doi: 10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00062.

[17] D. Ding, X. Fan, Y. Zhao, K. Kang, Q. Yin, and J. Zeng, 'Q-learning based dynamic task scheduling for energy-efficient cloud computing', *Future Gener. Comput. Syst.*, vol. 108, pp. 361–371, July 2020, doi: 10.1016/j.future.2020.02.018.

[18] J. Jiang *et al.*, 'Computing Resource Allocation in Mobile Edge Networks Based on Game Theory', in *2021 IEEE 4th International Conference on Electronics and Communication Engineering (ICECE)*, Xi'an, China: IEEE, Dec. 2021, pp. 179–183. doi: 10.1109/ICECE54449.2021.9674451.

[19] S. Mangalampalli, S. K. Swain, G. R. Karri, and S. Mishra, 'SLA Aware Task-Scheduling Algorithm in Cloud Computing Using Whale Optimization Algorithm', *Sci. Program.*, vol. 2023, pp. 1–11, Apr. 2023, doi: 10.1155/2023/8830895.

[20] Y. Li and S. Wang, 'An Energy-Aware Edge Server Placement Algorithm in Mobile Edge Computing', in *2018 IEEE International Conference on Edge Computing (EDGE)*, San Francisco, CA: IEEE, July 2018, pp. 66–73. doi: 10.1109/EDGE.2018.00016.

[21] Z. Xie and X. Song, 'A renewable-energy-driven energy-harvesting-based task scheduling and energy management framework', *ICT Express*, vol. 10, no. 1, pp. 39–45, Feb. 2024, doi: 10.1016/j.icte.2023.04.006.

[22] N. Mills, P. Rathnayaka, H. Moraliyage, D. De Silva, and A. Jennings, 'Cloud Edge Architecture Leveraging Artificial Intelligence and Analytics for Microgrid Energy Optimisation and Net Zero Carbon Emissions', in *2022 15th International Conference on Human System Interaction (HSI)*, Melbourne, Australia: IEEE, July 2022, pp. 1–7. doi: 10.1109/HSI55341.2022.9869465.

[23] S. Wang *et al.*, 'Energy-Efficient Resource Optimization in Collaborative Cloud-Edge Elastic Optical Networks', in *2022 20th International Conference on Optical Communications and Networks (ICOCN)*, Shenzhen, China: IEEE, Aug. 2022, pp. 1–3. doi: 10.1109/ICOCN55511.2022.9900986.

[24] 'Sinan: ML-based and QoS-aware resource management for cloud microservices | Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems'. Accessed: June 09, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3445814.3446693

[25] Y. Zhang, R. Isaacs, Y. Yue, J. Yang, L. Zhang, and Y. Vigfusson, 'LatenSeer: Causal Modeling of End-to-End Latency Distributions by Harnessing Distributed Tracing', in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, in SoCC '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 502–519. doi: 10.1145/3620678.3624787.

[26] 'PERT-GNN: Latency Prediction for Microservice-based Cloud-Native Applications via Graph Neural Networks | Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining'. Accessed: June 09, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3580305.3599465

[27] J. Park, B. Choi, C. Lee, and D. Han, 'GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices', *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol.*, pp. 154–167, Dec. 2021, doi: 10.1145/3485983.3494866.

[28] A. Ikram, S. Chakraborty, S. Mitra, S. Saini, S. Bagchi, and M. Kocaoglu, 'Root Cause Analysis of Failures in Microservices through Causal Discovery', *Adv. Neural Inf. Process. Syst.*, vol. 35, pp. 31158–31170, Dec. 2022.

[29] K. Budhathoki, L. Minorics, P. Bloebaum, and D. Janzing, 'Causal structure-based root cause analysis of outliers', in *Proceedings of the 39th International Conference on Machine Learning*, PMLR, June 2022, pp. 2357–2369. Accessed: June 09, 2025. [Online]. Available: https://proceedings.mlr.press/v162/budhathoki22a.html

[30] S. Jha, J. Rios, N. Abe, F. Bagehorn, and L. Shwartz, 'Fault Localization Using Interventional Causal Learning for Cloud-Native Applications', in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, June 2024, pp. 141–147. doi: 10.1109/DSN-S60304.2024.00040.

[31] B. Żurkowski and K. Zieliński, 'Root Cause Analysis for Cloud-Native Applications', *IEEE Trans. Cloud Comput.*, vol. 12, no. 1, pp. 232–250, Jan. 2024, doi: 10.1109/TCC.2024.3358823.

[32] T. Tournaire, Y. Jin, A. Aghasaryan, H. Castel-Taleb, and E. Hyon, 'Factored Reinforcement Learning for Auto-scaling in Tandem Queues', in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2022, pp. 1–7. doi: 10.1109/NOMS54207.2022.9789809.

[33] Q. Wang, L. Shwartz, G. Ya. Grabarnik, V. Arya, and K. Shanmugam, 'Detecting Causal Structure on Cloud Application Microservices Using Granger Causality Models', in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, Sept. 2021, pp. 558–565. doi: 10.1109/CLOUD53861.2021.00072.

[34] 'CAUSALITY, 2nd Edition, 2009'. Accessed: June 09, 2025. [Online]. Available: https://bayes.cs.ucla.edu/BOOK-2K/

[35] J. PEARL, 'Causal diagrams for empirical research', *Biometrika*, vol. 82, no. 4, pp. 669–688, Dec. 1995, doi: 10.1093/biomet/82.4.669.

[36] 'An Algorithm for Fast Recovery of Sparse Causal Graphs - Peter Spirtes, Clark Glymour, 1991'. Accessed: June 09, 2025. [Online]. Available: https://journals.sagepub.com/doi/abs/10.1177/089443939100900106

[37] J. Runge, 'Discovering contemporaneous and lagged causal relations in autocorrelated nonlinear time series datasets', Jan. 05, 2022, *arXiv*: arXiv:2003.03685. doi: 10.48550/arXiv.2003.03685.

[38] R. Tibshirani, 'Regression Shrinkage and Selection via the Lasso', *J. R. Stat. Soc. Ser. B Methodol.*, vol. 58, no. 1, pp. 267–288, 1996.

[39] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. Vapnik, 'Support Vector Regression Machines', in *Advances in Neural Information Processing Systems*, MIT Press, 1996. Accessed: June 09, 2025. [Online]. Available: https://papers.nips.cc/paper_files/paper/1996/hash/d38901788c533e8286cb6400b40b386d-Abstract.html

[40] T. Chen and C. Guestrin, 'XGBoost: A Scalable Tree Boosting System', in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 2016, pp. 785–794. doi: 10.1145/2939672.2939785.

[41] 'MEF Leads SD-WAN Service Standardization & Certification: Q&A with Nan Chen', *IEEE Commun. Stand. Mag.*, vol. 3, no. 3, pp. 6–8, Sept. 2019, doi: 10.1109/MCOMSTD.2019.8928157.

[42] S. Messaoudi, A.-E. Meliani, A. Mokhtari, and A. Ksentini, 'Security and Trust Management in Cloud Edge Continuum: AC^3 Project Approach', in *2024 IEEE 29th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, Athens, Greece: IEEE, Oct. 2024, pp. 1–6. doi: 10.1109/CAMAD62243.2024.10942825.

[43] S. Messaoudi, A. Ksentini, F. Messaoudi, and C. Bonnet, 'SDN-based L4S Congestion Control in Beyond 5G', in *2024 IEEE 25th International Conference on High Performance Switching and Routing (HPSR)*, Pisa, Italy: IEEE, July 2024, pp. 99–105. doi: 10.1109/HPSR62440.2024.10636004.

[44] A. Mokhtari and A. Ksentini, 'SD-WAN for Cloud Edge Computing Continuum interconnection', in *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, Cape Town, South Africa: IEEE, Dec. 2024, pp. 2533–2538. doi: 10.1109/GLOBECOM52923.2024.10901751.

[45] S. K. Chari, J. S. Vardakas, K. Ramantas, A. Ksentini, and C. Verikoukis, 'Reinforcement Learning Driven Sustainable Resource and Power Management for the MEC', in *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, Cape Town, South Africa: IEEE, Dec. 2024, pp. 565–570. doi: 10.1109/GLOBECOM52923.2024.10901623.

[46] S. Thananjeyan, C. A. Chan, E. Wong, and A. Nirmalathas, 'Energy-Efficient Mobile Edge Hosts for Mobile Edge Computing System', in *2018 IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*, Colombo, Sri Lanka: IEEE, Dec. 2018, pp. 1–6. doi: 10.1109/ICIAFS.2018.8913354.

[47] D. Jiang, Y. Wang, Z. Lv, W. Wang, and H. Wang, 'An Energy-Efficient Networking Approach in Cloud Services for IIoT Networks', *IEEE J. Sel. Areas Commun.*, vol. 38, no. 5, pp. 928–941, May 2020, doi: 10.1109/JSAC.2020.2980919.

[48] L. Li, Z. Kuang, and A. Liu, 'Energy Efficient and Low Delay Partial Offloading Scheduling and Power Allocation for MEC', in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, Shanghai, China: IEEE, May 2019, pp. 1–6. doi: 10.1109/ICC.2019.8761160.

[49] M. Qin, L. Chen, N. Zhao, Y. Chen, F. R. Yu, and G. Wei, 'Power-Constrained Edge Computing With Maximum Processing Capacity for IoT Networks', *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4330–4343, June 2019, doi: 10.1109/JIOT.2018.2875218.

[50] M. Song, Y. Lee, and K. Kim, 'Reward-Oriented Task Offloading Under Limited Edge Server Power for Multiaccess Edge Computing', *IEEE Internet Things J.*, vol. 8, no. 17, pp. 13425–13438, Sept. 2021, doi: 10.1109/JIOT.2021.3065429.

[51] Y. Huo *et al.*, 'DRL Driven Energy-efficient Resource Allocation for Multimedia Broadband Services in Mobile Edge Network', in *2020 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, Paris, France: IEEE, Oct. 2020, pp. 1–6. doi: 10.1109/BMSB49480.2020.9379443.

[52] X. Feng, J. Jia, T. Wang, J. Li, H. Xu, and Q. Li, 'Deep Reinforcement Learning based Energy Efficient Resource Allocation for Wireless Powered Edge Computing Network', in *2023 4th International Symposium on Computer Engineering and Intelligent Communications (ISCEIC)*, Nanjing, China: IEEE, Aug. 2023, pp. 144–148. doi: 10.1109/ISCEIC59030.2023.10271104.

[53] J. Zhang, J. Du, C. Jiang, Y. Shen, and J. Wang, 'Computation Offloading in Energy Harvesting Systems via Continuous Deep Reinforcement Learning', in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, Dublin, Ireland: IEEE, June 2020, pp. 1–6. doi: 10.1109/ICC40277.2020.9148938.

[54] Y. Yang, Y. Hu, and M. C. Gursoy, 'Deep Reinforcement Learning and Optimization Based Green Mobile Edge Computing', in *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, Las Vegas, NV, USA: IEEE, Jan. 2021, pp. 1–2. doi: 10.1109/CCNC49032.2021.9369566.

[55] A. Katal, S. Dahiya, and T. Choudhury, 'Energy efficiency in cloud computing data center: a survey on hardware technologies', *Clust. Comput.*, vol. 25, no. 1, pp. 675–705, Feb. 2022, doi: 10.1007/s10586-021-03431-z.

[56] *Google cluster-usage traces v3. [Online].* Accessed: Aug. 06, 2025. [Online]. Available: Available: https://github.com/google/cluster-data/blob/master/ClusterData2019.md

[57] Z. Ji, S. Wu, and C. Jiang, ‘Cooperative Multi-Agent Deep Reinforcement Learning for Computation Offloading in Digital Twin Satellite Edge Networks’, *IEEE J. Sel. Areas Commun.*, vol. 41, no. 11, pp. 3414–3429, Nov. 2023, doi: 10.1109/JSAC.2023.3313595.

[58] Y. Xiao, Y. Song, and J. Liu, ‘Towards Energy Efficient Resource Allocation: When Green Mobile Edge Computing Meets Multi-Agent Deep Reinforcement Learning’, in *ICC 2022 - IEEE International Conference on Communications*, Seoul, Korea, Republic of: IEEE, May 2022, pp. 4056–4061. doi: 10.1109/ICC45855.2022.9838659.

[59] J. Xu, Z. Xu, and B. Shi, ‘Deep Reinforcement Learning Based Resource Allocation Strategy in Cloud-Edge Computing System’, *Front. Bioeng. Biotechnol.*, vol. 10, p. 908056, Aug. 2022, doi: 10.3389/fbioe.2022.908056.

[60] F. Khoramnejad, A. Syed, W. Sean Kennedy, and M. Erol-Kantarci, ‘Energy and Delay Aware General Task Dependent Offloading in UAV-Aided Smart Farms’, *IEEE Trans. Netw. Serv. Manag.*, vol. 21, no. 5, pp. 5033–5048, Oct. 2024, doi: 10.1109/TNSM.2024.3391664.

[61] X. Jiao *et al.*, ‘Deep Reinforcement Learning for Time-Energy Tradeoff Online Offloading in MEC-Enabled Industrial Internet of Things’, *IEEE Trans. Netw. Sci. Eng.*, pp. 1–14, 2023, doi: 10.1109/TNSE.2023.3263169.

[62] B. Hazarika *et al.*, ‘Generative AI-Augmented Graph Reinforcement Learning for Adaptive UAV Swarm Optimization’, *IEEE Internet Things J.*, vol. 12, no. 8, pp. 9508–9524, Apr. 2025, doi: 10.1109/JIOT.2025.3529904.

[63] X. Zhang *et al.*, ‘Beyond the Cloud: Edge Inference for Generative Large Language Models in Wireless Networks’, *IEEE Trans. Wirel. Commun.*, vol. 24, no. 1, pp. 643–658, Jan. 2025, doi: 10.1109/TWC.2024.3497923.

[64] M. Tian, Z. Liu, C. Qiu, X. Wang, D. Niyato, and V. C. M. Leung, ‘Enabling Collaborative and Green Generative AI Inference in Edge Networks’, in *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, Cape Town, South Africa: IEEE, Dec. 2024, pp. 3709–3714. doi: 10.1109/GLOBECOM52923.2024.10901548.

[65] H. Zhou *et al.*, ‘Generative AI as a Service in 6G Edge-Cloud: Generation Task Offloading by In-context Learning’, 2024, *arXiv*. doi: 10.48550/ARXIV.2408.02549.

[66] Y. He, J. Fang, F. R. Yu, and V. C. Leung, ‘Large Language Models (LLMs) Inference Offloading and Resource Allocation in Cloud-Edge Computing: An Active Inference Approach’, *IEEE Trans. Mob. Comput.*, vol. 23, no. 12, pp. 11253–11264, Dec. 2024, doi: 10.1109/TMC.2024.3415661.

[67] Y. Liu *et al.*, ‘Optimizing Mobile-Edge AI-Generated Everything (AIGX) Services by Prompt Engineering: Fundamental, Framework, and Case Study’, 2023, *arXiv*. doi: 10.48550/ARXIV.2309.01065.

[68] Y. Liu, X. Lin, S. Li, G. Li, Q. Mao, and J. Li, ‘Towards Multi-Task Generative-AI Edge Services with an Attention-based Diffusion DRL Approach’, 2024, *arXiv*. doi: 10.48550/ARXIV.2405.08328.