TRANSPARENT APPLICATION DEPLOYMENT IN A SECURE, ACCELERATED AND COGNITIVE CLOUD CONTINUUM

Grant Agreement no. 101017168

Deliverable D5.4
Intelligent Service and Resource Orchestration Mechanisms

Programme:	H2020-ICT-2020-2
Project number:	101017168
Project acronym:	SERRANO
Start/End date:	01/01/2021 – 31/12/2023

Deliverable type:	Report
Related WP:	WP5
Responsible Editor:	NBFC
Due date:	31/07/2023
Actual submission date:	31/07/2023

Dissemination level:	Public
Revision:	FINAL

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017168

Revision History

Date	Editor	Status	Version	Changes
29.05.23	NBFC	Draft	0.1	Initial ToC
04.07.23	ICCS	Draft	0.2	Initial contribution in Sections 3, 5, 7, 9
14.07.23	NBFC	Draft	0.3	Initial contribution in Section 10
18.07.23	ICCS	Draft	0.4	Updates in Sections 5,7, 9
19.07.23	USTUTT/HLRS	Draft	0.5	Contribution in Sections 8 and 9
20.07.23	INNOV, UVT	Draft	0.6	Contribution in Section 4
21.07.23	UVT	Draft	0.6	Contribution in Section 6
21.07.23	NBFC	Draft	0.7	Contribution in Sections 1,2, and 11
24.07.23	ICCS	Draft	0.8	Finalize Sections 5 and 9
25.07.23	NBFC	Draft	0.9	Ready for internal review
28.07.23	ICCS, NBFC	Draft	0.10	Address internal review comments
31.07.23	ICCS	Final	1.0	Final version for submission

Author List

Organization	Author
ICCS	Aristotelis Kretsis, Panagiotis Kokkinos, Polyzois Soumplis, Ippokratis Sartzetakis, Fotis Kouzinos, Emmanouel Varvarigos
MLNX	Yoray Zack, Juan Jose Vegas Olmos, Sandra Starck
USTUTT/HLRS	Kamil Tokmakov, Javad Fadaie Ghotbi
INTRA	Paraskevas Bourgos, Makis Karadimas
INNOV	Efthymios Chondrogiannis, Efstathios Karanastasis, Filia Filippou, Andreas Litke, Kassi Papasotiriou, Stelios Pantelopoulos
UVT	Gabriel Iuhasz, Adrian Spătaru
NBFC	Anastassios Nanos, Christos Panagiotou, George Ntoutsos, Charalampos Mainas, Dimitris Karadimas, Alexandros Karantzoulis, Matias Vara Larsen, Shenghao Qiu

Internal Reviewers

Gabriel Iuhasz - UVT

Polyzois Soumplis - ICCS

Abstract: Deliverable D5.4 summarizes the outcomes from all five tasks of *Work Package 5 - Intelligence Service and Resource Orchestration.* It presents the research and development activities during the second iteration of the SERRANO incremental implementation plan (M16-M31). The deliverable builds upon the initial developments, which were reported in M15 at deliverables D5.1, D5.2, and D5.3, to provide the remaining functionality and implement the complete interfaces for inter-component communication. The deliverable presents the final design and developments for: (i) the ARDIA (A Resource reference model for Data-Intensive Applications) modelling framework, (ii) AI-Enhanced Service Orchestrator, (iii) multi-objective resource allocation and service orchestration algorithms, (iv) AI/ML-driven service assurance and re-optimization mechanisms, (v) energy and resource-aware flow mappings, (vi) novel network and cloud telemetry framework, (vii) hierarchical resource orchestration, and (viii) lightweight virtualization mechanisms. The provided developments are integral parts of the cognitive orchestration and transparent deployment mechanisms of the SERRANO complete platform prototype that will be used for the final performance evaluations.

Keywords: ARDIA framework models, AI-enhanced Service, Service Assurance, Multi-objective Optimization, Resource Optimization Toolkit, HPC Services, Telemetry, Resource Orchestration, Lightweight Virtualization, Containers, Unikernels, vAccel.

Disclaimer: *The information, documentation and figures available in this deliverable are written by the SERRANO Consortium partners under EC co-financing (project H2020-ICT-101017168) and do not necessarily reflect the view of the European Commission. The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The reader uses the information at his/her sole risk and liability.*

Copyright © 2023 the SERRANO Consortium. All rights reserved. This document may not be copied, reproduced or modified in whole or in part for any purpose without written permission from the SERRANO Consortium. In addition to such written permission to copy, reproduce or modify this document in whole or part, an acknowledgement of the authors of the document and all applicable portions of the copyright notice must be clearly referenced.

Table of Contents

1 Executive Summary ... 15
2 Introduction ... 16
2.1 Purpose of this document ... 16
2.2 Document structure ... 17
2.3 Audience ... 17
3 SERRANO Intelligent Service and Resource Orchestration Mechanisms ... 18
4 Intelligent Service Orchestration ... 20
4.1 AI-enhanced Service Orchestrator ... 20
4.2 Abstraction Models and Mapping Rules ... 21
4.3 Telemetry Data Analysis and ML Model Development ... 23
4.4 Translation Mechanism ... 25
4.5 Integration with a Graphical Interface ... 26
4.5.1 SERRANO-TOSCA ... 27
4.6 Example of Usage ... 29
4.6.1 Application requirements, intent specification and deployment description ... 29
4.6.2 Mapping Rules and Translation Mechanism ... 32
5 Algorithmic Framework ... 34
5.1 Cloud-native Applications' Workload Placement in the SERRANO Edge-Cloud Continuum ... 34
5.1.1 Related work ... 35
5.1.2 Problem formulation ... 36
5.1.3 Resource allocation mechanisms ... 38
5.1.4 Performance evaluation ... 41
5.1.5 Conclusions ... 43
5.2 Security-aware Resource Allocation in the SERRANO Edge-Cloud Continuum ... 44
5.2.1 Related work ... 44
5.2.2 Infrastructure description ... 45
5.2.3 Problem formulation ... 47
5.2.4 Performance evaluation ... 53
5.2.5 Conclusions ... 56
5.3 Intent-based Allocation of Cloud Computing Resources Using Q-Learning ... 57
5.3.1 Related work ... 58
5.3.2 System Model and Infrastructure-Agnostic Operations ... 59
5.3.3 Q-learning based Intent Translation ... 60
5.3.4 Evaluation ... 63
5.4 Resource Optimization Toolkit ... 65
5.4.1 Final implementation and interfaces ... 65
5.4.2 Algorithms integration and Python API ... 68
5.4.3 Deployment and configuration ... 71
6 Service Assurance and Remediation ... 72

6.1 Architecture 72
6.1.1 Configuration and REST API 77
6.2 Methods for Detection and Analysis 78
6.2.1 Supervised ML methods 79
6.2.2 Unsupervised ML methods 81
6.3 Discussion 84
7 Network and Cloud Telemetry Framework 85
7.1 SERRANO Telemetry Framework 85
7.1.1 Central Telemetry Handler and Enhanced Telemetry Agent 86
7.1.2 Monitoring Probes 89
7.1.3 Operational Database 93
7.1.4 Deployment of telemetry services and data visualization 94
7.2 Inventory and telemetry parameters 95
7.3 Telemetry interfaces 96
7.4 Persistent Monitoring Data Storage 98
7.5 Identifying Network Congestion Using Knowledge Graphs and Link Prediction 102
7.5.1 Previous Work 103
7.5.2 Proposed Methodology 104
7.5.3 Experiments 109
7.5.4 Discussion 111
8 Energy and Resource Aware Flow Mapping 112
8.1 Excess Cluster, Hardware, and Tools 112
8.2 Power Measurement Utilities 113
8.3 Power Measurement Conversion and Visualization 114
8.4 CPU Frequency Utility 115
8.5 Kernels Benchmarking 115
9 Resource Orchestration Mechanisms 117
9.1 SERRANO Resource Orchestrator 118
9.2 Orchestration Drivers 124
9.3 SERRANO HPC Gateway 126
9.4 Integration with SERRANO Services 128
9.4.1 Secure storage policies cognitive creation 128
9.4.2 Cloud-native applications deployment 131
9.4.3 SERRANO HW/SW accelerated kernels execution 135
10 Lightweight Virtualization Mechanisms 138
10.1 Efficient Sandboxing of Containers on Edge Nodes 140
10.2 Sandboxed Containers 142
10.3 Unikernels as Containers 145
10.3.1 bima: unikernel container images 145
10.3.2 urunc: a unikernel container runtime 146
10.4 microVM Optimizations 147
10.5 Hardware Acceleration 151

11 Conclusions 152
12 References 153

List of Figures

Figure 1: SERRANO high-level architecture 18

Figure 2: The SERRANO platform, utilizing edge, cloud and HPC resources and empowering the Everything as a Service (EaaS) notion towards the cloud continuum 19

Figure 3: AI-enhanced Service Orchestrator Interface 20

Figure 4: Security Tiers for the SERRANO platform 22

Figure 5: Execution speedup in (a) Kalman Filter and (b) Wavelet Transformation using different types of resources 23

Figure 6: Energy gain in (a) Kalman Filter and (b) Wavelet Transformation using different types of resources 24

Figure 7: ML model for predicting the Total Execution Time of a particular microservice for different workloads in (a) an Edge Device and (b) in HPC 24

Figure 8: ML model for predicting the Total Energy Consumption of a particular microservice for different workloads in (a) an Edge Device and (b) in HPC 25

Figure 9: Class diagram for SERRANO-TOSCA entities 27

Figure 10: Application definition using SERRANO-TOSCA definitions in Alien4Cloud Topology Editor 30

Figure 11: Intent Dialogue for one component in Alien4Cloud 30

Figure 12: Intent dialogue for the application performance dimension 31

Figure 13: Deployment runtime interface 32

Figure 14: Overview of the given YAML and JSON files – AISO input 33

Figure 15: AISO Deployment Scenario(s) – JSON File Provided to Resource Orchestrator 33

Figure 16: Flowchart of the GRAA heuristic. 39

Figure 17: Multi-agent Rollout options for serving the *i-th* microservice of application *a* 40

Figure 18: The pareto efficiency chart 42

Figure 19: The number of microservices allocated at the various layers 42

Figure 20: Operational and networking cost for the different objective co-efficients 43

Figure 21: Heterogenous resources across the edge-cloud continuum. 45

Figure 22: Different levels of workload isolation 46

Figure 23: Flowchart of the greedy best fit heuristic 51

Figure 24: Optimality gap for the different optimization criteria 54

Figure 25: Allocation of microservices at the different layers of the edge-cloud continuum . 54
Figure 26: Operational cost overhead for the different optimization criteria 55
Figure 27: Experienced latency for the different optimization criteria 55
Figure 28: Intent-driven resource allocation 57
Figure 29: The average reward over time for different cost resource levels 63
Figure 30: The average reward over time for different ε values 64
Figure 31: The average reward over time for different number of intent parameters........... 64
Figure 32: The Q-Table's heatmap for ε=0.5 and 50000 timesteps....................................... 65
Figure 33: Resource Optimization Toolkit (ROT) architecture and main components............ 66
Figure 34: Resource Optimization Toolkit REST API 67
Figure 35: ROT asynchronous communication over SERRANO Message Broker – Final implementation 68
Figure 36: ROT - Workflow for executing an orchestration algorithm in SERRANO 69
Figure 37: Integrated orchestration algorithms.. 70
Figure 38: Code snippet for interacting with the ROT through the provided Python API 71
Figure 39: Event Detection Engine – General architecture... 73
Figure 40: Pearson Correlation Raw Data 74
Figure 41: Feature reduction (t-SNE) 75
Figure 42: SAR Response Example 77
Figure 43: SAR REST Configuration.. 77
Figure 44: SAR REST Control.. 78
Figure 45: Class distribution 79
Figure 46: Learning curve XGBoost overlapping anomalies.. 80
Figure 47: ROC curve for XGBoost... 81
Figure 48: Decision Boundary Comparison 83
Figure 49: Shapley value-based feature importance 83
Figure 50: Shapley value-based feature importance 84
Figure 51: SERRANO hierarchical telemetry architecture.. 85
Figure 52: Central Telemetry Handler and Enhanced Telemetry Agent architecture 86

Figure 53: SERRANO telemetry framework – Inventory workflow 87
Figure 54: SERRANO telemetry framework – Monitoring workflow 88
Figure 55: General architecture of SERRANO monitoring probes 89
Figure 56: Autonomous monitoring of deployed cloud-native and short-lived applications.. 91
Figure 57: Monitoring data collected by SERRANO HPC monitoring probe 92
Figure 58: SERRANO telemetry framework deployment in project integration testbed 94
Figure 59: Memory usage for a selected worker node in the NBFC K8s cluster...................... 95
Figure 60: Collected inventory and monitoring parameters in the SERRANO platform.......... 96
Figure 61: Telemetry framework REST interfaces – Control and management methods....... 97
Figure 62: Telemetry framework REST interfaces – High-level CTH methods........................ 98
Figure 63: Persistent Monitoring Data Storage (PMDS) architecture.................................... 99
Figure 64: Persistent Monitoring Data Storage (PMDS) RESTful interface 99
Figure 65: PMDS Python API – Historical telemetry data for a specific worker node within a K8s cluster 101
Figure 66: PMDS deployed in main SERRANO Kubernetes cluster 102
Figure 67: Overview of proposed KG-based modelling and event detection methodology . 104
Figure 68: Knowledge graph meta-graph... 106
Figure 69: The three-step process of the GraphSAGE inductive representation method..... 107
Figure 70: The link prediction process. 108
Figure 71: Link prediction confusion matrix.. 110
Figure 72: Hardware components of the EXCESS cluster 113
Figure 73: Hardware components of the EXCESS cluster 115
Figure 74: Energy consumption of Kalman filter with different frequencies and different numbers of cores... 116
Figure 75: SERRANO distributed and cognitive resource orchestration mechanisms, unifying different edge, cloud, and HPC platforms.. 117
Figure 76: SERRANO Resource Orchestrator architecture and services 118
Figure 77: Resource Orchestrator RESTful interface.. 119
Figure 78: Resource Orchestrator RESTful interface – Methods related to inter-component communication.. 120

Figure 79: SERRANO Orchestration API objects 121
Figure 80: Relationship among main SERRANO Orchestration API objects 122
Figure 81: SERRANO Orchestration API objects and federated application deployment 123
Figure 82: SERRANO Orchestration Drivers 124
Figure 83: Orchestration Driver workflow 125
Figure 84: Interaction between HPC Gateway and HPC infrastructure 126
Figure 85: REST API endpoints exposed by HPC Gateway 127
Figure 86: Secure storage policy cognitive creation – Orchestration workflow 130
Figure 87: Code snippet for creating and using a SERRANO secure storage policy 130
Figure 88: Application deployment – High-level cognitive orchestration workflow 132
Figure 89: Kubernetes application deployment description enhanced by the SERRANO Resource Orchestrator 133
Figure 90: Cloud-native application deployment – Transparent deployment workflow 134
Figure 91: Terminating cloud-native application deployment 135
Figure 92: Kernel execution and data handling from the end user's perspective, common approach for all supported modes and platforms 136
Figure 93: A Virtual Machine running on a generic user-space VMM on top of KVM 139
Figure 94: High-level overview of generic container spawning in a k8s environment 140
Figure 95: Container sandboxing 141
Figure 96: Packing a unikernel as an OCI-compatible container image 145
Figure 97: Running an unpacked container image as a unikernel 147
Figure 98: A unikernel running as a VM on HEDGE 149

List of Tables

Table 1: The Parameters of a Mapping Rule 21
Table 2: TOSCA parameters for updating Kubernetes ConfigMap 29
Table 3: MILP variables 37
Table 4: Characteristics of the computing nodes of the basic and extended topologies 41
Table 5: The total cost and the execution time for w=0.01 for the different mechanisms 42

Table 6: Multipliers of the computing and storage requirements for the different security and trustworthiness tiers 47

Table 7: MILP variables 49

Table 8: Characteristics of the computing nodes of the different topologies 53

Table 9: Cloud-native applications' workload characteristics 53

Table 10: ROT plug-in mechanism – AlgorithmInterface abstract class 69

Table 11: ROT Python API – Provided methods and events 70

Table 12: Unsupervised method scores 82

Table 13: Central Telemetry Handler and Enhanced Telemetry Agent configuration options 87

Table 14: PMDS Python API – Available input parameters 100

Table 15: Link prediction evaluation metrics 110

Table 16: Comparison between compute nodes of Excess and Hawk 113

Table 17: Power consumption of parallel implementation of Kalman filter in Turbo Mode 116

Table 18: Datastore topics (keys) for the main SERRANO Orchestration API objects 123

Table 19: Process execution environment 143

Abbreviations

A4C	Alien4Cloud
ABI	Application Binary Interface
AE	Autoencoders
AES	Advanced Encryption Standard
AI	Artificial intelligence
AIoT	Artificial Intelligence of Things
AISO	AI-enhanced Service Orchestrator
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARDIA	A Resource reference model for Data-Intensive Applications
AWT	Average Waiting Time
ARFF	Attribute Relation File Format
CBLOF	Clustering-Based Local Outlier Factor
CI/CD	Continuous Integration / Continuous Development
CO	Central Office
CPU	Central Processing Unit
CRI	Container Runtime Interface
CTH	Central Telemetry Handler
CU	Cost Unit
D	Deliverable
DBMS	Database Management System
DevOps	Development and Operations
DM	Device Mapper
DNN	Deep Neural Network
DPUs	Data Processing Units
DU	Data Unit
ECC	Edge-Cloud Computing
EDE	Event Detection Engine
ELF	Executable and Linkable Format
ETA	Enhanced Telemetry Agent
FaaS	Function as a Service
FPGA	Field-programmable Gate Array
GCM	Galois/Counter Mode
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GRAA	Greedy Resource Allocation Algorithm
GUI	Grapical User Interface
HPC	High Performance Computing
HPO	Hyper-Prameter Optimization
HW	Hardware
IaaS	Infrastructure-as-a-Service
ILP	Integer Linear Programming
IoT	Internet of Things
JSON	JavaScript Object Notation

K8s	Kubernetes
KG	Knowledge Graph
KVM	Kernel-based Virtual Machine
KVVM	in-Kernel Virtual Machine Monitor
L.U.	Latency Units
LOF	Local Outlier Factor
M	Month
MAS	Multi-Agent System
MEC	Mobile Edge Computing
MILP	Mixed-Integer Linear Programming
ML	Machine learning
MP	Mapping Rule
MPI	Message Passing Interface
MQTT	MQ Telemetry Transport
NBI	North Bound Interface
NN	Neural Network
OCI	Open Container Initiative
OS	Operating System
PA	Personal Agent
PMDS	Persistent Monitoring Data Storage
PU	Period Unit
PV	Persistent Volume
PVC	Persistent Volume Claim
PyG	PyTorch Geometric
QoS	Quality of Service
REST	Representational State Transfer
RL	Reinforcement Learning
ROT	Resource Optimization Toolkit
SAR	Service Assurance and Remediation
SARSA	State-Action-Reward-State-Action
SDK	Service Development Kit
SLA	Service Level Agreement
SSH	Secure Shell Protocol
SW	Software
TD-L	Temporal Difference Learning
TOSCA	Topology and Orchestration Specification for Cloud Applications
t-SNE	t-distributed Stochastic Neighbor Embedding
TU	Task Unit
UC	Use Case
VAE	Variational AutoEncoders
VM	Virtual Machine
VMM	Virtual Machine Manager
WP	Work Package

1 Executive Summary

SERRANO envisages the development and deployment of disaggregated federated cloud and edge infrastructures that incorporate hardware-accelerated edge and cloud nodes as integral parts of the overall computation and storage chain. In addition, the SERRANO ecosystem expansion includes HPC infrastructures that can be utilized for exceptionally computationally intensive simulations and data analysis, bridging the gap between these currently largely separated computing paradigms.

Deliverable 5.4 presents a comprehensive report on the progress made in WP5 during the second iteration (M16-M31) of the SERRANO implementation plan. The main focus of this deliverable is to outline the work accomplished in all five tasks within WP5, which are dedicated to the implementation of the SERRANO intelligent service and resource orchestration mechanisms, along with lightweight virtualization mechanisms. These tasks build upon the initial developments described in D5.1 (M15), D5.2 (M15), and D5.3 (M15) to provide the final version of the envisioned mechanisms.

The deliverable offers an overview of the SERRANO platform and comprehensively details all the critical technical developments for finalizing the end-to-end SERRANO orchestration and deployment mechanisms. This is a major milestone that mars a significant achievement for the SERRANO project, as it includes the following key functionalities: (a) workload deployment modeling using the ARDIA framework to efficiently utilize available resources, (b) cognitive workload hierarchical orchestration, incorporating resource- and service-oriented optimization algorithms, (c) AI-enabled service assurance and re-optimization mechanisms, considering energy and resource-aware dimensions, (d) enhanced telemetry mechanisms for improved monitoring and data collection, and (e) support for workload deployment in diverse execution modes, such as lightweight virtualization, containerization, and unikernels.

The information provided in this deliverable significantly contributes to the development of the SERRANO full platform prototype (M31). Furthermore, the developed mechanisms play a significant role in supporting the final evaluation of SERRANO use cases, which will be documented in deliverable D6.8 “Final version of business, end user and technical evaluation” (M36). In addition, this progress us closer to the final release of the SERRANO platform, which will be integrated and comprehensively documented in deliverable D6.7 “Final version of SERRANO integrated platform” (M36).

2 Introduction

2.1 Purpose of this document

Deliverable D5.4 presents the outcomes of all tasks carried out in WP5 throughout the second phase (M16-M31) of the work package implementation. The initial progress during the first iteration of the implementation plan (M07-M15) was reported on D5.1 (M15), D5.2 (M15), and D5.3 (M15).

T5.1 is dedicated to the development of a series of abstraction models for representing and describing resources, services, and applications along with telemetry data. These models serve as the building blocks of the ARDIA modelling framework. Additionally, T5.1 includes the development of the SERRANO AI-enhanced Service Orchestrator, which effectively translates high-level and infrastructure-agnostic deployment requirements into resource-specific deployment scenarios. All these developments are described in Section 4.

T5.2 focuses on the development of multi-objective optimization algorithms for application deployment across edge, cloud, and HPC resources, and the data management within the distributed secure storage infrastructure of the SERRANO platform. As part of T5.2, the Resource Optimization Toolkit has been created, incorporating the developed algorithms. These developments are presented in Section 5. Furthermore, this task encompasses the development of data-driven service assurance mechanisms and the development of the Event Detection Engine, a critical component of the Service Assurance and Remediation service within the SERRANO platform. These aspects of T5.2 are presented in Section 6.

Moving to T5.3, its main objective is to implement an autonomous and data-driven telemetry framework within the SERRANO platform. This framework autonomously collects telemetry data from multiple and heterogeneous infrastructure resources and as well as metrics from the deployed applications. Section 7 covers the design and implementation of the final release of the SERRANO telemetry framework.

T5.4 is responsible for the development of a framework to assist developers in incorporating performance and power model functionality into the design and programming of their digital services, particularly within the SERRANO platform. Furthermore, T5.4 establishes the necessary HPC infrastructure and conducts measurements of the energy efficiency of the developed HPC services. Section 8 includes these developments.

T5.5 focuses on the development of SERRANO hierarchical resource orchestration mechanisms, leveraging well-established orchestration solutions at edge, cloud, and HPC platforms. Section 9 provides detailed insights into these developments, including technical details regarding the integration of the SERRANO orchestration and deployment mechanisms with other key SERRANO services. Additionally, T5.5 involves the development of essential software components that enable the seamless execution of workloads in various lightweight virtualization solutions, hypervisors, and unikernel frameworks. These novel developments are described in Section 10.

2.2 Document structure

The present deliverable is split into eight major chapters:

- SERRANO Intelligent Service and Resource Orchestration Mechanisms
- Intelligent Service Orchestration
- Algorithmic Framework
- Service Assurance and Remediation
- Network and Cloud Telemetry Framework
- Energy and Resource Aware Flow Mapping
- Resource Orchestration Mechanisms
- Lightweight Virtualization Mechanisms

2.3 Audience

The deliverable is public and available to anyone interested in the final release of the SERRANO intelligent service and resource orchestration mechanisms. Moreover, this document can also be useful to the general public for obtaining a better understanding of the framework and scope of the SERRANO project.

3 SERRANO Intelligent Service and Resource Orchestration Mechanisms

The SERRANO architecture was initially introduced in deliverable D2.3 "SERRANO architecture" (M09), and subsequently refined in its final version in D2.5 "Final version of SERRANO architecture" (M18), incorporating valuable insights from the development activities conducted during the first iteration of implementation (M1-M18). D2.5 (M18) presents a comprehensive architecture overview, encompassing the SERRANO components, their interfaces, and supported workflows. In this section, we offer a concise description of the architecture (Figure 1) to facilitate the presentation of the final developments in WP5 regarding the intelligent service and resource orchestration mechanisms in the SERRANO platform.

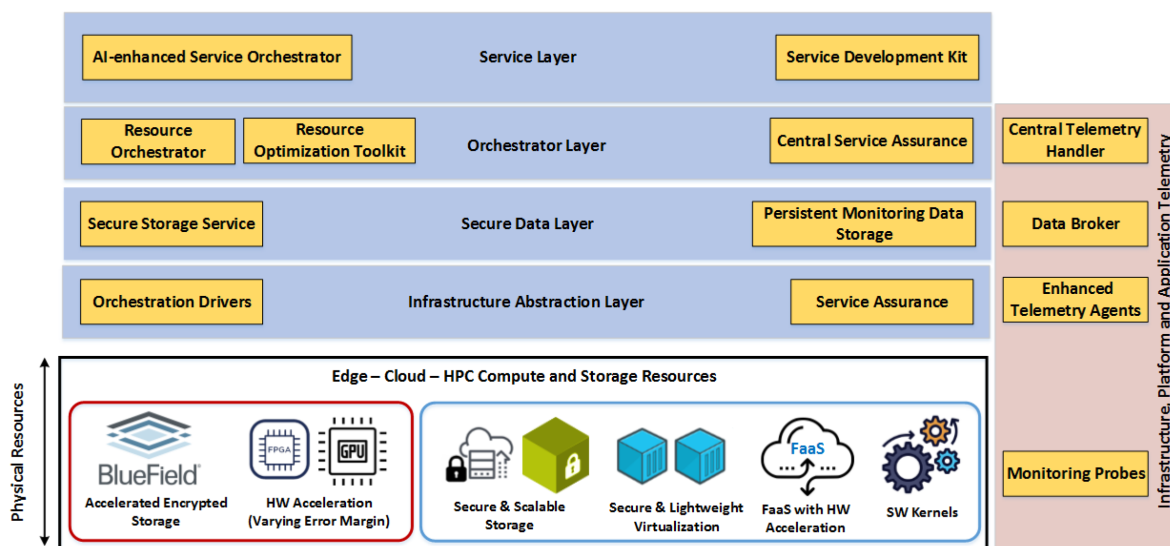

Figure 1: SERRANO high-level architecture

The Service Layer includes the *AI-enhanced Service Orchestrator* (Section 4) that analyses applications to determine the possible deployment scenarios and translates the given application requirements (high-level requirements) to lower-level ones. The Orchestration Layer ensures efficient service orchestration and resource management through the SERRANO *Resource Orchestrator* (Section 9). The *Resource Optimization Toolkit* (Section 5) provides joint computational and storage resource allocation and service placement algorithms, leveraging various optimization techniques. The *Central Service Assurance* manages the runtime lifecycle of each application deployment across the SERRANO heterogeneous infrastructure. It receives notifications from the *Service Assurance and Remediation* mechanisms (Section 6) that include data-driven mechanisms that facilitate the identification of critical situations and trigger proactively and reactively re-optimization actions to maintain the required performance level.

Across the SERRANO ecosystem resides the Infrastructure, Platform, and Application Telemetry stack (Section 7) that collects metrics from the SERRANO infrastructure and deployed applications. The main components are the *Central Telemetry Handler*, the *Enhanced Telemetry Agents*, and *Monitoring Probes*. In addition, the *Persistent Monitoring Data Storage* allows the management of the historical monitoring data, which is required mainly by the service assurance and remediation system. In addition, the Resource Layer includes heterogeneous edge, cloud, and HPC computational and storage resources encompassing the SERRANO-enhanced resources (Sections 8 and 10), while the *Orchestration Drivers* (Section 9.2) enable efficient and transparent deployment of services across the heterogeneous infrastructure.

The developed intelligent service and resource orchestration mechanisms provide an abstraction layer that automates the operation and maximizes the utilization of available diverse resources, supporting a develop once, deploy everywhere approach. This integration seamlessly links edge, cloud, and HPC resources, facilitating the processing of low-latency services that necessitate immediate action at their source. At the same time, computationally- and data-intensive applications are intelligently distributed across a diverse set of cloud and HPC platforms. The SERRANO platform (Figure 2) is a self-optimizing system that continuously adapts based on its ability to sense (detect what is happening), discern (interpret senses), infer (understand implications), decide (choose a course of action), and act (take action), within an infinite time horizon control loop. Leveraging SERRANO's abstraction mechanisms, cloud-native applications are supported towards the edge-cloud-HPC continuum.

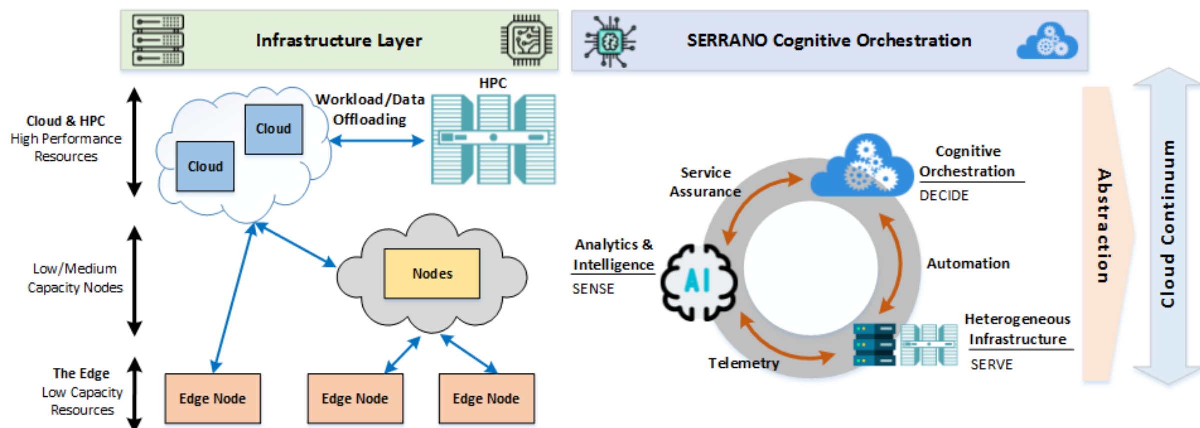

Figure 2: The SERRANO platform, utilizing edge, cloud and HPC resources and empowering the Everything as a Service (EaaS) notion towards the cloud continuum

4 Intelligent Service Orchestration

4.1 AI-enhanced Service Orchestrator

The intelligent service orchestration in SERRANO is enabled by the cooperation of several components of the SERRANO platform, including mainly the AI-enhanced Service Orchestrator (AISO) and the Resource Orchestrator but also the Central Telemetry Handler, and the Service Assurance Mechanism. For utilising intelligent service orchestration, the users interact with the SERRANO platform using the services provided by the AISO either directly or via a GUI. The final deployment considerations, as well as the actual deployment of an application to the available resources, are performed by the Resource Orchestrator. On the other hand, the Central Telemetry Handler and the Central Service Assurance are responsible for collecting performance and other data and analysing them to ensure that the application follows the specific performance levels. In this section, particular focus is given to the AISO and the ARDIA Framework.

The functionality provided by the AI-enhanced Service Orchestrator and the components' architecture have already been described in the deliverables D5.1 (M15) and D2.5 (M18). In a nutshell, the AISO provides a REST API (Figure 3) that facilitates the efficient and intelligent deployment of applications to the resources linked with the SERRANO platform. For this purpose, the application owners should have already containerised the application. Next, they provide the application deployment description in a YAML file and express the application requirements and their intent based on the parameters specified in the Application Model (part of the ARDIA Framework). The AISO employs underlying mechanisms that utilise domain-experts-defined mapping rules and telemetry-data-driven ML models and undertakes the translation of the given high-level constraints to the ones appropriate for application deployment, based on the parameters specified in the Resource Model (also part of the ARDIA Framework). Finally, it invokes the relevant Resource Orchestrator services to request the application deployment. Considering that the given application requirements and user intent can usually be satisfied in more than one way, the output provided to the Resource Orchestrator includes several suggested deployment scenarios that are most appropriate in each case.

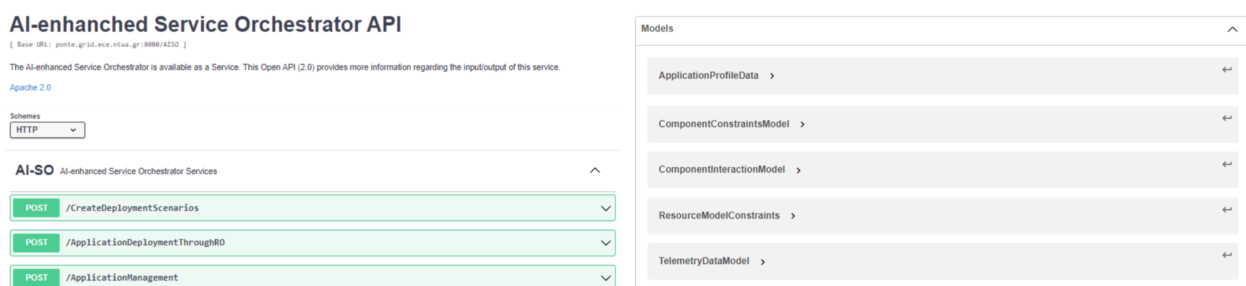

Figure 3: AI-enhanced Service Orchestrator Interface

4.2 Abstraction Models and Mapping Rules

The three abstraction models developed in SERRANO, as part of the ARDIA framework, enable the interaction among the software components that provide the service orchestration. These are the Application, Resource, and Telemetry Data Models that have already been described in deliverable D5.1 (M15). The Application Model provides the terminology required to express the application requirements, including the user intent. In particular, it enables users to specify the internal components (aka microservices) of each application, the relations among them, and, more importantly, the particular constraints that they should satisfy, either independently from one another or as a whole (an example is presented in Section 4.6). The Resource Model specifies the parameters of particular importance for different types of resources, including standalone nodes and accelerators (e.g., GPU, FPGA) as well as the relations among them. The elements specified in this model are used to formally express possible deployment scenarios of each application so that the Resource Orchestrator can further process the relevant application requirements. The Telemetry Data model specifies the parameters being collected by the Central Telemetry Handler during the deployment and execution of an application. The data collected and expressed using this model are used by several components of the SERRANO platform, including the AISO.

Table 1: The Parameters of a Mapping Rule

Parameter		Brief Description
Main Parameters	**Source**	One or more Application Model parameter
	Target	One or more Resource Model parameter
	Transformation	The process that should be followed for expressing application to resource model constraints.
Prerequisites	**Conditions**	The conditions that should be satisfied so that this mapping rule can be potentially applied
Metadata	**Origin**	Indicates if this mapping rule has been specified by domain experts or through the analysis of collected telemetry data
	Direction	Indicates if this mapping rule can be used when "moving" from source to target or vice versa

The functionality provided by the AISO is based on the Mapping Rules specified. The mapping rules were specified either manually (in close collaboration with domain experts) or automatically through the analysis of collected telemetry data. Each mapping rule has several parameters (Table 1), including but not limited to source and target elements, along with the process that should be followed (aka transformation) for moving from one data representation to the other one. The source elements are subsets of parameters specified in the Application Model, whereas the target elements are the appropriate ones specified in the Resource Model. The transformation specifies the process applied for expressing the conditions defined based on the source elements to the corresponding ones based on the target elements, and it may internally use a pre-trained ML model (as presented in Section 4.3). Apart from the aforementioned parameters, each mapping rule includes additional data,

such as the prerequisites that should be fulfilled so that this mapping rule can be used, and metadata, such as the origin of the mapping rule, its direction of usage (i.e., for "moving" from source to target elements), etc.

Several mapping rules have been specified to bridge the gap between Application and Resource Models. The design of these mapping rules was driven by (a) analysis of relevant publications in this field, (b) ongoing work in other SERRANO tasks / WPs in close collaboration with the respective partners and (c) analysis of data collected from the execution of the applications. For instance, based on publication [1] it can be presumed that when the aim is to avoid high network utilisation or to achieve low response latency, it is preferable to deploy an application to an edge device (or fog node) rather than to a cloud provider. Also, when security is of great concern during the execution of an application, the particular node tiers (Figure 4) should be taken into consideration during deployment by the Resource Orchestrator, as described in the deliverable D3.4 (M30). For instance, when isolation is of great importance, a Tier 4 node should be selected.

	Tier-0	Tier-1	Tier-2	Tier-3	Tier-4
Isolation	minimal	Yes	Yes	Yes	Maximum
Encryption	No	No	No	Could have	Yes
Trusted Execution	No	No	No	Yes	Yes
CPU/MEM Footprint	Low	medium	Low	low	Medium
Spawn Time	Fast	Fair	Ultra-fast	Fast	Fair
Specialized software	No	Yes	Yes	Yes	Yes
Specialized hardware	No	No	No	Yes	Yes

Figure 4: Security Tiers for the SERRANO platform

Regarding hardware accelerators such as GPU and FPGA, it is common knowledge that an FPGA consumes less energy that a GPU (or CPU) [2] and can instantly respond to a user's request. Nevertheless, their usage often depends on the application design and development (often some parts of an application should be redesigned or even developed from scratch using vendor-specific hardware languages) and their capabilities to adapt to computing environment changes (e.g., usage of a GPU or FPGA for some parts of the application) through their proper configuration.

The definition of the aforementioned mapping rules was specified in close collaboration with the domain experts involved in the SERRANO project. Nevertheless, in many cases, the relation among the application and resource model parameters is much more complicated. For this purpose, data were collected from the execution of applications (i.e., particular tasks) using different SERRANO resources, and the collected data were accordingly analysed and used to develop ML models that capture the exact relation among the relevant source and target entities.

4.3 Telemetry Data Analysis and ML Model Development

The telemetry data collected from the execution of different parts of the three UC applications were further examined, and relevant mapping rules were specified. More complicated relations among the elements included in the Application and Resource model were specified using ML techniques. More precisely, the respective telemetry data from the execution of a particular task of an application under different resource configurations were collected, filtered, and accordingly used for the development of the respective ML models. In particular, regression models were trained based on the data collected to be accordingly used for prediction purposes.

Data encryption and decryption (UC1) are resource-demanding processes that can be significantly improved through GPUs or FPGAs. The data collected indicated that total execution time and energy gained can be significantly improved, especially in the case of AES-GCM Encryption. Nevertheless, the expected improvement level also depends on other parameters, such as the number of instantiated computing units, which has to do with the particular algorithm implementation. The data available about each one of these two tasks (i.e., encryption/decryption) were filtered and accordingly used for training two different polynomial regression models that can predict the expected execution speedup and energy gain based on the resource type and number of instantiated computing units.

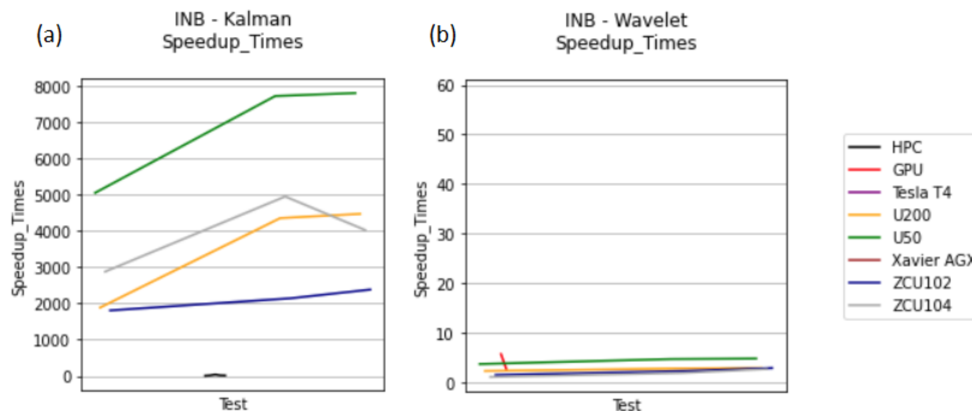

Figure 5: Execution speedup in (a) Kalman Filter and (b) Wavelet Transformation using different types of resources

The analysis of data collected regarding the Portfolio Analysis (UC2) tasks indicated that the time required, and energy consumed for the execution of Kalman filters can be significantly improved through the usage of the particular resources (Figure 5 and Figure 6). In this case, the type of accelerator used has a tremendous impact on the execution speed-up and the energy consumption gains, with the most significant improvement coming from using an Alveo U50 accelerator [3]. The time needed and the energy consumed for the execution of a Wavelet transformation are also affected by the particular resource type. Nevertheless, in this case, the type of resource has a lower impact on the execution time and energy consumption in comparison with the corresponding figures noticed for Kalman Filters. The aforementioned data were used for training an ML model that can predict the expected energy gain based on the type and particular brand/model of the resource device.

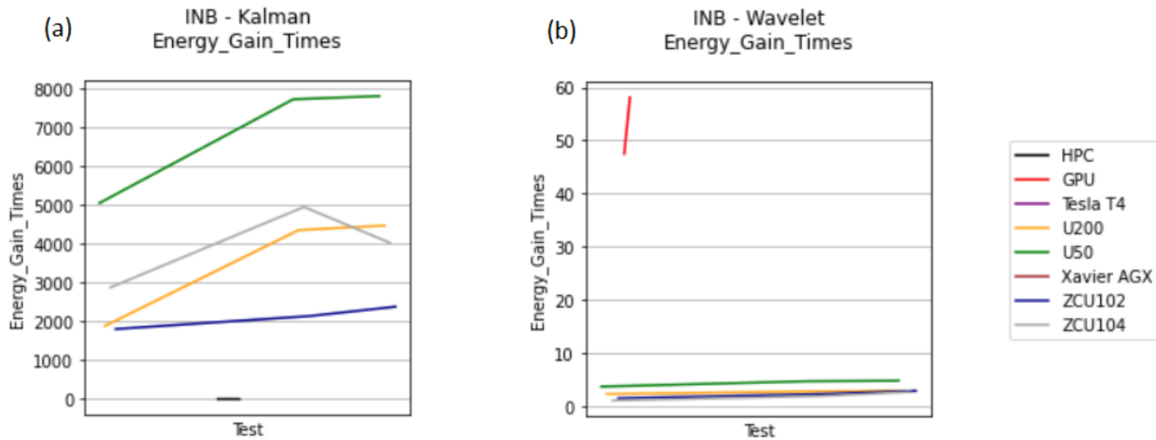

Figure 6: Energy gain in (a) Kalman Filter and (b) Wavelet Transformation using different types of resources

Anomaly detection in manufacturing (UC3) is a challenging process since a considerable amount of data should be gathered and analysed to detect potential discrepancies that indicate a failure is about to happen. In this manner, equipment can be utilised for their whole lifespan and their replacement can be programmed in advance, thus avoiding unexpected delays in the production line and unneeded expenses. Hence, continuously monitoring and assessing anomalies in real-time is of great importance. The amount of energy ML techniques consume in this process is another critical factor [3]. The applications and the respective microservices were tested under different resource configurations (i.e., in an edge or HPC device) for the processing of different workloads, and the relevant data regarding the total execution time of the particular tasks and the energy consumed were recorded (as part of the WP4 tasks).

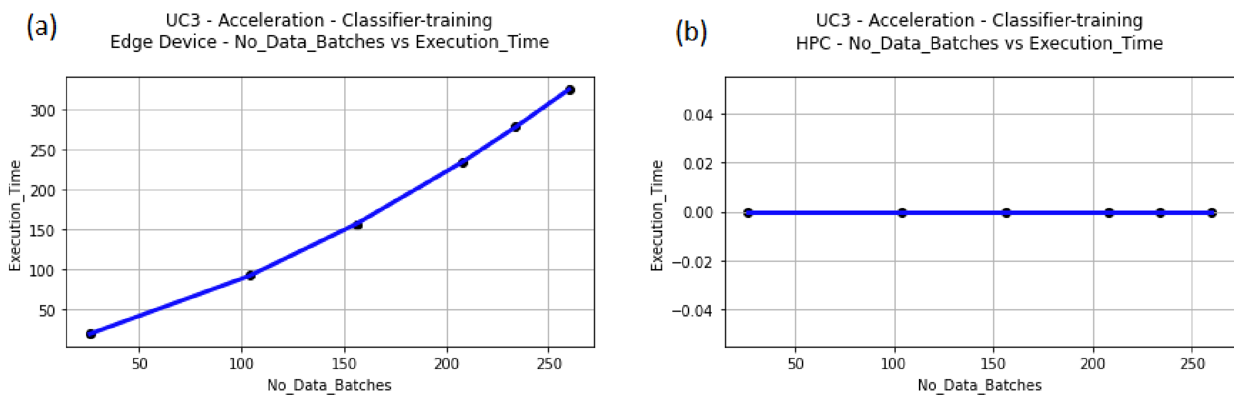

Figure 7: ML model for predicting the Total Execution Time of a particular microservice for different workloads in (a) an Edge Device and (b) in HPC

Then, several polynomial regression models were developed to predict the expected execution time and energy consumption of different application microservices respectively (Figure 7 and Figure 8). For this purpose, the data were split in three groups, i.e., training, validation, and testing, so that they could be used for hyper-parameters tuning, model training and validation purposes.

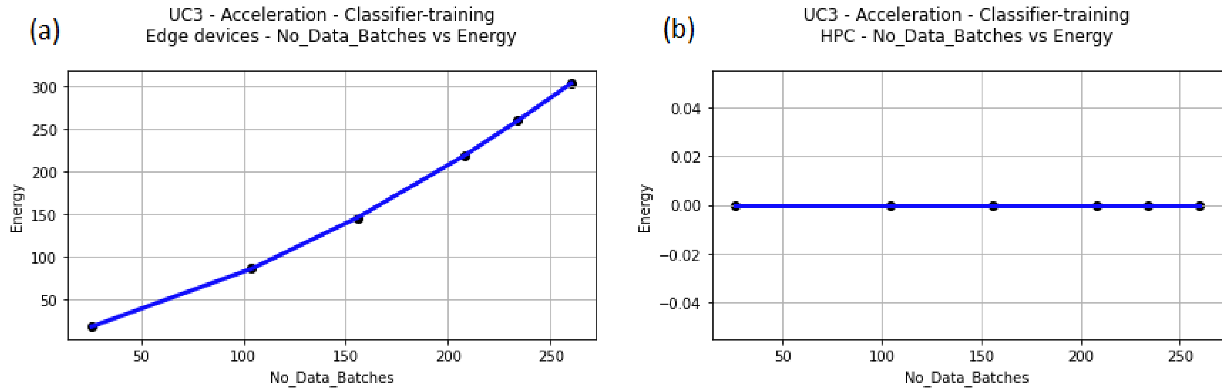

Figure 8: ML model for predicting the Total Energy Consumption of a particular microservice for different workloads in (a) an Edge Device and (b) in HPC

4.4 Translation Mechanism

The data provided by the end-user regarding the particular application requirements and user intent are used by the AISO to detect the potential deployment scenarios feasible in each case so that the given restrictions are satisfied. Each deployment scenario specifies the type and, in some cases, the suggested concrete details of the resource. The deployment scenarios are expressed in JSON format based on the elements specified in the Resource Model.

The AISO examines the data provided by the end user along with the Mapping Rules (MRs) already specified for translating the given parameters to the appropriate resource constraints. More precisely, it focuses on the source and target elements of the MRs defined in order to find the ones that can be directly or indirectly applied to the given parameters. These MRs are accordingly applied one by one based on their relative order of execution (i.e., priority is considered). Also, a branch is created if there is more than one way to achieve the same purpose (i.e., satisfy the respective condition). Through this process, the potential deployment scenarios are built, each containing several restrictions to the value of the resource model parameters that should be simultaneously satisfied without any contradiction among them.

In the following paragraphs, the focus is given to the detection and usage of a particular MR.

- Presence of Source Data

In case all source elements are available (i.e., a constraint has been specified regarding the appropriate set or range of their values), a MR can be directly applied for the translation of the source parameters (expressed based on the elements of the Application Model) to the appropriate target parameters (expressed based on the elements of the Resource Model). For instance, when low data transfer latency is necessary, the system will propose deploying the application in the Edge Device (i.e., close to the location where the data are being produced) rather than in a cloud provider.

- Presence of Target Data

On the other hand, when all target parameters are available, the AISO examines possible deployment configurations to satisfy the given application requirements. More precisely, for each possible configuration, it uses the predefined ML models for detecting/predicting the expected outcome and hence selecting the ones the outcome of which is compatible with the given constraints. For instance, when a particular task should be completed in a limited amount of time, the system uses the ML models for predicting the expected amount of time (or a relevant parameter that can be directly linked with this one, such as execution speed-up factor) for different types of resources and proposes the usage of those resources that should produce an outcome that is compliant with the initial requirement.

It should be noted that both source and target elements may already exist in the parameters specified by the end user. In this case, the output of the respective mapping rules should be compatible with the user data provided. In particular, the output of the mapping rules fired based on the given source data should be a superset of the given one. Also, the expected outcome of the possible resource configurations that can take place should be compatible with the constraints specified by the end user. For instance, if both constraints above about data transfer latency and energy consumption have been specified, the proposed deployment scenario should contain those resources that simultaneously satisfy both constraints.

4.5 Integration with a Graphical Interface

The creation of the deployment scenarios and the allocation of the appropriate resources for the deployment of an application, considering the application requirements and user intent, are driven by appropriate JSON and YAML descriptions. The owners of each application should prepare both (either manually or with the aid of a GUI) and accordingly provide them to the respective SERRANO orchestration and deployment services.

The application requirements and user intent are enclosed in a SERRANO-specific JSON description with a predefined structure based on the Application Model elements (part of the ARDIA framework – described in the deliverable D5.1). The AISO checks the JSON structure to ensure that it complies with the required one and that the elements included are expected. The allocation of the appropriate type and resource quantities for the deployment and execution of an application is done using a deployment descriptor. Since the edge and cloud platforms in SERRANO are managed by Kubernetes instances, the technical details regarding the deployment of each application are expressed in a Kubernetes-specific YAML file using the Kubernetes YAML Generator [5].

An effort was put into integrating the Alien4Cloud (A4C) [6] platform with the AISO and the SERRANO platform to simplify the process above and improve user experience. The A4C platform was adequately configured to be able to deploy cloud-native applications to the SERRANO platform. A4C is an open-source software platform for managing applications using the DevOps paradigm. This platform is compatible with TOSCA [7], which is a standard modelling specification language for describing applications on cloud computing platforms. The TOSCA specification was extended for our purposes and an Alien4Cloud Orchestrator Plugin was developed to deploy applications on the SERRANO ecosystem. The plugin

generates the Kubernetes-based deployment configuration for the application and the intent specification JSON description as required by the AISO for deploying the application to the appropriate resources.

The successful integration of the Alien4Cloud platform with the AI-enhanced Service and Resource Orchestrators enables users to deploy their applications in the SERRANO platform through a user-friendly environment. The Alien4Cloud platform allows users to check the status and logs of application components. To this end, the appropriate logic should be implemented in the developed plugin for the SERRANO platform. Thus, the plugin uses SERRANO telemetry data to show the current status and application information, such as kernel executions, component performance, and component restarts.

4.5.1 SERRANO-TOSCA

The SERRANO extension to the TOSCA specification contains two main parts: the intent model and the Kubernetes model. The intent is modelled using TOSCA *data* types (strings, numbers, lists, maps, scalar units for size), and some input fields are constrained. For example, the Data Storage Duration intent allows only two values: Short-Term and Long Term; the Service Level Up-Time intent is constrained to match the pattern *"`>/=[0-9]{1,2}\%`"*: greater than or equal with a number (1 or 2 digits) and a percent sign.

Figure 9 presents a class diagram for the developed specification. On the right part, there are data types, i.e. entities that only contain properties. On the left part, there are Node types, i.e. entities that have properties, requirements, and capabilities. A node requirement must be satisfied through the connection with another entity that exposes the required capability.

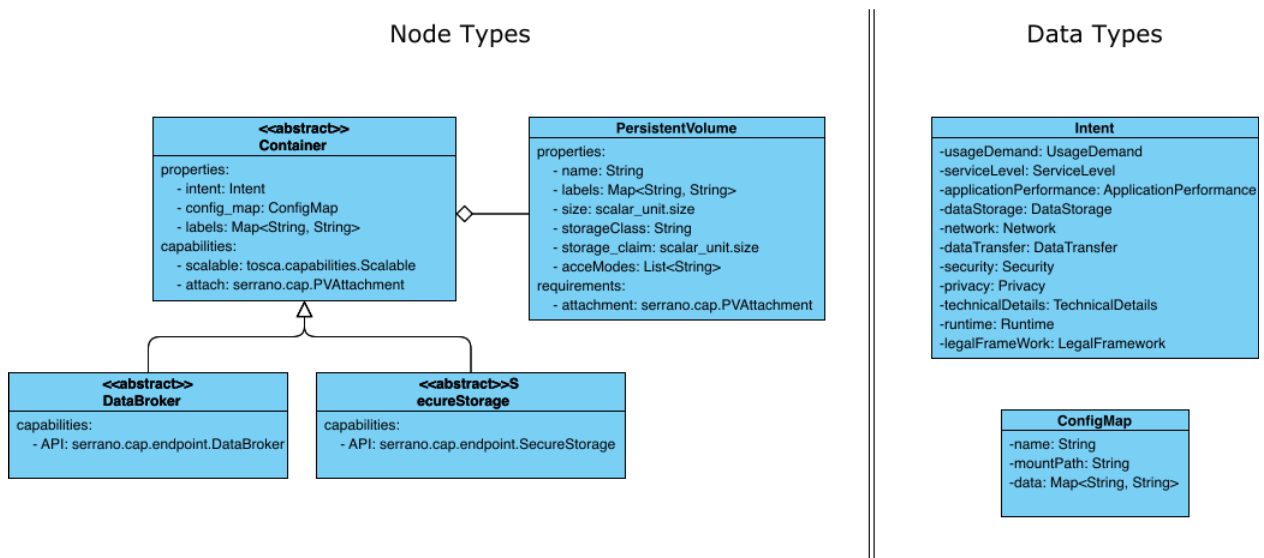

Figure 9: Class diagram for SERRANO-TOSCA entities

The Kubernetes entities selected for modelling the TOSCA extension are Persistent Volumes, and Config Maps. The Config Map is a data type defining a name, a mount path, and the data map (where each key is a file name, and the corresponding values are the file content). The

repository TOSCA type is used to model private Docker Registries, and a new artefact type derived from `tosca.artifacts.Deployment.Image` is introduced to refer to container images.

Containers and Persistent Volumes are modelled using TOSCA *node* types. The proposed extension defines an abstract SERRANO Service named *Container,* extending the root type and defining the following:

- Properties – fields that allow the user to change values during application definition
  - Intent – the previously mentioned data type modelling all parameters from the Application model of the ARDIA Framework.
  - Config Map – files required for the configuration of the component
  - Labels – labels for the Kubernetes specification of this component
- Capabilities – fields that expose a given functionality of this component
  - Scalable – allows the user to set a default, min, and max number of replicas for this component; this capability exists in the default TOSCA definition.
  - Attach – exposes the capability `serrano.cap.PVAttach` to attach one or more Persistent Volumes to this component; this capability is defined for the project

The Persistent Volume definition contains all necessary fields extracted from the Kubernetes Specification for persistent volumes. This node expresses a requirement for the `serrano.cap.PVAttach` capability, which is exposed by the Container node and all further extensions of this node.

Two additional abstract service definitions have been constructed: Data Broker and Secure Storage. Both services are core services the SERRANO platform provides, and the use case components can impose requirements over their capabilities. If an application topology contains abstract services, these will not be deployed, but all components that depend on abstract services will have their configuration files updated with the endpoint of the SERRANO-provided core service. The abstract services provided by SERRANO must first be registered in the Alien4Cloud administration panel. Moreover, an explicit version of these services has been defined, in case the end-user wants to have a private deployment of the Data Broker or Secure Storage services. For example, if users want to deploy the microservices at the edge, they can also deploy an instance of the Data Broker component nearby instead of using the cloud-based Data Broker service provided by SERRANO.

Finally, the plugin uses the TOSCA life-cycle parameters to update the ConfigMap of a component that depends on another component (e.g., uses its API). The developer of a SERRANO-TOSCA component has specific keywords in the ConfigMap that are replaced during the generation of the YAML file. Then, the developer of the TOSCA definition will add these keywords to the create step of the TOSCA life-cycle interface, as presented in the following code listing (Table 2). In this case, the ConfigMap must contain the keywords *INPUT_DATABROKER_IP* and *INPUT_DATABROKER_PORT,* which will be replaced by the actual IP address and port of the target component satisfying the *mqtt* requirement of this component.

Table 2: TOSCA parameters for updating Kubernetes ConfigMap

```
interfaces:
      Standard:
        create:
          inputs:
            INPUT_DATABROKER_IP: { get_property: [REQ_TARGET, mqtt, ip_address]}
            INPUT_DATABROKER_PORT: { get_property: [REQ_TARGET, mqtt, port]}
```

4.6 Example of Usage

We utilized the AISO and ARDIA Framework to facilitate the infrastructure-agnostic deploy. This deployment comprised three microservices. For a more detailed technical insight into this application, refer to Deliverable D6.5 (M27). The requested user intent was the provision of instant response to events coming from sensors while keeping energy consumption as low as possible. The application was containerised, and the Kubernetes deployment descriptions were prepared in advance so that it could be accordingly used for resource allocation and data collection for the ML model training. The provided Kubernetes descriptors were used to check if the Orchestrator plugin generates the correct configuration. Also, an initial JSON file with the application requirements was developed by the partners. The example presented here shows how users can create these documents and deploy an application using the plugin developed for the Alien4Cloud platform.

4.6.1 Application requirements, intent specification and deployment description

The Alien4Cloud framework has been properly configured so that the users can express the application requirements and their intent along with the application deployment description based on the SERRANO Abstraction Models (part of the ARDIA framework). Figure 10 presents the visual representation of the application in Alien4Cloud. The application has been composed in the Topology Editor interface using the SERRANO-TOSCA definitions specific to this use case. The considered application has three components (on the left side in the figure), all expressing a requirement (dependency) on the API capabilities exposed by the Data Broker and Secure Storage services. In this example and in the Topology Editor, the API dependencies are abstract, meaning the SERRANO platform provides them. The Data Manager component (the top one in the figure) also requires a Persistent Volume to be mounted in this container file system. The volume is used to store data received from a remote location that will later be used by the other two components. The other two components (i.e., Classifier Trainer and Classifier Inference) receive messages from the Data Manager and use the Storage Service to read data for training and classification.

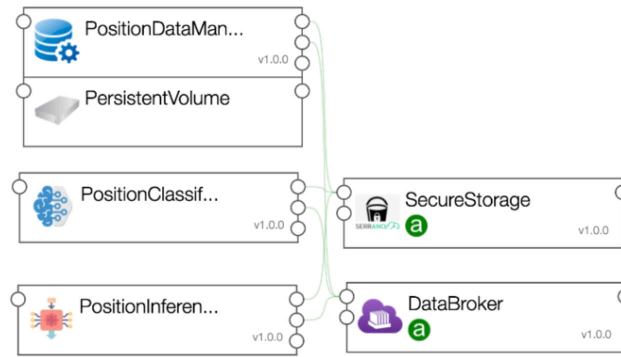

Figure 10: Application definition using SERRANO-TOSCA definitions in Alien4Cloud Topology Editor

The intent can be specified for each component extending the Container definition. When clicking on one of the components, for example, the Classifier component, the user can set different properties, such as the intent. Figure 11 shows the main dialogue for setting the intent parameters. The intent is categorised based on different dimensions of the possible constraints, however some remain top level, such as Energy Consumption and Overall Cost. The figure shows the dropdown when setting the Energy Consumption parameter. The user can access the different categories from the left side panel or click the edit button from the right panel.

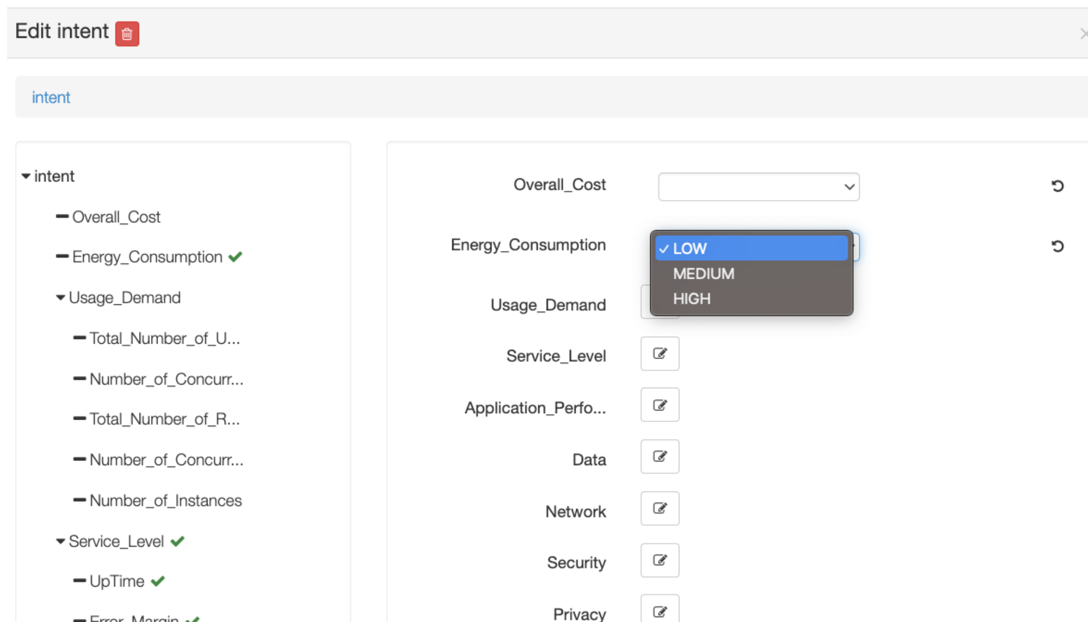

Figure 11: Intent Dialogue for one component in Alien4Cloud

Figure 12 presents the intent dialogue for the Application Performance dimension. The total execution time is set to low, and all other parameters are not set. Here, as can be observed, the user can set constraints on the type of accelerator the component needs (e.g., GPU). The example application does not explicitly require any accelerator because the computationally intensive part is executed using the Functional as a Service (FaaS) execution model that allows the on-demand deployment of accelerated kernels through the SERRANO SDK (Section 9.4.3).

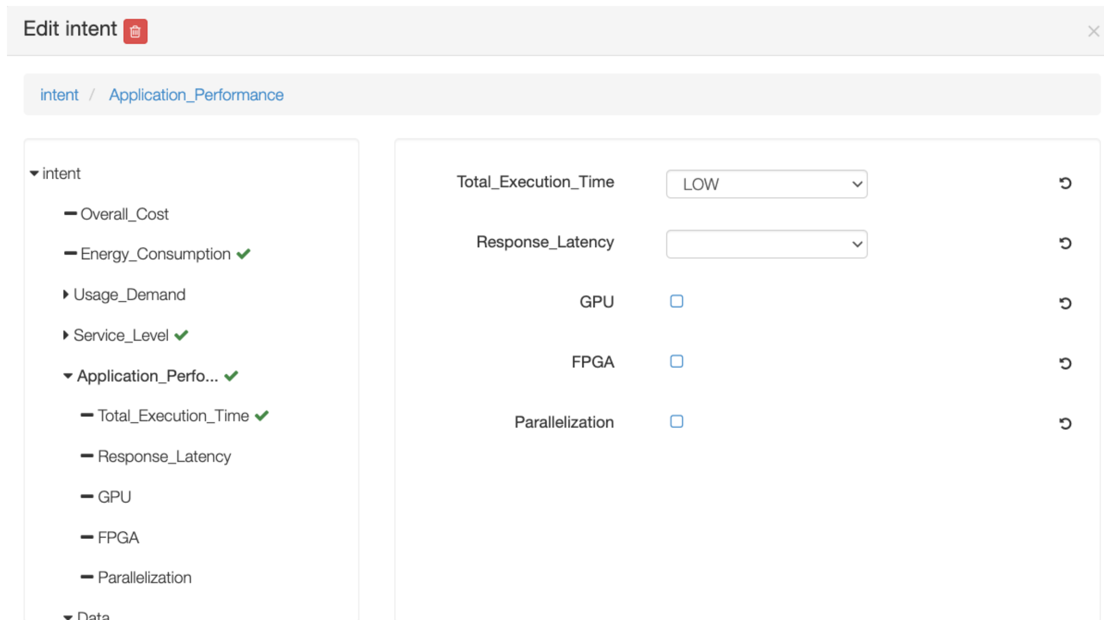

Figure 12: Intent dialogue for the application performance dimension

The Orchestrator plugin developed for Alien4Cloud reads the SERRANO-TOSCA topology definition, which also contains the intent, and generates the corresponding JSON and YAML descriptions. Generating the intent JSON is straightforward. All that is needed is a transformation from the TOSCA to JSON format. The generation of Kubernetes-based deployment entities is more challenging. Each component is processed, and a Deployment object is created. The next step is to create the ConfigMaps associated with each component and add it to the Deployment object. Persistent Volumes and Persistent Volume Claims are then created and linked with the deployment. Finally, the TOSCA requirements of each component are inspected to see if the current component, C1, depends on another component, C2. Component C2 can be either a user-defined component or an abstract service the SERRANO platform provides. In the first case, a Kubernetes Service is defined for C2, and the config map of C1 is updated to use the Service name as the IP address. In the latter case, the plugin is configured to know the address of the SERRANO-provided services that will later be used to update the ConfigMap of component C1. Finally, dependencies between components are considered when populating the intent JSON, specifically the application workflow field. The Orchestrator will impose that C2 will start before C1.

After all the nodes have been investigated, the request will be sent to the AISO. The AISO will respond with the unique deployment identifier the Resource Orchestrator provides. This identifier will be used to query the SERRANO telemetry services for information about the component status and logs. Figure 13 shows the status during deployment. The two abstract services provided by SERRANO are up and running, and the three components from the use case application are under deployment. The user can inspect the deployment-related events using the *Events* tab on the right sidebar. Logs and performance metrics can be accessed in the Logs interface by clicking the last button on the left sidebar.

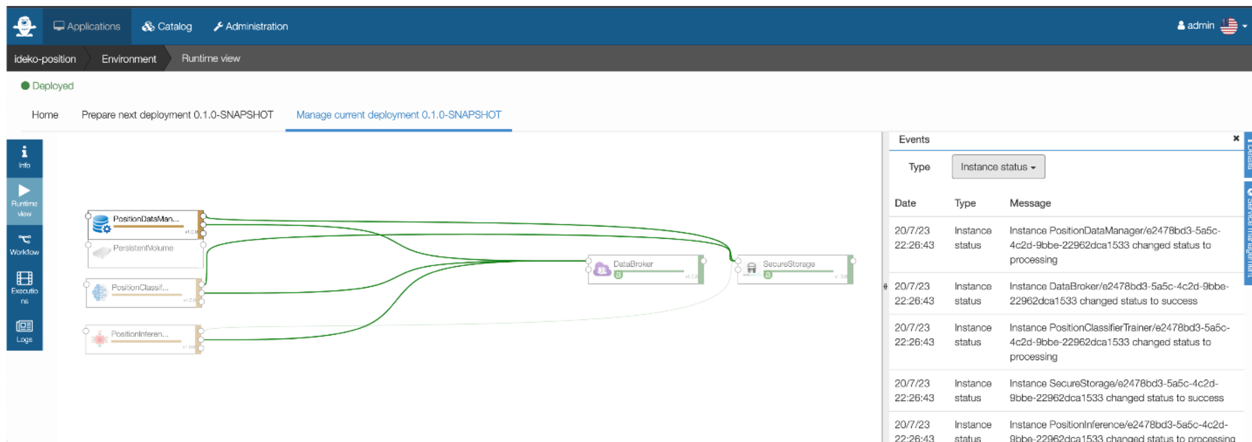

Figure 13: Deployment runtime interface

SERRANO users, who are not necessarily the developers of an application, can use the components in the catalogue to compose their application topology. They can use the SERRANO storage and messaging services or deploy their own instance. The developed Orchestrator plugin transforms this topology dynamically in a Kubernetes specification. User intent can be formulated for each component using a user-friendly dialogue; the intent will be transformed to the JSON required by the AISO, taking into consideration dependencies. The two descriptions are compiled and sent to the AISO, which in turn contacts the Resource Orchestrator. The deployment unique identifier is returned to the Orchestrator plugin and can be used to inspect the status of the components and performance metrics.

4.6.2 Mapping Rules and Translation Mechanism

The process output described in the previous section is the JSON description with the application requirements and user intent, along with the YAML deployment descriptor file with additional parameters about the application – microservices deployment (Figure 14).

The AISO further processes the given descriptions, especially the data recorded in the provided JSON description, taking into account the already specified Mapping Rules. In brief, the given workload amount is used by the AISO in order to figure out the anticipated execution time and energy consumption for resources of different types and eventually select and suggest the most appropriate one. For example, based on the data and modes presented in the previous section, the AISO detects that HPC performs much better than an edge device. However, considering the size of data and the amount of time necessary for their transmission, it proposes the usage of an edge device. The output of the above process is a JSON description (Figure 15) with the potential deployment scenarios that can take place (in this example, only one deployment scenario is available).

Finally, the SERRANO Resource Orchestrator utilizes the created descriptions to allocate suitable resources and deploy the corresponding microservices. Once the deployment is completed, the SERRANO Resource Orchestrator returns a unique identifier, facilitating further actions. With this identifier, users can access additional information about the application deployment and manage it seamlessly through both the AISO and the Alien4Cloud GUI.

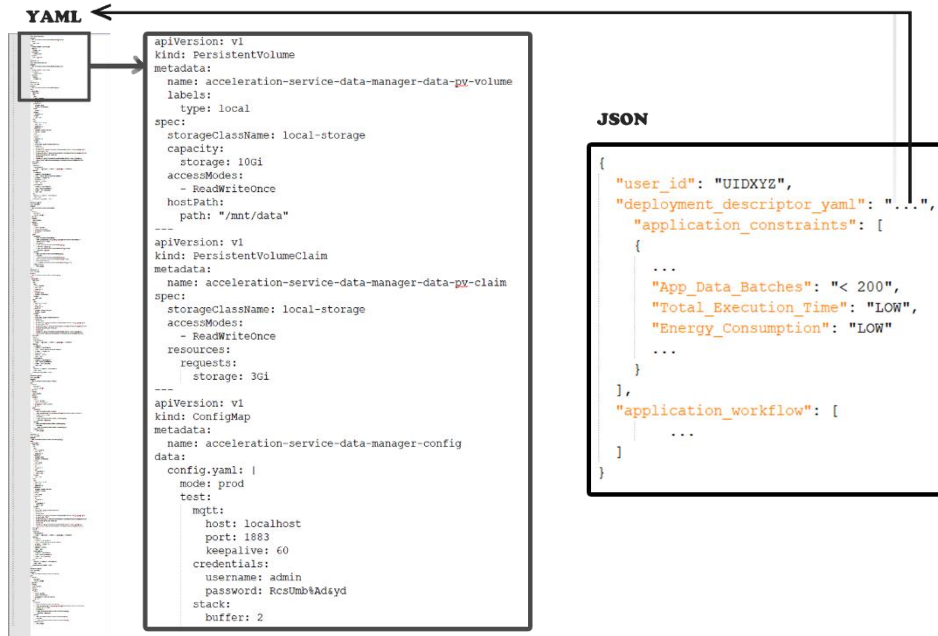

Figure 14: Overview of the given YAML and JSON files – AISO input

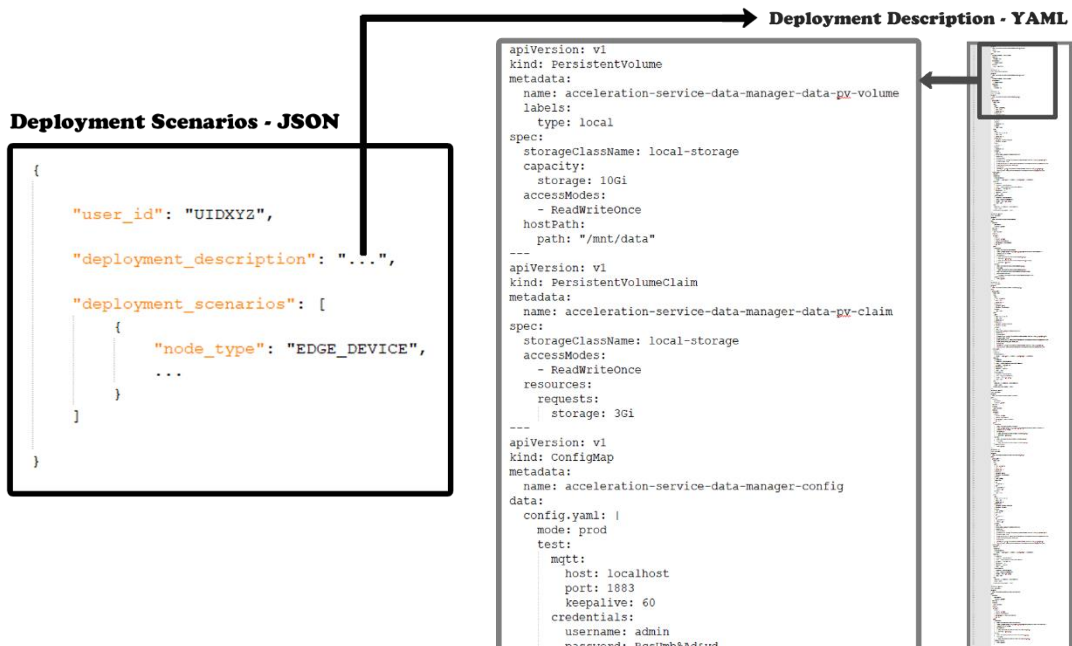

Figure 15: AISO Deployment Scenario(s) – JSON File Provided to Resource Orchestrator

5 Algorithmic Framework

The resource allocation problem in heterogeneous, dynamic, and multi-technology environments is highly complex, mainly due to the multitude of conflicting objectives involved. To address this complexity, the utilization of multi-objective optimization algorithms becomes imperative. Towards this direction, we developed a set of algorithms that leverage multi-objective optimization, AI/ML techniques, and heuristics. These algorithms offer a range of trade-offs between optimality and complexity, enabling efficient satisfaction of the diverse and stringent requirements of heterogeneous and distributed applications. In addition, a selection of these algorithms has been integrated into the Resource Optimization Toolkit (ROT). Next, we present the algorithms developed during the second iteration of the implementation period (M16-M31), the final developments in the ROT, and the successful integration of these algorithms into the toolkit

5.1 Cloud-native Applications' Workload Placement in the SERRANO Edge-Cloud Continuum

The standard monolithic application architectures, where all logic resides within a unified and inseparable entity, proved quite efficient in the past. Nonetheless, the gradual establishment of cutting-edge ICT technologies (5G/6G, optical networks, virtualization) have escalated the application design complexity. Coupled with the relentless need for updates to satisfy the ever-increasing Quality of Service (QoS) demands, the traditional monolithic approach stands inadequate in this rapidly evolving landscape, thereby necessitating a novel application architecture. The cloud-native approach presents itself as a compelling alternative: By taking full advantage of the cloud computing model and decomposing the applications into microservices, it offers the flexibility, scalability, and robustness. Moreover, emerging services, interconnected products, and other digitized assets generate massive amounts of data at the network's edge, often requiring ultra-low processing delays. To address these challenges, the edge computing paradigm has arisen, where computing units are placed at various locations close to the data sources. Moreover, edge resources can be utilized in conjunction with the cloud, forming a robust edge-cloud continuum.

The present work focuses on developing a novel mechanism to appropriately allocate the available resources across the various layers of an edge-cloud infrastructure to support the incoming workload from cloud-native applications. The aim is to jointly optimize a weighted combination of the average (per application) delay and average service cost while simultaneously guaranteeing that the delay between dependent microservices and the available infrastructure resources align with the applications' requirements. Initially, the problem is modelled as a Mixed Integer Linear Programming (MILP) problem. To tackle the excessive execution time of finding the optimal solution, a fast heuristic algorithm is implemented, referred to as the Greedy Resource Allocation Algorithm (GRAA). This algorithm is further employed by a novel Rollout technique to optimize further the generated solution (namely Rollout based on GRAA), relying on Reinforcement Learning (RL) principles.

5.1.1 Related work

The resource allocation problem in virtualized environments is a multi-dimensional research area that has attracted the interest of the research community. The modelling of the problem among the different works varies according to the considered topology and the adopted technologies, while the proposed solutions employ techniques from the wider realm of mathematics and computer science.

Authors in [9] developed "Foggy", an architectural framework based on open-source tools that handles requests from end users in a multi-level heterogeneous fog/edge environment. The requests arrive in a FIFO queue, and at each stage, the available nodes are ranked by their processing power and their networking towards the end user to extract the best match. The authors in [10] proposed a dynamic resource scheduling scheme for critical smart-healthcare tasks in a edge-cloud topology. Their model consists of a multi-agent system (MAS) with four kinds of agents named personal agent (PA), master personal agent (MPA), fog node agent (FNA), and master fog node agent (MFNA). The scheduling strategy relies on effective prioritization of the tasks according to their criticality and on balancing network load. In [8] a system for microservices placement in a multi-layered fog/edge environment is implemented, targeting to place them as close as possible to the data sources.

Reinforcement learning is a technique that has been gaining momentum in the context of resource allocation. The authors in [11] present a deep reinforcement learning approach, based on state-action-reward-state-action (SARSA), for addressing the problem of task off-loading and resource allocation in Mobile Edge Computing (MEC) environments. They model user requests as a sequence of sub-tasks, which can be executed by either the nearest edge server, the adjacent edge server, or the central cloud. The proposed solution aims to minimize service delay and energy consumption by dynamically making offloading decisions and allocating resources based on the current state of the infrastructure. Wang et al. [12] present a solution for the microservice coordination problem in mobile edge computing environments where mobile users (e.g., autonomous vehicles) offload computation to the edge clouds. The authors aim to minimize a weighted combination of delay and migration costs by determining the optimal deployment locations for microservices. They first propose an offline algorithm able to derive the optimal objective and then a Q-learning-based reinforcement learning approach that produces a near-optimal solution in real-time. Chen et al. [13] propose a deep reinforcement learning solution for microservice deployment in heterogenous edge-cloud environments. They consider microservices as a service chain, in which the microservices must be executed in a pre-specified order. Simulations are conducted with a combination of real and synthetic data, with the objective of minimizing the Average Waiting Time (AWT) of the microservices.

In this work, we explore the assignment of microservice-based applications in a distributed edge-cloud infrastructure, considering key operational aspects. Contrary to the mentioned works, we address the dependencies formed by communicating microservices as delay constraints between the corresponding service nodes to guarantee their seamless communication, which is a crucial concern when considering geographically dispersed

infrastructures. These dependencies, often in the form of information exchange requirements or service chains, are directly affected by communication latency during runtime. In addition, to the extent of our knowledge, we are the first to introduce the multi-agent Rollout technique in such a scenario. This unique optimization approach, grounded in Dynamic Programming and Reinforcement Learning principles, utilizes greedy heuristics to approximate future decisions. Albeit easy in understanding and implementation, it can provide significantly improved solutions.

5.1.2 Problem formulation

We consider a hierarchical edge-cloud infrastructure, with multiple layers of edge resources (e.g., on-device, near-edge, far-edge) to serve the incoming cloud-native workload. We assume that the edge layers consist of machines with relatively limited resources, such as raspberry Pi's, NVIDIA Jetson, servers, mini – Datacenters, etc. while the cloud layer has practically unlimited resources.

The hierarchical edge-cloud infrastructure is denoted as an Undirected Weighted Graph $G = (V, E)$. Each node $v \in V$ is described by the tuple $\tau_v = [c_v, r_v, o_v, n_v]$, where c_v is node's v CPU capacity measured in CPU units, r_v is the node's RAM capacity measured in RAM units, o_v is the node's operating cost and n_v is the node's networking cost coefficient. Operational cost relates to the expenses made for purchasing, deploying, and operating the respective computing/storage systems. This cost is small for the cloud layer, since providers achieve economies of scale, and gradually increases for the edge layers, due to their limited resources, the small number of customers and their geographically dispersed placement. Networking cost coefficient n_v results from the usage of any link from the nodes where data are generated to the node(s) v where computing operations take place and is multiplied by the ingress data to deduce the actual networking cost of service. The coefficient is minimal for the near edge nodes, where links are shorter in distance and cheaper to install, while it gradually increases up to the massive links connecting the cloud nodes. Generally, data is generated at the lower levels of the infrastructure that can be either equipped with computing resources or not. As they are typically located in the near edge, the delay is small for transferring the data to a subset of near edge nodes as they are located closer to the data-source, given their plurality and thus higher geographical density, while it increases for the higher layer nodes (far edge, cloud). Finally, each link $e \in E$ between two nodes v and v' is characterized by a weight $l_{v,v'}$, representing the communication (propagation) delay of nodes v and v'.

The workload under consideration consists of a set A of cloud-native applications. Each application $a \in A$ is described by an Undirected Weighted Graph $G^a = (V^a, E^a)$, with the nodes V^a corresponding to the microservices that make up the application and the arcs E^a the inter-dependencies (communication requirements) among them. Each cloud native application has a source node $\pi_a \in V$ and each microservice $i = 1, \dots, |I^a|$ of application a, has specific resource requirements described by the tuple $[\varepsilon_{a,i}, \rho_{a,i}, s_{a,i}]$, where $\varepsilon_{a,i}$ is the microservice's CPU demand, $\rho_{a,i}$ is its memory demand and $s_{a,i}$ is the size of the input data. Furthermore, each arc $e \in E^a$ between two microservices $\iota, \iota' \in V^a$ has a weight $\lambda_{\alpha,\iota,\iota'}$ that represents the maximum acceptable delay between the corresponding service nodes v, v' of

these microservices. This is a measure of the intensity of the dependency between these two microservices, in a sense that highly dependent microservices should be served by the same or geographically approximate nodes to reduce communication costs and guarantee application's efficiency with in-time calculations.

In what follows, we present the mathematical formulation of the cloud native resource allocation problem over a cloud-edge infrastructure. The optimization objective is a weighted combination of the average (operational and networking) cost and the maximum delay per application assignment, with respect to computing and networking constraints imposed by the applications requirements and nodes' resource availability.

5.1.2.1 MILP formulation

Table 3: MILP variables

Notation	Interpretation
V	Total number of nodes
A	Total number of applications
I_a	Total number of microservices for the a'th application
o_v	Operating cost of node
$\lambda_{a,i,i'}$	Relative upper delay limit between micro-services i, i' of an application
$l_{v,v'}$	Communication delay between nodes v and v'
c_v	Total available CPU units of node v
r_v	Total available memory units of node v
$\varepsilon_{a,i}$	CPU units required by the i'th microservice of application a
$\rho_{a,i}$	Memory units required by the i'th microservice of application a
n_v	Networking cost coefficient of node v
$s_{a,i}$	Weighting coefficient to control the optimization objective
$x_{v,a,i}$	Binary variable, which is equal to 1 if the i'th microservice of application a is assigned to node i, and 0 otherwise
τ_α	Integer variable that denotes the monetary cost for serving the application
θ_α	Integer variable that denotes the maximum propagation latency of the cloud-native application

Objective function:

$$\min w \cdot \sum_{\alpha=1}^{A} \tau_\alpha + (1-w) \cdot \sum_{\alpha=1}^{A} \theta_\alpha \qquad (1)$$

Subject to the following constraints:

C.1. Placement of the microservices to nodes. For each application $a = 1, \dots, A$ and for each microservice $i = 1, \dots, I_a$

$$\sum_{v=1}^{V} x_{v,a,i} = 1 \qquad (2)$$

C.2. Respect of the relative latency between the applications' microservices. For each application $a = 1, \dots, A$, and each pair of microservices of application a, $i, i' = 1, \dots, I_a$,

$$l_{v,v'} x_{v,a,i} + l_{v,v'} x_{v',a,i'} \leq \lambda_{a,i,i'} + l_{v,v'} \qquad (3)$$

C.3. The allocated CPU units of the assigned microservices cannot surpass the number of available CPU units at each node. For each node $v = 1, \dots, V$,

$$\sum_{a=1}^{A} \sum_{i=1}^{I_a} \varepsilon_{a,i} x_{v,a,i} \leq c_v \qquad (4)$$

C.4. The allocated Memory units of the assigned microservices cannot surpass the number of available Memory units at each node. For each node $v = 1, \dots, V$,

$$\sum_{a=1}^{A} \sum_{i=1}^{I_a} \rho_{a,i} x_{v,a,i} \leq r_v \qquad (5)$$

C.5. Total monetary application cost $\tau_a$calculation. For each application $a = 1, \dots, A$

$$\tau_\alpha = \sum_{v=1}^{V} \sum_{i=1}^{I_a} (o_v + n_v \cdot s_{a,i}) \cdot x_{v,a,i} \qquad (6)$$

C.6. Maximum per application latency (propagation) calculation. For each node $v = 1, \dots, V$, for each cloud native application $a = 1, \dots, A$, and each of its microservices $i = 1, \dots, I_a$,

$$\theta_\alpha \geq x_{v,a,i} \cdot l_{\pi_a,v} \quad (7)$$

The objective function (Eq. 1) is the weighted sum of the maximum delay and cost per applications' assignments, where $w = 0$ considers purely the delay minimization problem, while $w = 1$ deals with the cost minimization problem. Any intermediate value of w considers both of the aforementioned parameters with different contribution in the calculation of the total cost. Note that our considered formulation supports general workloads (not strictly cloud-native applications) that can take the form of an application with a single microservice.

5.1.3 Resource allocation mechanisms

Given the problem is of the NP-hard class [16], the proposed MILP is computationally intensive, with prohibitively large execution times even for small-scale problems. Therefore, we developed sub-optimal mechanisms. Firstly, we present the Greedy Resource Allocation Algorithm (GRAA), designed to deduce the optimal placement for each microservice in a greedy fashion. Subsequently, we introduce the multi-agent Rollout mechanism, a meta-heuristic algorithm that exploits GRAA to deliver an improved solution through an iterative process.

5.1.3.1 Greedy Resource Allocation Algorithm (GRAA)

GRAA is a greedy heuristic that seeks to obtain a satisfactory, albeit sub-optimal, solution by addressing the application demands in a best-fit manner. GRAA takes as input the infrastructure graph $G = (V, E)$ along with all the applications' demands and its microservices described by graph $G^a = (V^a, E^a)$ for application a, $\forall a = 1, \dots, A$. Applications are handled sequentially. After selecting an application, its first microservice is selected and the candidate infrastructure nodes with enough resources are calculated in order to accommodate it. These nodes are ranked based on the objective function considering the cost and the latency introduced by the assignment of the microservice $i = 1, \dots, I_a$. to each node. The best node $v \in V$ is selected and the demanded by the microservice computing and memory resources are reserved. If the application consists of more than one microservices, the next microservice is selected. The same process is followed for the following microservice with the addition of the relative latency constraint between the communicating microservices. Hence, given the first microservice's location, the nodes $v' \in V$ with communication latency below the microservices limit are considered, $l_{v,v'} \leq \lambda_{\mu_{a,1},\mu_{a,2}}$. If multiple nodes meet the criteria, the second microservice is placed in the optimal one, which could be identical to the first microservice's node. This process is repeated until the I_a-th microservice of the application is served. If a suitable node to host an application's microservice cannot be found, the procedure is re-initiated for the same application considering the second-best node for the first microservice and so forth. Once a solution is found, the utilization of the resources is updated and the application is marked as served. The above process is repeated for all applications, returning the final assignment and the value of the objective function (Eq. 1).

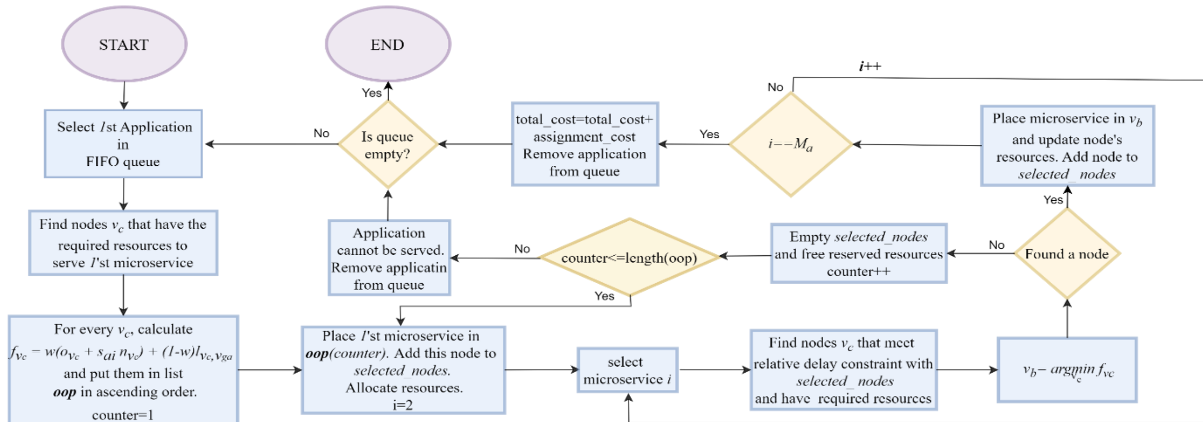

Figure 16: Flowchart of the GRAA heuristic.

From the description of the aforementioned procedure, it is possible that the selection of the first node can result in an infeasible solution due to the latency constraints among the application's microservices. Although this may occur for edge resources which are characterized by limited capacity, this not the case for the abundant cloud resources, which can handle application demands at the price of increased propagation latency.

The complexity of this approach is polynomial, with a worst-case execution time of $O(|A| \cdot |I_a| \cdot |V|)$, assuming all the nodes $|V|$ are candidate locations to serve the first microservice of each application. Figure 16 illustrates a typical iteration of GRAA.

5.1.3.2 Multi-agent Rollout

To further improve the performance of GRAA, we developed a multi-agent Rollout mechanism. Rollout [14],[15], one among the most recognized reinforcement leaning techniques, aims to provide a close to optimal solution by leveraging a base policy (like GRAA). It is an iterative process that takes each time as input an instance of the resource assignment problem (concerning applications with microservices) along with a partial solution (some microservices assigned to nodes) and constructs the complete solution step-by-step. This technique becomes particularly useful when the exact methods are too slow and/or when solutions provided by heuristics are inefficient.

Assuming that the first (a-1) applications have been served and application a is up next, the multi-agent rollout heuristic gets as input a solution path $o = [o_1, \dots, o_{a-1}]$ of size $\sum_{k=1}^{a-1} o_k \cdot I_k$, where states o_k, for $k = 1, \dots, a-1$ contain the assignment of the microservices of application $k = 1, \dots., a-1$ to processing nodes. State o_a is then broken down into I_a stages each corresponding to the assignment of one of the I_a microservices of application a to processing nodes. Initially, a number of possible placements $P_{a,i}$ for each microservice $i = 1, \dots, I_a$ are calculated. Then, to determine the placement of a microservice i, one of the available placement options $p \in P_{a,i}$ is selected and the respective service cost is calculated based on the provided objective function. Meanwhile, the cost for the remaining microservices and applications is computed using the GRAA heuristic (base policy), resulting in a total cost σ_p. When all the possible placements P_i of microservice *i* have been evaluated, the one yielding the lowest cost σ_i is selected (Figure 17). The utilization of the node that serves the microservice is updated accordingly, the microservice is marked as served and the procedure continues with the following microservice. The placement of the microservice I_a of application a indicates the transition to state o_{a+1} and the same procedure is repeated until all the application demands *A* are served. Finally, the allocation of resources to nodes is returned along with the objective value of the performed assignment.

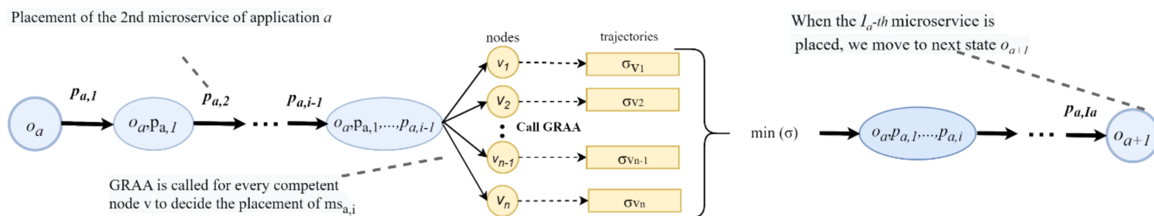

Figure 17: Multi-agent Rollout options for serving the *i-th* microservice of application *a*

Consider an application a consisting of I_a microservices. Each microservice can be placed (in the general scenario) in any node of the infrastructure, resulting in a state size of $|V|^{Ia}$ for the collective decision of the application's placement. When the allocation of resources of an application *a* is segmented into $|I_a|$ sequential decisions by agents, the state space is reduced

into $|V| \cdot |I_a|$ states. In this case, the control space complexity from the different options when serving the applications is traded off with state space complexity.

5.1.4 Performance evaluation

For our experiments, we considered two topologies for the hierarchical cloud-edge infrastructure, with different characteristics regarding the number of nodes at the different layers and their computing capacity (Table 4): a basic that consists of 19 nodes and an extended one with 53 nodes, with the cloud having enough capacity to serve the examined workloads. We assumed that both topologies are organized into a hierarchical infrastructure consisting of nodes (locations) belonging to three different layers: the near-edge, far-edge, and cloud.

The basic topology was used for performance comparison between our GRAA heuristic, the multi-agent Rollout, and the built-in optimal MILP solver of MATLAB. The execution times for the optimal solver became prohibitively large for larger configurations, hence using the basic topology. The extended topology considers the same node attributes, but their numbers are scaled to 40 near-edge nodes, 10 far-edge nodes, and 3 central cloud locations. The extended topology was used for the rest of the experiments to provide a closer-to-real-world scenario and demonstrate the scalability of the proposed algorithms.

Table 4: Characteristics of the computing nodes of the basic and extended topologies

	Near-Edge	Far-Edge	Cloud
Basic topology (#Nodes)	15	3	1
Extended topology (#Nodes)	40	10	3
c_v	[4, 8]	[80-120]	500
r_v	[4, 16]	[120-200]	1000
o_v	[2, 3]	[1,1.5]	[0.3,0.7]
n_v	0.1	0.25	0.5

With regards to the workload, the demands were generated randomly at the near-edge nodes, with the number of microservices per application drawn from a uniform distribution in the close interval [1,5]. The workload size of each application was also randomly selected in the interval [1,5], measured in normalized size units. Dependencies between pairs of microservices were created randomly with probability equal to 0.3, while the latency constraint among them varies in the close interval [0.5,3.5] latency units. The processing and memory requirements of each microservice are drawn from the uniform distribution in the close interval [1, 4] and [1, 8] respectively.

The proposed mechanisms were developed in MATLAB and the experiments were conducted on a 6 core 2.6 GHz Intel Core i7 PC with 12 GB of RAM.

Initially, we benchmarked the performance of the multi-agent rollout and the greedy heuristic against the optimal solution provided by the MILP in means of execution time and optimality. This was done for randomly selected application demands (ranging from 50 to 300) and for

weighting coefficient 0.01. This coefficient corresponds to the latency optimization problem, while still considering a minimal cost factor.

Table 5: The total cost and the execution time for w=0.01 for the different mechanisms

Application demands	MILP		Multi-agent Rollout		GRAA	
	Obj. value	Exec. Time (sec)	Obj. value	Exec. Time (sec)	Obj. value	Exec. Time (sec)
50	55.92	92.37	56.31	17.3	56.84	0.12
100	115.15	507.42	116.17	69.42	117.90	0.22
150	228.51	2453.16	236.49	147.72	252.38	0.35
200	409.94	10000	421.6	252.09	438.62	0.47
250	657.8	10000	675.42	349.74	702.37	0.67
300	-	10000	1051.8	348.82	1079.2	0.89

Regarding the performance of the proposed mechanisms, GRAA exhibited the worst performance, with a gap up to 10% from the optimal solution, whereas the Multi-agent Rollout managed to generate solutions within 3.5% of the optimal in all cases. In terms of execution time, GRAA exposed the shortest, in the order of milliseconds, even for higher workloads. Rollouts execution time while grew polynomially with the workload increment. Finally, the MILP solver showcased exponentially increasing execution times, while it was unable to produce a feasible solution within the set period for the largest workload.

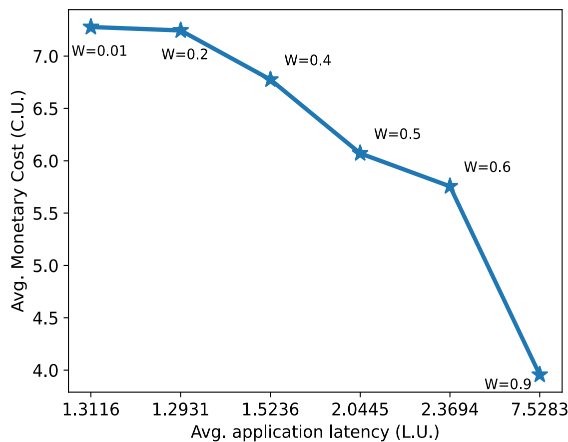

Figure 18: The pareto efficiency chart

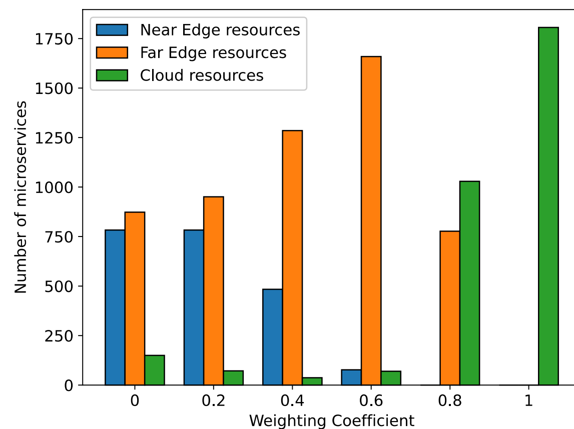

Figure 19: The number of microservices allocated at the various layers

Figure 18 portrays the allocative efficiency chart for the two objectives that are taken into consideration, namely the monetary cost for the application execution and the average latency per application for the different weighting co-efficients used in the objective function. As anticipated, the lowest cost is attained when cloud resources are highly utilized and thus the propagation latency increases as cloud resources are located in a few distant locations to which the data are transferred. Conversely, when the single optimization criterion is the minimization of latency, the propagation delay is reduced by 70% compared to the previous case, while the monetary cost is increased by almost 75%.

Next, we examined the utilization of edge and cloud resources for serving 600 cloud native application demands for the different weighting coefficients *w* (Figure 19). Edge resources are utilized more in small weight values, as the objective is approaching the delay minimization and edge layers consist of nodes in geographic proximity to the data-source. In this case the microservices of an application expand over the resources of the edge layer. On the other hand, cloud resources are heavily utilized in high *w* values, as the objective gravitates towards monetary cost minimization, thereby favoring the "cheap" cloud nodes. For intermediate *w* values, applications microservices are distributed over the edge-cloud continuum. This showcases the importance of edge resources in the minimization of the applications latency for time critical operations.

Finally, we examined the contribution of networking and operational cost for the different weighting co-efficient values (Figure 20). When the objective function targets the minimization of the monetary cost, the cloud resources are preferred with the operational and networking cost contributing almost equally to the total cost, as the processing cost is low while the networking cost increases for the transferring the application data to the cloud. On the other hand, when the objective is the minimization of latency and edge resources are utilized, the processing cost of the edge resources is the main factor of the total monetary cost, while networking costs constitute only 12% of the overall cost.

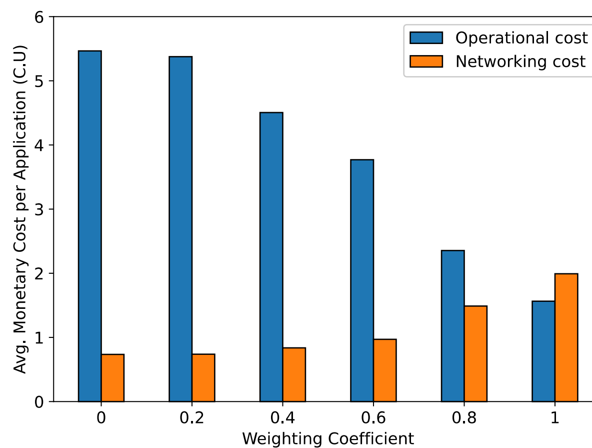

Figure 20: Operational and networking cost for the different objective co-efficients

5.1.5 Conclusions

In this study, we addressed the problem of resource allocation in multi-layered edge-cloud infrastructures for optimally serving cloud-native applications. We considered multiple important yet often overlooked parameters, such as the delay constraints posed by the dependencies among microservices. GRAA was developed to provide a sub-optimal solution that is further optimized by the Rollout technique. We demonstrated the trade-off between delay and monetary cost of service and proved the efficiency of the Rollout technique, which provided a significant improvement in the GRAA's solution, while also maintaining a tolerant computational time.

5.2 Security-aware Resource Allocation in the SERRANO Edge-Cloud Continuum

Allocating resources in a distributed multi-tenant infrastructure poses challenges for a centralized orchestrator. In order to address these challenges, hierarchical orchestrator architectures are employed to enable a more efficient resource allocation. SERRANO Resource Orchestrator follows a declarative approach, instead of an imperative one, for describing the workload requirements to the Local Orchestrator. This provides several degrees of freedom to the Local Orchestrator for serving in an optimal manner the "request", satisfying both the central orchestrator and the resource's objectives. Then, the control is passed to Local Orchestrators that are responsible for the actual deployment based on the desired performance requirements.

In this work, we assume varying levels of workload isolation achievable through lightweight virtualization mechanisms, establishing distinct tiers of security and trustworthiness, each with its own quantified computational and storage requirements. We model the respective resource allocation problem, i.e., of provisioning edge-cloud continuum resources for cloud-native applications subject to applications' performance and security requirements, as a Mixed Integer Linear Program. Additionally, a best-fit heuristic is introduced to reduce the execution time for real-size scenarios, leveraging clustering algorithms to perform a fast assignment of applications to resources while maintaining a tolerable optimality gap. Finally, a Multi-agent Reinforcement Learning based mechanism is also proposed to trade off execution time of the proposed heuristic with performance. Through extensive simulation experiments, we demonstrate the merits of our proposed mechanisms and explore the several trade-offs that emerge from conflicting objectives.

5.2.1 Related work

To enable the secure execution of cloud-native applications, frameworks are introduced that support container execution in a sandboxed environment based on micro-VMs. Recent works also recommend unikernels [18][19] that have minimal memory/system footprint, achieve high performance, and provide strong isolation equivalent to that of virtual machines. These trends give rise to several fundamental challenges related to application deployment, the support of heterogeneous infrastructures, and the provided security. The authors in [20] focus on the challenges and requirements for building a scalable and trustworthy multi-tenant AIoT (Artificial Intelligence of Things) cloud-native platform. They first identify several key challenges, including security, privacy, and trust and highlight how these challenges differ in a multi-tenant edge environment compared to a central cloud. They also present the state-of-the-art methods for addressing these challenges and describe open research areas.

In [21], the authors propose a security-aware dynamic scheduling approach for cloud-based industrial applications in a two-tier infrastructure. They introduce a three-level security model corresponding to public, semi-public, and private data. Then, a distributed Particle Swarm Optimization heuristic is developed to perform resource allocation and a dynamic scheduling

mechanism for real-time optimization. The authors in [22] propose a security-aware offloading model for a multi-user environment. A new security layer is introduced utilizing the AES cryptographic algorithm to prevent attacks such as sniffing, jamming and eavesdropping. The resource allocation problem is formulated with the optimization objective of minimizing the latency and energy overhead of mobile users, leveraging a Deep Reinforcement learning algorithm. The work in [23] presents a security-aware task offloading method for maximizing the total profit of edge nodes in an Edge-Cloud computing (ECC) environment. A security model is constructed, which utilizes several confidentialities (IDEA, DES, AES etc.) and integrity (Tiger, SHA1, MD5 etc.) services for coping with security threats. A genetic algorithm is developed to solve the resource-allocation problem.

Indeed, the dependencies among an application's microservices, typically manifesting in the form of information exchange or service chains, are frequently overlooked. Guaranteeing seamless communication among interdependent components is of paramount importance when dealing with geographically dispersed infrastructures. In our model, we represent these dependencies as communication delay requirements. In addition, application isolation mechanisms should be considered, such as virtualization and containerization techniques, where applications are executed in sandboxes [24], or even unikernels. Coupled with hardware extensions [25], these mechanisms can provide increased security for multi-tenant execution. These requirements of applications and resources across the edge-cloud infrastructure introduce, from an algorithmic perspective, a high number of constraints that need to be addressed simultaneously considering different optimization criteria.

5.2.2 Infrastructure description

We focus on a multi-layer edge-cloud infrastructure (Figure 21), encompassing computing and storage resources across various layers. The considered infrastructure comprises devices positioned at different locations, spanning from "near-edge" (i.e., from on-premises to tens of kilometres) to "far-edge" devices (i.e., some hundreds of kilometres) and cloud datacentres (i.e., typically several thousand kilometres away, situated in various geographic regions worldwide).

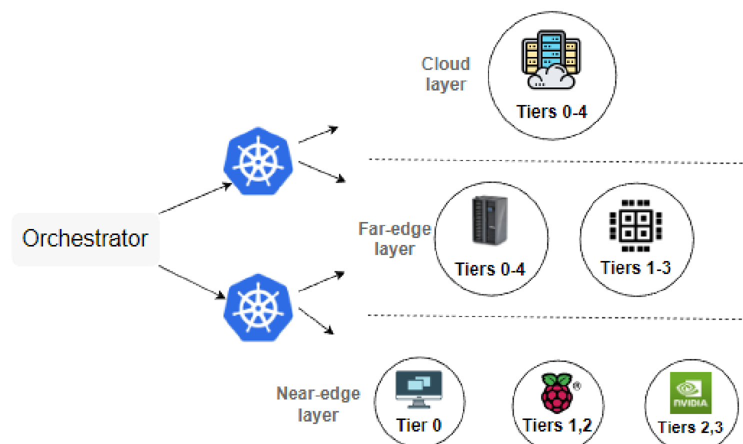

Figure 21: Heterogenous resources across the edge-cloud continuum.

The management of the infrastructure and the service of the applications is performed through a hierarchical two-level system. The high-level orchestrator assigns application requests to local orchestrators, each controlling a subset of infrastructure nodes. This provides several degrees of freedom to the Local Orchestrator to serve in a highly efficient manner the deployment request, satisfying both the central orchestrator and the resource's objectives, with a minimal decision-making timeframe.

The resources can vary both in size and capabilities, with common examples including micro-datacentres, modular datacentres in shipping containers, specialized computing devices (e.g., FPGA, GPU), and IoT devices (e.g., Raspberry Pi, NVIDIA Jetson). These can be deployed on providers' premises (e.g., the Central Office - CO), or on other large and small premises (e.g., stadiums, malls, businesses, houses). Special hardware can enable trusted execution. Various networking mechanisms using wired (optical) and wireless (e.g., 5G) technologies provide the required interconnection of the individual edge and cloud layers. These multi-domain and multi-technology network paths are typically controlled and managed by multiple telco operators. In this work, we abstract the communication paths between the resources in the same or different layers as virtual links with specific latency. These values depend on the networking locality of the resources, with those nearby resulting in lower latency than those far apart. Hence, the propagation delay increases in accordance with the physical distance of the data generation point.

To ensure secure application execution, the infrastructure leverages advanced software mechanisms and, in some cases, peripheral hardware. In this way it facilitates varying workload isolation levels and trusted execution across layers, even amid untrusted physical nodes typical of edge devices. Figure 22 illustrates the diverse levels of workload isolation that can be achieved using the novel mechanisms that are also developed in the context of the SERRANO project. For clarity, we provide an overview of the supported SERRANO security tiers. More technical details are available in the deliverables D3.3 (M15) and D3.4 (M30).

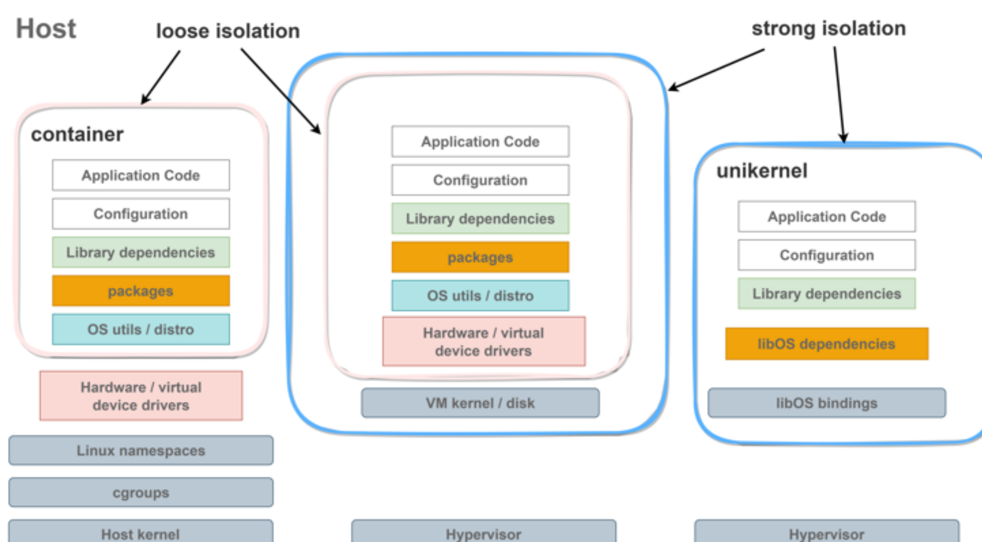

Figure 22: Different levels of workload isolation

Tier-0 represents generic containers. Tier-1 embodies microVM sandboxing [18],[26], where applications run atop a microVM. This requires booting a complete virtualization stack (including VMM, kernel, rootfs), which remains active until the application's termination. Despite progress in minimizing the overhead of VMMs regarding CPU and memory footprint, even the latest VMMs display a 30% overhead in memory management and address translation, plus additional CPU usage for handling I/O and context/mode switches. This also considers the extra memory used by the VMM and the necessity for a full OS system (the micro-VM) to be active for container spawning. Storage overhead is proportional to the application. However, a microVM can support container execution with a minimum rootfs, typically in the tens of MBs, with typical applications in the hundreds of MBs.

Tier-2 is defined by unikernel execution, where CPU, memory, and storage footprints are minimized as the application is compiled as a machine image, thereby eliminating unnecessary OS and library software stacks. According to [25], this results in at least a 20% reduction in CPU and memory overhead, while the application binary footprint decreases by at least 60%. This reduction is achieved by excluding the OS/libraries from the application, apart from the optimized build. Tier-3 and Tier-4 are similar to Tier-0 and Tier-1, respectively, but with enhanced security provided by secure boot. In these cases, a simple peripheral hardware (known as Trusted Platform Module) is required to provide hardware-based, security-related functions. Additionally, trusted execution in Tier 4 requires the use of an attestation mechanism in the hypervisor layer.

Table 6: Multipliers of the computing and storage requirements for the different security and trustworthiness tiers

Multipliers	Tier 0	Tier 1	Tier 2	Tier 3	Tier 4
CPU	1	1.3	0.8	1	1.3
RAM	1	1.3	0.8	1	1.3
Storage	1	1.1	0.4	1	1.1

Each tier imposes distinct demands on computing and storage resources, which we have quantified in Table 6. The presented values are normalized with respect to the generic workload requirements of Tier 0. Hence, the value of 1.3 of the CPU overhead for Tier 1 indicates that Tier 1 execution requires 30% more processing resources than Tier 0, whereas Tier 2 requires 20% less. Hence, when deploying a cloud-native application, it is essential to provide: (i) the computing and storage requirements for each microservice, (ii) specify the maximum delay between them for optimal execution in the infrastructure and additionally, (iii) the minimum level of security and isolation for each microservice to ensure the application's secure and efficient operation.

5.2.3 Problem formulation

We assume a hierarchical edge-cloud infrastructure that is denoted by a Complete Undirected Weighted Graph $G = (V, E)$. The set of nodes V corresponds to distinct geographical areas where a set M_v of computing resources are available, as well as the locations where workloads are generated (which may or may not be capable of local processing). A fixed communication

(propagation) latency $l_{v,v'}$ is introduced among different nodes $v, v' \in V$. This latency takes into consideration the nodes' propagation delay, as well as additional delays incurred within the nodes during the communication process. Machines M_v (virtual and/or physical) are deployed on the different nodes $v \in V$ and are controlled by low-level orchestrators O. Each low-level orchestrator $o \in O$ controls a subset of nodes $V_o \subseteq V$ and therefore controls $M_o = \bigcup_{v \in V_o} M_v$ machines, with two orchestrators controlling distinct set of resources ($V_o \cap V_{o'} = \emptyset$, for $o, o' \in O$).

The machines serve the workloads at different security tiers $S = \{0, 1, 2, 3, 4\}$, where integers from 0 to 4 are used to represent the different workload isolation levels. Also, a subset of the machines $\mathrm{M}_{\mathrm{v_s}} \subseteq M_v$ are equipped with hardware peripherals (secure boot) to support the execution of tier 3 and 4 workloads $S' = \{3,4\}$. Each machine m is described by the tuple $\tau_m = [c_m, r_m, h_m, s_m, p_m]$, where c_m is the CPU capacity of the machine measured in CPU units, r_m is the RAM capacity of the machine measured in RAM units, h_m is the storage capacity of the machine measured in GB's, s_m indicates the existence of secure boot (value 1) or not (value 0) and p_m is the operational cost of the machine that is the cost of use for a given period of time (time unit).

The workload in our scenario consists of a set A of cloud-native applications. Each application $a \in A$ is represented by an Undirected Weighted Graph $G^a = (V^a, E^a)$, where the nodes V^a denote the microservices that make up the application, and the edges E^a denote the existence of inter-dependencies among them. We adopted an undirected graph representation of the cloud-native applications, as we are concerned with the delay constraint formed by their communication dependency, which is assumed to be bi-directional in that case. The data of each application is generated at node g_a. Each microservice $i_a \in V^a$, has specific requirements described by the tuple $[\varepsilon_{a,i}, \rho_{a,i}, \omega_{a,i}, \sigma_{a,i}, \lambda_{a,i}]$, where $\varepsilon_{a,i}$ is the microservice's CPU demand, $\rho_{a,i}$ is its memory demand, $\omega_{a,i}$ is the storage demand, $\sigma_{a,i}$ is the minimum security tier requirement and $\lambda_{a,i}$ is the duration of microservice in time units. Note that the computing and storage resources are specified assuming Tier 0 execution. This eliminates the need for users to profile the requirements of their applications for the different security tiers. Hence, when deploying the microservices in a machine with respect to the specified security tier requirement, the CPU, RAM, and storage requirements of the microservices need to be considered based on the selected security tier and thus with the respective multipliers $\hat{\varepsilon}_\sigma, \hat{\rho}_\sigma, \hat{\omega}_\sigma$ (Table 6) to calculate the security-tier-specific computing and storage requirements.

Moreover, each link e_{i_a, i'_a} that connects two microservices $i_\alpha, i'_a \in V^a$, with $i \neq i'$ denotes a maximum acceptable latency requirement δ_{i_a, i'_a}; this implies that microservices i_α, i'_a can be assigned to machines m, m' and corresponding service nodes v, v' only if $\delta_{i_a, i'_a} \geq l_{v,v': m \in v, m' \in v'}$. This delay constraint measures the intensity of the dependency between them in the sense that highly dependent microservices should be placed on the same or geographically approximate nodes. Finally, each application $a \in A$ has a delay limit D_a, which is the maximum acceptable delay between any node that hosts any of the application's microservices and the source node where the application's demand is generated. This is a

general measure of the application's overall time-sensitivity, in a sense that a time-sensitive application requires all its microservices to be processed by nodes with low delay.

5.2.3.1 MILP formulation

Table 7: MILP variables

Variable	Interpretation
$x_{a,i,o}$	Binary variable equal to 1 if microservice $i = 1, \dots, I_a$ of application $a = 1, \dots, A$ is assigned to low level orchestrator $o = 1, \dots, O$
$y_{a,i,m,\sigma}$	Binary variable equal to 1 if microservice $i = 1, \dots, I_a$ of application $a = 1, \dots, A$ is placed at machine $m = 1, \dots, M_v$ and is served at security level $\sigma = 0, \dots, 4$
θ_a	Integer variable that denotes the latency of application $a = 1, \dots, A$
T_a	Integer variable that denotes the total monetary cost of serving the cloud native application $a = 1, \dots, A$
w_i	Weighting coefficients for $i = 1,2,3$ to control the contribution of operational cost and latency in the objective function with $\sum_{i=1}^{3} w_i = 1$

- Objective function. Minimize a weighted combination of the operational cost, communication delay and security tier.

$$\min w_1 \cdot \sum_{\alpha=1}^{A} T_a + w_2 \sum_{\alpha=1}^{A} \theta_\alpha + w_3 \sum_{\alpha=1}^{A} \sum_{i=1}^{I_a} (|S| - \sigma_{a,i} - 1)$$

Subject to the following constraints:

-C.1. Each microservice $i = 1, \dots, I_a$ of each application $a = 1, \dots A$ must be assigned to a low-level orchestrator.

$$\forall a \in A, \forall i \in I_a, \sum_{o=1}^{O} x_{a,i,o} = 1$$

-C.2.The microservices $i = 1, \dots., I_a$ of each application $a = 1, \dots A$ that are executed with security Tier $\sigma = 0, .., 4$ must be assigned to a machine of the selected orchestrator o.

$$\forall a \in A, \forall i \in I_a, \forall o \in O, \forall m \in M_o, \sum_{m=1}^{M_o} \sum_{\sigma=1}^{|S|} y_{i,a,m,\sigma} \geq x_{a,i,o}$$

-C.3. The microservices that are executed with security Tier 3 and 4 need to be placed at nodes with extra peripheral hardware.

$$\forall a \in A, \forall i \in I_a, \forall o \in O, \forall m \in M_{o,}, \forall s \in S', y_{i,a,m,\sigma} \leq s_m$$

-C.4. The total CPU required from all the microservices $i = 1, \ldots, I_a$ of application $a = 1, \ldots, A$ deployed at a machine m must not exceed its capacity.

$$\forall o \in O, \forall m \in M_o, \sum_{a=1}^{A} \sum_{i=1}^{I_a} (\varepsilon_{\iota,a} \cdot \hat{\varepsilon}_\sigma) \cdot y_{i,a,m,\sigma} \leq c_m$$

-C.5. The total RAM required from all the microservices $i = 1, \ldots, I_a$ of application $a = 1, \ldots, A$ deployed at a machine m must not exceed its capacity.

$$\forall o \in O, \forall m \in M_o, \sum_{a=1}^{A} \sum_{i=1}^{I_a} (\rho_{\iota,a} \cdot \hat{\rho}_\sigma) \cdot y_{i,a,m,\sigma} \leq r_m$$

-C.6. The total Storage required from all the microservices $i = 1, \ldots, I_a$ of application $a = 1, \ldots, A$ deployed at a machine m must not exceed its capacity.

$$\forall o \in O, \forall m \in M_o, \sum_{a=1}^{A} \sum_{i=1}^{I_a} (\omega_{\iota,a} \cdot \widehat{\omega}_\sigma) \cdot y_{i,a,m,\sigma} \leq h_m$$

-C.7. The trusted execution tier of a machine that is assigned a microservice must be equal or greater than the tier demanded by the microservice.

$$\forall a \in A, \forall i \in I_a, \forall m \in M_{o,} \forall o \in O, \forall \sigma \in S$$

$$y_{i,a,m,\sigma} \cdot \sigma \geq \sigma_{a,i}$$

-C.8,9. The microservices $i = 1, \ldots, I_a$ of application $a = 1, \ldots, A$ must be assigned to a machine that is situated in a node v that respects the application's delay limit.

$$\forall a \in A, \forall i \in I_a, \forall m \in M_o, \forall o \in O,$$

$$l_{m,g_a} \cdot y_{i,a,m,o} \leq \theta_a, \theta_a \leq D_a$$

-C.10. For each pair of connected microservices i, i' of an application $a = 1, \ldots, A$, the selected machines must respect the dependent microservices delay limit.

$$l_{v,v':m \in v,m' \in v'} \cdot y_{i,a,m,o} + l_{v,v':m \in v,m' \in v'} \cdot y_{i',a,m\prime,o} \leq \delta_{a,i,a,i'} + l_{v,v':m \in v,m' \in v'}$$

-C.11 Monetary cost calculation for application $a = 1, \ldots, A$

$$\forall a \in A, T_a = \sum_{i=1}^{I_a} \sum_{o=1}^{O} \sum_{m=1}^{M_o} y_{i,a,m,o} \cdot p_m \cdot \lambda_{a,i}$$

5.2.3.2 Best-fit heuristic

The presented MILP approach is computationally intensive and exhibits a prohibitively large execution time, even for medium-sized problems. To address this, we developed sub-optimal mechanisms. The first mechanism is a greedy best-fit heuristic. It takes as input the infrastructure graph G and application demands A and allocates resources sequentially for the cloud-native applications concerning computing and storage capacity, security, and latency constraints while simultaneously optimizing the set objective function. To do so, it examines each microservice independently and allocates resources in a best-fit manner according to the

specified objective function. When it fails to serve a microservice due to either communication latency or computing or storage capacity constraints, it backtracks and re-allocates resources for the problematic microservices.

The algorithm begins by ordering the cloud-native application demands based on their application delay limit D_a. As applications consist of dependent microservices, the pairs of microservices are also ordered based on their latency requirements (in latency units l.u.) from the strictest to the loosest. This way, the algorithm prioritizes applications and microservices with stricter latency requirements to maximize the chances of meeting the requirements while decreasing any reallocations due to backtracking.

The allocation of resources for cloud-native applications is performed sequentially. Given a microservice of an application a, the algorithm identifies the candidate orchestrators to serve it. These orchestrators are selected based on their ability to meet the application's latency requirement D_a and their machines' ability to fulfil communication constraints with already assigned microservices. Afterward, the selected orchestrators are sorted in ascending order based on their objective value, which is the weighted average of their machines' cost, security, and latency towards the data generation node. The orchestrators are examined sequentially, beginning with the one offering the best objective value.

If an application contains only one microservice, the algorithm selects the top-ranked orchestrator and subsequently identifies candidate machines. These machines possess the required computing and storage resources and an equal or higher trusted execution tier than the one demanded by the microservice. Additionally, these machines must be situated in nodes that satisfy the application's delay requirement. The algorithm then assigns the microservice to the candidate machine that yields the best objective value. If the application contains multiple microservices, the above process applies for the first microservice. However, for every subsequent microservice, the identification of the candidate machines also considers the latency requirements among interconnected microservices.

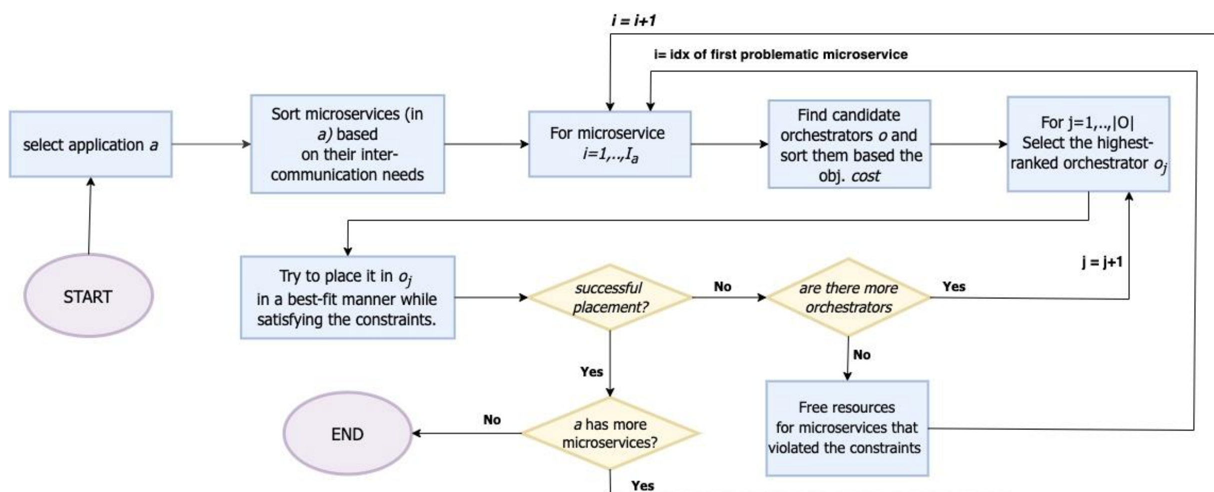

Figure 23: Flowchart of the greedy best fit heuristic

If no feasible placement is found for a microservice due to communication, resource, or security constraints, the affected inter-connected microservices that have already been served are de-allocated, freeing up the occupied resources. The algorithm will then attempt to re-embed the impacted microservices (possibly in a new orchestrator) as well as the one examined in this step until a feasible solution is found. This process is repeated until all microservices within an application are served, at which point the algorithm proceeds by selecting the next application in line. The algorithm terminates once all applications have been served. Figure 23 illustrates a typical iteration of the algorithm.

5.2.3.3 Multi-agent Rollout Heuristic

We also developed a multi-agent rollout [14][15] mechanism to enhance the performance of the greedy best-fit heuristic and trade-off execution time with performance. Rollout is a well-known reinforcement learning technique that provides a near-optimal solution by leveraging a base policy, which in this case is the greedy best-fit heuristic of the previous sub-section. It follows an iterative process that takes, at each step, an instance of the problem along with a partial solution and constructs the final solution incrementally.

After selecting an application, the multi-agent rollout algorithm assigns an agent to each application's microservice. These agents co-operate/compete with time in order to fulfil the assigned applications requirements based on the set objective function. Each agent acts sequentially by examining all possible placements across the different orchestrators and their nodes. As the search space can be large, nodes that do not include machines that fulfil the following requirements are pruned: (i) the minimum latency requirements of the already-served communicating microservices of the applications, (ii) the CPU, RAM, and storage capacity, (iii) the minimum trusted execution requirements and (iv) the application latency constraint D_a. Furthermore, for each node, if more than one machines meet the problem's constraints, only the placement in the one that yields the best objective is evaluated. This way, each agent, in the worst case, evaluates at most $\sum_o v_o$ possible placements for a microservice (instead of $\sum_o M_o$).

The best-fit heuristic discussed in the previous sub-section is utilized to approximate the cost of the remaining microservices of the examined application and the microservices of the other applications. The process is repeated for all potential placements of the given microservice, and the one that exhibits the lowest cost, including the cost of the allocation of the remaining microservices that is provided by the greedy heuristic algorithm, is selected. The allocation for the current microservice is marked as completed and resources are updated for the machines of the selected nodes. This marks the transition to the next state, the next microservice of the current application a. When all the application microservices are served, the application is marked as served and the aforementioned process is repeated for the next application.

The purpose of using the multi-agent version of Rollout is to reduce the state space of the problem. The state space is reduced by breaking down the allocation of resources for an application a and a microservice $i \in I_a$ taking sequential decisions and applying one-agent-at-a-time instead of all-agents-at-once. By pruning the example nodes and evaluating only one machine per node, as explained earlier, the state space is further reduced. In this way, the

control space complexity stemming from the various options for serving the applications is traded off with state space complexity, and the computational requirements are proportional to the number of microservices I_a of the different applications a and the number of nodes within the different orchestrators.

5.2.4 Performance evaluation

5.2.4.1 Experimental setup

We performed several simulation experiments to examine the performance of the proposed mechanisms. The mechanisms were developed in MATLAB, and the experiments were conducted on a 6 core 2.6 GHz Intel Core i7 PC with 12 GB of RAM. We assumed a hierarchical infrastructure that spans the edge-cloud continuum and is split into three layers that correspond to near edge, far edge, and cloud nodes. We introduced two different topologies, namely “basic” and “extended”, each consisting of nodes with computing machines of distinct characteristics and capacities, as summarized in Table 8. Note that values exhibited in the close interval $[a, b]$ are sampled from the uniform distribution over that range.

Table 8: Characteristics of the computing nodes of the different topologies

	Near-edge	Far-edge	Cloud
Nodes (basic)	25	4	1
Nodes (extended)	40	7	2
Machines per node (basic)	1	[7,10]	50
Machines per node (extended)	2	[10,15]	100
CPU (CPU units)	[4,8]	[5,10]	[8,12]
RAM (RAM units)	[1,4]	[2,8]	[4,16]
STORAGE (GB units)	[4,16]	[8,32]	[16,64]
Monetary COST (Cost Units)	[6,7]	[3,4]	[1.5,2]

In both topologies, the near edge layer comprises many nodes with few low-capacity computing systems placed close to the data sources. Conversely, the cloud layer comprises a limited number of nodes that host an abundance of high-powered machines. The cost of the near-edge nodes was taken to be around 4 times higher than the central cloud.

Table 9: Cloud-native applications’ workload characteristics

Number of microservices	[1,7]
Delay constraint	[2,10]
Microservices’ CPU demand	[1,2]
Microservices’ RAM demand	[0.5,1]
Microservices’ storage demand	[1,5]
Dependency chance for a pair of microservices	25%
Dependency delay constraint	[0.5,3.5]

Regarding the communication delay between infrastructure nodes, we assumed that near-edge resources require between [0.5, 1.5] l.u., far-edge resources [3, 4] l.u., and cloud resources [7, 8] l.u. from the data generation points. Although the exact values are not standardized, we used [8] as a guideline.

For the workload, we focused on two scenarios: (i) a small and (ii) a medium-sized consisting of cloud-native applications of a maximum of 7 microservices (Table 9). Note that an application with a single microservice can represent a generic end-user demand, while microservice replicas are considered as microservices with identical resource profiles. We set the dependency probability between any pair of microservices to 25% and the respective delay constraint to range between 0.5 and 3.5 l. u.

5.2.4.2 Experimental results

Initially, we compared the performance of the proposed sub-optimal mechanisms, the greedy heuristic, and the multi-agent rollout with respect to the optimal solution provided by the MILP mechanism. For the evaluation we considered the following optimization criteria: (i) minimization of the operational cost ($w_1 = 1$), (ii) minimization of the applications latency ($w_2 = 1$), (iii) maximization of trusted execution ($w_3 = 1$), and (iv) all optimization criteria ($w_1 = w_2 = 0.4, w_3 = 0.2$). We used the "small" topology described in Table 9 and a small workload of 50 applications. The execution time for the optimal solver was limited to 60 minutes, and the presented results are averaged over 20 simulations. The results of the simulation experiments are illustrated in Figure 24.

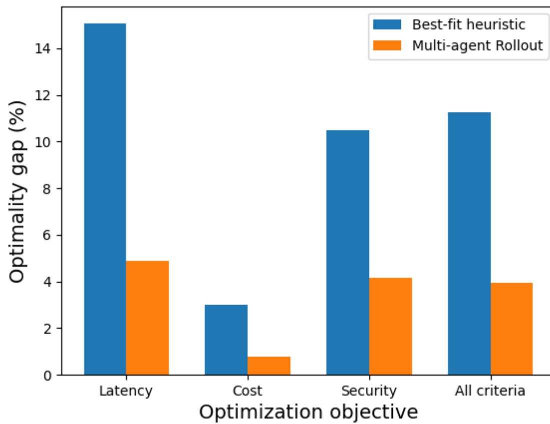

Figure 24: Optimality gap for the different optimization criteria

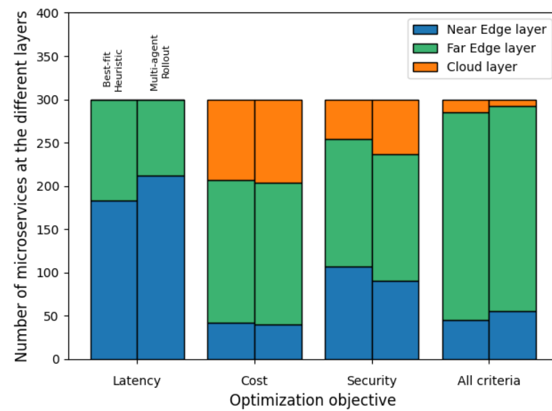

Figure 25: Allocation of microservices at the different layers of the edge-cloud continuum

The heuristic has an optimality gap of 14.5% when the main optimization criterion is latency minimization. This happens due to the high competition for the limited near-edge resources, which requires a more sophisticated resource allocation approach to allocate these resources effectively. For the same reason, the "all-optimization criteria" and "trusted execution" lag by 11% and 10% from optimal, respectively. However, when the optimization criterion minimizes the operational cost, the search space is much smaller; thus, the heuristic's performance is close to optimal, underperforming only by about 3%.

On the other hand, the Multi-agent Rollout exhibited significantly better performance, with the worst case being the latency optimization. However, it substantially improved the greedy heuristic solution due to the consideration of future placements, providing a 4.5% optimality

gap. Additionally, in the case of monetary cost minimization, the optimality gap provided was smaller than 1%, indicating that Rollout found an almost optimal assignment.

As for the execution time, the best-fit heuristic provided an almost instantaneous assignment, with an average time of 0.01 seconds per application placement. On the other hand, the Rollout algorithm performed much slower, at an average of 0.9 seconds per application, with a standard deviation of 0.3 seconds. Finally, the optimal solver exceeded the 3600 time limit in all cases, resulting in an average of 72 seconds per application.

Next, the multi-agent rollout mechanism was evaluated for the extended topology with 300 microservices and was compared to the best-fit heuristic, which is the baseline scenario for this set of experiments. We began by analysing the allocation of microservices for the different mechanisms and optimization criteria across the edge-cloud continuum (Figure 25). The experiments showed that resource allocation patterns varied based on the optimization objective. When cost or the trusted execution was prioritized, cloud resources were favoured due to their high capacity and the higher availability of trusted execution tiers. Conversely, when latency minimization was the main objective, near and far edge resources were heavily utilized. Additionally, when all the optimization criteria were simultaneously optimized, the solution proved beneficial in allocating resources tailored to the application's specific needs. This highlights the advantages of considering all the optimization criteria in a multi-objective optimization approach during the resource allocation process and the ability of the rollout mechanism to achieve an improved allocation of resources by leveraging the decisions of the heuristic in a reinforcement learning manner.

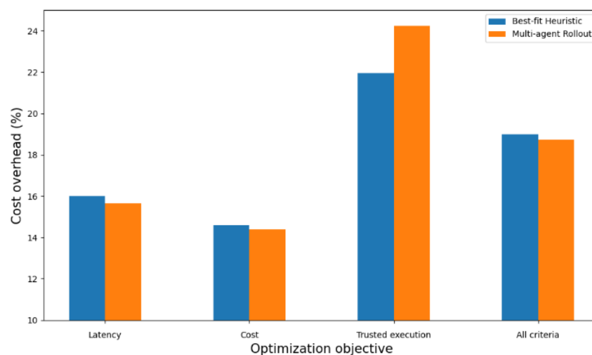

Figure 26: Operational cost overhead for the different optimization criteria

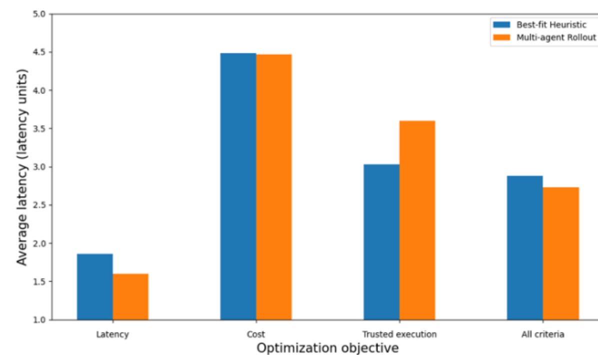

Figure 27: Experienced latency for the different optimization criteria

In Figure 26, we present the results of experiments regarding the average cost overhead associated with security as an additional constraint, compared to generic workload demands, which acts as the baseline scenario for this case for the different optimization objectives. The cost overhead for a microservice's placement is determined as the percentile increase in cost between its deployment in a default container, (Tier-0), and the deployment method chosen in the assignment.

As anticipated, the highest cost overhead is incurred in the maximization of trusted execution, where machines with higher tiers, which are more expensive, are favoured. Trusted execution is also considered in the “all optimization criteria” scenario, therefore producing increased

cost overhead. Conversely, the effect of additional security on cost overhead is lower for the other two optimization objectives, where trusted execution is not considered.

Finally, in Figure 27, we present the effect of the different optimization criteria on the average propagation latency. As expected, the lowest average latency is achieved when the optimization criterion minimizes latency. Conversely, when the operational cost is optimized, as many microservices as possible are placed on the "cheaper" far-edge and cloud layers, increasing latency. In the trusted execution optimization case, microservices are placed on machines with high-security tiers generally located in the upper layers (extra hardware), leading to higher average propagation latency. The weighted optimization approach trades off the requirements of different criteria and achieves a relatively smaller latency.

By comparing the Multi-agent Rollout with the greedy heuristic mechanism performance, we observe that the Rollout approach generally results in a marginally lower cost overhead and communication latency for all objectives except for trusted execution. As the Rollout mechanism produces an enhanced solution, when the cost contributes to the objective, Rollout discovers the most cost-efficient machines that usually possess lower security levels and subsequently lower security cost overhead. Similarly, with latency as an objective, Rollout manages to place more microservices on edge nodes and machines with adequate security tiers, thus lowering overhead costs. Similarly, for the trusted execution objective, the rollout mechanism improves the solution by placing more microservices on machines with a higher security tier, leading to a higher security cost overhead. These findings highlight the importance of considering multiple optimization criteria when allocating resources for cloud-native applications across the continuum. While prioritizing a single objective may lead to optimal results for that specific objective, it may negatively impact other criteria, such as latency or operational cost. Therefore, a comprehensive approach that balances multiple objectives can lead to a more efficient allocation of resources, resulting in improved application performance, reduced costs, and better resource utilization.

5.2.5 Conclusions

In this study, we aimed to address the challenge of allocating resources to cloud-native applications within a hierarchical edge-cloud infrastructure. Our approach considered critical factors such as the inter-dependencies among microservices and the trusted execution requirements of cloud-native applications. To meet microservices' varied security and isolation demands, we considered SERRANO's innovative technologies, such as sandboxing and unikernels. To model the resource allocation problem, we formulated a multi-objective optimization problem that balances various conflicting objectives, such as minimizing operational costs and propagation latency from data generation points, while considering the workloads' security tier requirements. We developed optimal and sub-optimal mechanisms that efficiently trade-off performance for execution time, as demonstrated in our experiments.

Our results showed that the greedy best-fit heuristic fell short of optimal performance by an average of almost 11% for all optimization criteria. However, the multi-agent rollout

mechanism significantly improved the greedy heuristic's performance, achieving close to optimal levels at 3.7%. Furthermore, our experiments highlighted the trade-offs between delay, cost, and security. In conclusion, our study provides a novel approach to resource allocation in a hierarchical edge-cloud infrastructure, addressing crucial factors such as security, isolation, and inter-dependencies among microservices.

5.3 Intent-based Allocation of Cloud Computing Resources Using Q-Learning

Resource allocation is a critical operation regarding the efficient use of the infrastructures. The majority of the formulated resource allocation problems and respective mechanisms assume a model where workload requirements are provided with certainty (e.g., from a user), while orchestration mechanisms have a clear view of the resources' characteristics and status.

In practice, however, these assumptions are not always valid. Users often have a subjective notion of their needs (e.g., what one considers low or high cost) and an abstract view of the available infrastructures. As a result, they are not able to specify in a certain, numeric manner, their requirements or match them to an infrastructure's actual characteristics. Also, orchestration mechanisms cannot always monitor efficiently the resources due to their high number and the dynamicity of their status. In addition, since not all resources belong to the same providers, it is reasonable that some providers are not willing to share the same level of details regarding their resources.

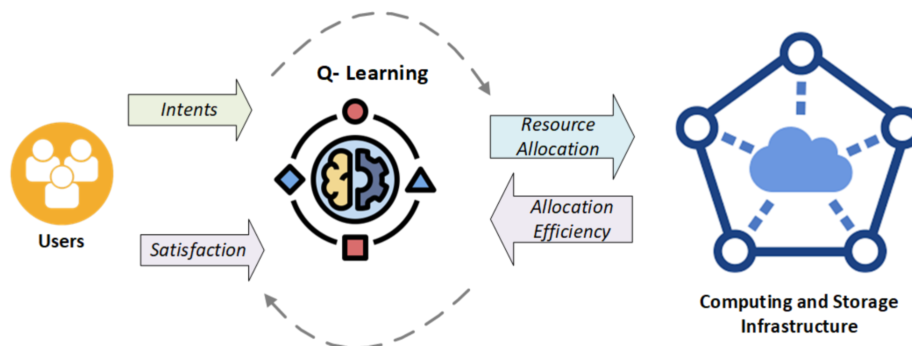

Figure 28: Intent-driven resource allocation

Recently, intent-based operations have been presented by various actors (providers, standardization organizations, academia) [28][29] as a way for applications and users to express their requirements regarding the use of Information and Communication Technology infrastructures, e.g., computing, networking, etc.

Overall, the goal is to focus on what one needs from an infrastructure instead of how to achieve it. In this context, our work considers intent-based resource allocation for cloud computing infrastructures (Figure 28). The idea is that application requirements are provided in an infrastructure-agnostic manner, assuming that application owners cannot provide the numeric requirements of their workload. The main contribution of our work is a Q-learning based Reinforcement Learning (RL) methodology that translates the users'/applications' intentions to efficient resource allocations.

5.3.1 Related work

Reinforcement Learning (RL) is a machine learning method that has recently gained a lot of attention from the research community. It is based on one or more agents that learn the environment of interest by interacting with it [31]. The RL agent gives recommendations and receives rewards from the environment. The ultimate goal of the agent is to maximize the total reward. If the reward is positive, the agent will continue to try its effort. If the reward is negative, the agent should change its policy to have a good value in the next step of the iteration.

In RL, there are problems that use models and are characterized as model-based and others that do not use a model of the environment and are called model-free. In the latter, agents learn to make decisions without having a model of the environment through trial and error. The most commonly used model-free RL methods include Q-learning, SARSA (State-Action-Reward-State-Action), Monte Carlo methods, TD-learning (Temporal Difference learning), Actor-Critic methods and Deep RL. These model-free RL methods are well-suited for problems where the environment is difficult to model; however, they may require more data for training and computational resources to learn an optimal state-to-action policy compared to model-based RL methods.

RL methods have been employed in various problems, including resource allocation: For optimal wireless resource allocation in order to avoid interference by hidden nodes in CSMA/CA method [32], in 5G services using deep Q-learning [33][34][35], in hybrid networks that contain access points, radio frequency and multiple visible light communications [36], in satellite-terrestrial networks [37] and in optical networks [38]. RL methods have also been used for resource allocation in edge and cloud computing. [34] proposes a joint task assignment and resource allocation approach in a multi-user WiFi-based mobile edge computing architecture. [39] proposed a Q-learning scheme to efficiently allocate edge-cloud resources for IoT applications. In [40] a computation offloading methodology for deep neural networks in edge-cloud environments is formulated. [41] use a Deep Reinforcement Learning-based approach to balance, in an edge computing environment, workload from mobile devices, so as to decrease service time and reduce failed task rate. In [42] a model-free Deep Reinforcement Learning approach is also introduced, in order to orchestrate the resources at the network edge and minimize the operational cost at runtime.

Intent-driven operations have the goal to overcome the complexity of utilizing complex infrastructures, decoupling the users' intentions regarding "what" should be done, from the actual resource orchestration, which specifies "how" it is done. Intent-driven operations have initially focused on networks [43][44][45], but recently, their application in cloud and edge computing is also investigated [46][47][48][49]. Authors in [46] define rules that enable users to express service-layer requirements. The Label Management Service in [47] helps cloud administrators model their policy requirements.

In [48], a learning-based intent-aware task offloading framework for air-ground integrated vehicular edge computing is developed. [49] proposes a framework to translate cloud performance-related intents into specific cloud computation resource requirements. [50]

proposes a strategy that matches multi-attribute tasks to cloud resources. In [51], it is used an intent-based network system to automate the deployment of virtual network functions in a cloud-based infrastructure.

The work we present next differentiates from the state of the art by utilizing a Q-learning RL methodology to translate users' intentions to resource allocations in a cloud infrastructure. In this process, the provided rewards are based both on the users' feedback and the infrastructure's status.

5.3.2 System Model and Infrastructure-Agnostic Operations

5.3.2.1 Infrastructure

In our work, we assume a computing infrastructure composed of N interconnected resources (edge and cloud) with different characteristics in terms of:

- Capacity $C=\{c_1,c_2,...,c_N\}$. This can be expressed as the number of (virtual) CPUs in case of a computing resource or the number of GB in case of a storage resource.
- Cost of use $U=\{u_1,u_2,...,u_N\}$. This can be formulated in different ways either as a fixed price or as cost per quantity per time unit (e.g. GB per hour used).
- Security $E=\{e_1,e_2,...,e_N\}$. This may depend on particular security features that the respective resource employs

Other parameters of interest can also be considered.

We also assume these characteristics are discrete and selected from a set of possible values. This is reasonable to assume based on the cloud computing paradigm of virtualized instances. In particular, all public cloud providers offer various types of instances, comprising varying combinations of (virtual) CPU, memory, storage, and networking capacity and are optimized for different workloads, e.g., compute or memory intensive.

In this context, the considered infrastructure's virtualized resources have capacity, cost, and security capabilities with discrete values from the following sets:

- N_c levels of capacity: $S_C = \{TC_1, TC_2, \dots, TC_{N_C}\}$, and $c_i \in S_C$ where $1 \le i \le N$
- N_u levels of cost: $S_U = \{TU_1, TU_2, \dots, TU_{N_u}\}$, and $u_i \in S_U$ where $1 \le i \le N$
- N_e levels of security: $S_E = \{TE_1, TE_2, \dots, TE_{N_e}\}$ and $e_i \in S_E$ where $1 \le i \le N$

5.3.2.2 Workload

The application requests for computing workload or storage space are submitted to an orchestration entity that manages the infrastructure. The request is described by the static characteristics of the workload to be submitted, such as the requested computing capacity (e.g., in terms of virtual CPUs) or the size of the data to be stored (e.g., 2 GB). It also includes

infrastructure-agnostic parameters, such as regarding the preferable cost, security, and performance, in the form of intents. This intent can take various forms and shapes, e.g., by characterizing the need to execute a workload "fast" or to store data with "high" security or with "low" cost. In our work, we formulate this with a small number of what we call "intent levels" for the different types of parameters of interest:

- $\widehat{N}_c$ levels of capacity: $\hat{S}_C = \{\hat{T}C_1, \hat{T}C_2, \dots, \hat{T}C_{N_c}\}$ where $\widehat{N}_c \ll N_c$
- $\widehat{N}_u$ levels of cost: $\hat{S}_U = \{\hat{T}U_1, \hat{T}U_2, \dots, \hat{T}U_{N_u}\}$ where $\widehat{N}_u \ll N_u$
- $\widehat{N}e$ *levels of security:* $\widehat{S}_E = \{\hat{T}E_1, \hat{T}E_2, \dots, \hat{T}E_{N_e}\}$ *where* $\widehat{N}e \ll Ne$

So, the j submitted workload of user k, w_{jk}, can be described with the tuple $\{\widehat{T}C_{jk}, \widehat{T}U_{jk}, \widehat{T}E_{ik}\}$, where $\widehat{T}C_{jk} \in \hat{S}_c$, $\widehat{T}U_{jk} \in \hat{S}_U$, and $\widehat{T}E_{jk} \in \hat{S}_E$. The way these infrastructure-agnostic intent levels match to the different infrastructure-related resource levels (Section 5.3.2.1) is the key for the intent-based operations that we research on the present work.

5.3.2.3 Example of Infrastructure-Agnostic Operation

Based on the above, we describe the following example of an infrastructure-agnostic storage workload request *R* to be served by the infrastructure. The request's static parameters include the size of the data to be stored, e.g., measured in GB. Also, the request is accompanied with intents specifying that this should be served with "low" cost, "high" security: $R = \{\hat{u}, \hat{e}\}$ = {"low", "high"}. Assuming that we have $\hat{S}_U$ = 1, 2, 3 and $\hat{S}_E$ = 1, 2, 3 "intent levels" for cost and security and the intention "low" matches to the value 1, while the "high" to value 3, then these intents can also be expressed numerically with the tuple $R = \{\hat{u}, \hat{e}\}$ = 1, 3

The goal of the methodology we present next is to efficiently translate the provided intents to specific decisions regarding how the tasks will be served to match as closely as possible to the users' or applications' intentions. For example, we assume that our infrastructure has $N_u = 10$ different cost levels for a storage resource, in terms of euros per GB per month stored, e.g., $S_u = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. So, when a user has an intent for cost $\hat{e}$ = 1 this means that the methodology has to match this ("intent level") to an actual cost value ("resource level") from the set S_u. In general, we may expect that $\hat{e}$ = 1 ("intent level") of cost, matches to 10, 20, 30 or even 40 euros per GB per month ("resource level"). In practice, however, this "intent level" is user-specific and can match any available "resource levels" or even none.

5.3.3 Q-learning based Intent Translation

In our work, we are using Q-learning, model-free Reinforcement Learning (RL), approach to translate the infrastructure-agnostic intent of a user regarding submitted workload to infrastructure-aware parameters.

5.3.3.1 State, Action spaces and reward

The basic design principles used in our RL-based method and need to be defined, include the state space *S*, the action space *A* and the reward *r*. The RL process is executed in time steps *t*.

The **state of the system environment** $s \in S$ at time *t*, describes the current status of the cloud infrastructure in terms of the availability of the resources. For simplicity, we assume that a single task/workload fully utilizes a resource from the *N* available ones.

As a result, the environment can be represented through a tuple that shows the availability of the resources: $\mathrm{s} = \{\mathrm{o}_1, \dots, \mathrm{o}_\mathrm{N}\}$, where $\mathrm{o}_\mathrm{i} \in \{-1, 1\}$ indicates whether the respective resource i is utilized or not.

The **action space** A contains all possible actions that can be taken, defining the transfer rules between states. As the agent explores the environment, it experiments with different actions to learn which are most effective in achieving the goals set. In our work, we assume that at each step *t* we can either assign a new task of a user to an available resource or migrate an existing task to another available resource. As a result, $A_t = \{r_1, \dots, r_N\}$, where *N* are all the available resources. In practice, though, only some transitions/actions from one state to another are possible since we assume that at a single step only one new task can be served or one existing task can migrate to a different resource. For example, in an infrastructure with N=4 resources, from the state $s_1 = \{-1, -1, 1, 1\}$, indicating that the third and the fourth resources are utilized, an action is possible to the state $\mathrm{s}_2 = \{1, -1, -1, 1\}$, indicating that a task migrates from the third resource to the first one, while no action is possible to the state $\mathrm{s}_3 = \{1, 1, -1, -1\}$, since this requires multiple task migrations.

After the agent takes an action at state *s* at time *t*, it will receive a **reward *r***, which can be used to evaluate the action performed. In order to design the reward function, it is necessary to determine the objectives based on which a positive or a negative reward will be provided.

One important novelty of our work is that rewards depend not only on the infrastructure (the typical environment in most related works) but also on the user that submits the task. On the user side, the reward relates to the level of satisfaction for serving the submitted task according to (or close to) the user's intention. On the infrastructure side, we focus on the efficiency with which the infrastructure is actually utilized. These objectives are interrelated since failing to serve a task due to poor utilization of the available resources results in unsatisfied users of the provided services.

In practice, user satisfaction can be provided by the user through an immediate feedback mechanism (e.g., using a User Interface [28]), after the infrastructure serves a task request, while the infrastructure's utilization can be monitored through a respective system, such as the SERRANO telemetry framework

In our work, we consider the following reward function: $R_t = a \cdot sf + b \cdot ul$, where *sf* is the satisfaction level based on the action performed at time slot *t* and *ul* the utilization of the resources at time slot *t*.

The *a* and *b* weights balance user feedback and resource utilization objectives. Also, we quantitatively calculate the satisfaction level as the difference between the user's intents and the parameter (cost, security, etc.) levels of the resources to which these tasks have been assigned to. For example, let's assume that a user has submitted two tasks with cost intent levels equal to $\hat{T}U_1 = 1$ and $\hat{T}U_3 = 3$ that correspond to cost *u=15* and *u'=45* respectively

(based on the user's actual intention). If these tasks are assigned to resources with cost levels $TU_4 = 10$ and $TU_7 = 30$ then we calculate the satisfaction level as equal to:

$$\mathrm{sf} = \frac{1}{1 + |\mathrm{TU}_4 - \mathrm{u}| + |\mathrm{TU}_7 - \mathrm{u}'|} = \frac{1}{1 + |(10 - 15)\ +\ (30 - 45)|} = \frac{1}{21}.$$

In practice of course, a user submitted satisfaction level will be somewhat subjective and will deviate from this "optimal" calculated value.

5.3.3.2 Q-learning methodology

As with any machine learning mechanism, Q-learning has two phases: training and inference. During the training phase, the agent iteratively interacts with the infrastructure and (in our work) with the user and learns the optimal action-value function that maps states and actions to maximize the expected cumulative reward. Through this process, the agent explores the infrastructure characteristics and the user's intentions. The Q-learning algorithm updates the estimate of the action-value function for each state-action pair visited by the agent using the well-known Bellman equation:

$$Q(s, a) = Q(s, a) + \alpha \cdot [r + \gamma \cdot \max(Q(s', a')) - Q(s, a)],$$

where Q(s, a) is the estimated value of taking action a in state s, α is the learning rate that determines how much weight to give to new information, r is the immediate reward received for taking action a in state s, γ is the discount factor between 0 and 1, which determines the importance of future rewards relative to immediate rewards, max(Q(s’, a’)) is the estimated value of the best action a' in the next state s'. These so-called Q-values for all possible state-action pairs are stored in a table, namely the Q-Table. Different selection strategies are possible for the agent in every state; for example, select an action randomly, select the action that it has executed the least number of times, or select the action with the largest Q-value.

In many formulations of the Q-learning process, an ε probability parameter sets a trade-off between exploitation that is choosing the optimal action for the next step, based on the Q-Table and exploration that is choosing a random action. In all cases, the reward after an action is taken, leads to the update of the respective state-action Q-value Q(s,a) and the update of the Q-Table.

The cumulative reward at each time step t is defined as:

$$G_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \dots$$

where R_{t+1} is the immediate reward received by the agent at time step $t+1$ for taking action a_t in state s_t and γ is again the discount factor. The agent aims to find the optimal policy that maximizes the expected value of G_t over all possible sequences of actions and states: the expected cumulative reward. In the inference phase, the trained agent exploits the learned action-value function and serves new demands based on the current state and user satisfaction level, using the action with the highest expected reward.

5.3.4 Evaluation

We performed a number of simulation experiments to evaluate how the Q-learning methodology succeeds in identifying a user's intentions in the submitted infrastructure-agnostic requests. Our experiments focus on the training phase of the Q-learning mechanism.

We consider a cloud environment consisting of storage resources, with N_c = 10 resource levels of capacity: $S_c = \{10, 20, \cdots, 100\}$ GB and with various combinations of the available characteristics in terms of cost and security levels. We also assume that storage capacity is a deterministic parameter of the submitted tasks, while capacity and security are expressed through respective intents. In the experiment performed, we employ various scenarios for translating intents to resource levels that correspond to different user intentions. It is clear that there is not necessarily a linear match between the intent and the resource levels. This means for example that an intent level $\hat{T}U_1$ is not necessarily equal to TU_1, but depends on the user preferences. In what follows, for simplicity we assume a single user that submits storage task requests in an infrastructure-agnostic manner through intents.

Initially, in the experiments performed we assumed resources that have different cost resource levels: 3, 5, 7, while the generated task requests had 2 intent levels. We run the training process for over 10000 timesteps. The Bellman's Equations parameters had the following values α = 0.5, γ = 0.9, while ε = 0.5.

Figure 29 illustrates the average reward over time for the first 1k timesteps. We observe that in all scenarios, the average reward increases over the first 100 timesteps and then stabilizes, increasing just slightly till 10k timesteps (not illustrated in this figure). This is reasonable considering the learning processes and the fact that we selected ε = 0.5, meaning that, on average half of the selected actions are random, that is not based on the calculated Q-values. What is essential to notice is the effect of the number of cost resource levels in the learning process. A higher number of cost resource levels results in a smaller average reward and vice versa lower number of resource levels results in larger average reward. This is due to the fact that a small number of resource levels means that there is a close relation between resource and intent levels, making their matching easier.

Figure 29: The average reward over time for different cost resource levels

Figure 30 illustrates the average reward over time for different ε values *{0.1, 0.2, …, 0.9}* over 10k timesteps. In this way, ε controls the agent’s rate of exploring the environment and identifying an optimal policy. The figure shows that the reward is the highest for ε=0.2, making it the optimal value for the specific problem and the goals set. Another approach is to use a variable epsilon strategy, where the value of epsilon changes over time, being high at first to enable more exploration and decreasing at some point to exploit the calculated Q-values.

Next, we considered the effect of multiple intent parameters (cost, security, and others) in the training process (Figure 31). Increasing the number of different intents a user provides for a single task request leads to a smaller average reward and a slower increase of its value over time. This due to the fact that it is more difficult to match an increasing number of intents to the actual parameters' resource levels.

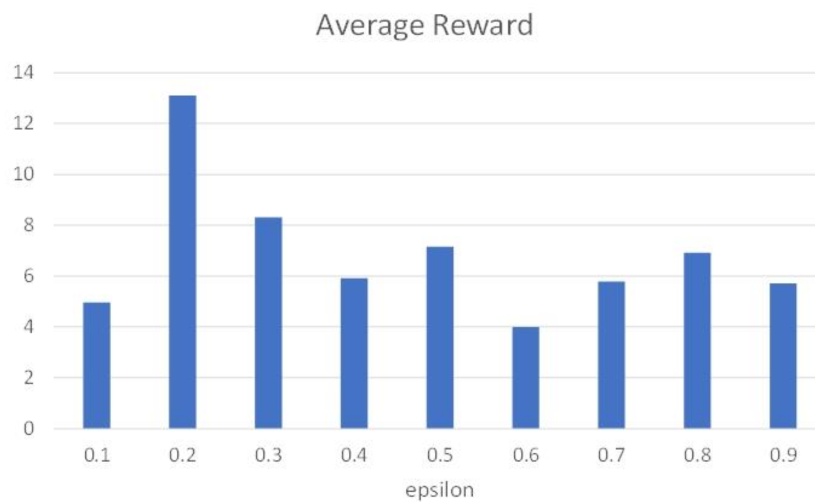

Figure 30: The average reward over time for different ε values

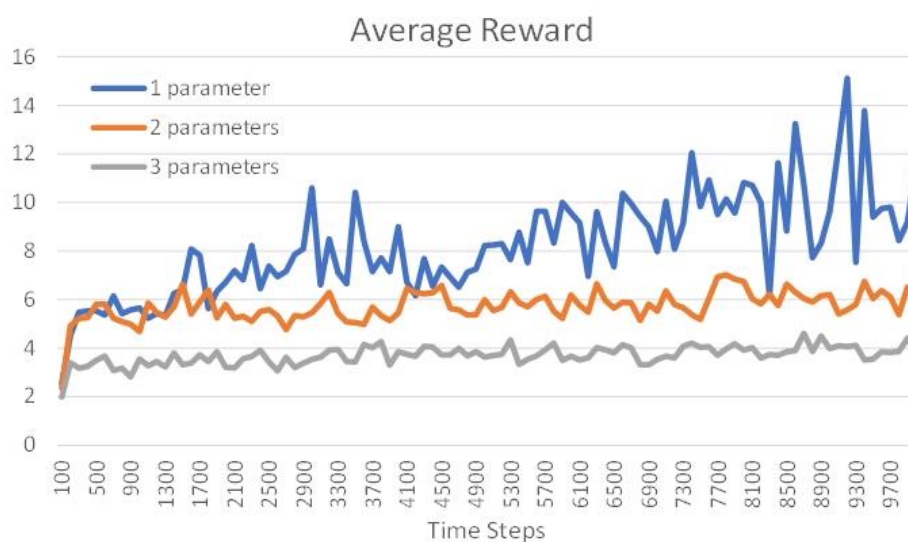

Figure 31: The average reward over time for different number of intent parameters

Finally, we also created a heatmap of the Q-Table (Figure 32) for ε=0.5, 50000 timesteps, and a single intent parameter. This was created out of curiosity, mainly in order to identify any properties of the Q-Table. One thing that we can observe is a kind of symmetry of the Q-Table. This is mainly due to the way the Q-Table is created in terms of step-pairs and the fact that we consider as valid actions those in which only a single task is migrated to a new resource.

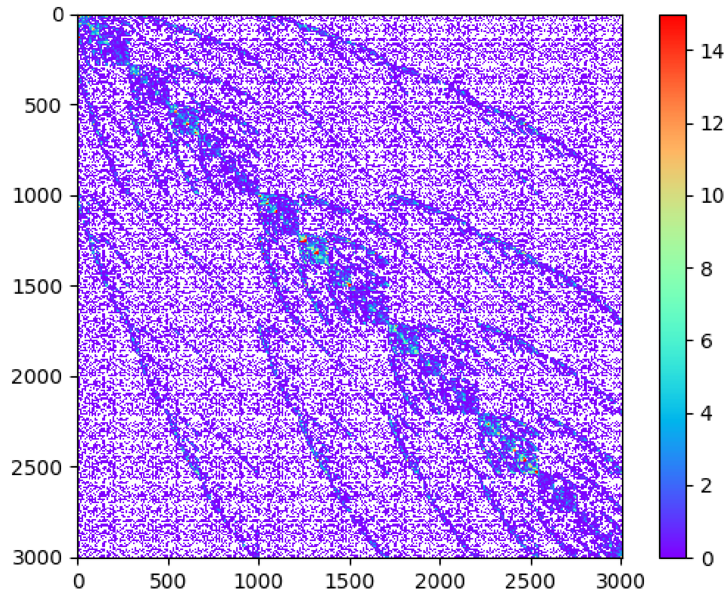

Figure 32: The Q-Table's heatmap for ε=0.5 and 50000 timesteps

5.4 Resource Optimization Toolkit

The SERRANO platform has to automatically determine the most appropriate resources across the distributed edge/cloud/HPC infrastructure to deploy the applications, execute accelerated kernels, and create secure storage policies. The Resource Optimization Toolkit (ROT) integrates the designed resource allocation algorithms in the SERRANO platform, implementing the deciding part at the envisioned closed-loop control based on observe, decide, and act principles. It provides to the SERRANO Resource Orchestrator (Section 9.1) the required logic to allocate the edge, cloud, and HPC resources to satisfy the applications' requirements, coordinate the efficient movement of required data across the selected resources, and support proactively and reactively re-optimization adjustments.

5.4.1 Final implementation and interfaces

Figure 33 presents the architecture of the ROT, its main components, and the interactions with other components within the SERRANO architecture. The deliverable D5.2 (M15), detailed the overall design, the main components, their roles, and the initial implementation. Next, we present their extensions along with the new developments. The architecture includes one ROT controller and multiple Execution Engines, the actual workers. The controller includes the Access Interface and Dispatcher components, while each worker comprises the Execution Engine and the library of the decision algorithms. This approach ensures that the

ROT will always be able to handle quickly any number of execution requests by the Resource Orchestrator, even in very complex infrastructures.

The ROT controller handles the interaction with the other services (i.e., Resource Orchestrator, Central Telemetry Handler) within the SERRANO platform. It exposes the appropriate interfaces that allow bidirectional communication for exchanging commands, information, and notifications. The Execution Engine receives instructions for starting or terminating algorithm executions from the ROT controller and performs all the required actions. It also monitors the node's resources where it is executed and returns related information. The ROT is implemented in Python, using additional frameworks such as Flask 2.0 [3], Pika [93], and PyQt [94].

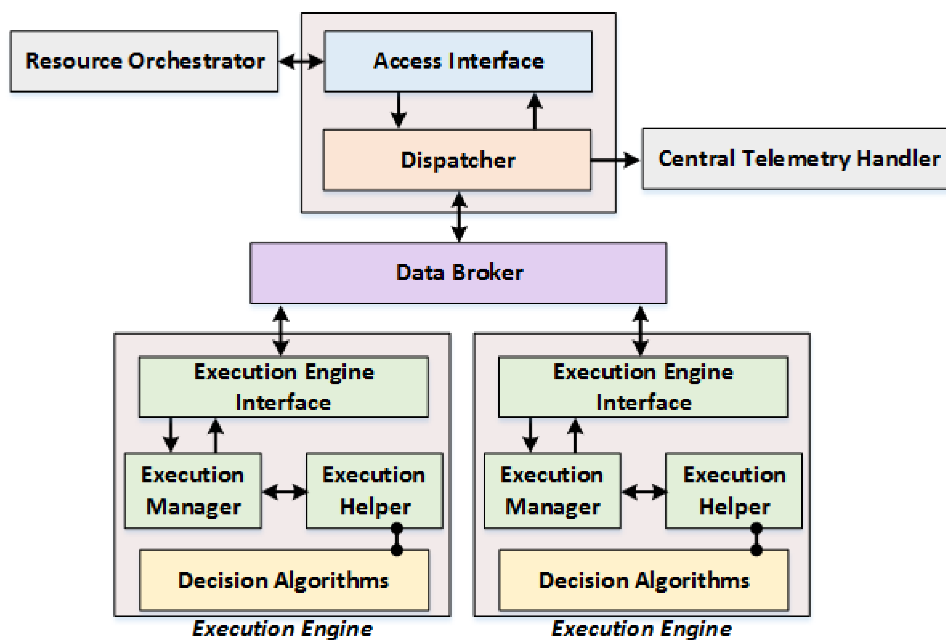

Figure 33: Resource Optimization Toolkit (ROT) architecture and main components

It offers two North Bound Interfaces (NBIs), the first is based on REST APIs, and the second is an asynchronous messaging interface based on the Advanced Message Queuing Protocol (AMQP). The former exposes control operations to manipulate and inspect the execution of deployment algorithms, get information for the available Execution Engines, and manage end users. The latter offers asynchronous communication between the ROT Controller and end users for exchanging notification messages and results. The Data Broker component of the SERRANO architecture implements the required asynchronous messaging interface. To this end, it provides several queues for the asynchronous exchange of messages that are described using a predefined syntax in JavaScript Object Notation (JSON) format. The detailed description and syntax of the messages are available in D5.2 (M15). Figure 34 summarizes the final version of the exposed REST API.

Algorithm Execution

GET /api/v1/rot/history Get the list of all executions.

GET /api/v1/rot/executions Get the list of all active executions.

POST /api/v1/rot/execution Start the execution of some specific algorithm with the requested input parameters.

DELETE /api/v1/rot/execution/{uuid} Terminate a specific algorithm execution.

GET /api/v1/rot/execution/{uuid} Get the details of a specific algorithm execution.

GET /api/v1/rot/statistics Get statistics for the completed executions.

GET /api/v1/rot/logs/{uuid} Get detailed logging information for a specific algorithm execution.

Execution Engines

GET /api/v1/rot/engines Get the available execution engines.

GET /api/v1/rot/engines/{uuid} Get details about a specific execution engine.

User Management

GET /api/v1/rot/users Get details about the registered users.

POST /api/v1/rot/users Register a new user.

GET /api/v1/rot/users/{uuid} Get details abot a specific user.

DELETE /api/v1/rot/users/{uuid} Delete a specific user.

Figure 34: Resource Optimization Toolkit REST API

In the second iteration of the implementation plan, we significantly extend the functionality of the ROT Controller, adding new methods in the REST interface and improving the operation of the Dispatcher component. The new functionality includes (i) support for multiple users, (ii) the use of different topics for results and notifications, (iii) improved handling of failures, and (iv) the implementation of a Python API to abstract the two exposed NBIs.

Moreover, we updated the initial implementation of the mechanisms that provide asynchronous communication between the ROT components (i.e., ROT Controller & Engines) and end users. The final implementation uses four different exchanges (Figure 35), supports multiple users better, and facilitates the scaling features of the ROT.

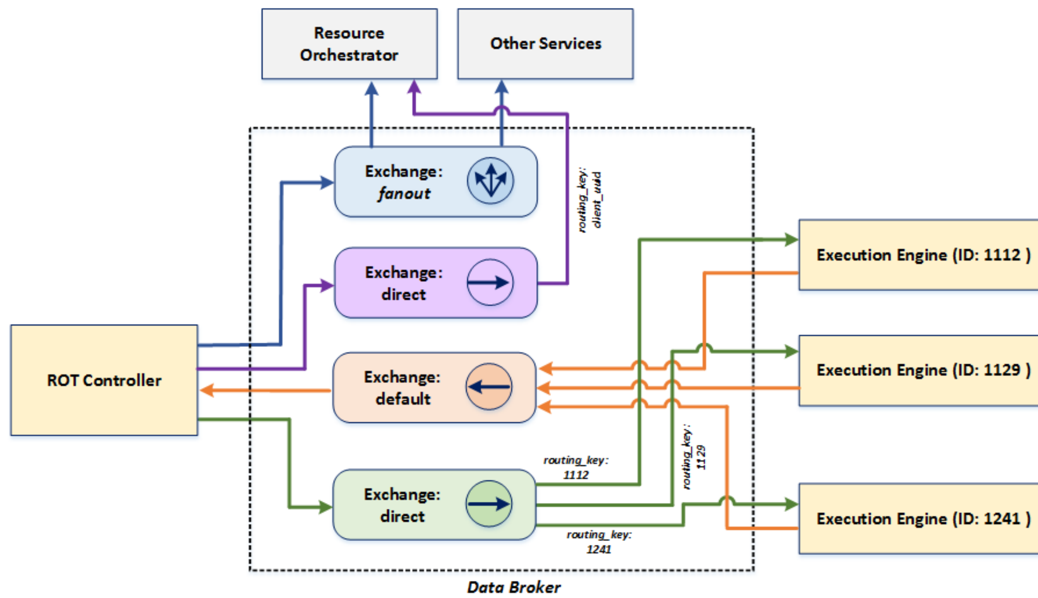

Figure 35: ROT asynchronous communication over SERRANO Message Broker – Final implementation

In addition, the updated syntax for the internal messages that provide the ROT responses has the following syntax:

- **uuid** (*string*): execution request unique identifier
- ***status*** (*string*): final status of the execution request:
  FAILED: execution request failed
  DONE: execution request finished successfully
  TERMINATED: execution request terminated
- ***results*** (string): algorithm execution output
- ***timestamp*** (*integer*): Unix time stamp

5.4.2 Algorithms integration and Python API

Figure 36 presents the overall workflow for executing resource allocation algorithms within the SERRANO platform, highlighting the roles of the ROT components and their interaction with other SERRANO services. The purple arrows indicate operations performed through the exposed northbound interfaces, the green arrows indicate interactions among the primary ROT services, the black arrows correspond to actions related to the preparation of an execution request, and the blue to the actual execution.

The resource orchestration algorithms selected for integration in the Resource Optimization Toolkit are implemented in Python. According to the ROT architecture, an algorithm is accessible from the Execution Engine's internal components through a custom plug-in mechanism. This mechanism exposes a standard interface for all integrated algorithms that determines the explicit syntax of the input parameters and the results for all algorithms. The interface uses JSON as the data-interchange format to provide input parameters and results.

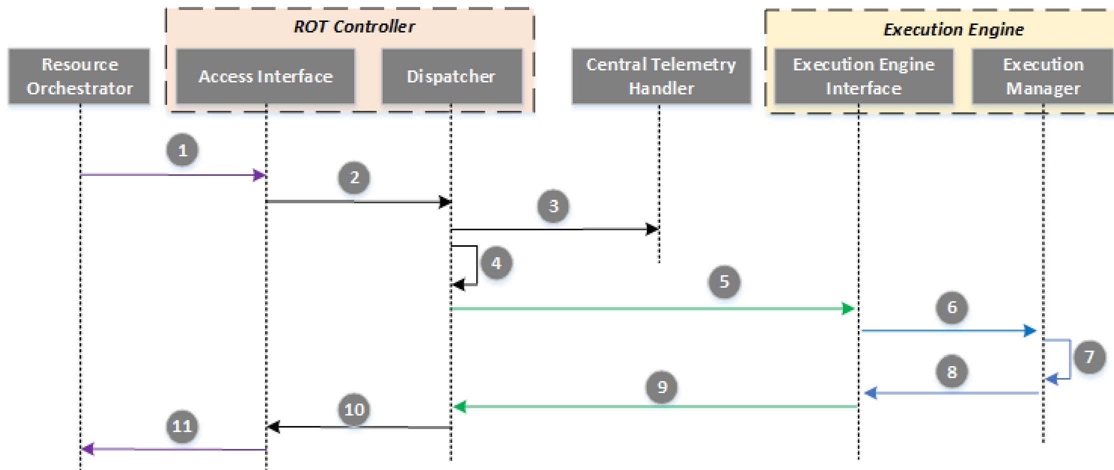

Figure 36: ROT - Workflow for executing an orchestration algorithm in SERRANO

The plug-in mechanism is implemented as a Python wrapper that facilitates access to algorithms and enables their execution through the Execution Helper component in the Execution Engines. The algorithms are organized within a predefined module called ***"algorithms"*** in the Execution Engine, with each algorithm implemented in a separate sub-package (Figure 37). The wrapper receives three parameters from the Execution Helper: (i) the name of the algorithm module, (ii) telemetry information from the SERRANO Central Telemetry Handler (CTH), and (iii) algorithm-specific input parameters. Before assigning the execution request to an Execution Engine, the ROT Controller automatically provides the necessary telemetry information. The ROT Controller queries the CTH service based on the requested orchestration algorithm since a different type of information is required to orchestrate a cloud-native application compared to creating a secure storage policy.

Table 10: ROT plug-in mechanism – AlgorithmInterface abstract class

```
import abc
import json

class AlgorithmInterface(metaclass=abc.ABCMeta):

    def __init__(self, parameters, infrastructure):
        self.__infrastructure = json.loads(infrastructure)
        self.__parameters = json.loads(parameters)

    def get_input_parameters(self):
        return self.__parameters

    def get_infrastructure_parameters(self):
        return self.__infrastructure

    @abc.abstractmethod
    def launch(self):
        pass
```

The plug-in mechanism requires that every algorithm's implementation should include a main file with the same name as the algorithm's module, where the execution starts. The file should include a class with a similar name that should extend the `AlgorithmInterface` class (Table 10). It is an abstract class defined by the plug-in interface, handles the telemetry information and input parameters, and provides the appropriate public methods to facilitate their usage. It also defines the abstract method `Launch()`, where an algorithm should implement its specific logic. The final version of the ROT implementation includes the following orchestration algorithms (Figure 37): (i) a simple first-fit allocation algorithm, (ii) the best-fit heuristic for the security-aware deployment, (iii) the greedy resource allocation algorithm (GRAA), and (iv) the heuristic for the distributed storage allocation (i.e., storage policies), which was presented in deliverable D5.2 (M15).

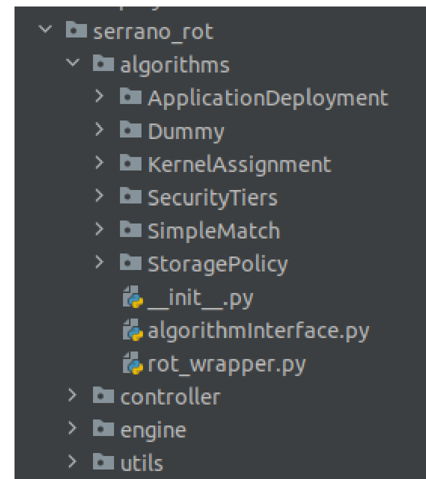

Figure 37: Integrated orchestration algorithms

To facilitate the ROT services integration within the SERRANO platform, we implemented a Python API that abstracts the integration with the ROT controller and the exposed northbound interfaces. The API is also part of the SERRANO SDK, while it can facilitate a more general adoption of the designed framework even outside of the SERRANO platform as a standalone service. Moreover, the Orchestration Manager, one of the primary services of the SERRANO Orchestrator (Section 9.1), uses the provided Python API to implement the required interactions with the ROT service. The API includes a set of methods to abstract the interaction with the REST interface and various events to handle the low-level operations for interacting with the asynchronous communication over the SERRANO Data Broker.

Table 11 summarizes the provided methods and events, and Figure 38 shows a related code snippet. The example presents the execution of various managing requests, such as getting the list of available Execution Engines (line 52) and execution requests (line 57) and the request of an algorithm execution (line 67), along with the results handling (lines 18, 33-38).

Table 11: ROT Python API – Provided methods and events

Name	Type	Description
get_engines()	Method	Get the available execution engines.
get_engine(engine_uuid)	Method	Get details about a specific execution engine.
get_executions ()	Method	Get the list of all active executions.
get_execution(execution_uuid)	Method	Get details about a specific algorithm execution.
get_logs(execution_uuid)	Method	Get detailed information for a specific execution.
get_statistics()	Method	Get statistics for the completed executions.
request_execution(algo, parameters)	Method	Start the execution for a specific algorithm with the provided input parameters.
terminate_execution(execution_uuid)	Method	Terminate a specific algorithm execution.
EventEnginesChanged	Event	Current state of a specific Execution Engine changed.
EventExecutionCompleted	Event	An execution request is completed successfully.
EventExecutionError	Event	Error during the execution of some request.

```python
    def start(self):
        self.client = clientInstance.ClientInstance()

        self.client.connect([clientEvents.EventExecutionCompleted,
                             clientEvents.EventExecutionError,
                             clientEvents.EventExecutionCancelled],
                            self.on_execution_response)
        self.client.connect([clientEvents.EventEnginesChanged], self.on_engines_changed)
```

```python
    def on_execution_response(self, evt):
        print("------------------------------")
        print("Event type: %s" % evt.evt_type)
        if evt.evt_type == "EventExecutionCompleted":
            print("Execution UUID: %s" % evt.execution_uuid)
            print("Results: %s" % evt.results)
```

```python
        engines = self.client.get_engines()
        for engine_uuid in engines:
            print(self.client.get_engine(engine_uuid))
            print()

        executions = self.client.get_executions()
        print(executions)

        input_params = {"size": 100,
                        "cost": 5,
                        "permanent_storage": 1,
                        "availability": 0,
                        "latency": 1,
                        "lat": 51.5074,
                        "lng": 0.1278}
        res = self.client.request_execution("StoragePolicy", input_params)
        print(res)
```

Figure 38: Code snippet for interacting with the ROT through the provided Python API

5.4.3 Deployment and configuration

The ROT framework components are implemented in Python language and packaged in container images using the SERRANO CI/CD services, ensuring a smooth and efficient development workflow. The source code will be available in the official repository of the SERRANO project [95] under an open-source licence (Apache 2.0). There are separate container images for the ROT Controller and ROT Execution Engine. The resulting container images are accessible through the official SERRANO Harbor image repository [96]. Moreover, corresponding Kubernetes YAML description files are also available to facilitate effortless deployment on Kubernetes platforms. These files enable the automatic deployment and scaling of the ROT Controller and ROT Execution Engines within Kubernetes.

6 Service Assurance and Remediation

In this section, we provide a comprehensive overview of the Service Assurance and Remediation (SAR) components. Specifically, we detail the Event Detection Engine (EDE) and how it is used for the detection analysis of the monitoring data received from the SERRANO telemetry framework. Once unwanted behaviour has been detected and analysed, the root cause is determined by computing Shapely values.

The following subsections detail the work performed during the second iteration of the implementation plan (M16-M31) on SAR. We should note that some information is also available in D5.2 (M15) however, for the sake of completeness we will include details regarding the overall architecture of SAR here as well.

6.1 Architecture

The Event Detection Engine (EDE) is a crucial component of the overall Service Assurance and Remediation mechanisms (SAR) as it enables the timely detection of any performance, behaviour and time/sequence related anomalies. The technology stack used for its implementation was chosen in order to create a robust, scalable solution that is also easy to extend with new state of the art detection methods.

When dealing with distributed systems deployed on heterogeneous hardware platforms, it is not a question of if but rather when anomalous events will occur. In order to initiate autonomous remediation of any such events we must first have a reliable anomaly detection mechanism. While simple point anomalies are quite easy to detect with relatively simple rule-based mechanisms, contextual and/or sequential anomalies of multivariate data are challenging. EDE provides a comprehensive set of ML methods that are specifically chosen to enable the detection of just such anomalous events.

In the context of SERRANO, we improved EDE with a few key features and capabilities. The main objectives for SERRANO are:

- Identify ML detection methods that are suitable for the detection of anomalous events in the SERRANO context. These include both supervised and unsupervised methods.
- Implement ML predictive model optimization mechanisms that can be used for both predictive performance optimization as well as user-defined constraints (i.e. inference times, computational resource utilization, model size etc.). Moreover, several visualizations that give insight into model performance have also been implemented, giving feedback to the end user at every stage of the optimization process.
- Implementation of Explainable AI mechanisms which can give meaningful insight into what caused a particular anomalous event to occur (root cause analysis). These methods stand as one of the main outputs of SAR.

- Integration into the SERRANO toolchain, especially for the EDE, which should report detected anomalies. The target integrations include the SERRANO orchestration and telemetry services.

Figure 39 shows the overall architecture of EDE. This robust framework consists of five primary components, each tasked with a specific functionality necessary for creating and exploiting ML-based detection methods. The SERRANO platform enables seamless deployment of cloud-native applications across distributed infrastructures with highly heterogenous hardware platforms. This leads to high volumes of data that need to be analysed. To efficiently handle such demanding workloads (EDE being implemented in Python) we have selected Dask [77] as the execution backend.

Dask allows for parallel/concurrent execution of training, optimization and prediction tasks, providing scalability to process the vast amount of data effectively. Furthermore, it can leverage an existing Dask cluster, simplifying configuration for EDE users that only have to configure the EDE scheduler. When Dask is deployed on top of Kubernetes autoscaling is also possible. We have developed a simple Dask scheduler that is capable of on-demand scaling of a cluster using Kubernetes. In the absence of a user-defined Dask scheduler, EDE will create a local 3-worker deployment during normal operation. Next, we will discuss some particularities of the architecture.

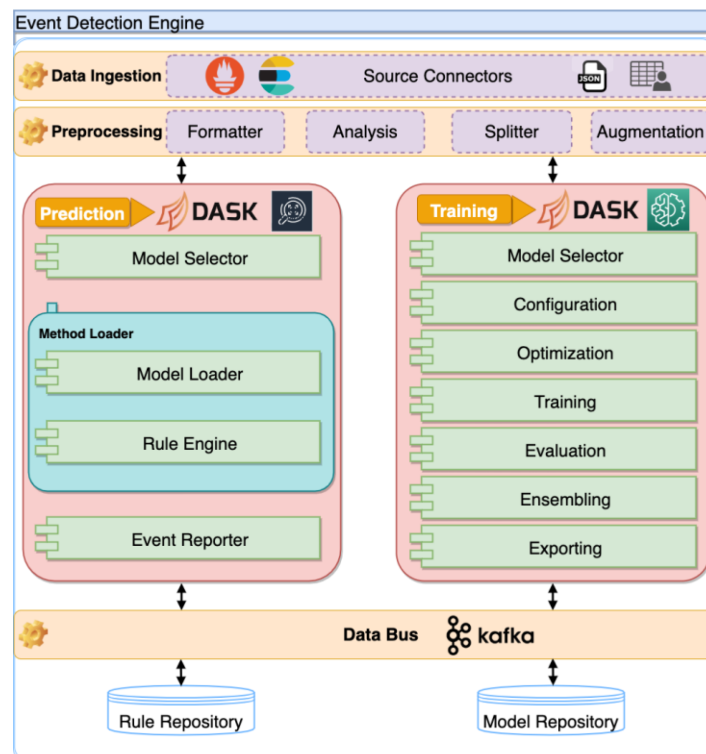

Figure 39: Event Detection Engine – General architecture

The first component is the ***Data Ingestion,*** which ensures data is available to the ML methods. We currently support fetching data directly from several data sources, such as Prometheus [78], ElasticSearch [79], and InfluxDB [101]. Of course, this mechanism is integrated with the SERRANO Telemetry Service. Local labelled data is also supported for supervised method training in CSV and JSON formats. This feature is designed to be used for analysis, training, and optimization phases. We also support MinIO data sources; however, the data will be fetched locally and not streamed in this scenario.

Legacy support for Attribute Relation File Format (ARFF) files is also enabled in the form of a custom conversion script. Initially, we had direct support for this Weka-based format however, as it is of little practical significance during production scenarios it was removed.

Preprocessing is implemented as a separate component and is capable of formatting as well as augmenting both local training and historical monitoring data to be used for predictive model creation. Data normalization and scaling is also handled by this component. We should note that the resulting scaler is also serialized and can/should be used on streamed live monitoring data.

In addition, statistical analysis is also executed by the pre-processing component. It is important to note that although there are some predefined data augmentation and analysis methods, EDE can execute user-defined methods as long as compatibility with the internal EDE processing pipeline is maintained. In essence, the processing pipeline uses data frames; thus, augmentation and analysis methods need to accept and return Pandas [80] DataFrames.

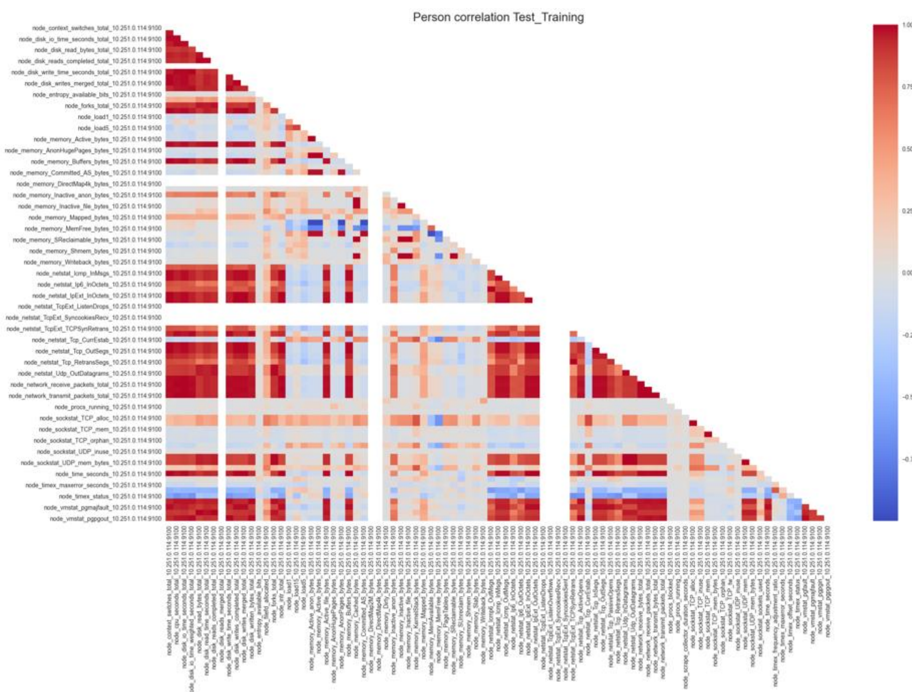

Figure 40: Pearson Correlation Raw Data

There are several example analysis and augmentation methods defined in EDE. Figure 40 shows the Pearson correlation between the features (system metrics) collected in case of one

monitored node. This visualization and analysis step can be used for feature engineering use cases as it measures the strength of the linear relationship between pairs of features.

Next, in Figure 41, we see a reprojection of the data with only three principal components using t-SNE (t-distributed stochastic neighbour embedding). Here we also highlight the five classes of labels present in the dataset. As we mentioned, these are only some of the methods already included in EDE, while users can easily add additional methods.

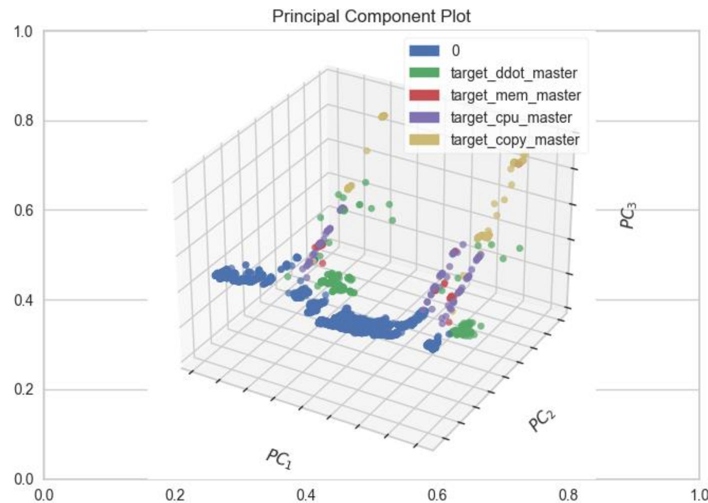

Figure 41: Feature reduction (t-SNE)

The ***Training*** component is used to select, configure, and optimize ML-based predictive models. In the case of supervised methods, models need to be trained using a labeled dataset. As with other components within EDE, users can define their own detection methods as long as they are in accordance with the processing pipeline and respect Scikit-learn [81] API naming conventions.

Optimization takes several forms. First, we have several Hyper-parameter optimization (HPO) methods ranging from unguided methods such as Grid and Random Search to guided approaches based on genetic algorithms, Bayesian methods, Tree of Parzen Estimators etc. Furthermore, model performance analysis methods and visualizations such as recursive feature elimination (based on feature importance), training instance selection based on learning curves etc. are also available. The main goal is to enable users to create predictive models with good predictive performance.

The optimization methods enumerated in the previous sections can be configured via the YAML configuration file. A typical use case would entail first running HPO on a selected ML method. If a Dask cluster is available, each worker will be assigned to evaluate one candidate solution, thus optimization is significantly faster. Once the best performing hyper-parameters have been identified users can define additional optimization analysis methods. All steps from this process are logged and visualization is created where applicable. Finally, the predictive models are exported. The following subsection details some of the experiments we have conducted using this methodology. Further research and analysis results can also be found in D4.4 (M30).

Prediction is handled by a separate component. It is capable of instantiating previously serialized methods. Usually, joblib [82] is used for model serialization and de-serialization, although ONNX [83] is also supported for DNN models. All detected anomalies are then forwarded to the EDE data bus using Kafka [84] topics. This means that other SERRANO tools and components can check this dedicated topic for any anomalous events being detected.

It is evident that simply detecting anomalies is not enough to enable assurance and remediation. An analysis of why a particular event is labelled anomalous is required. To this end, we currently support some explainable AI-based methods, such as the calculation of Shapley values.

Shapely values [85] can be used to select features with a high degree of impact on a prediction. This is especially useful in the case of unsupervised methods where this explanation can be used on a per prediction basis (i.e., why an event is deemed anomalous).

This measurement was first introduced for the study of coalition games. It is defined on a value function denoted as υ of players *S*. The Shapely value denotes the contribution to the pay out of a particular feature value, weighted and summed over all possible combinations:

$$\phi_i(\upsilon) = \sum_{S \subseteq \frac{\{1,\dots n\}}{\{i\}}}^{\infty} \frac{|S|!\,(n-|S|-1)!}{n!} (\upsilon(\mathrm{S} \cup \{\mathrm{i}\}) - \upsilon(\mathrm{S}))$$

Where *n* denotes the set of all players (in our case features), thus the Shapely value of game (υ,*n*) is used to distribute the total gain $\upsilon(n)$ to each player in accordance with each contribution [86]. Player *i* in our case corresponds to a feature from the dataset, thus *n* denotes the total number of input features. Conversely, the Shapely feature value $i \in n, \phi_i(\upsilon)$ is the weighted average of the marginal contribution. Resulting from the above equation, we can compute the nprediction for feature values in a set *S*, a subset of the features used to train the model, which are marginalized over features that are not in set *S* as follows:

$$\upsilon_x(\mathrm{S}) = \int f(x_1, \dots, x_n) dP(x \notin S) - E_X[f(X)]$$

For our purposes, *X* is the vector of values from the features of the instance (event) to be explained, and *n* is the number of features. Shapely values are symmetrical in the sense the sense that equal contribution results in equal Shapely values and non-contributing features have a Shapely value of 0.

The computed shapely values are used for remediations. For each event being analysed we provide the Shapely values for each feature as well as its base value. An example response can be seen in Figure 42. Notice that we provide data about the method, model, and query interval. This is in order to have a complete overview by SERRANO orchestration and the end user. Next, for each anomalous instance we provide the timestamp (in two formats) and the shapely and base values respectively.

```
{
  "method": "<detection_method>",
  "model": "<detection_model_name>",
  "interval": "<query_interval>",
  "anomalies": [
    {
      "utc": "<utc_time>",
      "hutc": "<human_readable_utc>",
      "analysis": [
        {
          "shape_values" :[
              ... # feature and impact score
          ],
          "base_values": [
              ...
          ]
        }
      ]
    }
  ]
}
```

Figure 42: SAR Response Example

6.1.1 Configuration and REST API

For the sake of brevity, we will not detail the entire configuration of EDE and SAR however, there is available a complete user manual in the official GitHub repository:

- https://github.com/ict-serrano/service-assurance-ede

We have implemented a REST API for controlling EDE inference settings for ease of integration and use. It is implemented using OpenAPI. Users are not able to train and validate predictive models using this API since it is only meant for inference. We did this as training is an inherently user-driven, iterative workflow.

Figure 43 shows the resources available for configuration. The functionality exposed here includes Connector setup for data sources, augmentation and analytics, data filtering, and predictive model selection.

config

Method	Endpoint
GET	/v1/config
PUT	/v1/config
GET	/v1/config/augmentation
PUT	/v1/config/augmentation
GET	/v1/config/connector
PUT	/v1/config/connector
GET	/v1/config/filter
PUT	/v1/config/filter
GET	/v1/config/inference
PUT	/v1/config/inference

Figure 43: SAR REST Configuration

data

GET /v1/data

GET /v1/data/{data_file}

PUT /v1/data/{data_file}

inference

POST /v1/detect

logs

GET /v1/logs

engine

GET /v1/service/jobs

GET /v1/service/jobs/{job_id}

DELETE /v1/service/worker

GET /v1/service/worker

POST /v1/service/worker

Figure 44: SAR REST Control

Figure 44 shows control resources used for detection job definition. Once a valid configuration has been set (see Figure 43), inference instances can be started using the *engine* and *inference* resources. If users or tools require the analysis of certain datapoints, the *data* resources can be used to push data instances directly to EDE.

6.2 Methods for Detection and Analysis

In D5.2 (M15) we detailed some of the experiments utilizing supervised methods specifically the HPO methods. For this deliverable, we will focus on some additional experimental results as well as results for unsupervised methods. In order to make this deliverable as self-contained as possible, we will briefly describe, the dataset used for these experiments.

The dataset used for these experiments was created using an anomaly induction tool created by UVT. It can induce anomalous events that mimic hardware anomalies. In the current dataset, we induced four anomalous event types:

- **CPU Overload** – Detects the number of physical CPU cores and saturates a user specified number of cores for a number of seconds. This simulates CPU overloading
- **Memory Eater/Leak** – Writes data into RAM, the amount is specified by the user in KB, MB, GB along with a multiplier and iteration step. This simulates memory interference fault and saturation. It is possible to define also how long this memory allocation is to be maintained.
- **DDOT** – Reputedly calculates the dot product between two matrices. The size of each matrix is calculated based on the CPU L2 cache size reported in the OS. This simulates CPU cache faults. Care should be taken when configuring this type of anomaly as large matrix sizes can cause unpredictable OS behaviour causing zombie and/or orphan

POSIX processes. Additionally, faulty logging of such events will yield contaminated labelled data.

- **COPY** – Generates and moves a large file from one location to another. Users can set allocation units (KB, MB, GB) and a multiplier. This simulates I/O interference, saturation and failing hard drives. A side effect of this type of anomaly is that it resembles the Memory Eater/Leak anomaly. This will help us quantify the ability of ML predictive models' capacity in differentiation of the two.

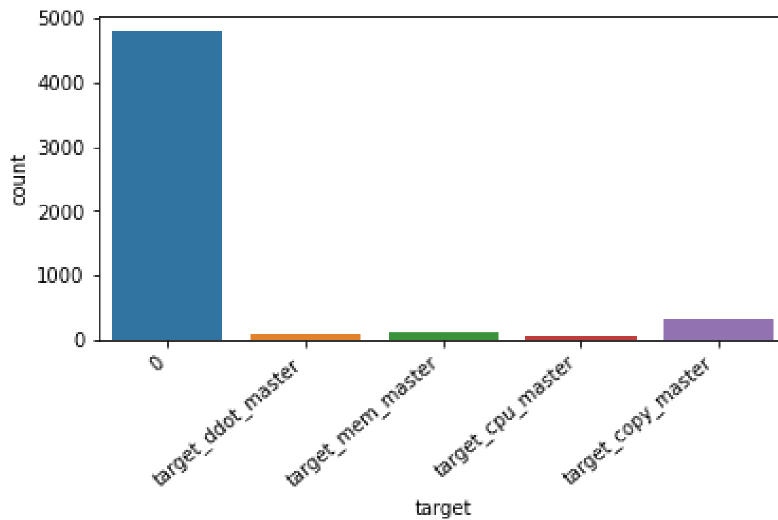

Figure 45: Class distribution

Figure 45 shows the distribution of event types in the dataset used. There is a total of 5400 events comprised of 90 features. Each feature represents a system level metric collected by Prometheus. We can clearly see that this dataset is extremely unbalanced: 4792 events are normal, 91 are DDOT, 132 are Memory Eater/Leak, 64 are CPU Overload, and 321 are COPY.

6.2.1 Supervised ML methods

In previous examples, we detailed method performance when dealing with single anomaly instances in a single node. This type of scenario is completely plausible. However, we also ran several experiments by inducting several anomalous events simultaneously since overlapping anomalies can also occur in practice. There are several ways of dealing with this scenario.

The best way in our opinion is to use a method called One-vs-Rest or One-vs-All, where a new predictive model will be trained for each class present in the training dataset. In our case, this means that for each training instance, four predictive models will be created.

This method has several advantages. First, we can use all the algorithms already tested for D5.2. We should point out that some of the methods previously tested support by default this type of scenario (i.e., XGBoost). However, we choose to implement our own method seeing that there are several ML methods used which are not fully compatible when it comes to how they report predictive performance.

Second, we can use a One-vs-Rest methodology on datasets that do not have overlapping anomalous instances. This results in a much-simplified training and validation cycle, where we can prepare for overlapping anomalous instances without having concrete examples in the training or validation set. For the sake of brevity, we will refrain from listing a complete list of all our experimental results. Instead, we will focus on those methods, which are also described for D5.2.

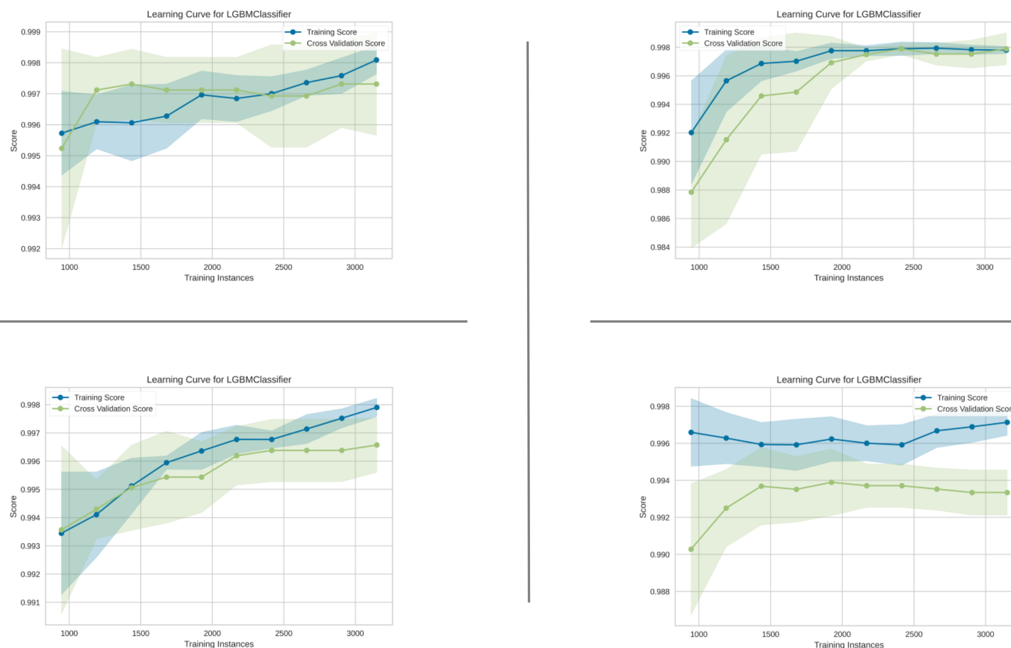

Figure 46: Learning curve XGBoost overlapping anomalies

Figure 46 shows how each trained XGBoost model handles a different number of training instances in the case of all 4 main anomaly classes. If we go from left to right, we have figures for COPY, MEM, CPU, and DDOT anomalies depicted. As before, the experiments were designed to gauge the performance of XGBoost with differing amounts of training data. All iterations used Stratified Shuffle split with 5-fold cross-validation.

The overall predictive performance for XGBoost, in particular, and all other ML methods in general can be seen as similar to the results shared in the past deliverable. Figure 47 shows the ROC (receiver operating characteristic) curves in the same order as in Figure 46. We can see that predictive performance is similar for overlapping anomalies and non-overlapping anomalies.

Regarding inference times, these are not overly affected as they are at most 10% slower than single model inference. This can be easily explained by the fact that the newly trained models are much simpler, being in essence binary classifiers. The final inference is obtained by stacking all model predictions. For ease of validation, we compute the predictive performance of each model separately.

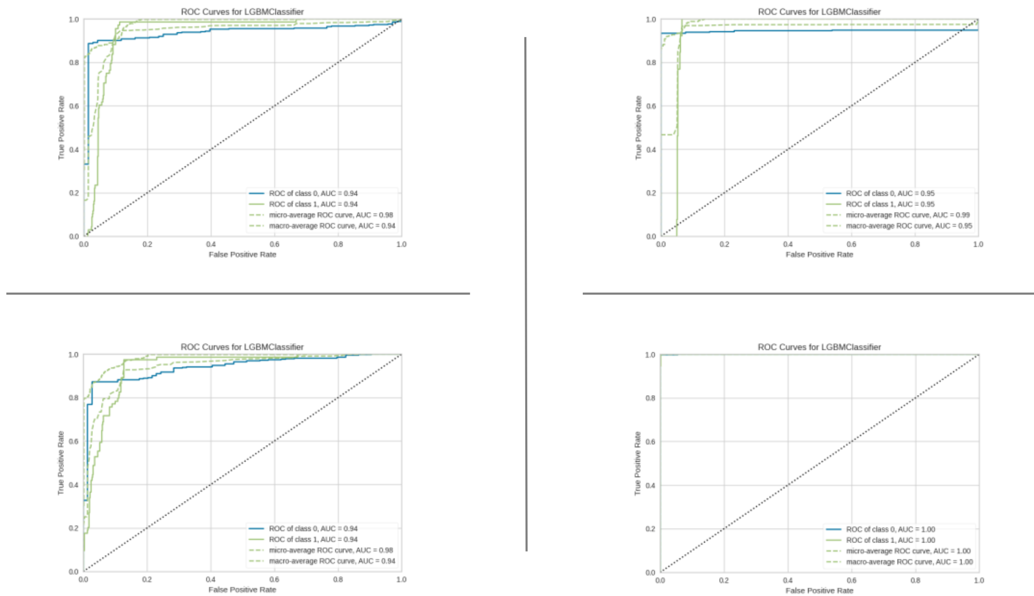

Figure 47: ROC curve for XGBoost

6.2.2 Unsupervised ML methods

Unsupervised ML methods are arguably more useful than supervised ML methods in the context of anomaly detection. These types of methods do not require labeled datasets for training. One major downside in our case is that most unsupervised anomaly detection methods are basically binary classifiers. This method detects whether an event is anomalous or not. There are some ways to give insight into which features caused a particular instance to be marked as anomalous. In our experiments, we used Shapely values to accomplish this.

In the initial experimental phase, we selected 17 ML methods for testing. As before, we will detail the three best models for brevity. These models were selected partially because of their performance and partially for the different underlying principles they are operating under. We should also mention that some methods, such as One class SVM, performed extremely poorly. A complete analysis of ML methods and their suitability for use in the SERRANO will be performed in the form of a journal paper.

Isolation Forest is an outlier ensemble-based algorithm that is constructed from multiple isolation trees [87]. It explores random subspaces from the data. In essence, it explores random local subspaces as each tree uses different splits. Scoring is done by qualifying how easy it is to find a local subspace of low dimensionality in which a particular event is isolated [88]. In other words, distance from the leaf to the root is used as the outlier score. Similar to Random Forest, a supervised method, the final score is obtained by averaging the path length of any particular data point in different isolation trees. In most scenarios, Isolation Forest works under the assumption that it is more likely to detect or isolate an outlier in a subspace of lower dimensionality created by the random splits.

Clustering-Based Local Outlier Factor (CBLOF) [89] is a proximity-based algorithm, which is a combination of Local Outlier Factor (LOF) and a clustering technique. LOF adjusts the anomaly (or outlier) score based on local density. The density is defined as an inverse of average distances. This approach results in events in local regions of high density being given higher anomaly scores even if they are isolated from the other events in their locality. This is mainly due to the definition of density that does not contain the number of anomalies in a cluster. In fact, CBLOF is a score in which anomalies are defined as a combination of local distance to nearby clusters and the size of the clusters to which each event belongs. Thus, events in small clusters that are at large distances from nearby clusters are flagged as anomalies.

Variational AutoEncoders (VAE) are deep neural network models designed for unsupervised training, which can be used for AD tasks [90]. They are often mentioned together with Autoencoders (AE), which are also deep learning models with seemingly similar topological components: encoder and decoder. The encoder tries to learn a lower-dimensional representation of the input data (similar to PCA), and a decoder attempts to reproduce the input data in the original dimension (AEs are usually symmetrical). AEs try to encode the data in such a way that they reduce the reconstruction error. When used for AD AEs reconstruction error can be used as a form of anomaly score. Provided the AE has sufficient training data to provide a minimal reconstruction error for normal data.

Table 12: Unsupervised method scores

Method	Class	pre	rec	spe	F1	geo	iba
Isolation Forest	0	0.90	0.90	0.61	0.90	0.74	0.57
	1	0.61	0.61	0.90	0.61	0.74	0.53
Avg/total	all	0.84	0.84	0.67	0.84	0.74	0.56
CBLOF	0	0.92	0.92	0.71	0.92	0.81	0.67
	1	0.71	0.71	0.92	0.71	0.81	0.64
Avg/total	all	0.88	0.88	0.75	0.88	0.81	0.64
VAE	0	0.98	0.98	0.93	0.98	0.95	0.92
	1	0.93	0.93	0.98	0.93	0.95	0.91
Avg/total	all	0.97	0.97	0.94	0.97	0.95	0.91

Table 12 shows the overall scores obtained by the 3 ML methods. It is split on a per class basis where 0 is a normal data instance and 1 is an anomalous instance. We should also mention that for the sake of validation we created a special label for the dataset used during the previous supervised experiments. Basically, we transformed the problem into a binary classification problem.

Overall, the VAE is the best performing method, although there are some considerations to mention. IF consumes the least amount of resources for training and inference while VAE requires substantially more computational resources and specialized hardware in the form of GPGPUs. CBLOF is all-around the best performing model.

Figure 48 highlights the differences between the three ML methods. It shows the decision boundaries as projected in a 2-dimensional space. In order to accomplish this visualization, we re-projected the original data to a 2D space, using PCA, while at the same time keeping the anomalous instance markers as returned from the original data.

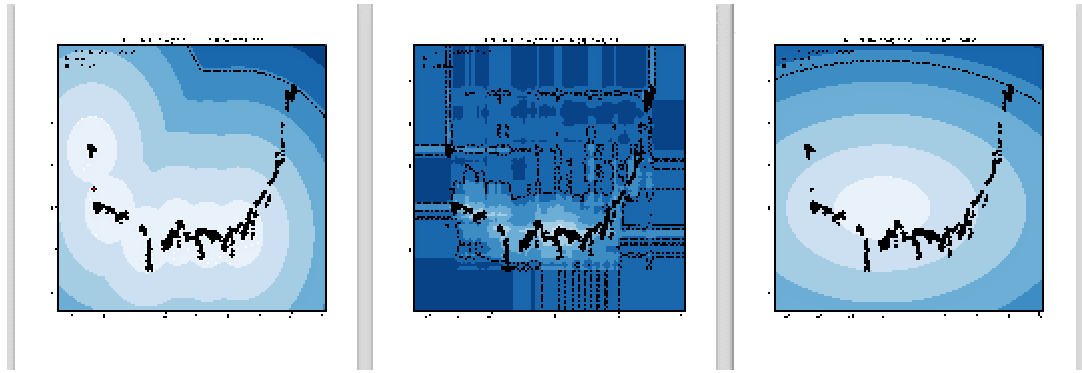

Figure 48: Decision Boundary Comparison

The decision boundaries also show some of the downsides of each method. IF tends to overfit quite easily, leading to poor out of sample performance. This is clearly seen in the decision boundary. The data is tightly circumscribed, while in the case of CBLOF, we see that the decision boundary shows several hotspots or cluster.

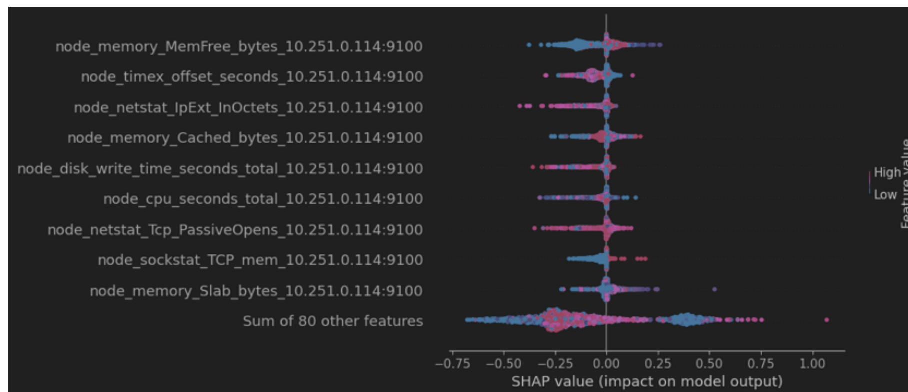

Figure 49: Shapley value-based feature importance

In order to derive the maximum amount of information for use in SAR when using unsupervised methods, we use Shapely values. Figure 49 shows how Shapely values can be used for calculating which feature from the dataset impacted the detection of anomalous instances. Based on this ranking, memory-related features are ranked among the highest. This is a direct result of how the anomaly induction methods work. MEMORY, COPY, and DDOT anomalies have a significant memory component to them.

Figure 50 shows how Shapely values can be used on a per instance basis, not just globally. We can see an anomalous instance that has CPU, memory and disk related features. In fact this is a DDOT anomalous instance. Although inferences are not as clear as in the case of supervised methods, we can still give insight into what caused each anomaly to occur.

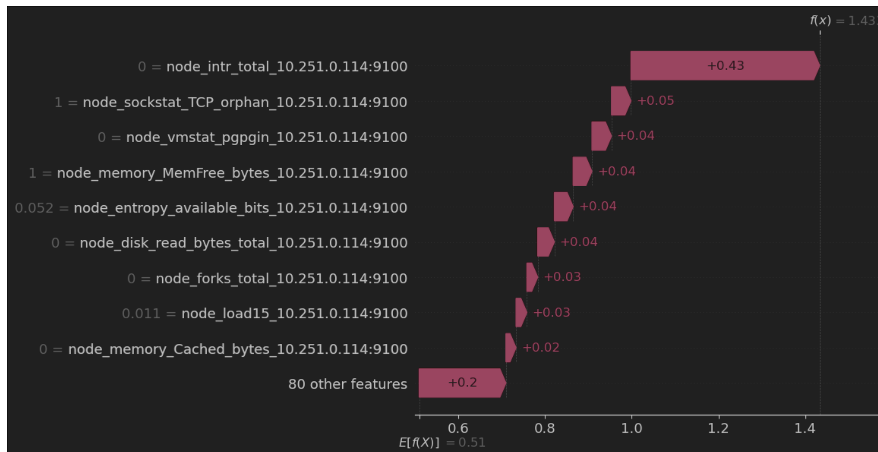

Figure 50: Shapley value-based feature importance

6.3 Discussion

In this section, we presented the overall architecture of the SAR, specifically the event detection component, EDE. We showed how the SAR component can analyse telemetry data and signal which event is anomalous. We also detailed how SAR can be configured and used both via configuration file and REST API.

In the case of supervised methods, we extended our initial work by adding support for detecting overlapping anomalous instances. While the results presented in this deliverable are relatively few, we aim to finalize a journal paper detailing our results before the end of the project. Furthermore, we aim to incorporate part of the transprecision work done in WP4 and was reported in D4.4 (M30). For unsupervised methods, our experimental results show that we can get meaningful insight into what caused an anomalous instance to occur. While these are basically binary classification methods, we can still train high-quality models with good predictive performance. The root cause analysis of detected anomalous events can be successfully done using Shapely values. Some of the results and experiments have already been published in a journal article [91], and we aim to have at least one more journal article by the end of the SERRANO project.

Regarding integration, SAR requires access to only two SERRANO components. The first is the telemetry services (i.e., Central Telemetry Handler and Persistent Monitoring Data Storage). Once these integrations are set up SAR can analyse the incoming data. Second, SAR requires access to a Message broker to send all analytics reports from where other components and end-users can fetch the data.

7 Network and Cloud Telemetry Framework

7.1 SERRANO Telemetry Framework

A heterogeneous and distributed infrastructure, like any system, must be observable before it can become subject to optimization. Towards this direction, Task 5.3 developed the SERRANO telemetry framework that includes autonomous and scalable mechanisms to provide the sense (detect what is happening) and discern (interpret senses) operations in the envisioned closed-loop control. SERRANO's hierarchical monitoring infrastructure (Figure 51) aims to facilitate orchestration decisions, detect problems, and trigger proactive or reactive adjustments to SERRANO-enhanced resources and deployed applications. The SERRANO telemetry stack consists of three key building blocks: (a) the Central Telemetry Handler, (b) Enhanced Telemetry Agents, and (c) Monitoring Probes.

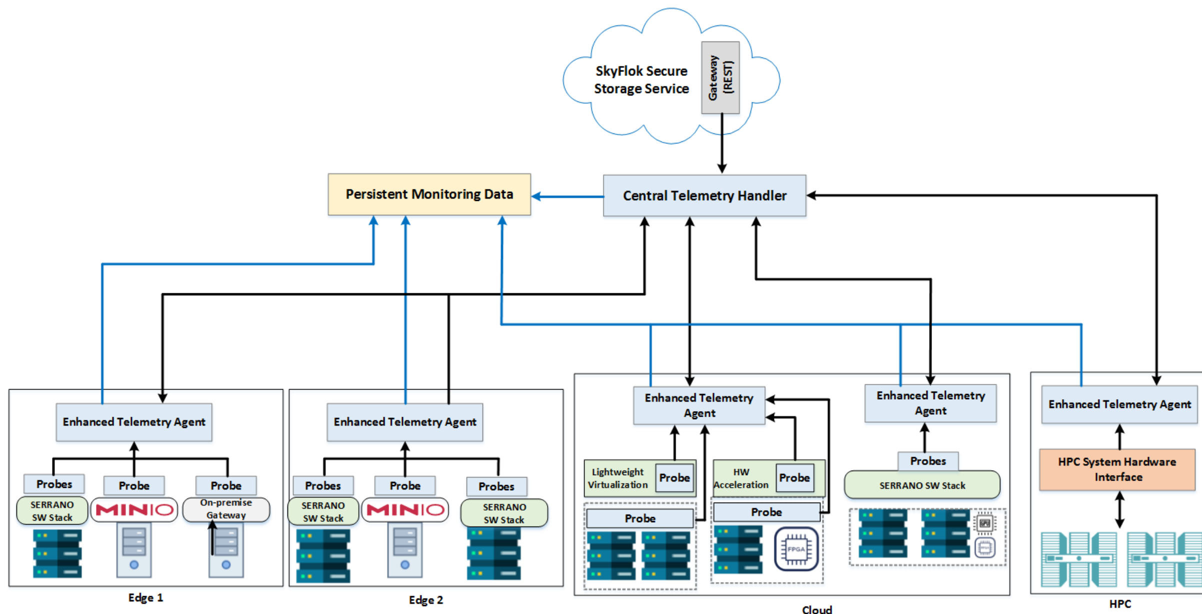

Figure 51: SERRANO hierarchical telemetry architecture

The Central Telemetry Handler is the root element of the SERRANO hierarchical telemetry infrastructure. The various Enhanced Telemetry Agents are responsible for a specific set of Monitoring Probes. The collection and exchange of monitored information is performed periodically, while the granularity can be adapted, and other telemetry operations can be activated based on detected events or explicitly by entities at upper layers. The telemetry functionalities are spread into several layers to meet the scalability requirement while enabling immediate reaction to events that affect the performance of the deployed applications at individual parts within the SERRANO platform. Deliverable D5.3 (M15) provides the overall design of the telemetry framework along with the technical details for the initial implementation of its main components. Next, we present the developments during the second iteration of the implementation period (M16-M31), the final developments in the Persistent Monitoring Data Storage (PMDS) service, and the successful integration of the telemetry framework mechanisms with other services in the SERRANO platform.

7.1.1 Central Telemetry Handler and Enhanced Telemetry Agent

The Central Telemetry Handler and Enhanced Telemetry Agents provide the same core functions at different scales and views of the infrastructure resources and deployed applications. Hence, they share the same design (Figure 52) and a joint implementation for their core components. A more detailed description of the individual components is available in D5.3 (M15). Next, we present the extensions and new developments during the second iteration of the SERRANO implementation plan (M16-M31).

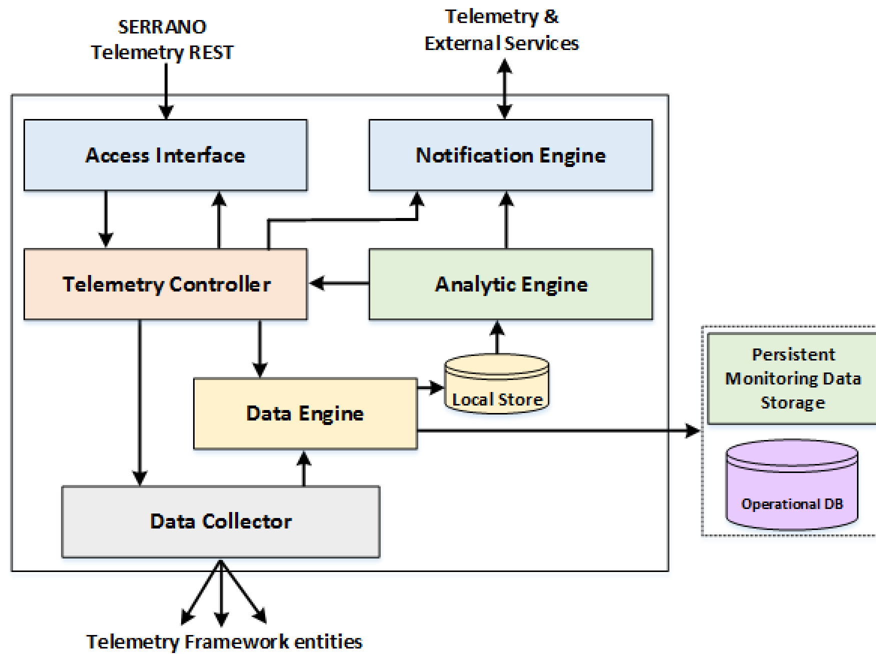

Figure 52: Central Telemetry Handler and Enhanced Telemetry Agent architecture

The Central Telemetry Handler and Enhanced Telemetry Agent are implemented in Python using popular frameworks such as Flask 2.0 [77], Pika [93], and PyQt [94]. These components have been fully containerized and are packaged in separate container images using the SERRANO CI/CD services. To facilitate easy deployment on Kubernetes platforms, there are also available corresponding Kubernetes YAML description files, including Deployment, Service, and ConfigMap. These descriptions enable the automatic deployment and scaling of the Central Telemetry Handler and Enhanced Telemetry Agent services within Kubernetes environments.

In the final release, the Central Telemetry Handler and Enhanced Telemetry Agent offer comprehensive configuration options through their respective REST APIs (*PUT methods /api/v1/telemetry/central & /api/v1/telemetry/agent*). The methods enable the on-demand change of the current operational configuration of the services. More specifically, by passing a JSON description as a parameter, various operational parameters can be adjusted, as listed in the following table.

Table 13: Central Telemetry Handler and Enhanced Telemetry Agent configuration options

Parameter Name	Parameter Type	Description
active_monitoring	Boolean	Defines if the service will or will not query the registered entities.
active_notifications	Boolean	Defines if the service will or will not emit notifications related to the operation of the telemetry framework.
query_interval	Integer	Period, in seconds, for retrieving the monitoring information by each registered entity.
query_timeout	Integer	Timeout period, in seconds, for getting the requested monitoring information from a telemetry service.
data_retain_period	Integer	Maximum period, in seconds, for retaining the collected monitoring data in the operational database prior to their automatic deletion from the telemetry services.
excluded_entities	List	A list of unique identifiers for telemetry framework entities that will be excluded from automatic collection of monitoring data.

As part of our efforts to enhance system efficiency, we have significantly improved the internal workflow of the Central Telemetry Handler and Enhanced Telemetry Agent. The updated workflow facilitates seamless data collection from the SERRANO Enhanced Telemetry Agents while offering external services access to the collected information. The information includes comprehensive inventory data about available resources within the SERRANO platform and real-time monitoring data reflecting their current operational state. We implemented the required modifications within the Access Interface, Telemetry Controller, and Data Engine components. These improvements have resulted in a more robust and efficient data flow, empowering the system to better cater to the needs of our users and external services.

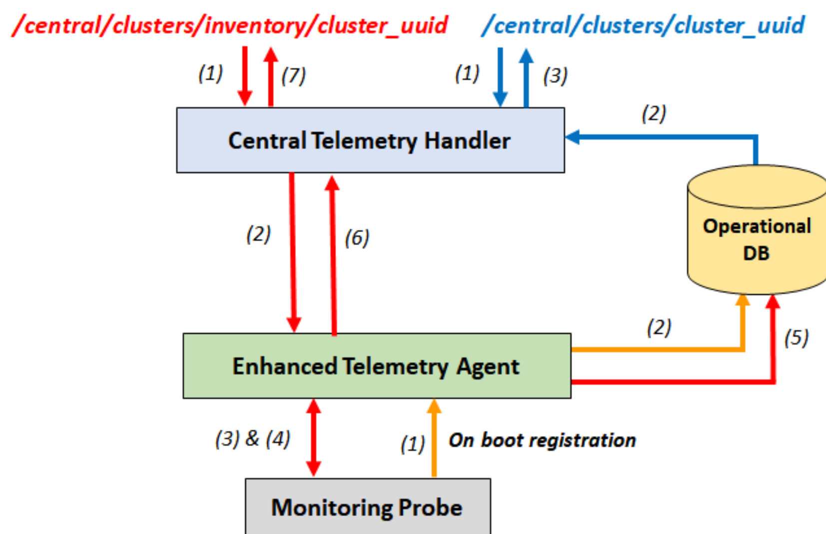

Figure 53: SERRANO telemetry framework – Inventory workflow

Figure 53 illustrates the information workflow among the telemetry components involved in retrieving inventory information for a specific edge, cloud, or HPC infrastructure within the SERRANO platform. Each SERRANO Monitoring Probe is automatically registered in a specific Enhanced Telemetry Agent. During the registration phase, a probe sends the inventory data for its type of resources, among other parameters. Next, the Enhanced Telemetry Agent

updates its internal services for the new Monitoring Probe and stores the received inventory data in the corresponding Operational Database.

The Central Telemetry Handler (CTH) offers two distinct methods to provide the requested inventory information. These methods facilitate data retrieval either by explicitly retrieving the inventory data from the appropriate Enhanced Telemetry Agent (ETA) or by directly querying the Operational Database. To this end, the CTH exposes two different methods. The first method (*GET - /api/v1/telemetry/central/clusters/cluster_uuid*) directly fetches the inventory data from the Operational Database. This workflow is represented in the diagram in blue colour. The second method (GET - */api/v1/telemetry/central/inventory/cluster_uuid*) triggers an internal procedure that notifies the corresponding Enhanced Telemetry Agent to execute the inventory operation for the specific platform. This workflow is depicted using red colour. More specifically, the CTH, using the information from the Operational Database, finds the ETA that manages the specified platform and forwards to it the inventory request (Step 2). Then, the ETA queries the respective Monitoring Probe to get the information data (Steps 3 & 4) and updates the Operational Database with the retrieved information (Step 5). Next, the ETA returns the requested inventory information to the CTH (Step 6).

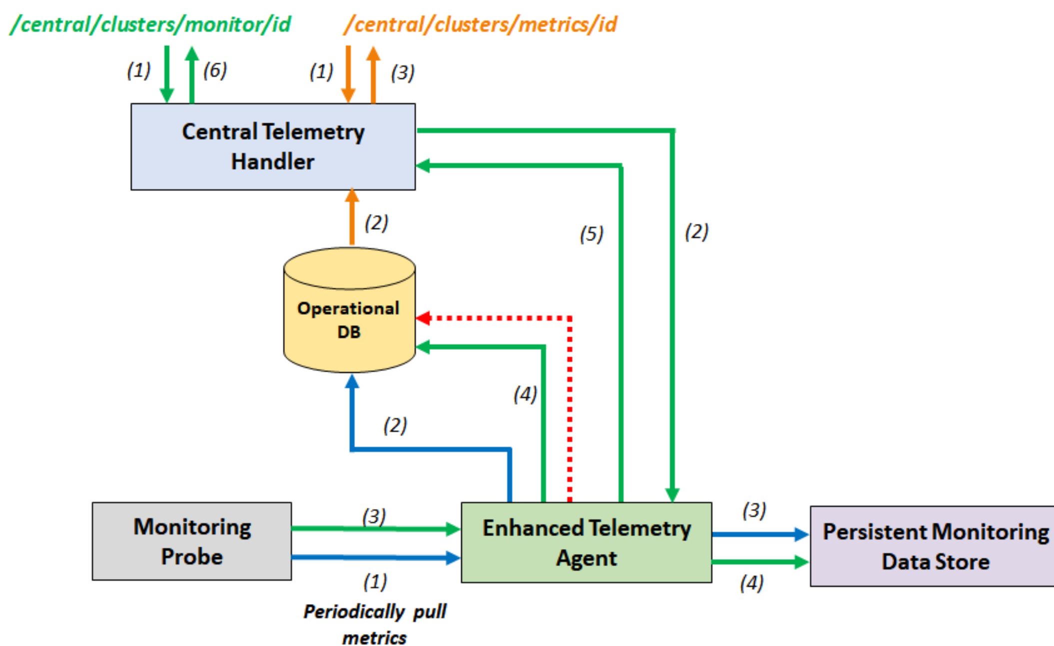

Figure 54: SERRANO telemetry framework – Monitoring workflow

Figure 54 shows the information workflow for retrieving monitoring data for a specific edge, cloud, or HPC infrastructure. The corresponding ETA automatically queries each SERRANO Monitoring Probe to provide the respective monitoring data. The ETA stores the received information in the Operational Database and the Persistent Monitoring Data Storage (PMDS). This workflow is executed over an infinite horizon and is depicted in blue colour in the above figure. Again, two distinct methods are available that provide the monitoring data either by explicitly retrieving them from the appropriate ETA or by directly querying the Operational Database. The first method (*GET - /api/v1/telemetry/central/cluster/metrics/cluster_uuid*)

directly fetches the monitoring data from the Operational Database and is represented in the diagram by an orange colour.

The second method (GET - */api/v1/telemetry/central/cluster/monitor/cluster_uuid*) provides the most up-to-date monitoring data by querying the appropriate Monitoring Probe through the corresponding ETA. In parallel, the framework automatically updates the operational database with the collected information and also stores it in the PMDS service. This workflow is illustrated with green colour. In the final release, an ETA stores only a certain number of monitoring samples in the Operational Database whose actual count is determined by the selected values in two of the supported configuration variables: the query interval and the maximum retain period. To this end, an ETA automatically removes the outdated entries, an operation represented with the red dotted line in the diagram. Furthermore, methods are available to retrieve detailed monitoring information for deployed applications within the SERRANO platform. The operation of the involved components is similar.

7.1.2 Monitoring Probes

Monitoring probes are the components of the SERRANO telemetry framework that collect valuable information about the infrastructure resources, services, and deployed applications within the SERRANO platform. Given the diverse nature of information sources targeted by the platform, the SERRANO telemetry framework relies on a collection of specialized probes, each dedicated to monitoring a specific resource type. As presented in D5.3 (M15), we adopted a single design for all SERRANO monitoring probes (Figure 55). This design not only ensures autonomous operation of the telemetry components but also facilitates seamless integration of these monitoring probes with the Data Collector component of the Enhanced Telemetry Agents and Central Telemetry Handler. This integration allows the telemetry framework to dynamically adjust the level of monitoring granularity, supporting both periodic and on-demand monitoring.

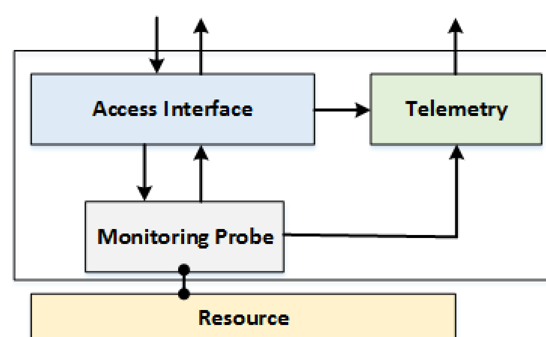

Figure 55: General architecture of SERRANO monitoring probes

During the second iteration of the implementation plan, we finished the implementation of the three different monitoring probes: Kubernetes Monitoring Probe, HPC Monitoring Probe, and SERRANO Edge Devices Monitoring Probe.

Before describing each developed monitoring probe, we present the operational workflow that implements every probe within the SERRANO telemetry framework to ensure the seamless operation of the framework. Each monitoring probe during its start-up procedure registers to some specific Enhanced Telemetry Agent using the appropriated exposed REST method (*PUT - /api/v1/telemetry/agent/register/{probe_uuid}*). The target Enhanced Telemetry Agent (ETA) is specified in the probe's configuration file, among several other operational parameters. Upon successful registration, the ETA can transparently manage and interact with the monitoring probe through the exposed REST methods (Figure 61) to collect inventory information and monitoring data. In the registration phase, the probe sends the ETA all the necessary operational information along with inventory information for the resources it monitors. Below is an example of the registration message from a SERRANO Edge Devices Monitoring Probe, along with the exchanged inventory information.


```
{"cluster_uuid": "7628b895-3a91-4f0c-b0b7-033eab309891", "probe_uuid": "5c781e1
9-344f-436e-b259-8cdf5b5eab97", "url": "https://serrano-edge-storage-probe.serv
ices.cloud.ict-serrano.eu", "type": "Probe.EdgeStorage", "inventory":[{"lat": 4
5.7472357, "lng": 21.2316107, "minio_node_disk_total_bytes": 8333520896, "name"
: "edge-storage-devices-0", "node": "serrano-k8s-worker-02", "timestamp": 16860
52883},{"lat": 45.7472357, "lng": 21.2316107, "minio_node_disk_total_bytes": 83
33520896, "name": "edge-storage-devices-1", "node": "serrano-k8s-worker-02", "t
imestamp":1686052883}]}
```


7.1.2.1 Kubernetes monitoring probe

In the SERRANO, we consider that the edge and cloud platforms that are unified under the control of the SERRANO platform are individual clusters managed by Kubernetes instances. Monitoring a Kubernetes cluster is crucial for maintaining its health and performance along with the overall stability of the SERRANO platform. To this end, the Kubernetes monitoring probes are implemented to monitor the clusters effectively.

One instance of the monitoring probe efficiently handles monitoring the resources and applications within a K8s cluster. However, our implementation also supports the on-demand deployment of additional monitoring probes, based on the instructions of the Enhanced Telemetry Agents, to ensure unhindered operation in large Kubernetes clusters. This probe is a Python-based containerized application that follows the overall architecture and operation workflow for the monitoring probes within the SERRANO telemetry framework. It also utilizes and integrates into a single solution several well-established tools to facilitate its operation, such as kube-state-metrics [97] that generates Kubernetes-specific metrics derived from the cluster's state, and Prometheus Node Exporter [99] that exposes a wide variety of hardware- and kernel-related metrics.

The probe is designed to automatically discover and monitor all available worker nodes within each Kubernetes cluster. This operation uses the Kubernetes APIs to fetch information about the available resources and Node Exporter to get the hardware metrics. It can monitor and provide both periodic and on-demand telemetry data for the following key cluster-level components:

- Nodes: Monitor the health and resource usage of worker nodes (CPU, memory, disk space, network).
- Deployments and Pods: Keep an eye on the status of pods and their resource consumption.
- Persistent Volume (PV) and Persistent Volume Claims (PVC): Monitor storage capacity, utilization, and any error conditions related to storage provisioning.

Beyond cluster-level monitoring, the monitoring probe autonomously monitors the cloud-native applications deployed across the SERRANO heterogeneous and distributed resources (Section 9.4.2). The probe also collects application-specific metrics exposed through custom endpoints. It can also dynamically collect relevant performance metrics on a per-function invocation basis for the SERRANO-accelerated kernels that are executed as short-lived applications (Sections 9.4.3 and 10.5). Figure 56 illustrates the interactions among the SERRANO orchestration mechanisms and the telemetry framework components so as to autonomously collect the most up-to-date telemetry data for the deployed cloud-native and short-lived applications, regardless of the individual platforms that host them. This operation is critical in order to ensure that the envisioned SERRANO continuous control loop mechanisms will always be able to adjust resources and migrate workload based on feedback regarding the application’s state.

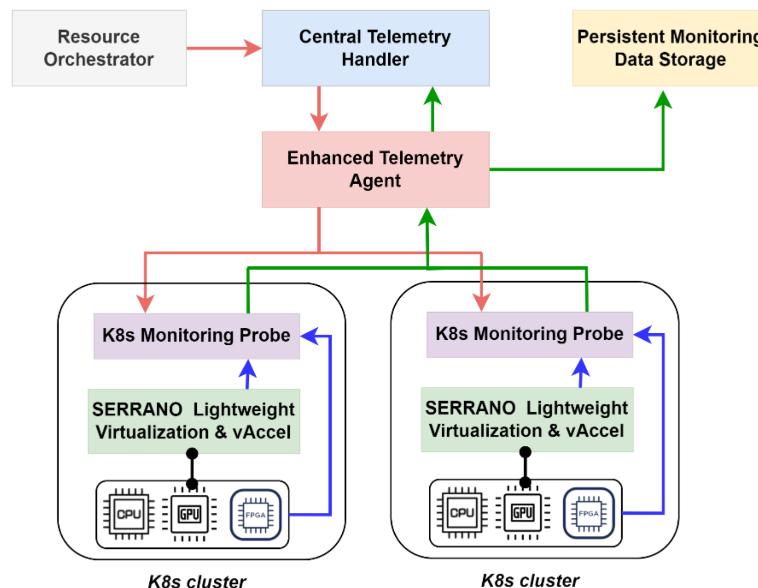

Figure 56: Autonomous monitoring of deployed cloud-native and short-lived applications

7.1.2.2 HPC monitoring probe

The HPC monitoring probe is a Python-based implementation that adheres to the overall architecture and design principles established for all SERRANO monitoring probes. This approach ensures consistency and ease of integration within the telemetry framework. The HPC monitoring probe implements all the necessary interfaces (Figure 61) to enable seamless communication with other components of the telemetry framework. Moreover, the HPC monitoring probe is containerized to ensure portability and scalability, facilitating its straightforward deployment and management in various environments.

The HPC monitoring probe interacts with the SERRANO HPC Gateway, utilizing its exposed REST methods for data exchange. By leveraging the HPC Gateway REST API, the monitoring probe efficiently collects inventory information and continuously monitors the status of HPC resources and the execution progress of deployed jobs. The integration with the HPC Gateway involves utilizing specific REST methods (Figure 85) that facilitate the retrieval of essential metrics, performance data, and status updates from the HPC infrastructure and currently running jobs. This integration empowers the telemetry framework to capture real-time insights into the HPC system's health, resource utilization, and job execution efficiency.

Figure 57 showcases an example of the collected resource descriptions and performance monitoring parameters for the SERRANO HPC platform.


```
{
  "hpc_monitoring_data": {
    "name": "excess_slurm",
    "partitions": [
      {
        "avail_cpus": 160,
        "avail_nodes": 2,
        "name": "profile",
        "queued_jobs": 0,
        "running_jobs": 0,
        "total_cpus": 160,
        "total_nodes": 2
      }
    ],
    "scheduler": "slurm"
  },
  "type": "Probe.HPC",
  "uuid": "078de729-9362-4920-af87-529b8bd329c3"
}
```


Figure 57: Monitoring data collected by SERRANO HPC monitoring probe

7.1.2.3 SERRANO edge storage devices monitoring probe

The SERRANO edge storage devices offer decentralized storage locations at the network edge while providing an S3 interface for seamless data access. These devices are containerized applications built upon the MinIO [103], a high-performance, highly customizable object storage solution. Deployment of these SERRANO edge devices is efficiently managed by SERRANO's orchestration mechanisms, which integrate smoothly with various Kubernetes platforms. For detailed information about these SERRANO-enhanced devices, refer to deliverables D3.2 (M15) and D3.4 (M30).

To this end, the monitoring probe utilizes the MinIO server's capabilities in exposing inventory and monitoring data over Prometheus-compatible endpoints. The monitoring probe uses these endpoints to collect information about the current state of the SERRANO edge storage devices. The probe is designed to automatically discover and monitor all available edge storage devices within each Kubernetes cluster. One instance of the monitoring probe efficiently handles monitoring for all SERRANO edge storage instances in a K8s cluster. This is achieved by querying the available pods within the Kubernetes cluster and scanning for specific labels associated with SERRANO edge storage device instances. Following this approach, the monitoring probe dynamically determines the number of SERRANO edge storage devices in each Kubernetes cluster, allowing it to seamlessly collect inventory and monitoring information. The probe is a containerized application implemented in Python.

7.1.3 Operational Database

The SERRANO telemetry framework also includes a number of operational databases that store information related to the deployed components of the framework along with their configuration and relationships. Moreover, it includes the most up-to-date information for the available infrastructure resources, their current state, and details about the deployed applications and executed SERRANO-accelerated kernels. These databases are based on MongoDB [102], an open-source document-oriented database that stores data in flexible format JSON-like documents. The primary role of the operational databases is to facilitate the autonomous operation of the SERRANO telemetry framework components and provide the most up-to-date monitoring data. To this end, to retain historical analytical data to feed the various AI/ML-based decision mechanisms within the SERRANO platform, the Persistent Monitoring Data Storage (PMDS) service is also available, described in Section 7.4.

The Central Telemetry Handler stores details for the Enhanced Telemetry Agents it manages in its operational database, along with high-level details about the available resources in the overall SERRANO platform and the deployed applications. The information is organized into four primary documents: *entities, infrastructure, serrano_state_metrics, and serrano_deployments.* The first document facilitates the operation of the telemetry framework and provides details for the deployed Enhanced Telemetry Agents within the SERRANO platform as well as the Monitoring Probes registered to each Enhanced Telemetry Agent. Below is an example of the available information for one Monitoring Probe for a Kubernetes cluster and one Enhanced Telemetry Agent.


```
_id: ObjectId('6381ceb1569d6e2826fe9688')
uuid: "25042d77-041f-4e80-88b0-182fce441661"
type: "Probe.k8s"
url: "http://serrano-k8s-probe-service:9080"
cluster_uuid: "3984f92a-21a0-4ce5-85a4-7febd261b794"
timestamp: 1669997815
```

```
_id: ObjectId('6380ce9b9d5f9532a1dd5b1a')
uuid: "4790a1e3-aa38-4461-8b81-8f771c60dbb5"
type: "Agent"
url: "http://85.120.206.26:30090"
> probes: Array
timestamp: 1674034349
```


The *infrastructure* document provides a high-level description of the capabilities of the available computational and storage resources in each edge, cloud, and HPC infrastructure within the SERRANO platform. The *serrano_state_metrics* includes high-level monitoring data for the usage of the available resources, with less granularity and details compared to the corresponding information available in the operational database for each Enhanced Telemetry Agent. Finally, the objects in *serrano_deployments* keep track of the applications' deployments and their allocation within the SERRANO platform. The Central Telemetry Handler manages the contents of its operational database exclusively by interacting with the available Enhanced Telemetry Agents.

Similarly, the information in the operational database of an Enhanced Telemetry Agent is organized into seven documents: *entities*, *clusters*, *cluster_deployment_metrics*, *cluster_state_metrics*, *edge_storage*, *edge_storage_metrics*, serrano_kernels_metrics. The *entities* document provides details for the available Monitoring Probes, Objects in *clusters* and *cluster_state_metrics* provide details for the inventory and monitoring information of the edge, cloud, and HPC platforms controlled by the Enhanced Telemetry Agent. The *edge_storage* and *edge_storage_metrics* documents collect information for SERRANO

edge storage devices. The *cluster_deployment_metrics* and *serrano_kernels_metrics* documents store the most up-to-date monitoring information for the applications and SERRANO-accelerated kernels that are deployed in the part of the SERRANO platform that the Enhanced Telemetry Agent monitors.

7.1.4 Deployment of telemetry services and data visualization

Figure 58 presents the SERRANO telemetry framework services deployment in the project integration testbed. During the preparation of the deliverable, the deployment included the Central Telemetry Handler, two Enhanced Telemetry Agents, one probe for each of the two available Kubernetes clusters, one probe for the HPC platform, one probe for the SERRANO edge storage devices, and the Persistent Monitoring Data Storage service. All the SERRANO telemetry framework services have been deployed using the defined Kubernetes YAML description files and the corresponding container images.

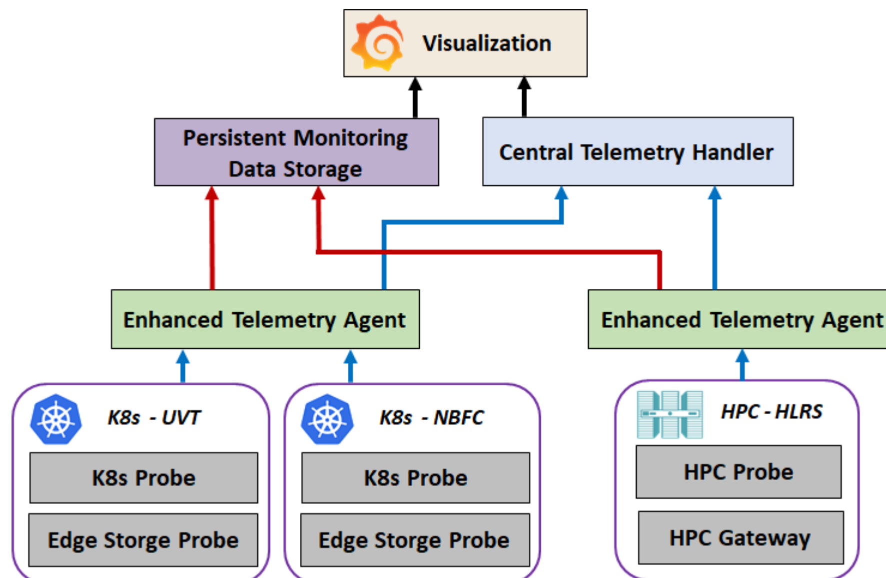

Figure 58: SERRANO telemetry framework deployment in project integration testbed

Moreover, the telemetry framework has been extended to visualize the collected inventory and monitoring information from the SERRANO platform through a web-based user interface. To this end, the final release includes a visualization module based on Grafana [97], an open-source analytics and interactive visualization web application with charts and graphs. We created several custom dashboards to visualize inventory and monitoring data retrieved from the Central Telemetry Handler and the Persistent Monitoring Data Storage service. The visualization module enables data aggregation and filtering by time range, workload, and infrastructure. Figure 59 presents an example of the provided information.

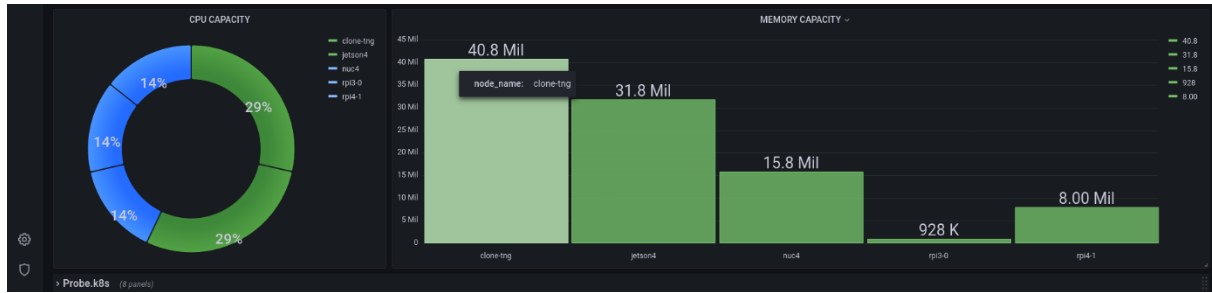

Figure 59: Memory usage for a selected worker node in the NBFC K8s cluster

7.2 Inventory and telemetry parameters

The SERRANO telemetry framework automatically discovers and monitors heterogeneous resources and deployed applications in edge/cloud and HPC platforms. Through the Operational Database, the framework maintains a detailed catalogue (i.e., inventory data) with the available computational and storage resources along with their capabilities and characteristics within the individual edge, cloud, and HPC platforms that constitute the SERRANO platform. Moreover, it constantly monitors the available resources and automatically gathers performance monitoring data (i.e., telemetry data) for all the deployed applications and executed SERRANO-accelerated kernels.

The orchestration and service assurance mechanisms leverage the collected inventory and telemetry data to improve the orchestration, deployment, and re-optimization decisions over time, depending on the status of the system as well as previous executions of the applications and kernels. To this end, the appropriate monitoring and telemetry data is collected by five main categories of resources: (i) computational and storage resources in edge/cloud platforms, (ii) HPC hardware resources, (iii) SERRANO-enhanced hardware resources (e.g., multi-level approximate hardware accelerators), (iv) SERRANO-enhanced software resources (e.g., SERRANO edge devices, on-premise storage gateway, lightweight virtualization, hardware acceleration abstractions) and (v) deployed cloud-native and short-lived (serverless) applications.

Compared to the initial version of the telemetry framework, we extend the monitoring parameters for the cloud and edge storage locations. Moreover, we updated and restructured the inventory and monitoring parameters for the edge and cloud computational resources to facilitate their more efficient handling by the PMDS service. In addition, we expanded the telemetry parameters that are automatically collected for the deployed applications and SERRANO-accelerated kernels. Finally, the telemetry framework is now able to collect application-specific parameters. Figure 60 summarizes the resource description and monitoring parameters that collect the final version of the SERRANO telemetry framework.

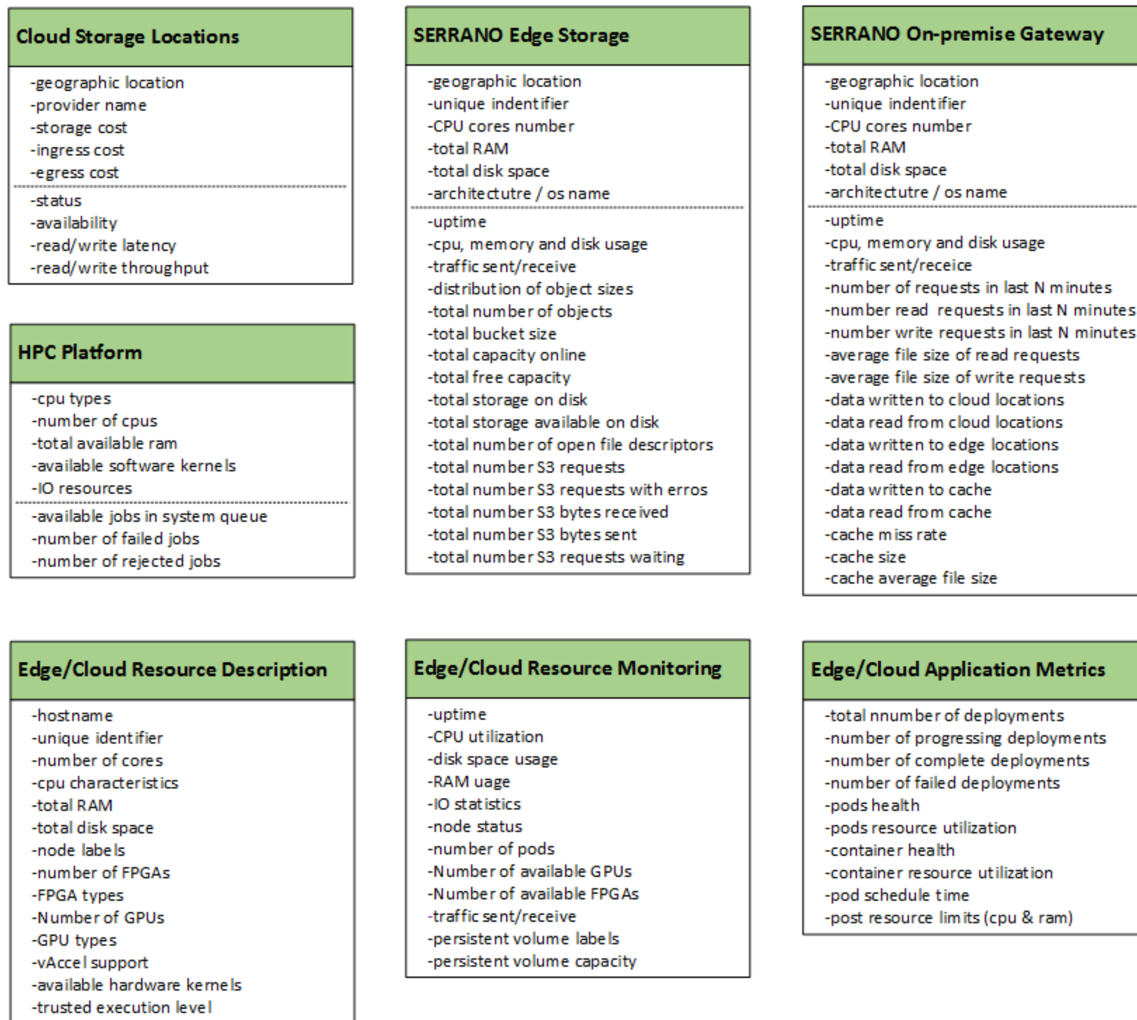

Figure 60: Collected inventory and monitoring parameters in the SERRANO platform

7.3 Telemetry interfaces

Regarding the telemetry interfaces, we have made significant enhancements to the functionality of the Access Interface, as well as the exposed REST API in Central Telemetry Handler, Enhanced Telemetry Agent, and available Monitoring probes. These new developments greatly facilitate the control and management of telemetry entities across the hierarchical infrastructure while enabling automatic monitoring of deployed cloud-native applications and SERRANO-accelerated kernels throughout the unified SERRANO platform.

The final version of the REST API exposed by the SERRANO telemetry framework includes several methods organized into two main categories. The first set of methods allows other SERRANO services (such as the AI-Enhanced Service Orchestrator, Event Detection Engine, Resource Optimization Toolkit, and Resource Orchestrator), end users, and even third-party applications to easily interact with the SERRANO telemetry framework via the Central Telemetry Hander. The second set includes methods exposed by the main components of the telemetry framework, supporting their operation, configuration, and exchange of inventory and telemetry data within the hierarchical telemetry architecture. Figure 61 and Figure 62

provide a summary of the final version of the RESTful API for the SERRANO telemetry framework.

Furthermore, the telemetry framework facilitates the exchange of events about the framework itself and the status of resources among the various components of the hierarchical telemetry infrastructure. The SERRANO Data Broker component enables this functionality by providing a dedicated topic exchange. An exchange serves as a messaging routing mechanism capable of supporting various routing logics. In this case, messages published by the Notification Engine component in the Enhanced Telemetry Agents or Central Telemetry Handler are associated with a routing key adhering to a predefined syntax. These messages are presented in JavaScript Object Notation (JSON) format. A detailed description of the specific topic exchange and the structure of all available notification messages is available in deliverable D5.3 (M15).

Monitoring Probes

GET /api/v1/telemetry/probe/monitor Get monitoring data from a specific probe.
GET /api/v1/telemetry/probe/inventory Get resource description parameters from a specific probe.
POST /api/v1/telemetry/probe/collection Configure data collection operation for a specific probe.
DELETE /api/v1/telemetry/probe/streaming/{sessionId} Configure data collection operation for a specific probe.

Central Telemetry Handler / Enhanced Telemetry Agent

POST /api/v1/telemetry/agent/register Registration of telemetry instance (i.e., Monitoring Probe, Enhanced Telemetry Agent) at a specific telemetry entity.
DELETE /api/v1/telemetry/agent/register/{uuid} Remove registered telemetry instance from a specific telemetry entity.
GET /api/v1/telemetry/agent/register/{uuid} Get details about a telemetry instance.
PUT /api/v1/telemetry/agent/register/{uuid} Update the configuration parameters of a registered telemetry instance.
POST /api/v1/telemetry/agent/streaming Indicates that a streaming session is available with a specific probe.
GET /api/v1/telemetry/agent/monitor/{uuid} Get monitoring data from a specific probe.
GET /api/v1/telemetry/agent/inventory/{uuid} Get resource description parameters from a specific probe.
GET /api/v1/telemetry/agent/entities Get the registered telemetry entities at the specific agent.

Figure 61: Telemetry framework REST interfaces – Control and management methods

Central Telemetry Handler

GET /api/v1/telemetry/central Get current configuration parameters.

PUT /api/v1/telemetry/central Change current configuration parameters.

GET /api/v1/telemetry/central/inventory List the available platform level telemetry entities.

GET /api/v1/telemetry/central/clusters List the available K8s and HPC clusters.

GET /api/v1/telemetry/central/clusters/{uuid} Get description parameters for a specific cluster.

GET /api/v1/telemetry/central/clusters/inventory/{uuid} Get resource description parameters for a specic cluster.

GET /api/v1/telemetry/central/clusters/monitor/{uuid} Get monitoring data for a specific cluster.

GET /api/v1/telemetry/central/clusters/metrics/{uuid} Get monitoring data for a specific cluster from the Operational Database.

GET /api/v1/telemetry/central/deployments/monitor/{uuid} Get monitoring data for a specific application deployment.

GET /api/v1/telemetry/central/kernels/monitor/{uuid} Get monitoring data for a specific SERRANO-accelerated kernel execution.

Figure 62: Telemetry framework REST interfaces – High-level CTH methods

7.4 Persistent Monitoring Data Storage

A fundamental piece of supporting an effective monitoring and orchestration pipeline is the availability of a central repository to retain historical telemetry data for the current state of the heterogeneous resources and deployed applications to feed the various AI/ML-based decision mechanisms within the SERRANO platform.

To this end, the SERRANO platform includes the Persistent Monitoring Data Storage (PMDS) service. The PMDS acts as long-term storage for the collected timestamped telemetry data that provides historical data to the SERRANO orchestration and service assurance mechanisms. It is based on InfluxDB [101], an open-source time-series database. InfluxDB provides fast, highly available storage for time-series data and can also be used as a data source for many other solutions, such as the Grafana, an open-source analytics and interactive visualization web application.

Figure 63 presents the PMDS architecture. The service is implemented in Python using the Flask 2.0 and PyQt frameworks. The Access Interface component is a REST controller that exposes methods that allow end users and external services to retrieve historical telemetry data. The available REST methods are depicted in Figure 64. The Data Engine implements the interaction with the InfluxDB by abstracting the required data manipulation for retrieving the telemetry data according to the requested parameters. It is also responsible for properly updating the stored information.

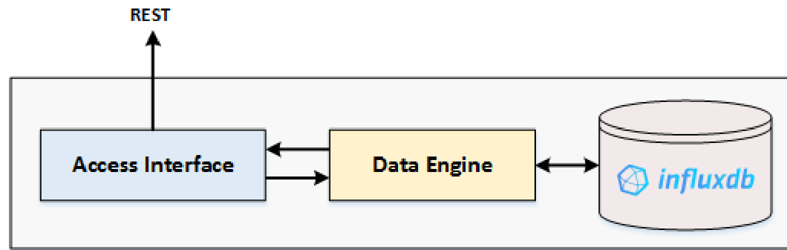

Figure 63: Persistent Monitoring Data Storage (PMDS) architecture

PMDS

Method	Endpoint	Description
GET	/api/v1/pmds/nodes/{cluster_uuid}	Retrieve historical telemetry data for the worker nodes within a K8s cluster.
GET	/api/v1/pmds/pvs/{cluster_uuid}	Provide historical telemetry data for the available Persitent Volumes (PVs) within a K8s cluster.
GET	/api/v1/pmds/deployments/{cluster_uuid}	Provide historical telemetry data for the available Deployments within a K8s cluster.
GET	/api/v1/pmds/pods/{cluster_uuid}	Provide historical telemetry data for the available Pods within a K8s cluster.
GET	/api/v1/pmds/edge_storage_devices/{cluster_uuid}	Provide historical telemetry data for the available SERRANO Edge Storage devices within a K8s cluster.
GET	/api/v1/pmds/serrano_deployments/{deployment_uuid}	Provide historical telemetry data for a specific application deployed through the SERRANO orchestration mechanisms.
GET	/api/v1/pmds/kernels/{deployment_uuid}	Provide historical telemetry data for a specific SERRANO-accelerated kernel execution.
POST	/api/v1/pmds/nodes	Store historical telemetry data for the worker nodes within a K8s cluster.
POST	/api/v1/pmds/pvs	Store historical telemetry data for the available Persitent Volumes (PVs) within a K8s cluster.
POST	/api/v1/pmds/deployments	Store historical telemetry data for the available Deployments within a K8s cluster.
POST	/api/v1/pmds/pods	Store historical telemetry data for the available Pods within a K8s cluster.
POST	/api/v1/pmds/edge_storage_devices	Store historical telemetry data for the available SERRANO Edge Storage devices within a K8s cluster.
POST	/api/v1/pmds/serrano_deployments	Store historical telemetry data for a specific application deployed through the SERRANO orchestration mechanisms.
POST	/api/v1/pmds/kernels	Store historical telemetry data for a specific SERRANO-accelerated kernel execution.

Figure 64: Persistent Monitoring Data Storage (PMDS) RESTful interface

We have developed a Python API to enhance the interaction with the PMDS service, simplifying the utilization of its RESTful interface. This API encapsulates the necessary functionality to query the PMDS service endpoints, offering a range of filtering parameters. To clarify the available options, we have compiled a comprehensive summary of the supported parameters for each Python method in the following table. Additionally, Figure 65 displays part of the provided telemetry for a specific worker node (*"serrano-k8s-worker-02"*) within

the SERRANO UVT K8s cluster. The figure demonstrates the telemetry data presented in both formatting options, enabling users to choose the format that suits their needs.

Furthermore, as part of our ongoing development efforts, we plan to seamlessly integrate the PMDS Python API into the final version of the SERRANO SDK. This integration will further enhance the capabilities of the SDK, providing users with a comprehensive solution for interacting with the PMDS service.

Table 14: PMDS Python API – Available input parameters

Category	Name	Description
Timeframe options *Accepted formats: relative duration (e.g., -30m, -1h, -1d) or Unix timestamp in seconds (e.g., 1644838147).*	start	Earliest time to include in results, by default is the last 24 hours.
	stop	Latest time to include in results. Default is now().
Required parameters	cluster_uuid	Determines the K8s cluster.
	namespace	Determines the target namespace.
Filtering parameters	node_name	Limits the results only for data related to the specified node name.
	field_measurement	Limits results only for the selected parameter.
	name	Limits results only for the selected service.
	group	Valid for querying telemetry data for available worker nodes within a K8s cluster. Limits results to parameters from the selected group. Supported values: ***"general"***, ***"cpu"***, ***"memory"***, ***"storage"***, ***"network"***. If not specified the "general" metrics will be returned.
Filtering parameters	format	Determines the format of the response. Supported values ***"raw"*** and ***"compact"***. - ***raw***: provides data in a time series way - ***compact***: organizes data at a per target parameter basis.

```
[
  {
    "_field": "node_memory_Buffers_bytes",
    "_time": "2023-07-04T09:08:25.216700+00:00",
    "_value": 1054969856,
    "group": "memory",
    "node_name": "serrano-k8s-worker-01"
  },
  {
    "_field": "node_memory_Cached_bytes",
    "_time": "2023-07-04T09:08:25.216700+00:00",
    "_value": 30781259776,
    "group": "memory",
    "node_name": "serrano-k8s-worker-01"
  },
  {
    "_field": "node_memory_MemAvailable_bytes",
    "_time": "2023-07-04T09:08:25.216700+00:00",
    "_value": 59630399488,
    "group": "memory",
    "node_name": "serrano-k8s-worker-01"
  },
  {
    "_field": "node_memory_MemFree_bytes",
    "_time": "2023-07-04T09:08:25.216700+00:00",
    "_value": 24235716608,
    "group": "memory",
    "node_name": "serrano-k8s-worker-01"
  },
  {
    "_field": "node_memory_MemTotal_bytes",
    "_time": "2023-07-04T09:08:25.216700+00:00",
    "_value": 67436380160,
    "group": "memory",
    "node_name": "serrano-k8s-worker-01"
  },
  {
    "_field": "node_memory_MemUsed_bytes",
    "_time": "2023-07-04T09:08:25.216700+00:00",
    "_value": 43200663552,
    "group": "memory",
    "node_name": "serrano-k8s-worker-01"
  },
  {
    "_field": "node_memory_usage_percentage",
    "_time": "2023-07-04T09:08:25.216700+00:00",
    "_value": 64.06,
    "group": "memory",
    "node_name": "serrano-k8s-worker-01"
  }
]
```


(Raw format)


```
{
  "serrano-k8s-worker-01": [
    {
      "node_memory_Buffers_bytes": 1054969856,
      "node_memory_Cached_bytes": 30781259776,
      "node_memory_MemAvailable_bytes": 59630399488,
      "node_memory_MemFree_bytes": 24235716608,
      "node_memory_MemTotal_bytes": 67436380160,
      "node_memory_MemUsed_bytes": 43200663552,
      "node_memory_usage_percentage": 64.06,
      "time": "2023-07-04T09:08:25.216700+00:00"
    }
  ]
}
```


(Compact format)

Figure 65: PMDS Python API – Historical telemetry data for a specific worker node within a K8s cluster

The PMDS service and its configuration file are packaged as Python applications using the SERRANO CI/CD services. We also defined all the required Kubernetes YAML description files (i.e., ConfigMap, Deployment, Services, Ingress) to facilitate its deployment in Kubernetes. Figure 66 shows the PMDS deployment in the primary SERRANO Kubernetes cluster, which the UVT provides in the project.

```
Name:                   serrano-pmds
Namespace:              integration
CreationTimestamp:      Thu, 08 Jun 2023 12:44:19 +0300
Labels:                 <none>
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=serrano-pmds
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=serrano-pmds
  Containers:
   serrano-pmds:
    Image:      ictserrano/serrano:pmds-v1.2
    Port:       10030/TCP
    Host Port:  0/TCP
    Command:
      python3
    Args:
      /home/serrano/PMDSInstance.py
    Environment:  <none>
    Mounts:
      /etc/serrano from serrano-pmds-config (rw)
  Volumes:
   serrano-pmds-config:
    Type:      ConfigMap (a volume populated by a ConfigMap)
    Name:      serrano-pmds-config
    Optional:  false
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   serrano-pmds-7877f45b47 (1/1 replicas created)
Events:          <none>
```

Figure 66: PMDS deployed in main SERRANO Kubernetes cluster

7.5 Identifying Network Congestion Using Knowledge Graphs and Link Prediction

Data and computation distribution approaches through the use of content distribution networks and edge computing have alleviated part of the network load in core networks. However, the growth of Internet of Things (IoT) devices and the diversification of data sources, significantly affects the efficiency and reliability of communication networks that interconnect these distributed storage and computing units. As a result, there is an increasing need for smart, adaptive network management solutions that observe, decide, and act in real-time. Such solutions would allow for proactive and reactive adjustments in network traffic management, based on current and predicted link status.

In what follows, we propose a methodology driven by machine learning that enables the prediction of potential network congestion events (i.e., over-usage of network links). We implement a graph-based network representation method to encapsulate both topological and traffic-related information for each node into vector embeddings. Subsequently, we use link prediction methods on these generated embeddings to identify patterns that may identify potential network congestion and require preventative measures. The application of knowledge graphs in the modelling of communication networks has been limited. However, leveraging a Knowledge Graph (KG) to model network infrastructures is crucial for capturing

the full state of each link in the network. This holistic approach differs significantly from other Machine Learning (ML) applications, which tend to focus solely on individual node properties without accounting for their interconnections. Our approach provides a more comprehensive representation of the network, capturing not just the state of each node, but also the relationships between nodes. The use of graph embeddings allows us to transform these network entities (nodes, links) into fixed-length vectors that represent the network in a low-dimensional space, preserving its topology. This approach facilitates the application of traditional data-driven ML algorithms for predicting congestion events without neglecting the topological information of each sample, a common limitation of many ML-based approaches.

7.5.1 Previous Work

Communication networks are highly dynamic, complex systems with intricate dependencies between their elements, which often make traditional ML models inadequate [53]. Moreover, these models typically require extensive, annotated data for training, which is not always available or feasible to obtain in real-world scenarios. Recently, graph-based models, particularly Graph Neural Networks (GNNs), have emerged as a powerful approach for dealing with complex, interconnected data like that found in communication networks. GNNs have been successful in a variety of applications, including recommendation systems, social network analysis, and bioinformatics [54]. However, their application in the domain of network management and specifically in the context of routing and traffic optimization, is relatively new and largely unexplored [55][56].

In particular, the use of GNNs for predicting potential network congestion points, a critical aspect of traffic management, has shown promising results. Graph-based link prediction algorithms, can be used to predict potential network congestion points by learning the underlying patterns of network traffic and predicting when and where link utilization might exceed a certain threshold [57]. The integration of GNNs with anomaly detection methods to identify anomalous network events, such as processing and memory failures, is a promising research direction [58][59][60]. The convergence of graph-based models and ML techniques provides a robust framework for automated network management. Knowledge Graphs (KG) can be a natural platform for integrating multi-modal data from heterogeneous sources, enabling representation and reasoning about their in-between dependencies and relationships [61]. Despite the adoption of KG approaches, it is worth mentioning that while most optimal resource allocation problems are typically modelled as graph problems [62][63], they are usually solved using queuing theory [64], Q-learning [65][66] or via traditional ML methods [66].

To the best of our knowledge, no previous work leverages an end-to-end GNN implementation by combining graph-based embeddings of network topologies with KG-based event detection via link prediction. Combining KGs with GNNs can provide a comprehensive and scalable solution to the complex problem of network management, with the ability to model the network’s topology and predict potential network congestion points or anomalies.

7.5.2 Proposed Methodology

Our approach materializes an ML-driven pipeline that utilizes knowledge graphs and link prediction, both graph oriented and data-driven, in order to address the traffic management needs. Recurring inference is a feature provided by the implemented pipeline, hence continuous control is feasible. Knowledge graphs comprise a powerful combination of intuitive representation of a network and prompt exploitable information. The former is a necessity for human operators, while the latter is prerequisite for producing quantifiable insights that can be leveraged by state-of-the-art data-driven approaches.

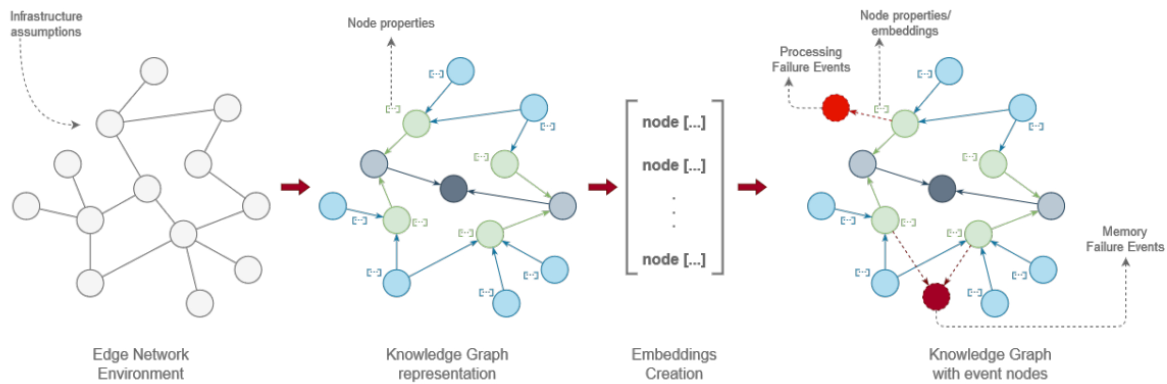

Figure 67: Overview of proposed KG-based modelling and event detection methodology

Our methodology comprises four primary steps, as illustrated in Figure 67. Initially, to address the challenge of limited real-world data availability, we use a simulated network topology and communication network infrastructure assumptions as inputs. Secondly, we convert this simulated infrastructure into a knowledge graph representation. This graph encompasses all the key infrastructure entities (datacentre nodes, routers, network nodes) and their relationships, encapsulating the network's overall structure and behaviour in a graph-based model. In the third step, we extract topological embeddings from this knowledge graph. Finally, instead of employing traditional unsupervised learning algorithms, we use the link prediction approach on the extracted embeddings to connect problematic network parts with relevant event nodes. The final output of our methodology is the predicted links that represent potential network congestion, effectively enabling proactive traffic management and optimization.

The network topology consists of four types of vertices: Datacentres, Routers, Subnetwork Nodes, and Exchange Points. The vertices are connected pairwise with different types of edges. Specifically, a Datacentre node is connected with a Router node via an edge of type *"data packet"*, which also carries two properties: total processing and memory usage. A Router node is connected to a Subnetwork node via an edge of type *"regional connection"* and a Subnetwork node is connected to an Exchange Point node via an edge of type *"backbone connection"*.

The assumptions made during the generation of the above infrastructure are listed below:

- A Datacentre node sends many data packets, but each data packet originates from one Datacentres node.
- Each Router node processes many data packets, but a data packet is processed by a single Router node. A Subnetwork node connects multiple Router nodes, and similarly a Router node is associated with a multiple Subnetwork node.
- The Exchange Point nodes are operating in a distributed manner, making it possible for them to handle traffic from multiple Subnetwork nodes.
- The data packets can traverse multiple paths based on the network's topology and current traffic conditions.

To support our experiments, we simulated two types of events within the infrastructure, each having 40 occurrences. The *"Processing Failure"* event encompasses cases where the cumulative processing power demanded by data packets processed at a Router node exceeds its capacity. Similarly, the *"Memory Failure"* event incorporates cases where the cumulative memory demanded by the data packets overflows the Router node's memory capacity. To this end, we assumed that the requested processing power and memory could exceed the provided capacity by up to thirty percent. This reflects a scenario where the network traffic surges unexpectedly, causing stress on the system and possibly leading to service degradation or failure.

7.5.2.1 Knowledge Graph Representation

We leveraged the simulated infrastructure of the previous step to populate a Knowledge Graph (KG) that can be queried using the Cypher query language [29] [30]. Each corresponding entity of the modelling step is assigned to a different node type, while their in-between relationships are represented as different edge types in the graph.

Overall, we represented the connection of 3200 Datacentre nodes with 2400 Routers via 9660 *"data packet"* relationships. These Routers are connected to 120 different Subnetwork Nodes through 3600 *"regional connection"* relationships. Finally, the Subnetwork Nodes are allocated to 80 Exchange Points via 240 *"backbone connection"* relationships. It is also noted that each Router includes a total processing usage and a total memory usage property, while each *"data packet"* relationship contains a computing demand and data size property, denoting the upcoming infrastructure needs.

Apart from the infrastructure-type nodes, the KG was enriched with two event-type nodes: Processing Failure Event node and Memory Failure Event node, corresponding to cases of processing and memory over usage of Router nodes, respectively. Given that link prediction relies on training data derived from a subset of edges that have ground-truth labels to predict similar connections on unseen data, the simulated instance encompassed 80 Router nodes with over usage properties: 40 of them with processing over usage and 40 of them with memory over usage. However, only 30 Router nodes of each type were connected to their corresponding event node, serving as a training set for the link prediction algorithm. The remaining edges were intentionally excluded in order to be predicted by the link prediction

model. The virtual graph that represents all node labels and relationship-types available in the above described knowledge graph is shown in Figure 68.

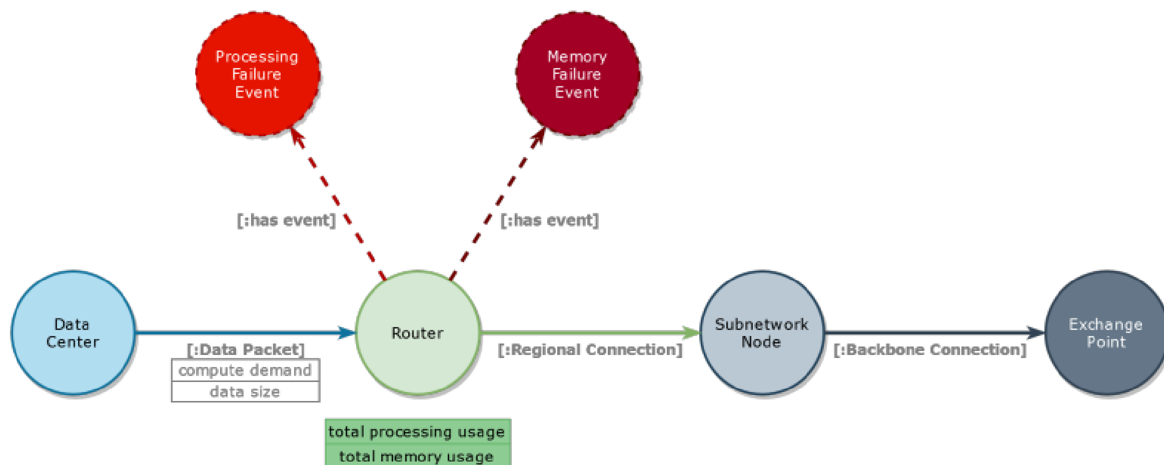

Figure 68: Knowledge graph meta-graph

7.5.2.2 Creation of Graph Embeddings

The graph created in the previous phase offers an insightful way to model our simulated communication infrastructure without relying on fixed-size representations. We employed a graph embedding method to transform graph entities (nodes, edges) into vectors in a low-dimensional space while preserving the graph's topology.

We used GraphSAGE [69], a neural-based graph embedding method, to generate predictive representations through unsupervised learning by sampling and aggregating features from a node's local neighbourhood using random walks. Instead of training standalone embedding vectors for each node, it trains a set of aggregator functions that combine feature information from its closest neighbours. This enables the simultaneous learning of the topological structure of the node's neighbourhood and the distribution of the node's features within it.

For each node $v \in V$ of the sub-graph, GraphSAGE creates a tree that has as root the corresponding node. The depth of the tree equals the defined search depth K inside the graph, whereas the children of each tree node are its adjacent nodes in the graph. In order to keep the computational footprint of each batch fixed, instead of using the full immediate neighbourhood sets, a fixed-size uniform sampling is performed and a sample of the immediate neighbours is leveraged. Step 1 in Figure 69 depicts the random tree of depth 3 created for a node of the graph. The aforementioned tree is then utilized by the aggregation functions in order to create the embeddings of the root node. The procedure is described below.

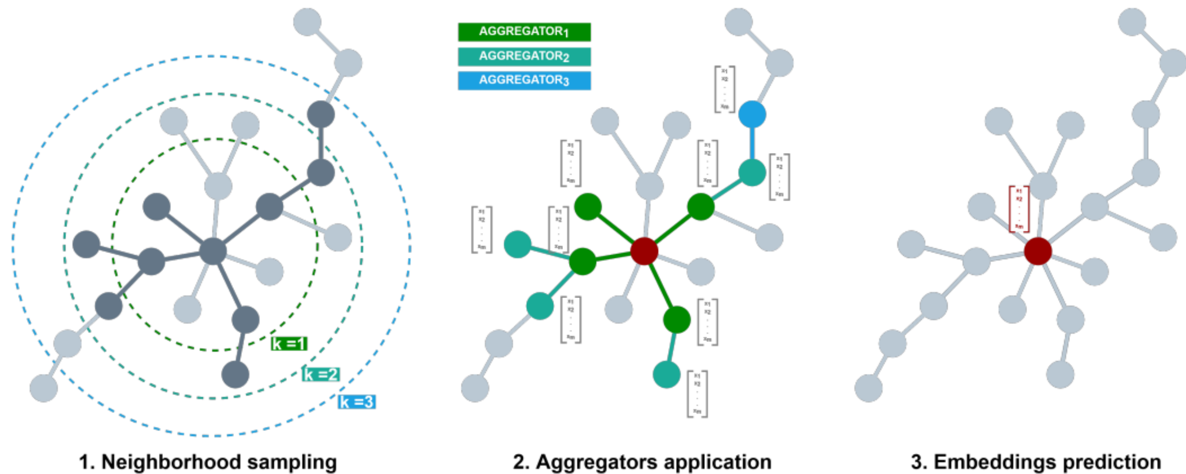

Figure 69: The three-step process of the GraphSAGE inductive representation method.

The algorithm follows an iterative approach, where the initial node representations (i.e., original node features) are updated based on the network topology and their neighbours' features. Specifically, given a target node and a defined range of search depths K, for each $k \in 1, ..., K$, the algorithm updates the representations of the nodes based on their immediate neighbours (i.e., the nodes in the $(k-1)^{th}$ layer are updated based on the features of the nodes in the kth layer). As shown in Step 2 of Figure 69, for $k = 1$, the representation of the red node will be updated with the aggregated information derived from its green neighbour nodes, for $k = 2$ the representation of the green nodes will be updated by that of their turquoise neighbour nodes, and so on. Finally, the target node (red) representation is derived from the aggregated updated representations of its immediate neighbours into a single vector (Step 3 of Figure 69). This vector representation forms the final embeddings for the target node; this operation is repeated for every node in our graph.

It should be noted that the update process comprises the following operations: First, the neighbourhood representation for each node is calculated by aggregating the previous representations of its immediate neighbours, using one of the aggregator architectures described in the following paragraph. Second, this neighbourhood representation is concatenated with the node's previous representation, and finally, this concatenated vector is fed through a fully connected layer with a nonlinear activation function, which transforms the representation to a fixed size (Figure 69). Hence, as the process iterates through search depths, nodes incrementally gain more and more information from further reaches of the graph. For $k = 0$ the algorithm initializes by setting as representations of each node its input node features.

During training, a graph-based loss function is used in a fully unsupervised learning setting to tune the learnable weight matrices W_k via stochastic gradient descent. The loss function incorporates a negative sampling term, promoting similar vector representations for nearby nodes while enforcing distinct representations for disparate nodes.

At inference time, the trained system generates embeddings for entirely unseen nodes by applying the learned aggregation functions. GraphSAGE follows an inductive approach, only exploiting local node attribute information. Thus, it can generalize to unseen data, unlike transductive embedding frameworks that can only generate embeddings from static graphs.

7.5.2.3 Event Prediction

The embeddings created as described in the previous section were utilized for link prediction, aiming to anticipate two types of events: processing failure and memory failure events. Each element of the vector was regarded as a feature for link prediction, with the size of the dataset derived from the number of nodes in the graph and the length of the vectors.

Link prediction algorithms aim to predict future or missing associations among nodes in a network. The idea behind link prediction is that the likelihood of an association between two nodes depends on their individual properties and the network structure. More specifically, similar pairs of nodes (with similar node properties and neighbourhoods) tend to connect with the same link type. Figure 70 depicts an example where a Router node (with properties: total processing usage = 110, total memory usage = 60, 1st embedding element = 8.23, 2nd embedding element = 0.75) is connected to the Processing Failure node while another Router node (with similar properties) is considered as candidate for linking. In our study, we specifically used a Graph Neural Network (GNN) based method to perform the link prediction task using the PyTorch Geometric (PyG) library [70].

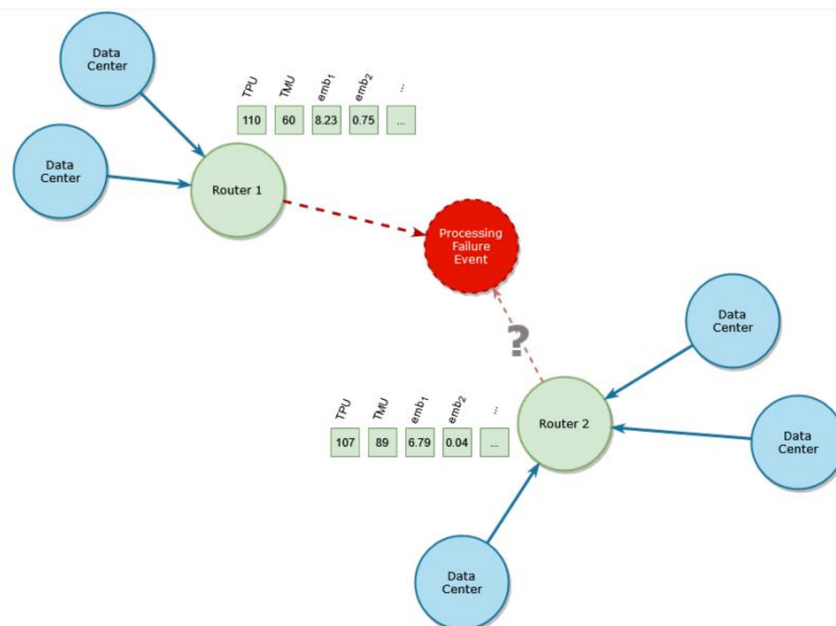

Figure 70: The link prediction process.

A GNN learns to generate node representations incorporating both local graph structures and node features [71]. By iteratively updating the node embeddings based on its neighbouring node representations, a GNN can model the dependencies among the nodes in the graph. Once the node embeddings are obtained, each node pair's similarity measure is calculated to predict whether a link exists or will form in the future.

In our setup, we trained a GNN on the initial graph where each node is associated with the GraphSAGE embeddings. After the GNN training phase, we obtained a new set of nodes embeddings, which were used for computing the similarity scores. We used Cosine Similarity as the similarity measure, where the similarity score for a node pair is computed as the cosine of the angle between their embedding vectors [72]. The similarity scores were then used to rank all possible node pairs in descending order, with the highest scores suggesting the most likely future links.

This approach allowed us to assign potential processing failure or memory failure events to our Router nodes. A potential processing failure event can be predicted if a high similarity score is obtained for a pair of a Datacentre node and a Router node, suggesting a likely future association between the two and potential over usage of processing resources leading to a network bottleneck. Similarly, a potential memory over usage event can be predicted if a high similarity score is obtained for a Datacentre node and a Router node, suggesting a potential demand for memory that exceeds the Router's capacity. In essence, our link prediction approach allowed us to anticipate and respond to potential failure events in our infrastructure, facilitating proactive resource management and performance optimization.

7.5.3 Experiments

We used the NetworkX Python package [73][74] to construct the infrastructure graph and generate the events. We utilized the Neo4j Python Driver [73] to import the graph into the Neo4j graph database management system [75], thereby representing it as a knowledge graph.

To create node embeddings, we relied on Neo4j's built-in GraphSAGE algorithm. We set up a 3-layer GraphSAGE architecture with a pool aggregation strategy, a random walk search depth of $w = 5$, and a sigmoid activation function [76]. The model was trained in batches of size $b = 10$ for 30 epochs, using a learning rate $l = 0.1$, to produce node embeddings of dimension $d = 16$. The training process took approximately 2.5 seconds, and the trained model was used to derive embeddings for all nodes of the sub-graph. These were then added as additional properties of type *embeddingGraphSage* to each node.

Following the embedding process, we utilized the PyTorch Geometric (PyG) library's link prediction method to forecast the processing failure and memory failure events. We trained the link prediction model on the generated node embeddings and used cosine similarity as the measure to compute similarity scores between node pairs. These scores were used to rank potential future links, with high scores indicating likely future connections. We trained our model for 300 epochs using the mean square error loss function on 1800 Router nodes: 1740 with normal usage properties and 60 with processing and memory over usage properties. The test set comprised of 580 Router nodes with normal usage properties plus the excluded 10 nodes with processing over usage and the 10 nodes with memory over usage properties for our evaluation. The training process was run on a computer with an Intel Core i7 processor and 16GB RAM. The computation time for the link prediction model was approximately 3 seconds, and the results obtained are presented next.

By using this approach, we successfully applied machine learning methodologies to a graph-based representation of our infrastructure. This not only allowed us to predict potential traffic congestion events, but also provided a clear and intuitive understanding of the underlying infrastructure and its usage patterns.

Next, we present the simulation results for a particular instance of the setting. The evaluation metrics used to represent the results of the implemented link prediction algorithm are Confusion Matrix, Accuracy, Precision, Recall, and F1 Score. We composed a summary metrics table (Table 15) that includes performance metrics for each class of events.

Table 15: Link prediction evaluation metrics

	Precision	Recall	F1 Score	Support
Memory Failure	0.32	0.70	0.44	10
Normal	0.99	0.96	0.98	580
Processing Failure	0.44	0.80	0.57	10
Accuracy			0.95	600
Weighted AVG	0.97	0.95	0.96	600

From Table 15, it can be seen that the accuracy of the link prediction model is quite impressive, exceeding 95%. Additionally, it is noteworthy that the model performs well in predicting both the ’Normal’ and the ’Failure’ classes, suggesting that the learned embeddings are robust enough to capture the complex interdependencies within the infrastructure. The model exhibits a higher recall score for both ’Memory Failure’ and ’Processing Failure’ cases, signifying that it can correctly identify a substantial proportion of over usage events. The lower precision in these classes results from the model predicting more false positives, which might be an acceptable trade-off in this context as a preventive measure. It is crucial to flag potential over usage events to prevent them from escalating into more significant issues. The ’Normal’ class shows high precision, recall, and F1 score, indicating the model’s effectiveness in correctly identifying normal system usage scenarios and reducing false alarms. It is important to remember that the performance metrics of the model are tied directly to the quality of the node embeddings created by the GraphSAGE algorithm.

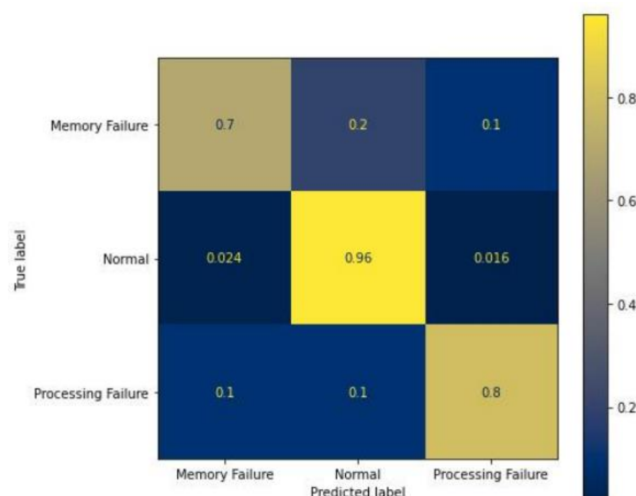

Figure 71: Link prediction confusion matrix

Since the embeddings encode both topological and node feature information, the success of the link prediction model in identifying possible over usage events speaks to the expressive power of the embeddings. Despite the imbalanced distribution of classes in the dataset, the weighted average of precision, recall, and F1 score surpasses the 96% mark, further reinforcing the efficacy of the link prediction model in this task. These high scores suggest that the model is able to generalize well across the different classes, exhibiting its ability to handle both normal and over usage events effectively. For an alternative visualization of model performance, we provide a confusion matrix for the link prediction model (Figure 71).

7.5.4 Discussion

In this section, we discuss our proposed methodology's potential benefits and implications from two main perspectives: network service providers and end users.

For network service providers, our approach offers a tangible means of capturing and understanding the complex, dynamic aspects of network management and orchestration needs of their clients. By modelling these interactions using a graph-based approach and learning embeddings, we provide a measurable way to address real-time communication event detection challenges. This proactive methodology facilitates the discovery of usage patterns and the prediction of unusual or outlier events, allowing service providers to better anticipate and manage network incidents. This proactive management leads to a range of benefits including, but not limited to, reduction of service level agreement (SLA) violations (such as availability, response time, reliability, and cost limit), and enhanced ability to manage heterogeneous resources across different domains efficiently, automatically, and in scalable manner. This translates into maximizing overall network efficiency, facilitating the implementation of complex billing models, and proactively preparing for future demands and capacity needs.

From the perspective of the end user, our event detection approach offers enhanced reliability and trust in the communication network services. Ensuring high quality of service (QoS) requirements, and proactively informing users about potential outlier events related to abnormal communication patterns, not only optimizes user experience, but also increases transparency. These outlier events can be a result of compromised services, malfunctioning components, configuration anomalies or even adversarial attacks, providing users with valuable insights into their network status. It is important to note that while individual resource usage values (e.g., network traffic bandwidth, latency) might not surpass a certain threshold, adverse or malicious behaviour can be inferred from the combination of these values, as depicted by the graph-based embeddings of the network structure.

This means our approach adds an additional layer of protection against adversarial attacks or similar malicious intents that target the integrity of network services, thereby further fortifying network security. Thus, our approach addresses a dual need: equipping service providers with the tools to efficiently manage their networks and helping end users experience a reliable, secure, and well-maintained communication service.

8 Energy and Resource Aware Flow Mapping

This section reports on the progress of Task 5.4 "Energy and Resource Aware Flow Mapping", which focuses on creating a framework to help developers integrate performance and energy modelling functionality into their digital services, specifically within the SERRANO platform. It involves considering hardware, operating systems, compilers, and drivers from an HPC application developer's perspective in relation to SERRANO's digital services. The task includes selecting energy-aware benchmarks running on HPC systems. The results were used to extend existing energy and performance models tailored to the specific hardware features, allowing developers to visualise application execution regarding power consumption.

In the previous deliverable D5.2 (M15), the main focus was on getting the test cluster (EXCESS cluster) ready. This cluster has a dual purpose: firstly, to evaluate the energy efficiency of HPC services, and secondly, to function as a testing platform for the SERRANO orchestrator. The testbed offers numerous opportunities to discover the optimal configuration of HPC services in terms of performance and energy efficiency. Additionally, it aids in preparation for utilising the Hawk supercomputer. This preparation encompasses activities such as installing hardware, developing, and executing tools, and analysing benchmarks. These actions aim to study the behaviour of the hardware and software employed in the SERRANO project, particularly concerning HPC Services.

During the second iteration of the implementation plan (M16-M31), the task focused on developing power measurement utilities and deriving an energy model of the EXCESS cluster using power measurement hardware and extensive benchmarking. This energy model was also used to extrapolate the energy consumption of the Hawk supercomputer using linear regression. This section presents the status of these activities as well.

8.1 Excess Cluster, Hardware, and Tools

One reason for installing the EXCESS cluster is the absence of an interface to measure the power consumption in Hawk. It is, therefore, impossible to investigate possible configurations of HPC services concerning energy efficiency. The EXCESS cluster contains the external hardware on one compute node to measure voltage and current. Therefore, the power consumption can be accurately derived.

Figure 72 illustrates the primary elements comprising the EXCESS cluster:

- *Login* node, which serves as a gateway for accessing the compute nodes within the cluster.
- *Node01* node, which functions as a compute node and shares the exact CPU model as the Hawk supercomputer. It is equipped with six sensors that measure electric power and voltage from three 12-volt power sources: CPU1, CPU2, and the motherboard (ATX). This enables recording power consumption details for the main memory (64 GB), CPU, voltage regulators, and InfiniBand Adapter.

- *Addi* server, which is equipped with AC/DC converters and is connected to the sensors of node01. The measurement process is initiated for each batch job, and the data is stored in the home directory once the job is completed. The recorded data can be utilised for subsequent energy profiling and analysis purposes.

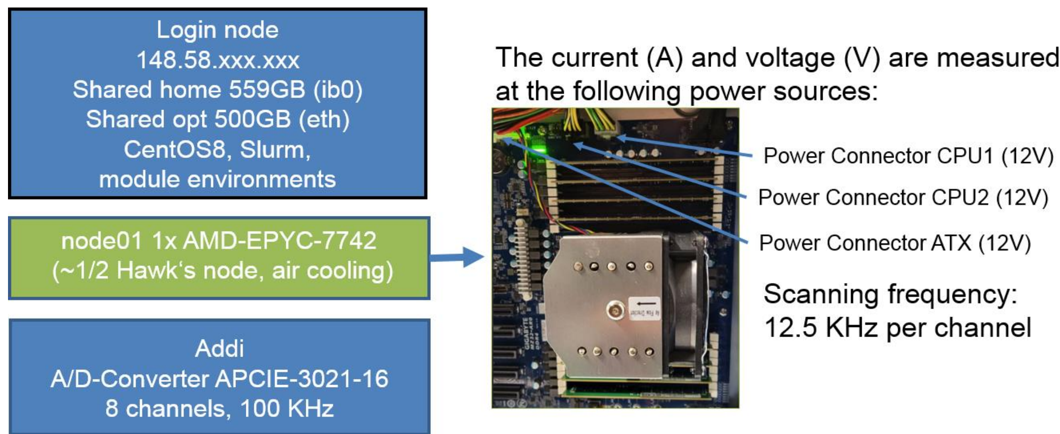

Figure 72: Hardware components of the EXCESS cluster

The compute nodes of EXCESS and Hawk are similar but not identical (Table 16). However, the energy efficiency of these two systems can be compared, given that the configuration of the operating mode of the processors and main memory is as close to each other as possible. To address this, BIOS and OS settings were tuned such that the configuration of EXCESS nodes is similar to the Hawk nodes.

Table 16: Comparison between compute nodes of Excess and Hawk

Hardware	EXCESS Compute Node	Hawk Compute Node
CPUs	1 x AMD-EPYC-7742	2 x AMD-EPYC-7742
Main memory	Samsung SDRAM DDR4 double rank, 3200 MT/s, 8x16GiB	Micron SDRAM DDR4 double rank, 3200 MT/s, 16x16GiB

The EXCESS cluster is equipped with utilities for interfacing with the power measurement hardware, obtaining measured data, transforming these data into suitable format, and visualising the power measurements.

8.2 Power Measurement Utilities

The power measurement utilities automatically start and stop external power measurement on compute nodes during the lifetime of a batch job. Slurm [105] batch scheduler provides a configuration for prolog and epilog scripts, which allow to integrate additional functionalities before and after a batch job execution, respectively. Therefore, power measurement utilities were integrated into the prolog and epilog.

A user of the EXCESS cluster can enable power measurements in batch jobs by creating empty files (*save_raw_data* and *copy_raw_data*) in a specific user home directory (*.pwm/node01/*). At the prolog stage, the existence of these files will trigger the *Addi* server to start the power

measurement service. At the epilog stage, this service will be stopped, and the raw binary data of the power measurements will be compressed and saved into a specific location: *`/nfs_home/power/pwm/node01/node01_{JOB_ID}.tar.gz`*, where JOB_ID is the unique identifier of the batch job. These data contain precise timestamps that can be utilised to synchronise with the energy-aware benchmark applications, i.e., the applications that also expose timestamps to signify the change in application phases, such as IO operations (read or write), enabled optimisation or arbitrary phases defined by the developers of the benchmarks.

8.3 Power Measurement Conversion and Visualization

In order to convert binary data obtained from power measurement sensors (channels) into the CSV format, a tool (namely the *`power_calculate`*) was developed that reads the data from each channel and converts it into a data format that has the following structure:

Field	Description
ID	ID of the application phases
Time	Execution time of the phase
Number of measures	Number of samples to take the measurements
CPU1 Average, Min, Max Power [Watt]	Average, minimum and maximum power consumption of the CPU1
CPU1 Average, Min, Max Energy [Joule]	Average, minimum and maximum energy consumption of the CPU1
CPU2 Average, Min, Max Power [Watt]	Average, minimum and maximum power consumption of the CPU2
CPU2 Average, Min, Max Energy [Joule]	Average, minimum and maximum energy consumption of the CPU2
ATX Average, Min, Max Power [Watt]	Average, minimum and maximum power consumption of the ATX motherboard
ATX Average, Min, Max Energy [Joule]	Average, minimum and maximum energy consumption of the ATX motherboard
Total Average, Min, Max Power [Watt]	Total average, minimum and maximum power consumption of the system
Total Average, Min, Max Energy [Joule]	Total average, minimum and maximum energy consumption of the system

This formatted data can then be analysed and used to determine HPC applications' power and energy consumption. Deliverables D4.2 (M15) and D4.4 (M30) elaborate more on how this data is used to determine the power and energy consumption of SERRANO-accelerated kernels running on HPC systems.

In order to visualise the power metrics, another tool (flow) was developed. It reads the raw binary data of power measurements and provides graphs for visual analysis, as shown in Figure 73.

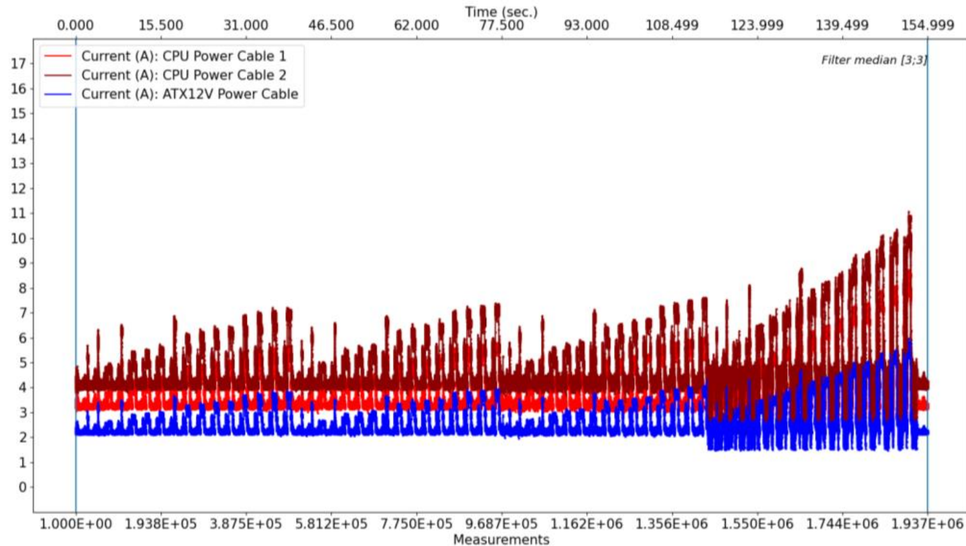

Figure 73: Hardware components of the EXCESS cluster

8.4 CPU Frequency Utility

Additionally, a utility for modifying the clock frequency of CPUs was provided, as the CPU's frequency plays a significant role in power draw. The utility is located at `/opt/power/rome-freq/bin/set_node01_frequency.sh`. This utility is based on cpupower[1] and has four frequency configurations:

- 1.5 GHz: Without the CPU boost, the minimum frequency is 1.5 GHz, maximum frequency is 1.5 GHz;
- 2.0 GHz: Without the CPU boost, minimum frequency is 2.0 GHz, maximum frequency is 2.0 GHz; BOOST = 0, MIN_FREQ=2.0 GHz, MAN_FREQ=2.0, GOV = userspace;
- 2.25 GHz: Without the CPU boost, minimum frequency is 2.25 GHz, maximum frequency is 2.25 GHz;
- Turbo Mode: With the CPU boost[2], minimum frequency is 2.25 GHz, maximum frequency is 2.25 GHz.

8.5 Kernels Benchmarking

Energy measurements of the developed kernels running in HPC leverage the utilities described earlier. In general, the implementation of kernels is energy benchmark aware. The kernel execution consists of three phases: reading the data required by the kernel, execution of the kernel, and writing the results. For each kernel, we executed energy benchmarks (described in Deliverables D4.2 and D4.4), which iterate over the number of processors, CPU frequencies, approximation, and precision parameters.

[1] die.net: cpupower(1) - Linux man page, https://linux.die.net/man/1/cpupower
[2] https://www.kernel.org/doc/Documentation/cpu-freq/boost.txt

As an example, Table 17 presents the measured power consumption of the parallel Kalman filter in Turbo frequency with different numbers of cores. It has been observed that as the number of cores increases, power consumption also increases.

Table 17: Power consumption of parallel implementation of Kalman filter in Turbo Mode

Number of Cores	Power (Watt)
1	228.97
2	231.97
4	234.97
8	237.87
16	240.49
32	243.65
64	249.53

Figure 74 shows the energy consumption of the Kalman filter, measured with different CPU frequencies and numbers of cores. The energy consumption decreases when more cores are employed in parallel applications. The same approach was applied to other kernels, the results of which can be found in Deliverable D4.4 (M30).

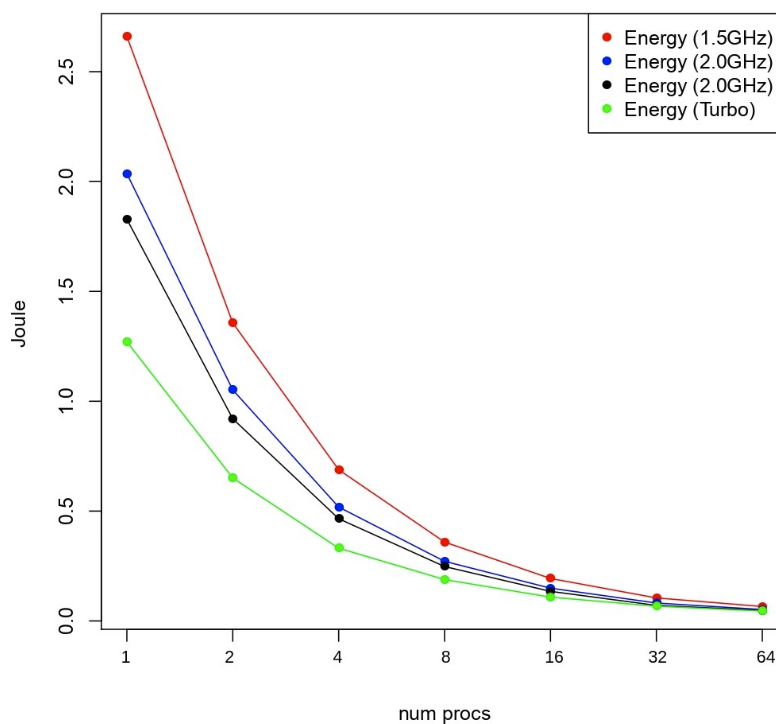

Figure 74: Energy consumption of Kalman filter with different frequencies and different numbers of cores

9 Resource Orchestration Mechanisms

The seamless orchestration of the available edge, cloud, and HPC resources is key for realizing SERRANO's objectives. To this end, SERRANO adopts a hierarchical architecture to enable end-to-end cognitive resource orchestration and transparent application deployment over heterogeneous resources. The detailed architecture and the overall design were presented in D5.3 (M15). For reference, we provide an overview of the selected design, while Figure 75 summarizes the architecture of the SERRANO resource orchestration mechanisms and their interactions with other SERRANO components.

The SERRANO Resource Orchestrator, developed in the context of the project, acts as the high-level orchestrator that interacts with multiple Local Orchestrators, where each handles individual parts of the overall unified infrastructure. During this process, it exploits the advanced scheduling capabilities of the Resource Orchestration Toolkit (ROT) that provide cognitive decisions. Then, it delegates the decision for the actual deployment operations to the corresponding Local Orchestrators at the selected platforms. To this end, the Resource Orchestrator adopts a declarative approach to describe the workload requirements to the selected Local Orchestrators instead of an imperative one. The adopted design enables the SERRANO Resource Orchestrator to manage the underlying heterogeneous infrastructure more abstractly and disaggregated than the Local Orchestrators.

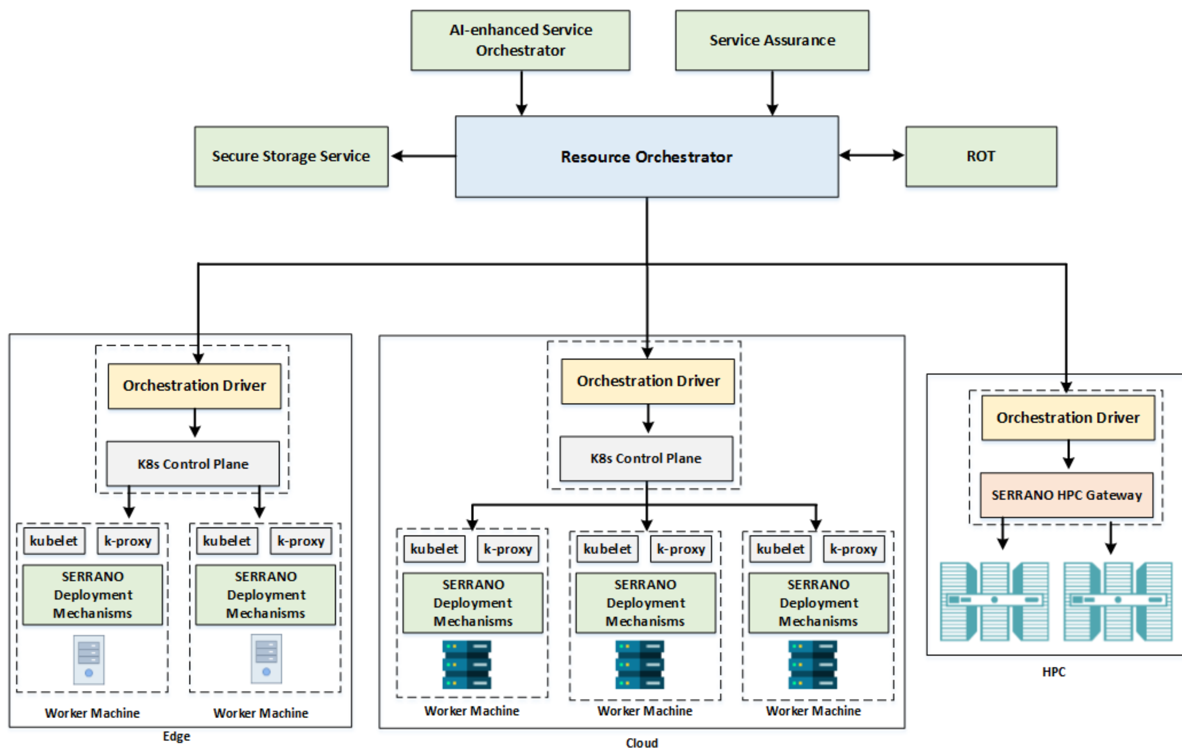

Figure 75: SERRANO distributed and cognitive resource orchestration mechanisms, unifying different edge, cloud, and HPC platforms

9.1 SERRANO Resource Orchestrator

The Resource Orchestrator is a cloud-native application implemented in Python that consists of two primary services: *Orchestration API Server* and *Orchestration Manager*. The Datastore, a critical component for the overall operation and coordination among the Resource Orchestrator services and SERRANO Orchestration Drivers, completes the architecture. These services and configuration files are packaged as Python applications using the SERRANO CI/CD services. The SERRANO image registry [96] includes a separate container image for each service. We also defined all the required Kubernetes YAML description files (i.e., ConfigMap, Deployment, Services) to facilitate deploying the developed services in Kubernetes. Figure 76 illustrates the SERRANO Resource Orchestrator architecture and its main components.

The Datastore is based on etcd [102], an open-source distributed key-value store, and stores the SERRANO API objects that include configuration and state data for the available platforms, deployed applications, and SERRANO hardware and software accelerated kernels. One of the essential features of etcd is the *"watch"* function that, through the Watch API, provides an event-based interface for asynchronously monitoring changes to keys in the etcd. An etcd watch waits for changes to keys by continuously watching from a revision and streams the key updates back to the registered client. We leverage this feature to facilitate communication among the Orchestration API Server, Orchestration Manager, and Orchestration Drivers. Hence, the Resource Orchestrator services can keep track of the actual and desired state of the deployed workloads across the unified infrastructure. The Orchestration API Server, Orchestration Manager, Orchestration Drivers, Resource Optimization Toolkit, and Datastore constitute the control plane of the SERRANO orchestration and deployment framework.

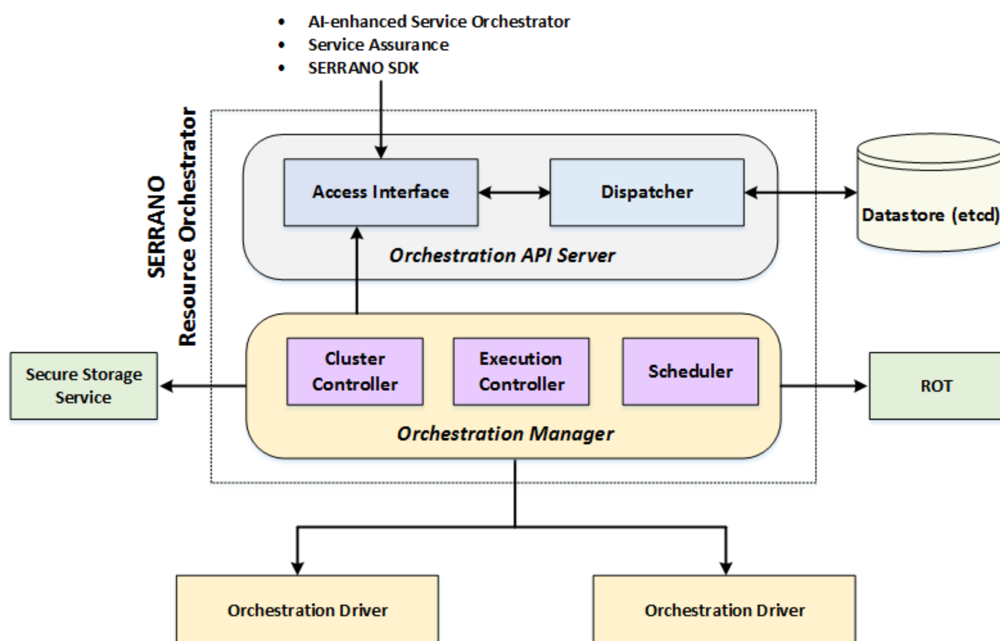

Figure 76: SERRANO Resource Orchestrator architecture and services

Regarding the Orchestration API Service, we significantly extend the functionality of the Access Interface and Dispatcher components to facilitate (i) the transparent application deployment, (ii) execution of SERRANO hardware and software accelerated kernels, and (iii) the intent-based creation of secure storage policies. The Access Interface provides loose coupling with the other components within the SERRANO platform, mainly with AI-enhanced Service Orchestrator and Service Assurance mechanisms. It exposes the appropriate interfaces to enable bidirectional communication for exchanging commands, information, and notifications. The Access Interface also validates all the requests before forwarding them to the Dispatcher that exclusively handles the interaction with the Datastore. The exposed RESTful API final version includes several methods organized into two main categories. The first set of methods (Figure 77) enables the deployment and management of cloud-native applications, execution of SERRANO accelerated kernels, and the cognitive creation of secure storage policies. The second set (Figure 78) abstracts the interaction of the Orchestration Manager and Orchestration Driver services with the Datastore by enabling them to create, update, and query relevant information along with their subscription for watching specific topics in the Datastore.

Deployments

Method	Endpoint	Description
GET	/api/v1/orchestrator/deployments	Get the list of all current application deployments.
POST	/api/v1/orchestrator/deployments	Request the deployment of a new application.
PUT	/api/v1/orchestrator/deployments	Request the re-optimization of a specific application deployment.
DELETE	/api/v1/orchestrator/deployments/{uuid}	Terminate a specific application deployment.
GET	/api/v1/orchestrator/deployments/{uuid}	Get information for specific application deployment.
GET	/api/v1/orchestrator/deployments/watch	Register for changes regarding deployment objects in etcd.

Kernels

Method	Endpoint	Description
POST	/api/v1/orchestrator/faas	Request the FaaS-like execution of a kernel within the SERRANO platform.
GET	/api/v1/orchestrator/faas/{uuid}	Get details for a specific FaaS-like kernel execution within the SERRANO platform.
POST	/api/v1/orchestrator/kernel	Request the Serverless-like execution of a kernel within the SERRANO platform.
GET	/api/v1/orchestrator/kernel/{uuid}	Get details for a specific FaaS-like kernel execution within the SERRANO platform.

Storage Policies

Method	Endpoint	Description
GET	/api/v1/orchestrator/storage_policies	Get the list of all current storage policies.
POST	/api/v1/orchestrator/storage_policies	Request the deployment of a new storage policy.
PUT	/api/v1/orchestrator/storage_policies	Request the re-optimization of a specific storage policy deployment.
DELETE	/api/v1/orchestrator/storage_policies/{uuid}	Delete a specific storage policy deployment.
GET	/api/v1/orchestrator/storage_policies/{uuid}	Get information for specific storage policy deployment.

Figure 77: Resource Orchestrator RESTful interface

Clusters

Method	Endpoint	Description
GET	/api/v1/orchestrator/clusters	Get the list of all registered clusters.
POST	/api/v1/orchestrator/clusters	Register a new cluster.
PUT	/api/v1/orchestrator/clusters	Update the information for a specific cluster.
DELETE	/api/v1/orchestrator/clusters/{uuid}	Delete a previously registered cluster.
GET	/api/v1/orchestrator/clusters/{uuid}	Get information for specific cluster.
GET	/api/v1/orchestrator/clusters/health/{uuid}	Get details about the health of the sepcific cluster.
GET	/api/v1/orchestrator/clusters/watch	Register for changes regarding cluster objects in etcd.

Assignments

Method	Endpoint	Description
GET	/api/v1/orchestrator/assignments	Get the list of available deployement assignments.
POST	/api/v1/orchestrator/assignments	Create a new deployment assignment.
PUT	/api/v1/orchestrator/assignments	Update a specific deployment assignment.
DELETE	/api/v1/orchestrator/assignments/{uuid}	Delete a specific deployment assignment.
GET	/api/v1/orchestrator/assignments/{uuid}	Get information for specific deployment assignment.
GET	/api/v1/orchestrator/assignments/watch/{uuid}	Register for changes regarding deployment assignment objects for a specific cluster in etcd.

Bundles

Method	Endpoint	Description
GET	/api/v1/orchestrator/bundles	Get the list of all bundles.
POST	/api/v1/orchestrator/bundles	Create a new bundle for some specific deployment assignment.
PUT	/api/v1/orchestrator/bundles	Update a specific bundle description.
DELETE	/api/v1/orchestrator/bundles/{uuid}	Delete a specific bundle description.
GET	/api/v1/orchestrator/bundles/{uuid}	Get information for specific bundle description.
GET	/api/v1/orchestrator/bundles/watch/{uuid}	Register for changes regarding bundle objects for a specific deployment assignment in etcd.

Figure 78: Resource Orchestrator RESTful interface – Methods related to inter-component communication

The Orchestration Manager implements the main part of the application logic and coordinates the resource allocation and application deployment, kernel execution, and secure storage policy management operations. In the original design, some of these tasks were handled by the Dispatcher component of the Orchestration API Server. However, in the revised design and final implementation, the Orchestration Manager, through its controllers, is exclusively responsible for all the coordination and management actions. The Orchestration Manager performs operations based on the SERRANO Orchestration API objects that are created through the API Server. To achieve its objectives, the Orchestration Manager incorporates various controllers, which watch SERRANO Orchestration objects in the Datastore. These

controllers execute the necessary operations to serve the requests and then communicate with the Orchestration Drivers on the underlying platforms.

More specifically, the *Scheduler Controller* interacts with the ROT to retrieve the instructions for the cognitive application deployment and definition of secure storage policies. The *Cluster Controller* attaches Kubernetes clusters and HPC platforms to Resource Orchestrator and oversees their operational state. The *Execution Controller* prepares the required application deployment instructions (declarative approach) with the assistance of the Scheduler controller, coordinates the required data movement by interacting with the SERRANO Secure Storage service, and finally triggers the actual deployment by interacting with the Orchestration Drivers at the selected edge/cloud and HPC platforms.

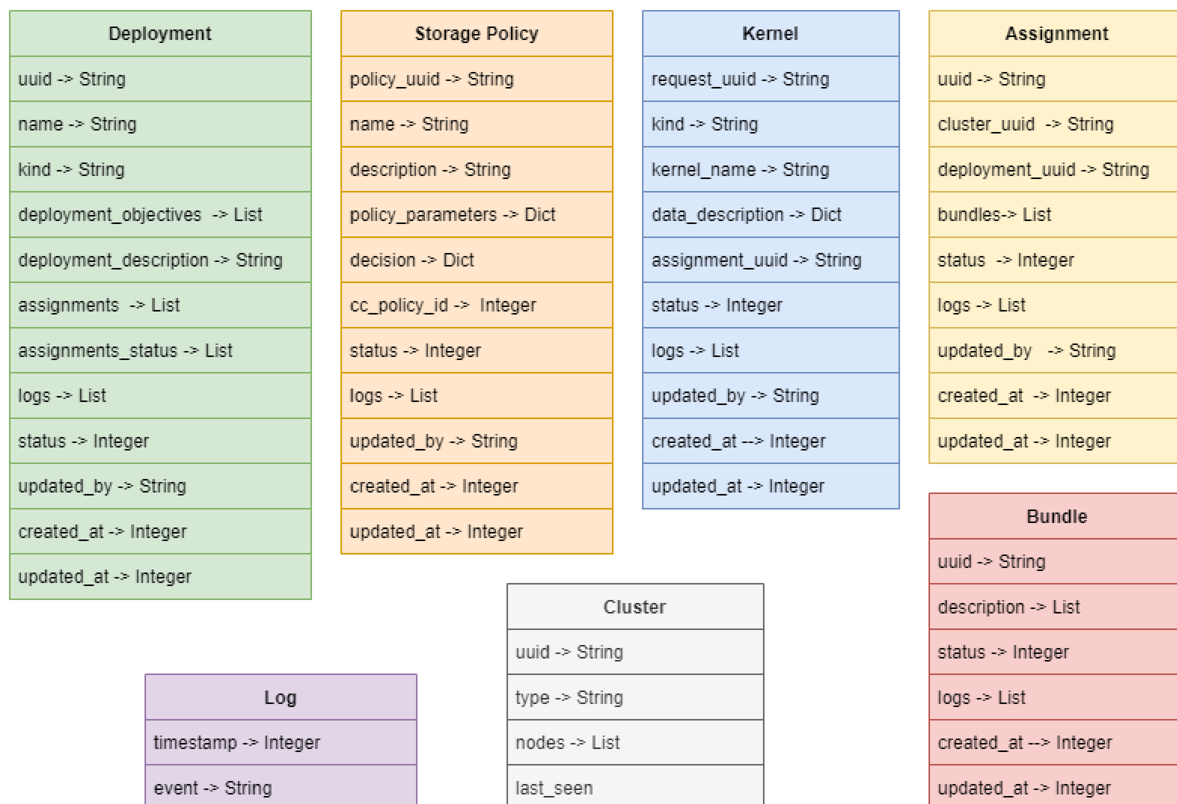

Figure 79: SERRANO Orchestration API objects

As previously stated, the Datastore, through the etcd service, offers reactive capabilities to the Resource Orchestrator services, enabling them to effectively orchestrate and manage the complete lifecycle of operations associated with service requests. By leveraging the Datastore, the Resource Orchestrator services can efficiently handle and respond to various requests. Service requests within the orchestration and deployment mechanisms are expressed as SERRANO Orchestration API objects (Figure 79). These objects serve as the primary means of communication between the different components of the system. They encapsulate the necessary information to serve, manage, and monitor the progress of service requests. To facilitate the interaction between the Resource Orchestrator services, specific pairs of services are responsible for creating, updating, deleting, and watching these SERRANO Orchestration API objects. This distributed responsibility ensures efficient handling of requests and enables the system to operate seamlessly.

The main SERRANO Orchestration API objects are the following:

- ***Cluster:*** It provides an overview of the available individual platforms (edge, cloud, HPC). These objects are created and updated based on the information from the Orchestration Drivers while watched and used by the Orchestration Manager.
- ***Deployment:*** It corresponds to the high-level description for deploying a cloud-native application in the SERRANO platform. It includes the application description along with the user intent for the deployment objectives. It is created and deleted by the Orchestration API server, while it is watched and used by Orchestration Manager. These entities are not changed during the orchestration and deployment phase since the high-level orchestration decisions and the infrastructure-specific instructions for the low-level orchestration mechanisms are expressed through the Assignment and Bundle objects.
- ***Kernel:*** It corresponds to the description for the deployment of a SERRANO accelerated kernel in the SERRANO platform. It is created and deleted by the Orchestration API Server and watched and used by the Orchestration Manager. More details are provided in Section 9.4.3.
- ***Storage Policy***: It is the high-level description from the intent-based creation of a secure storage policy. It is created and deleted by the Orchestration API Service and watched and used by the Orchestration Manager that will execute all the required operations. More details are provided in Section 9.4.1.
- ***Assignment:*** It is an internal object that captures the assignment of application microservices to a specific SERRANO cluster (i.e., edge/cloud or HPC platform). These entities are created, updated, and deleted by the Orchestration Manager according to the decisions from the ROT component while they are watched and used by the Orchestration Drivers. The unique identifier of the selected platform (i.e., CLUSTER_UUID) is part of the respective topic's description in the Datastore (Table 18). More details are available in Section 9.2.
- ***Bundle:*** It includes the application description along with parameters and platform-specific deployment objectives based on the ROT decisions that will guide the low-level orchestration mechanisms at the selected platforms (declarative approach). These entities are created, updated, and deleted by the Orchestration Manager and they are watched and used by the Orchestration Drivers.

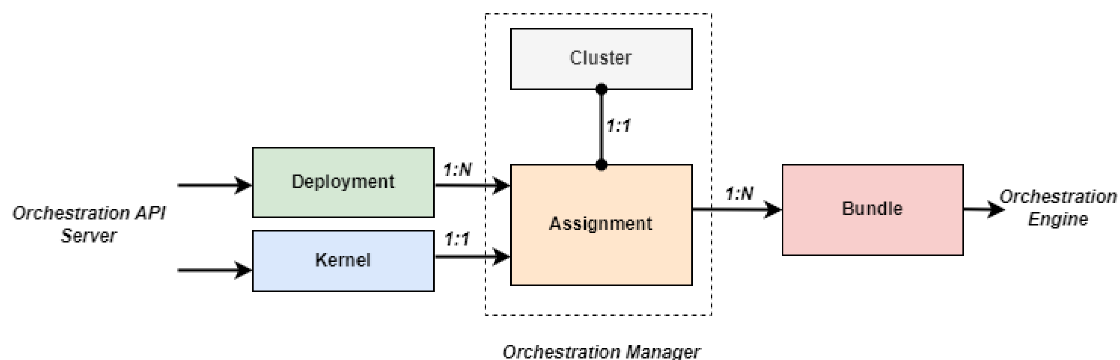

Figure 80: Relationship among main SERRANO Orchestration API objects

To provide a visual representation of the relationships and interactions between the SERRANO Orchestration API objects, Figure 80 illustrates their associations. Furthermore, Table 18 complements the previous description by listing the relevant topics for each SERRANO Orchestration API object stored and managed within the Datastore. It highlights the specific information and data points associated with the SERRANO Orchestration API objects that are crucial for the system's overall functioning.

Table 18: Datastore topics (keys) for the main SERRANO Orchestration API objects

API Object	Topic
Cluster	/serrano/orchestrator/clusters/cluster/*CLUSTER_UUID*
Deployment	/serrano/orchestrator/deployments/deployment/*DEPLOYMENT_UUID*
Kernel	/serrano/orchestrator/kernels/kernel/*REQUEST_UUID*
Storage Policy	/serrano/orchestrator/storage_policies/policy/*POLICY_UUID*
Assignment	/serrano/orchestrator/assignments/*CLUSTER_UUID*/assignment/*ASSIGNMENT_UUID*
Bundle	/serrano/orchestrator/bundles/bundle/*BUNDLE_UUID*

To elaborate more on the usage of the SERRANO Orchestration API objects from the SERRANO orchestration mechanisms, we present an example of how one of the SERRANO use cases is handled by the SERRANO Resource Orchestration services. Specifically, we considered the Position Service from the Anomaly Detection in Manufacturing Settings use case, which includes three microservices. Deliverable D6.5 (M27) provides a more comprehensive technical description of this application, and Section 9.4.2 describes the detailed deployment workflow. Next, we focus on the internal SERRANO API objects that the Orchestration Manager creates to serve the requested deployment within the SERRANO platform.

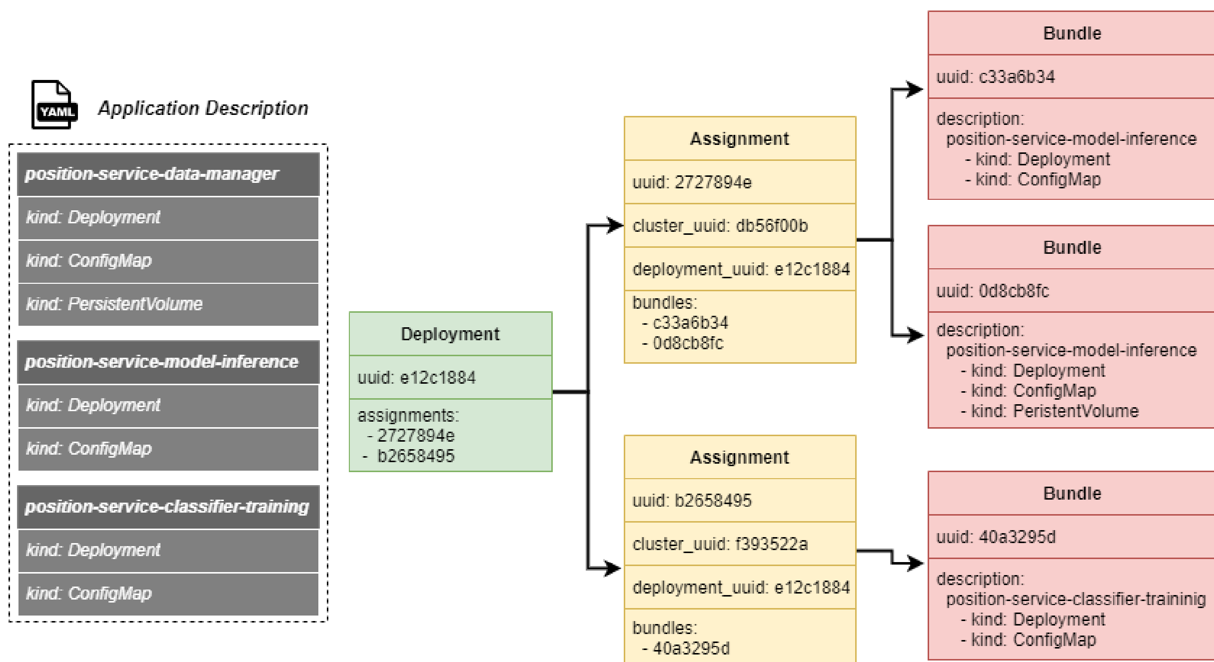

Figure 81: SERRANO Orchestration API objects and federated application deployment

We consider that the overall application is assigned to two different platforms. Two microservices are assigned in an edge cluster (UUID: *"db56f00b"*) and the third in a cloud cluster (UUID: *"f393522a"*). In this case, the Orchestration Manager will create two Assignment objects (UUIDs *"2727894e"* and *"b2658495"*), one for each selected platform. These Assignments will be linked with the initial Deployment object (UUID *"e12c1884"*). Moreover, the Orchestration Manager will create three Bundle objects (UUIDs *"c33a6b34"*, *"0d8cb8fc"*, and *"40a3295d"*), one for each microservice, and will map them with the appropriate Assignment objects. Each Bundle object includes the required deployment descriptions. In our example, the Bundle objects include K8s API objects, such as Deployment, ConfigMap, PersistentVolume, and PersistentVolumeClaim. The Bundles also include additional parameters added by the Orchestration Manager that will guide the platform-level scheduling of the microservices. Figure 81 summarizes this process.

9.2 Orchestration Drivers

The Orchestration Drivers complete the implementation of the hierarchical resource orchestration. An Orchestration Driver provides an abstraction layer for interacting with the specific edge, cloud, and HPC orchestration mechanisms, dealing with the low-level details of the heterogeneous Local Orchestrators at the individual platforms. SERRANO considers that Local Orchestrators are based on existing and well-established solutions. According to the final implementation, Kubernetes (K8s) is the orchestration platform for the edge and cloud resources, whereas HPC resource managers and batch jobs schedulers are considered for the HPC platforms.

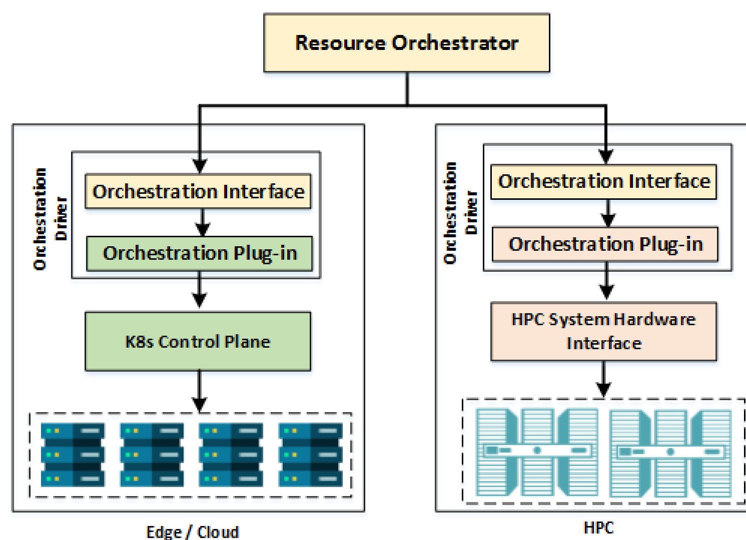

Figure 82: SERRANO Orchestration Drivers

Since SERRANO unifies platforms with different local orchestration mechanisms, two types of Orchestration Drivers are available. Figure 82 shows the final design of the Orchestration Drivers. The Orchestration Drivers are implemented in Python as plug-ins that share the same implementation for the ***Orchestration Interface***. This component leverages the exposed REST API from the Orchestration API Server to provide an infrastructure-agnostic interface between the Resource Orchestrator (i.e., Orchestration Manager) and the Local Orchestrators. It

facilitates the generic description of the deployment preferences and constraints. On the other hand, the ***Orchestration Plug-in*** component differs for each Orchestration Driver type since it interfaces with a specific Local Orchestrator based on its specific exposed APIs. More specifically, the Orchestration Plug-in for the Kubernetes platforms uses the exposed API by the Kubernetes API Server (i.e., kube-apiserver), while the Orchestration Plug-in for the HPC platforms uses the exposed interface by the SERRANO HPC Gateway (Figure 85).

The implementation of the Orchestration Driver includes a configuration file that utilizes JSON format. This file consists of multiple configuration parameters that allow for precise customization of Orchestration Driver operation and seamless integration with other SERRANO orchestration and deployment services. It is possible to designate the specific type of Orchestration Plug-in to be loaded for a particular instance of the Orchestration Driver using one of the available configuration parameters. In addition, each Orchestration Driver is associated with a unique identifier that determines the specific edge/cloud or HPC platform that manages. This identifier corresponds to the *CLUSTER_UUID* parameter in the Datastore topics presented in the previous section.

The Orchestration Driver and its configuration file are packaged as a Python application using the SERRANO CI/CD services, ensuring a smooth and efficient development workflow. The resulting container image is made accessible through the SERRANO image registry. There is a common image for both Orchestration Drivers. To facilitate effortless deployment on Kubernetes platforms, corresponding Kubernetes YAML description files are also available. These files enable the automatic deployment of the Orchestration Drivers within Kubernetes.

The following workflow (Figure 83) summarizes the operation of SERRANO Orchestration Drivers. During its initialization phase, a driver is registered through the Orchestration API Server to watch for any changes related to assignments in its dedicated topic (*/serrano/orchestrator/assignments/CLUSTER_UUID*) in the Datastore. Moreover, it sends the Orchestration API Server a summary of the available resources in the platform it manages. The Orchestration API Server uses this information to update the respective contents in Datastore. It also sends heartbeat messages periodically to the Orchestration API Server. These steps are common for both Orchestration Driver types and are handled by the Orchestration Interface using a set of methods that all Orchestration Plug-ins must implement.

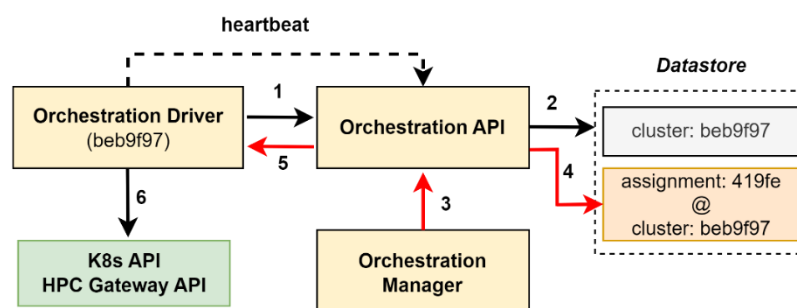

Figure 83: Orchestration Driver workflow

Next, the Orchestration Driver is notified of any change in the topic that watches, and based on the event type, it triggers the appropriate actions to serve the request from the Orchestration Manager. To this end, it formats the appropriate instructions to the Local Orchestrator and forwards them using their exposed interfaces. More details for the interaction with the SERRANO-enhanced and infrastructure-specific low-level mechanisms in K8s and HPC platforms are also provided in Sections 9.3, 9.4, and 10.

9.3 SERRANO HPC Gateway

The SERRANO HPC Gateway is the intermediate component between SERRANO's HPC services (WP4), the Intelligent Service and Resource Orchestration Layer (WP5), and the HPC infrastructure. The HPC Gateway supports popular batch job schedulers, such as Slurm [105] and the PBS-based OpenPBS [106].

Due to security restrictions and isolation imposed on the compute nodes of HPC clusters, only the front-end (or login) nodes of the clusters are usually used as the access point, where a user or automation tool can login via SSH, prepare software environments and workspaces, build applications and submit HPC jobs. The job submission commands are specific to the resource manager. For example, Slurm uses *sbatch* commands for job submission, whereas for PBS-based resource managers, the *qsub* command is used. Additionally, the job status can be monitored via *scontrol* and *qstat* commands of Slurm and PBS, respectively.

Similarly, the information about the partitions of the HPC system can be obtained via scheduler specific commands. For Slurm, *sinfo* and *squeue* commands are common to determine the state of the partitions, whereas *pbsnodes* and *qstat -Q* commands are used in PBS.

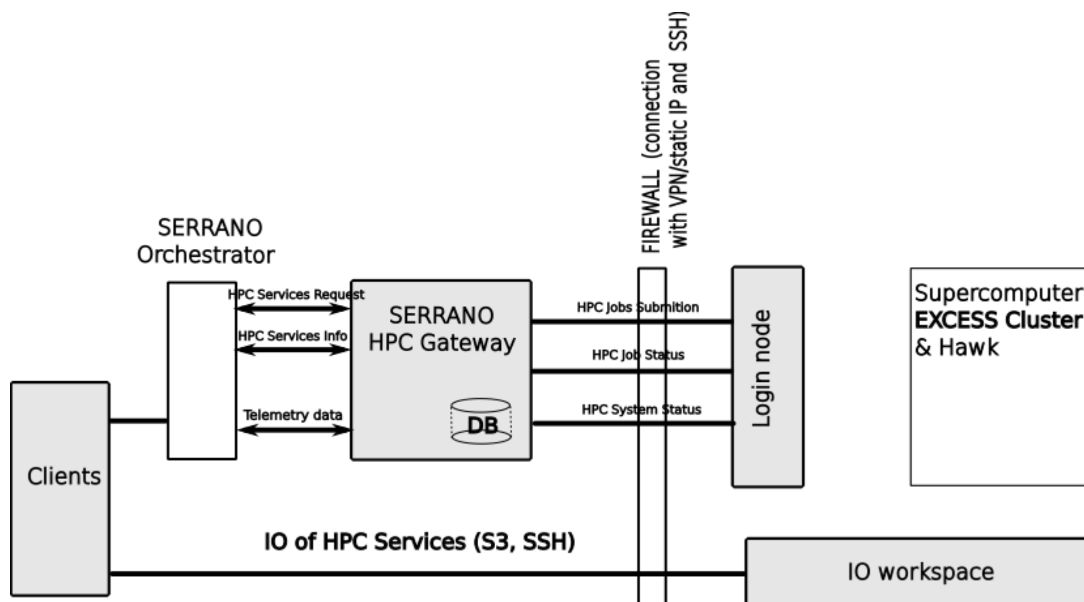

Figure 84: Interaction between HPC Gateway and HPC infrastructure

Therefore, SERRANO HPC Gateway communicates with the front-end (login) nodes via SSH and uses commands specific to the resource managers under use in order to prepare a batch job script for submission (i.e., to select the appropriate header), submit the job, and monitor the status of the job and the partitions, as shown in Figure 84. Moreover, SERRANO HPC Gateway provides endpoints for remote (HTTP, S3) file transfers into HPC infrastructure, as well as transferring results from HPC into S3.

The HPC System Hardware Interface (HPC Gateway) is integrated with the SERRANO platform. It exposes REST API endpoints (Figure 85) needed for the Resource Orchestrator and Telemetry Framework for the execution of HPC services/kernels (/job) and monitoring the state of the HPC infrastructure (*/infrastructure/infrastructure_name/telemetry*). Moreover, users can utilise data endpoints (*/data, /s3_data,* and */s3_result*) of the HPC Gateway to transfer data from HTTP and S3 endpoints, such as the SERRANO Secure Storage service (WP3), into HPC and move resulting data to S3. The HPC Gateway is implemented as a service[3] and interacts with the target HPC infrastructure using SSH protocol (as shown in Figure 84). The administrator maintains SSH keys that will be used for authentication with the infrastructure.

POST /data Transfer a remote HTTP file to the target infrastructure
GET /data/{file_transfer_id} Get file transfer status
POST /infrastructure Create a new infrastructure
GET /infrastructure/{infrastructure_name} Get infrastructure
GET /infrastructure/{infrastructure_name}/telemetry Get infrastructure telemetry
POST /job Submit a new job
GET /job/{job_id} Get job status
POST /s3_data Transfer an S3 object to the target infrastructure
GET /s3_data/{file_transfer_id} Get S3 file transfer status
POST /s3_result Transfer the results from HPC to an S3 bucket
GET /s3_result/{file_transfer_id} Get S3 result transfer status
GET /services Get all available services

Figure 85: REST API endpoints exposed by HPC Gateway

A user can perform the complete workflow using HPC Gateway, i.e. injection of the initial data, processing the data in HPC, and retrieving the results. For example, one can use */s3_data* endpoint to send the initial data from S3 into the target HPC infrastructure, then run signal processing kernels (e.g., Kalman and FFT filters) via */job* endpoint and move the results back to the S3 storage via */s3_result* endpoint. The requests chain is outlined below (the responses are omitted).

[3] https://hpc-interface.services.cloud.ict-serrano.eu

```
POST /s3_data
Body:
{
  "infrastructure": "cluster_name",
  "endpoint": "https://on-premise-storage-gateway.services.cloud.ict-serrano.eu
/s3",
  "bucket": "initial-data-bucket",
  "object": "initial-data",
  "region": "local",
  "access_key": "access_key",
  "secret_key": "secret_key",
  "dst": "/path/to/initial/data",
}

POST /job
Body:
{
  "infrastructure": "cluster_name",
  "services": [ "kalman", "fft" ],
  "params": {
    "read_input_data": "/path/to/initial/data",
    "input_data_double": "/path/to/double-precision/data",
    "input_data_float": "/path/to/single-precision/data",
    "num_mpi_procs": 64
  }
}

POST /s3_result
Body:
{
  "endpoint": "https://on-premise-storage-gateway.services.cloud.ict-serrano.eu
/s3",
  "bucket": "results-bucket",
  "object": "results.csv",
  "region": "local",
  "access_key": "access_key",
  "secret_key": "secret_key",
  "src": "/path/to/results",
  "infrastructure": "cluster_name"
}
```

9.4 Integration with SERRANO Services

9.4.1 Secure storage policies cognitive creation

The SERRANO platform supports creating automated secure storage policies based on significantly varying storage task requirements. In addition, SERRANO provides the intent-based definition of secure storage policies and their cognitive orchestration to abstract the infrastructure-specific requirements and operations from the end users regarding the choice of storage locations and redundancy, encryption, and compression parameters. This functionality is closely related to the developments in WP3 regarding the Secure Storage service and the respective use case. It also integrates the functionality of many platform

components, such as the Secure Storage service, the AI-Enhanced Service Orchestrator, the SERRANO Telemetry Framework, the Resource Optimization Toolkit, and the SERRANO Resource Orchestrator services.

From the SERRANO orchestration mechanisms perspective, the overall procedure is divided into three phases: (1) description of user intent and translation to infrastructure-specific objectives, (2) storage request orchestration, and (3) secure storage policy creation.

The initial phase is presented in Section 4.4. Next, the AI-Enhanced Service Orchestrator (AISO) triggers the execution of the second phase by passing to the SERRANO orchestration mechanisms the mapping of the user intent to infrastructure-specific parameters and high-level orchestration objections. To this end, the AISO uses the respective methods (*/api/v1/orchestrator/storage_policies*) (Figure 77) from the updated REST API of the SERRANO Orchestration API Server. The POST request supports the following parameters:

```
•  name: Optional[str] = None
•  description: Optional[str] = ""
•  policy_parameters: dict
```

The Orchestration API Server validates the request parameters and creates the corresponding Storage Policy object in the Datastore. The Orchestration Manager that watches the related topic (i.e., */serrano/orchestrator/storage_policies/policy*) for updates is triggered and, through its controllers, will initially handle the orchestration of the secure storage policy request (Steps 1-3 in the following workflow).

Next, the Orchestration Manager, through its Scheduler Controller, will request from the ROT the orchestration decision in order to create the secure storage policy based on the provided parameters (Step 4). The ROT Controller (Step 5) queries the Central Telemetry Handler (CTH) to get the available cloud and edge storage locations and creates the appropriate execution request (Step 6) that is assigned to one of the available Execution Engines. Next, the Orchestration Manager receives from the ROT the decision that includes parameters that will guide the storage policy creation (Step 7). Through the Orchestration API Server methods, the Orchestration Manager updates the *decision* field in the corresponding Storage Policy object (Step 8).

In the subsequent phase, the Orchestration Manager initiates the policy creation process by requesting it from the Secure Storage Service. In order to accomplish this, the Orchestration Manager formats the appropriate request based on the provided orchestration decision and triggers the creation process by executing the exposed REST method (*POST /storage_policy*) provided by the Secure Storage Service (Step 9). Regarding the request parameters, the users directly specify the *name* and *description* parameters. The remaining parameters, which determine the storage policy configuration, are compiled by the Orchestration Manager according to the ROT decision, considering the user intent, the deployment objectives, and the availability and characteristics of cloud and edge storage locations. Figure 86 summarizes the orchestration workflow for creating secure storage policies.

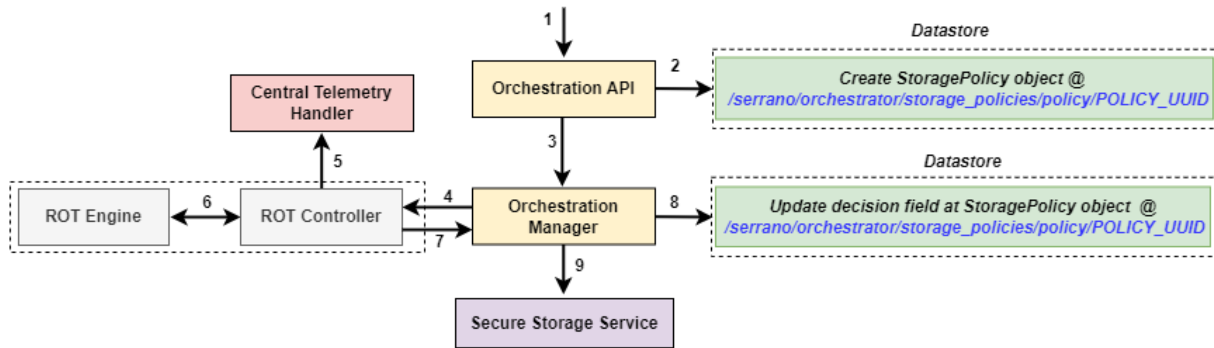

Figure 86: Secure storage policy cognitive creation – Orchestration workflow

Moreover, the SERRANO SDK provides the appropriate methods that abstract the overall workflow for defining, orchestrating, and creating secure storage policies. Figure 87 shows the corresponding code snippet. Users can use the provided Python methods to describe their intent (line 8) and request the creation of a storage policy (line 10). Then, they can check (line 12) and use the defined policy (line 36). The SERRANO-enhanced storage service exposes the Secure Storage API that allows SERRANO users to manage buckets and store and retrieve files. It is based on what can be considered the industry standard for object storage: Amazon Web Services S3. Deliverable D3.4 (M34) provides more technical details. Users can use the provided functionality using any S3 client library. In the example, we use the boto3 Python library to create a bucket based on the created storage policy (line 26).

```
1   import boto3
2   from botocore.client import Config
3   from botocore.exceptions import ClientError
4   import api.orchestrator.orchestratorAPI as serrano_api
5
6   orchestratorAPI = serrano_api.OrchestratorAPI()
7
8   description = {"size": 100, "cost": 5, "permanent_storage": 1, "availability": 0, "lat": 45.7472357," lng": 21.2316107}
9
10  policy = orchestratorAPI.serrano_create_storage_policy(name="demo-cc", policy_intent=description)
11
12  print(orchestratorAPI.serrano_get_storage_policy(policy["policy_uuid"]))
13
24  def create_bucket(bucket_name, storage_policy):
25      try:
26          return s3.create_bucket(Bucket=bucket_name,
27                                  CreateBucketConfiguration={'LocationConstraint': storage_policy})
28      except ClientError as e:
29          print(e.response['Error'])
30          return None
31      except Exception as e:
32          print(str(e))
33          return None
34
35
36  bucket = create_bucket("test_bucket", policy["name"])
37
```

Figure 87: Code snippet for creating and using a SERRANO secure storage policy

9.4.2 Cloud-native applications deployment

The SERRANO Resource Orchestrator implements essential functionalities to ensure efficient application deployment and resource orchestration in the disaggregated and heterogeneous SERRANO infrastructure. From a SERRANO orchestration perspective, the overall procedure is divided into three phases: (1) preparatory handling of deployment requests, (2) high-level cognitive resource orchestration, and (3) transparent application deployment. These phases are fully aligned with the workflows "*Cognitive resource orchestration operation within the SERRANO platform*" and "*Transparent application deployment operation within the SERRANO platform*" as outlined in the final SERRANO architecture specification in D2.5 (M18).

The initial phase is presented in Section 4.4. Next, the AI-Enhanced Service Orchestrator (AISO) triggers the execution of the second phase by passing to the SERRANO orchestration mechanisms the mapping of the user intent to infrastructure-specific parameters and high-level orchestration objectives. To this end, the AISO uses the appropriate REST methods (*/api/v1/orchestrator/deployments*) (Figure 77) exposed by the SERRANO Orchestration API Server. More specifically, the POST request supports the following parameters, while the PUT request requires the additional parameter *"deployment_uuid: str"*.

```
•   name: Optional[str] = None
•   user_token: Optional[str] = ""
•   deployment_description: str
•   deployment_objectives: Optional[List[dict]] = None
```

The Orchestration API Server validates the request parameters and creates the corresponding Deployment object in the Datastore. If the name is not defined, then the Orchestrator API service automatically sets as name the *deployment_uuid*, a parameter defined automatically by the Orchestrator API Server. Moreover, the schema includes two additional parameters corresponding to the two categories of input data that the Resource Orchestrator expects for handling the deployment requests. The *deployment_description* is mandatory and provides the YAML description of the application's microservices. The *deployment_objective* is optional and provides the objectives from the AISO for the orchestration algorithms. The Orchestration Manager that watches the related topic (i.e., */serrano/orchestrator/deployments/deployment*) for updates is triggered and, through its controllers, will initially handle the orchestration of the deployment request. In the following workflow, the initial phase corresponds to Steps 1-3, which trigger the ROT to provide the necessary orchestration decision for the application deployment.

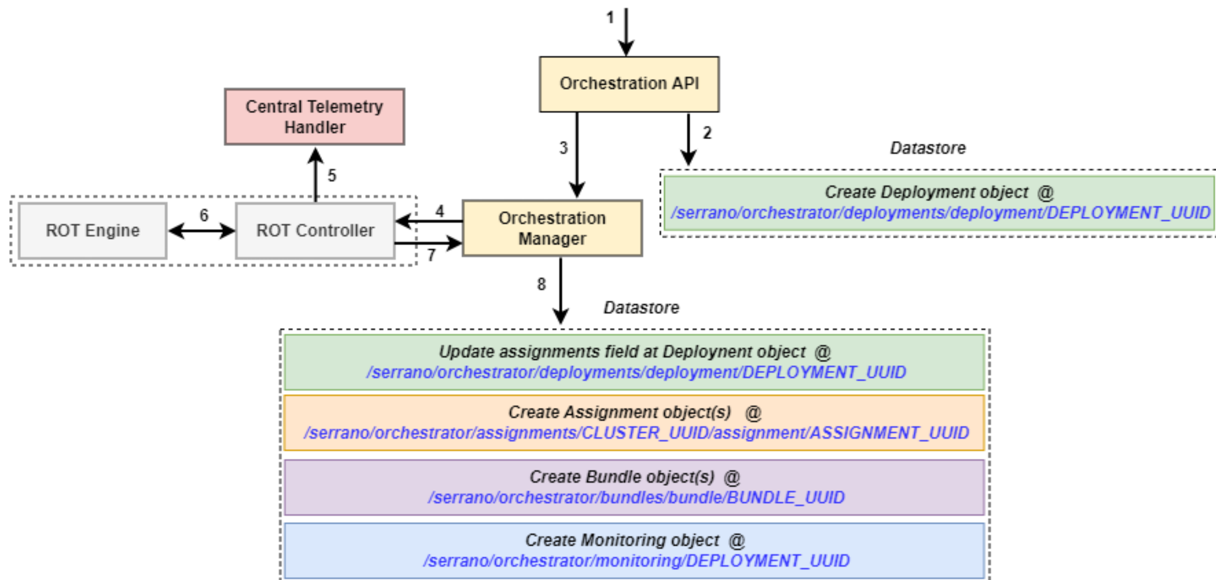

Figure 88: Application deployment – High-level cognitive orchestration workflow

Next, through its Scheduler Controller, the Orchestration Manager will request from the ROT the orchestration decision to guide the deployment of the requested cloud-native application in the SERRANO platform based on the user intent (Step 4). The request description to the ROT Controller includes the deployment objectives (i.e., *deployment_objectives* parameter), as provided by the AISO, and the application graph. The latter comes from analysing the provided application description (i.e., *deployment_description* parameter). This is required since the application description typically also includes objects such as ConfigMaps, Services, and Storage Volumes that are not required during the high-level orchestration. The ROT Controller (Step 5) queries the Central Telemetry Handler (CTH) to get the high-level description of the edge/cloud and HPC platforms that are under the management of the SERRANO platform and creates the appropriate execution request (Step 6) that is assigned to one of the available Engines. Next, the Orchestration Manager receives from the ROT the decision that includes parameters that will guide the application deployment (Step 7).

Then, the Orchestration Manager, through its Execution Controller, creates and stores in the Datastore the appropriate number of Assignment and Bundle objects (Step 8) according to the assignment of the application microservices into the individual edge, cloud, and HPC platforms as described in the ROT response. It also updates the corresponding Deployment object accordingly. The SERRANO orchestration mechanisms require all YAML descriptions corresponding to the same microservice to share common labelling (Figure 81). This is required when the ROT splits the application deployment in multiple SERRANO platforms since, in these cases, it is also required to include the respective supplementary descriptions, such as ConfigMaps, and Persistent Volumes, to the Bundles. To this end, we adopted a simple design approach in which all the related YAML descriptions share the same Label with the name ***"group_id"***. The mechanisms that provide the graphical-based definition and submission of applications (Section 4.5) also automatically support the definition of the required labelling.

The ROT orchestration decisions assign the applications' microservices to specific SERRANO platforms. These assignments are used to generate the Assignment objects in the Datastore. In addition to the assignments, ROT provides appropriate resource configurations to guide the platform-level orchestration mechanisms, such as the K8s scheduler, in making the final deployment decisions. The SERRANO Resource Orchestrator follows a declarative approach to describing the workload requirements to the Local Orchestrators. When creating Bundle objects corresponding to K8s Deployment descriptions, the Orchestration Manager automatically compiles the provided resource configurations to specific deployment requirements. The created description also includes a set of affinity and anti-affinity rules to be used by the Kubernetes scheduler to find out the most suitable nodes for the Pods deployment. These deployment requirements are added to the user-defined application description, resulting in cognitive-enhanced and platform-specific deployment requirements.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: position-service-classifier-training
spec:
  template:
    metadata:
      labels:
        project: serrano
        service: position
        microservice: classifier-training
    spec:
      runtimeClassName: kata-fc
      containers:
        - name: position-classifier-training
          image: serrano-harbor.rid-intrasoft.eu/position-classifier-training:0.1
      nodeSelector:
        serrano/security-tier:4
```

Figure 89: Kubernetes application deployment description enhanced by the SERRANO Resource Orchestrator

Figure 89 depicts a Kubernetes deployment description[4] that the SERRANO Resource Orchestrator has automatically enhanced for the *position-service-classifier-training* microservice from the application example in Figure 81. In the example, we consider a deployment intent that requested an advanced security layer for the microservice, while the orchestration mechanisms decided the deployment in nodes that provide maximum security, trust, and isolation (Tier-4). SERRANO builds on the confidential computing paradigm to provide different end-to-end secure tiers. Additional technical details for this topic are available in deliverable D3.4 (M30). The selected security level requires the existence of secure boot and trusted execution extensions in the worker node and the microservice deployment as a container sandboxed in microVM (Section 10.2). The YAML deployment description of the microservice is consequently enhanced with the appropriate labels and annotations to guide the platform-level scheduling mechanisms.

[4] For clarity, the deployment description includes only the parameters relevant to the provided example.

Finally, the Orchestration Manager creates the Monitoring object that will be updated with information about the deployed services in each platform during the deployment phase. The Orchestration API Server uses this information to configure the SERRANO telemetry services when a deployment request is served successfully.

The successful execution of the final step (Step 8) in the previous workflow triggers the third phase that handles the transparent application deployment across the SERRANO platform. In this phase, the Orchestration Drivers at the selected platforms receive the deployment instructions. Then, by interacting with the local orchestrator and the SERRANO-enhanced resources (Section 10), they trigger the actual deployment of the application's workloads with the selected runtime configurations. Figure 90 summarizes the workflow for deploying the application's workload in a selected cluster.

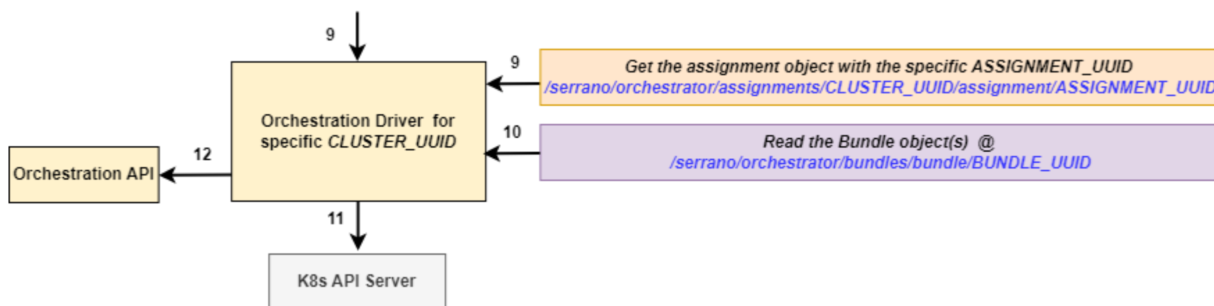

Figure 90: Cloud-native application deployment – Transparent deployment workflow

More specifically, the Orchestration Driver that manages a specific cluster detects a new assignment for its cluster (Step 9). The Assignment object that receives through the corresponding Datastore notification includes the list of all Bundles' unique identifiers related to the specific assignment. Next, for each Bundle's unique identifier in the list, the Orchestration Driver executes the following actions:

- It retrieves the Bundle description through the exposed GET method by the Orchestration API Server (Step 10).
- It uses the provided API by each local orchestration platform, such as the K8s API for the Kubernetes platforms, to apply the deployment actions that include the Bundle description (Step 11).

Next, the Orchestration Driver updates, through the Orchestration API Server, the status of the corresponding Assignment object (Step 12). The Orchestration API Server uses this information to determine if a deployment request has been served successfully.

Finally, the Orchestration Manager performs two additional actions when all the Assignments of a deployment request have been executed successfully. It informs, through the Central Telemetry Handler, the SERRANO telemetry framework to start the automatic monitoring of the deployed application and also registers the deployed application to the Service Assurance mechanisms.

9.4.2.1 Terminating application deployment

The SERRANO Resource Orchestrator also supports the deletion of a deployed application. The procedure involves all the Resource Orchestrator services (i.e., Orchestration API, Orchestration Manager, Orchestration Driver), while Figure 91 summarizes the workflow.

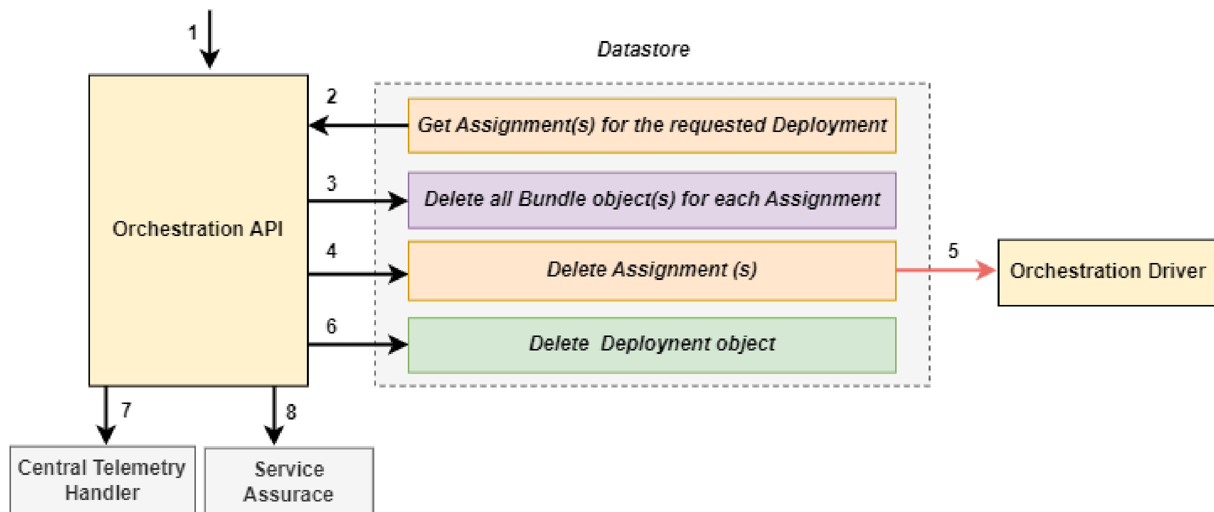

Figure 91: Terminating cloud-native application deployment

The Orchestration API Server receives the termination request through its exposed REST API (Step 1). It gets the list of Assignments that are related to the target deployment. For each Assignment, the Orchestration API Server gets the unique identifier for the Assignment's Bundles and uses them to delete the corresponding entries from the Datastore (Step 3). After deleting each Assignment's Bundles, the Orchestrator API Server deletes from the Datastore the Assignment itself (Step 4). This will trigger the involvement of the corresponding Orchestration Driver for that specific Assignment that, through the provided API by the local orchestration mechanisms, will terminate all the running instances in the specific platform (Step 5). Then, the Orchestrator API Server deletes the Deployment from the Datastore (Step 6). Finally, the Orchestration API service informs the SERRANO telemetry service (Step 7) and the Service Assurance mechanisms to update their operation accordingly (Step 8).

9.4.3 SERRANO HW/SW accelerated kernels execution

One of the innovations that SERRANO provides is the development of a library of accelerated kernels. These kernels harness both hardware and software acceleration techniques to enhance applications' performance and energy efficiency on cloud and edge devices, such as GPUs, FPGAs, and HPC platforms. The development of these kernels took place in WP4, and the associated deliverables (i.e., D4.1 (M15), D4.2 (M15), D4.3 (M15), and D4.4 (M30)) offer comprehensive technical information and extensive evaluation results for the kernels featured in SERRANO's use case applications. This section describes how the SERRANO orchestration mechanisms enable the seamless execution of these accelerated kernels across the heterogeneous SERRANO platform.

The SERRANO platform supports two deployment methods for the seamless execution of SERRANO-accelerated kernels across the heterogeneous edge, cloud, and HPC resources. These deployment methods are aligned with the serverless computing execution model, where users focus solely on developing their application's functions while the platform abstracts away the underlying servers and infrastructure, making it easier to deploy and manage applications.

The first method involves deploying the kernels alongside the application services. This method is suitable when the application services have a specific set of kernels that need to be executed repeatedly. The second method is based on the Functional as a Service (FaaS) execution model and allows on-demand deployment of accelerated kernels. FaaS is a specific serverless computing implementation that allows developers to trigger functions in response to events. These functions are stateless, meaning they do not maintain persistent connections or store data between executions. In this case, an application service running in the SERRANO platform through the SERRANO SDK can request the orchestration mechanisms to execute a specific kernel. The SERRANO orchestration and deployment mechanisms handle all the required operations and return the results to the application service.

The SERRANO platform ensures for both deployment methods that users receive seamless access to accelerated kernels without the need to manage the deployment and execution process. This approach also optimizes the use of resources, allowing kernels to be executed on the most appropriate resources. The SERRANO use cases use both deployment methods.

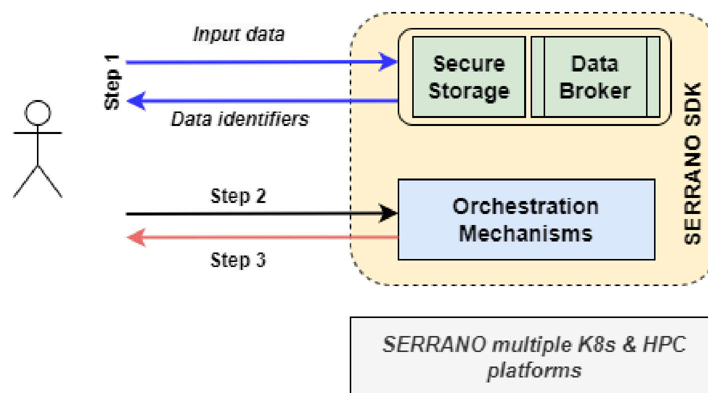

Figure 92: Kernel execution and data handling from the end user's perspective, common approach for all supported modes and platforms

To use either deployment method, data provisioning must be automated and abstracted into kernels, and results must be handled back transparently to users. From an end-user perspective, the overall process includes the following steps (as shown in Figure 92):

1. Move input data to SERRANO storage services (i.e., Data Broker, Secure Storage Service) and retrieve the corresponding description to pass to the execution request (Step 2). This description contains a set of identifiers for the SERRANO deployment mechanisms (e.g., Orchestration Drivers, HPC Gateway, Lightweight Virtualization Mechanisms) to download the data and prepare them for use by the kernels.

2. Submit the request to the SERRANO platform, specifying the kernel, input data description from Step 1, and any other options necessary for configuring the kernel execution.
3. Retrieve the results using the SERRANO Python API method provided.

The SERRANO SDK provides suitable APIs that abstract the interaction with the various SERRANO platform services to facilitate these operations. Deliverable D6.5 (M27) provides a complete example regarding the execution of a SERRANO-accelerated kernel through the provided Python API.

10 Lightweight Virtualization Mechanisms

Running applications in the cloud has changed the way users develop and ship their code. During the past decade, applications were deployed in the cloud using conventional Virtual Machines (VMs) following the Infrastructure-as-a-Service (IaaS) model. Users choose the setup of their virtual hardware, install their preferred OS, and deploy their application / service on top of that VM.

However, quite recently, the community has given rise to other approaches [105][108], which were quickly adopted by cloud vendors, towards solutions that follow the paradigm of Platform-, Software-, and Function-as-a-Service (PaaS, SaaS, and FaaS respectively). These approaches offer performance and flexibility improvements over IaaS by decoupling the application from the infrastructure. Providing a common OS stack, maintained by the provider and optimized for the specific hardware it is running on, is much more efficient than exposing a generic virtual hardware interface. Additionally, users seek to maximize the number of requests handled while minimizing request/response latency. Apart from the cloud paradigm, Edge computing is slowly adopting these modes of operation, especially in the context of IoT and 5G [109].

In the IaaS case, the burden of orchestrating and optimizing the systems software stack running on top of virtual hardware is passed to the user, while in the other cases, the vendor exposes a customized interface tailored to the application / service offered. The cloud-native [110] concept emerged from this trend as a need to reduce bloated interfaces and abstractions that introduced significant overhead for application deployment and execution. Containers played an important role towards cloud-native embracement; they have revolutionized deployment by facilitating application packing and dependency tracking, and reducing the overheads of execution; however, this comes at the cost of security and isolation [111]. As a result, cloud vendors fall back to generic virtualization techniques: Microservice offerings are essentially VMs running the vendor's custom systems stack, exposing a language runtime, a specific service such as a Database Management System (DBMS), a LAMP stack or just container host-side systems software. For instance, to provide a secure Serverless environment where users deploy their functions at will, cloud providers either: (a) spawn a VM per tenant, install their Serverless backends there, and keep it hot while the user submits functions to be executed; (b) spawn VMs which host containers per tenant, with the necessary software installed, and execute the user function there; or (c) spawn microVMs [112] per tenant where isolation is provided by the microVM monitor [113].

Figure 93 captures a snapshot of the traditional mode of execution for a generic VM on Linux/KVM using a standard user-space Virtual Machine Monitor (VMM). The Kernel-based Virtual Machine (KVM) module in the Linux kernel interfaces with the VMM, which essentially handles privileged operations (*VMExits*). So, when a privileged operation needs to be executed in the guest, the system traps it in the host's kernel-space (KVM), which, in turn, delivers this event to the monitor in user-space. Upon completion, the execution returns to the kernel, which, in turn, kicks the vCPU of the guest via a *VMEnter*. This process is a design choice: for

instance, QEMU supports a full operating system stack, emulates several architectures / features, and makes perfect sense for this code to be in user-space.

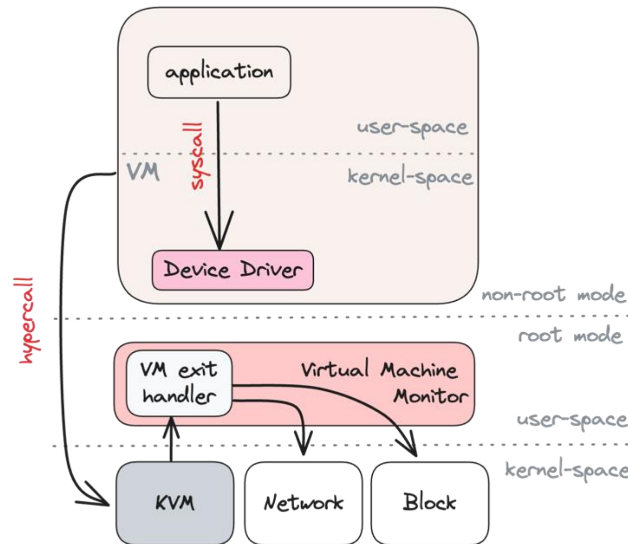

Figure 93: A Virtual Machine running on a generic user-space VMM on top of KVM

On the other hand, in the context of Serverless Computing [114], cloud-native applications, and lightweight execution, this process seems too complicated. Why should the system hand over the event to user-space since the only operation needed to be performed will probably be in the kernel (network, storage I/O, etc.)? This is the *vhost* approach [115] taken to decouple the control path from the data path when performing high-performance I/O.

Apart from optimizing the data path, researchers have done considerable work to minimize the overhead of VM spawn and execution. Several minimalistic approaches have been proposed regarding the systems software stack to facilitate fast and secure application execution in the cloud. For instance, Unikernel as Processes [116] describes a specialized VMM with a number of backends (e.g., seccomp, KVM, muen, etc.) that minimizes the attack surface by limiting the interface with the underlying layers. NEMU [117] is a stripped down QEMU, specifically built and designed to run modern cloud workloads on x86_64 and ARM CPUs. Amazon's Firecracker [113] is a fork of [118], a lightweight VMM, built to deploy microVMs, which feature enhanced security and workload isolation over traditional VMs.

Most of these approaches use the Linux kernel as the guest OS. This implies that although the user just needs to execute a function, the cloud provider must spawn a Linux kernel guest, or a container, from scratch and then run the function in this environment. More importantly, in all of the above cases, when a *VMexit* happens, the mode of execution still needs to be passed on to the monitor (first mode-switch) to service the exit and then back to KVM to resume the guest (second mode-switch).

Simplicity is key when designing an application execution stack: users want to run their code fast and get a result back. They do not care where the code will run as long as there is reproducible, fast, and secure execution. To this end, lightweight virtualization appears mutually beneficial to cloud vendors and users: the former increase resource utilization by

consolidating more tasks to nodes; the latter enjoy fast service response times and (potentially) lower-cost services.

We adopt a hybrid approach in the SERRANO platform to balance the trade-offs between lightweight application execution and workload isolation/security. To this end, we enable the deployment of workloads in various execution modes, such as generic containers, sandboxed containers, and unikernels, using the necessary virtualization mechanisms for each mode. The following sections detail the technologies involved in the SERRANO workload deployment mechanisms.

10.1 Efficient Sandboxing of Containers on Edge Nodes

Containers and their benefits

Containers are lightweight, self-contained execution environments that encapsulate applications and their dependencies, providing a consistent and reproducible runtime environment. They enable the packaging of software in a manner that allows it to run reliably and consistently across different computing environments, including any computer hardware, infrastructure, or cloud environment. Achieving this versatility is made possible through a combination of operating system-level virtualization and resource isolation techniques. By leveraging kernel features, containers create isolated environments with their file systems, network interfaces, and process trees, ensuring application isolation from other containers and the host system. Moreover, containers abstract away the underlying infrastructure, allowing applications to be developed and deployed with minimal concern for specific hardware or software configurations. Unlike virtual machines, containers do not require a guest OS in each instance, resulting in smaller, faster, and highly portable units that can be executed on desktops, traditional IT systems, or in the cloud. By utilizing the features and resources of the host operating system, containers provide an efficient and platform-independent execution environment, making software deployment portable, scalable, and manageable across various environments, from development to production, on-premises, or in the cloud. Figure 94 shows a high-level overview of the node-level flow for a container spawn.

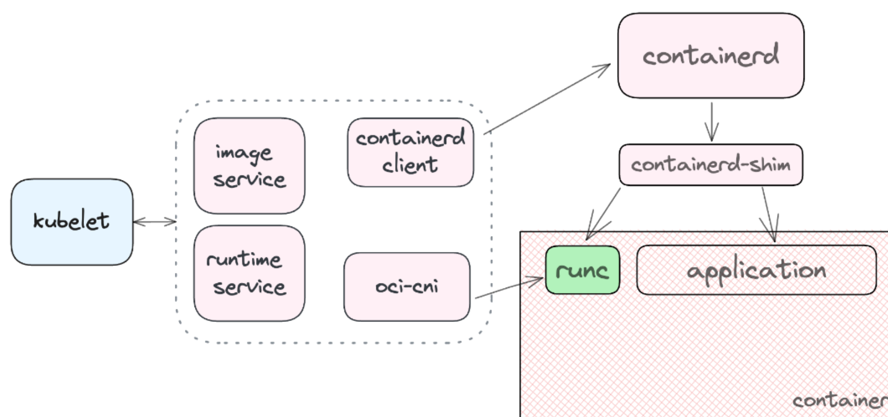

Figure 94: High-level overview of generic container spawning in a k8s environment

Limitations of containers

Despite their numerous advantages, containers have certain limitations, particularly in terms of security and isolation. Although containers provide a level of isolation by leveraging operating system features, they still share the underlying operating system kernel. This shared kernel introduces potential security risks, as a compromise within the kernel could impact the security and integrity of all containers running on the same host. Additionally, containers may not provide sufficient isolation for certain sensitive workloads or applications with strict security requirements. Furthermore, containers may face challenges when handling specific types of workloads, such as those with strict real-time requirements or resource-intensive applications that demand fine-grained control over hardware resources.

Sandboxing and its importance

To address the limitations of containers in terms of security, isolation and resource control, container sandboxing comes into play. By encapsulating containers within microVMs, each with its dedicated kernel instance, stronger isolation and security are achieved. The use of microVMs ensures that any compromise within a specific microVM remains contained, mitigating potential security risks from the shared underlying kernel. Moreover, container sandboxing resolves the insufficient isolation for sensitive workloads by providing a secure and isolated execution environment for individual containers within each microVM. With container sandboxing, containers can overcome their limitations in terms of security, isolation, and handling diverse workloads, making them more robust and suitable for a wide range of applications across various domains.

Figure 95 presents the high-level concept of container sandboxing using kata-containers.

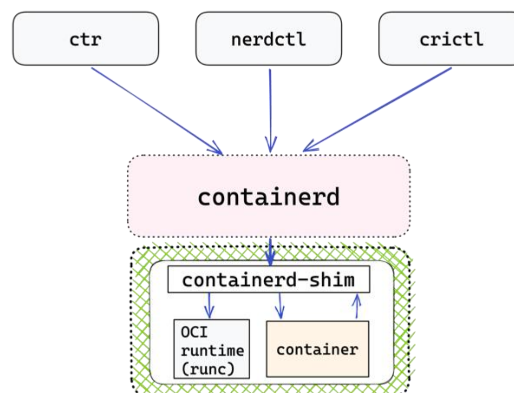

Figure 95: Container sandboxing

Sandboxing in Edge Devices

Edge devices are computing devices positioned in proximity to where data is generated or consumed. Edge computing offers multiple advantages. Firstly, it reduces the need for data transfer to centralized locations, resulting in reduced network congestion and latency. Local data processing on edge devices enables faster response times, crucial for real-time. Secondly, edge devices enhance privacy and data security. Processing data locally minimizes the risk of data breaches during transmission, ensuring compliance with stringent privacy regulations.

Lastly, edge devices improve reliability and availability. By distributing computing resources across multiple devices, the system becomes less reliant on centralized infrastructure, making it more resilient to network outages or connectivity issues.

Edge devices pose significant challenges in terms of limited computing resources and their heterogeneous nature. Firstly, edge devices often have constrained processing power, memory, and storage capacity, which can impact the performance and scalability of applications deployed on them. Secondly, the heterogeneous nature of edge devices introduces complexities. Different devices may have varying hardware capabilities, operating systems, and available hardware accelerators. This requires developing specialized software that can seamlessly run across diverse edge devices, accommodating their unique characteristics. Developers must account for compatibility issues, adaptability, and the need for device-specific optimizations. In addition to these challenges, multitenancy adds another layer of complexity. When multiple deployments share the same edge devices, isolation becomes crucial to ensure the security and integrity of each application and its data. Sandboxing techniques, such as microVMs, can be employed to provide isolated execution environments for individual deployments, mitigating the risk of interference or unauthorized access.

Deploying multiple sandboxed containers in a single edge device can be challenging due to resource limitations and potential application conflicts. Edge devices often have constrained processing power, memory, and storage capacity, making it difficult to allocate sufficient resources to each container without impacting performance. Concurrently running multiple containers on the same device can lead to resource contention and interference, jeopardizing isolation and security.

Container orchestration platforms, like Kubernetes, provide advanced management capabilities for scaling and load balancing containers, ensuring efficient resource distribution. Lightweight virtualization technologies, such as specialized container runtimes and microVMs, enable isolation between containers and enhance security. By encapsulating each container's execution environment, sandboxing techniques mitigate interference and unauthorized access risks.

10.2 Sandboxed Containers

In the cloud environments and microservices age, containers have become very prominent, which is why we need to safeguard their execution while keeping their speed and portability intact. Kata Containers [125] is an open-source project that aims to provide a secure and lightweight runtime environment for containerized applications. It leverages hardware virtualization technologies to offer strong separation between containers while maintaining the performance advantages of lightweight containers. Launching a container in a micro VM, managed by a hypervisor, with its own kernel and root file system, ensures enhanced security and isolation, defending the application from remote execution, memory leaks, or unprivileged access, as well as protecting the host in case of untrusted or untested programs.

Table 19 summarizes the properties of various environments where processes run.

Table 19: Process execution environment

Type	Name	Virtualized	Containerized	rootfs	Rootfs Device Type	Mount Type
Host	Host	No	No	Host specific	Host specific	Host specific
VM root	Guest (VM)	Yes	No	rootfs inside the guest image	Hypervisor specific	ext4
VM container root	Container	Yes	Yes	rootfs type requested by user	kataShared	virtioFS/ snapshotter

Kata Containers has a list of design requirements that its runtimes always guarantee to fulfil:

- OCI compatibility
- runc CLI compatibility
- CRI and Kubernetes support
- Multiple hardware architectures support
- Multiple hypervisor support
- Virtualization overhead reduction
- Networking & Storage compatibility
- I/O acceleration & scalability
- CI and structured logging

To address these requirements, the container runtime has been designed as a modular system, using various components, each with a unique task to actualize an end-to-end container spawn. The most prominent components are:

- Shim: containerd shimv2 implementation
  - handles the shim process
  - runs the ttRPC service in the shim side
- Service: services for containers
  - implements the ctr shim protocol and interacts with runtimes through messages.
- Runtimes: container runtimes
  - addresses messages from task services to manage containers.
  - contains the sandbox and the container manager.
- Resource: abstractions for resources
  - sandbox resources: network, share-fs
  - container resources: rootfs, volume, cgroup
- Hypervisor: manager for the VM
- Agent: used to communicate with the guest OS from the shim side

The runtime is compatible with the containerd runtime shimv2 architecture and complies with the Open Container Initiative (OCI) runtime specification, making it Kubernetes-compatible with either CRI-O or the equivalent containerd implementation. A single runtime shim is also sufficient to manage all the OCI containers of an entire pod. Kata utilizes its agent, a daemon process, to establish robust communication between the guest and the host through a vshock socket operating on a ttRPC-based protocol. This approach enables the exchange of container management commands as well as carry the standard I/O streams between a container and its manager, eliminating also the need for multiple runtime calls. Such a structure ensures that Kata remains agnostic to the sandbox mechanisms, allowing a plethora of different hypervisors and different types of containers such as WebAssembly (Wasm) or Linux, not just virt ones, suiting different demands and preferences.

Our job is to provide the container images and the VM resources (kernel and image) while Kata handles the rest. After loading the kata configuration file, a set of shimv2 API functions are called to commence the runtime instance, which starts the hypervisor. Inside the VM rootfs resides the kata agent, which is also initiated as part of the VM boot, and after the sandbox is ready, it gives the signal for the container spawn that uses the OCI bundle from the container image and has been passed from the host to the guest beforehand, to be used as its root file system. The container init process and all ensuing ones, as well as the I/O go through the VMM interface, and the desired isolation has been achieved.

There are currently two different runtimes. The *runtime* is the default one, written in Golang, while the *runtime-rs*, which utilizes Rust, is under development and was created by a need for better container startup speed, resource consumption, stability, and security. In the context of the SERRANO project, we ported AWS Firecracker to the Rust runtime. While the Go runtime features plenty of different hypervisors (ACRN, Cloud Hypervisor, Firecracker, and QEMU), the Rust runtime has the built-in option of Dragonball, the default hypervisor explicitly implemented for it. In contrast, the rest of the hypervisor options, like QEMU and Cloud Hypervisor, are not yet integrated.

Our implementation provides the ability to generate a working Firecracker hypervisor instance that can spawn a VM and subsequently containers with all the needed features of a VMM together with full networking functionality, the option to jail the sandbox and its resources along with the capability to hot plug block devices, by patching them to pre-inserted dummy drives, as Firecracker does not support filesystem sharing.

Additionally, we have integrated the vaccel-agent as a part of kata containers, both embedded in the runtime and also as standalone execution, providing the option for extra computational power if the workload requires it, through hardware acceleration, without involving direct hardware/device access.

10.3 Unikernels as Containers

To bridge the gap between containerized environments and unikernels, enabling seamless integration with cloud-native architectures, we introduce urunc. Designed to fully leverage the container semantics and benefit from the OCI tools and methodology, urunc aims to become “runc for unikernels”, while offering compatibility with the Container Runtime Interface (CRI). By relying on underlying hypervisors, urunc launches unikernels provided by OCI-compatible images, allowing developers and administrators to package, deliver, deploy, and manage their software using familiar cloud-native practices.

10.3.1 bima: unikernel container images

The first step to enable this functionality is to pack a unikernel into an OCI-compatible container image. To achieve this, we build *bima*, a software tool that embeds a unikernel image, metadata and its dependencies into a layered OCI container image.

Figure 96 shows how to pack a unikernel image into an OCI-compatible container image, using *bima*.

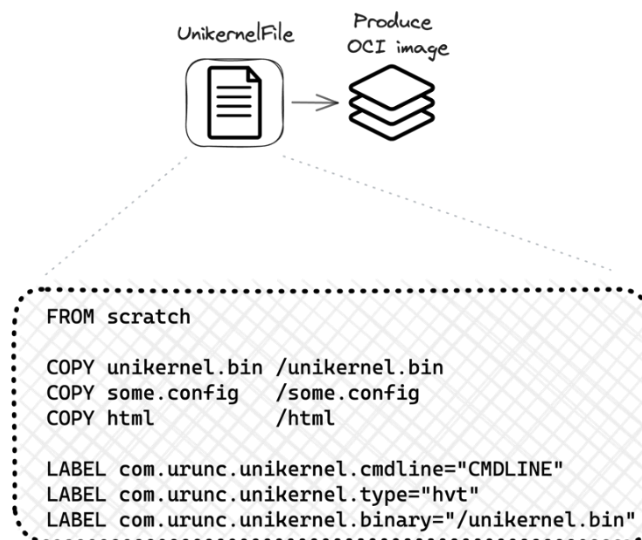

Figure 96: Packing a unikernel as an OCI-compatible container image

bima builds an OCI-compatible Container Image from a special type of Containerfile that supports a minimal set of instructions: FROM, COPY, and LABEL. The images built by bima are intended to be run by urunc, so there is no compatibility with other container runtimes. However, they can be pushed and pulled from generic container image registries, such as Docker Hub and on-premise Harbor installations.

- *FROM*: It is not taken into account in the current implementation, but we plan to add support for it.
- *COPY*: It works as in Dockerfiles. The current implementation supports only one copy operation per "instruction" (think one copy per line).

- *LABEL*: All LABEL "instructions" are added as annotations to the Container image. They are also added to a special urunc.json inside the container's rootfs.

Due to the tight coupling between bima and urunc, the few annotations required for urunc to work are also required by bima.

The required annotations are the following:

- *com.urunc.unikernel.unikernelType*: The type of the unikernel (can be rumprun, unikraft, etc)
- *com.urunc.unikernel.hypervisor*: The desired hypervisor to run the unikernel (e.g. qemu, hedge, hvt)
- *com.urunc.unikernel.binary*: The unikernel binary to run
- *com.urunc.unikernel.cmdline*: The cmdline used to run the unikernel

The produced image's platform OS is always Linux, while the platform architecture is automatically extracted from the ELF headers of the file defined in *com.urunc.unikernel.binary* annotation.

A sample Containerfile should look like the following:

```
# the FROM instruction will not be parsed
FROM scratch

COPY test-redis.hvt /unikernel/test-redis.hvt
COPY redis.conf /conf/redis.conf

LABEL com.urunc.unikernel.binary=/unikernel/test-redis.hvt
LABEL "com.urunc.unikernel.cmdline"='{"cmdline":"redis-server /data/conf/redis.conf",\
"net":{"if":"ukvmif0","cloner":"True","type":"inet","method":"static","addr":"10.0.66.
   2","mask":"24","gw":"10.0.66.1"},\
"blk":{"source":"etfs","path":"/dev/ld0a","fstype":"blk","mountpoint":"/data"}}'
LABEL "com.urunc.unikernel.unikernelType"="rumprun"
LABEL "com.urunc.unikernel.hypervisor"="qemu"
```

10.3.2 urunc: a unikernel container runtime

Figure 97 presents a high-level architecture diagram of urunc and its interaction with the rest of the components in a generic container environment.

To delve into the inner workings of urunc, the process of starting a new unikernel "container" via containerd involves the following steps:

1. Containerd unpacks the image onto a devmapper block device and invokes urunc.
2. Urunc parses the image's rootfs and annotations, initiating the required setup procedures. These include creating essential pipes for stdio and verifying the availability of the specified vmm.

3. Subsequently, urunc spawns a new process within a distinct network namespace and awaits the completion of the setup phase.
4. Once the setup is finished, urunc executes the vmm process, replacing the container's init process with the vmm process. The parameters for the vmm process are derived from the unikernel binary and options provided within the "unikernel" image.
5. Finally, urunc returns the process ID (PID) of the vmm process to containerd, effectively enabling it to handle the container's lifecycle management.

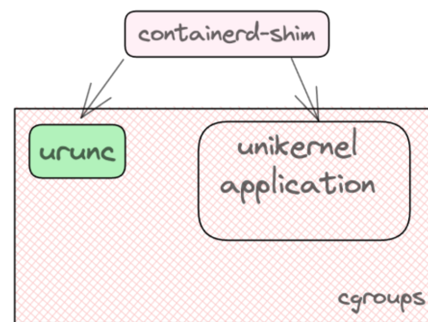

Figure 97: Running an unpacked container image as a unikernel

Unikernels hold great potential for utilization in serverless deployments. With their lightweight nature, ultra-fast boot times, and singular purpose, unikernels align perfectly with the requirements of short-lived, single-purpose serverless functions. By leveraging urunc, developers can seamlessly deploy and manage unikernel-based serverless applications in a cloud-native manner. Combining unikernels and serverless computing enables efficient resource utilization, rapid scaling, and optimal performance, opening up possibilities for building highly efficient and responsive cloud-native applications.

Incorporating unikernels into the container ecosystem through urunc unlocks the benefits of both technologies. Unikernels provide better performance, security, and resource efficiency, while urunc enables seamless integration into cloud-native environments by embracing container semantics and OCI compatibility. This powerful combination empowers developers to leverage the advantages of unikernels while utilizing the robust orchestration capabilities, scalability, and ecosystem of cloud-native architectures.

10.4 microVM Optimizations

The virtualization layer is a vital component of the software system stack in edge computing. Virtualization allows the abstraction of the underlying resources and enables the concurrent execution of workloads from various tenants in an isolated environment. Nonetheless, this comes at the cost of consuming more resources and adding overhead to the overall execution of a workload. Consequently, the virtualization layer must be as lightweight as possible while not compromising the isolation and fair execution among the different tenants.

Containers have dominated the cloud. Instead of virtualizing the entire system, containers use Operating System mechanisms to provide the necessary isolation. Such a design requires fewer resources than traditional virtualization since the virtualized environment is much

smaller. Furthermore, containers can achieve better performance, especially in the case of I/O and boot times, since the applications can directly communicate with the host Operating System without the mediation of any other software (e.g., hypervisor). On the other hand, relying on pure software solutions by sharing the Operating System among different tenants raises concerns regarding the level of isolation that containers provide. To this end, several recent studies have proved that container isolation is much weaker than traditional virtualization techniques. As a result, container deployment usually occurs inside virtual machines, increasing the overall system software stack.

Under these circumstances, traditional system-level virtualization is the only feasible solution in order to provide strong isolation. In system-level virtualization, a virtual machine monitor (VMM) creates an entire virtual machine, and a different Operating System runs inside. In most cases, the virtual machine monitor is a user-space application that interacts with the host Operating System to create and manage the virtual machines. As a result, researchers and engineers focus on reducing the overhead that the virtual machine monitor induces and optimizing the I/O performance.

In this context, microVMs have emerged. Instead of using an entire Operating System inside a virtual machine, microVMs use a minimal kernel and only the necessary components to execute applications. The lightweight virtual machines are much more scalable since they can quickly boot and shut down while reducing resource consumption. Such virtual machines also require fewer functionalities from the underlying hypervisor. As a result, new hypervisors can only support the necessary functionalities, specifically for microVMs, reducing their codebase and the overhead of setting up the environment for the virtual machine.

VMM constant mode switching between the host OS kernel and user-space can be expensive and redundant, but in some VMM designs, emulation of I/O devices makes it necessary. However, especially in the cloud, the VMM and the VMs mostly use virtual devices for I/O. In this context, the I/O request could be handled directly by the host OS without VMM mediation. Vhost follows such an approach and allows VMM to offload the data plane to another component, which could run inside the host OS. Vhost manages to improve the overall I/O performance significantly. Nonetheless, with vhost, the guest-host communication operates asynchronously, requiring a thread to poll for the latest data. Using threads for polling might be fine on high-end servers with multicore CPUs, but it can create issues for edge devices with limited cores. Thus, despite the benefits of vhost, such technology only applies to some edge devices.

Virtual Machine Monitor

A hypervisor can be simple and minimal; the example of the solo5 unikernel showcases an interesting aspect of I/O in hardware virtualization. When a privileged operation (like a network I/O request) occurs in the guest, the system traps (*VMExit*) in the host kernel (KVM), and then it is delivered back to the user space monitor. In turn, the monitor handles the request from the guest and asks KVM to resume the guest execution. While this path seems appropriate in the typical case (such as with general-purpose hypervisors, where different architectures or devices are emulated), in the case of lightweight virtualization, an additional

and unnecessary switch from kernel space to user space incurs significant overhead. For instance, during a network I/O request, the host kernel will return the control to the user space monitor in order to handle the guest's request, and the user space monitor will eventually make a system call to transmit or receive the network packet, returning the control to the host kernel. We want to explore how significant the overhead of these mode switches is and find solutions that can substantially reduce the overhead.

To entirely remove this overhead, we designed and implemented HEDGE, a minimal and simplistic VMM that resides inside the Linux kernel interacting directly with KVM without any intervention from the user space. HEDGE is a simple dispatch handler in the kernel that services a guest's needs. It provides an interface to the KVM API, a Virtual Machine execution environment for each of the VMs spawned, generic device handling (network & block), and a management layer to perform basic VM operations (create, destroy, dump console, etc.).

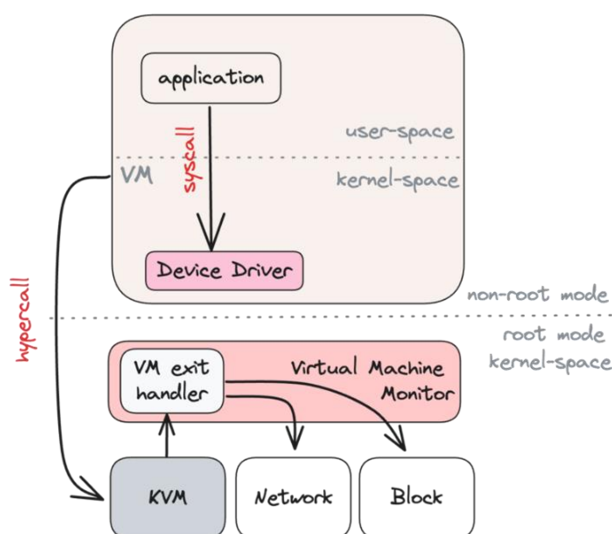

Figure 98: A unikernel running as a VM on HEDGE

A major challenge in this approach is that KVM targets user space processes, providing an API through file descriptors. Moreover, using KVM's API from inside the kernel is impossible because most needed functions are only used inside KVM. A way around this is to create a glue code, which is some wrappers of KVM functions, between HEDGE and KVM to expose all the needed functionality. For that reason, two small patches are required to be able to use HEDGE. In all other cases, HEDGE works similarly to most user space VMMs. As in the case of QEMU/KVM each VM is associated with one kernel thread, which implements the vCPU. The thread's life cycle begins when the HEDGE receives a request to spawn a new VM and handles all privileged operations (*VMExits*). A worth noting design choice that we made is that the new kernel thread will have its own memory mappings (*mm struct*). Moreover, HEDGE allocates a virtual memory, which will serve as the guest's memory, and maps it to a virtual address of the newly created kernel thread's memory area. Thereby, the kernel thread mimics a user space process, tricking KVM that it gets used from user space.

An important aspect of HEDGE's design is reducing the noise VMMs enforce to handle I/O requests. Performance is one of this project's primary goals; to achieve that, the guest needs

to run uninterrupted as much as possible. Besides removing the mode switch overhead, HEDGE handles I/O requests with the minimum possible overhead. The simple and minimal hypercall Application Binary Interface (ABI) from Solo5 helps in that direction. Network packages are formed from the guest, and when the I/O request occurs, the job of HEDGE is as simple as forwarding the frame to the appropriate network interface. Receiving packages follow the opposite route. Every guest is associated with a virtual interface (TAP), and we use raw ethernet sockets to receive and send network packets on behalf of the guest. Regarding block device support, HEDGE leverages the device mapper (DM) functionality to create a virtual block device mapped to a physical device. Using the block read/write hypercalls from Solo5 ABI, the guest makes I/O requests, which are translated to read/write calls in the kernel to the DM block device. However, the plan is to add support for VirtIO in an effort to host more unikernel frameworks and even basic functionality of a Linux guest.

As with every VMM, HEDGE provides its management interface. For the time being, it is minimal and can handle basic VM operations such as start, stop, etc. One can easily manage HEDGE both locally (user space) or remotely. In that manner, HEDGE can be easily managed in cases where user space access is impossible, such as edge nodes. In both cases, HEDGE can be managed by the following commands:

- *Load*: Loads a module (VM image) and prepares its deployment.
- *Start*: Executes the selected module.
- *Stop*: Stops the execution of a VM.

Moreover, a user can select which block or net device will be used, specify the command line arguments for the guest, and dump the guest's console output. Furthermore, a user can access statistics such as boot and setup times, I/O operations (both disk and network), and generic stats regarding HEDGE, such as the number of VMs, memory consumption, and more. Someone can interact with the management interface locally via a specialized filesystem in the Linux kernel, *procfs*. When HEDGE is loaded, two new files and one directory are created under */proc* directory:

- */proc/monitor*: I/O file that can be used to control the hypervisor and its virtual machines.
- */proc/vmcons/VMID*: I/O file which keeps the output of the virtual machine.
- */proc/vmstats/VMID*: A directory which keeps stats for hypervisor, and virtual machines

On the other hand, one can interact with the network management interface. In that case, the commands are sent over UDP, while the files can be transmitted over *tftp.*

In the context of the SERRANO platform, we have enhanced HEDGE to support generic Linux distributions as well. However, as our focus is on Unikernels, we have ported and successfully executed unikraft [123], rumprun [124], solo5, and OSv unikernels. Additionally, we are in the process of integrating HEDGE with urunc to support the secure and efficient end-to-end deployment of workloads through the SERRANO orchestrator to edge devices.

10.5 Hardware Acceleration

In the context of edge devices, utilizing specialized hardware components or accelerators to offload and enhance specific computing tasks plays a vital role in boosting performance and efficiency. By leveraging hardware accelerators like GPUs or FPGAs, sandboxed containers can delegate computationally intensive tasks, leading to swifter and more efficient execution. This approach not only improves the overall application performance but also optimizes the utilization of resources on edge devices. Hardware acceleration empowers edge nodes to effectively handle challenging workloads, such as real-time data processing, AI inferencing, or video transcoding. The result is improved responsiveness, decreased latency, and increased energy efficiency, ultimately enhancing the capabilities of edge computing.

In SERRANO, we build and enhance the vAccel framework to enable interoperable hardware acceleration to workloads deployed as container images in various modes of execution: containers, sandboxed containers (in microVMs), and unikernels. More details on the vAccel framework can be found in D4.4 (M30).

The integration of the vAccel framework with the custom container runtimes we build in the context of T5.5 is twofold:

- First, we integrate vAccel to the container runtimes we build and enhance (kata-containers) to support hardware acceleration functionality in instances that do not have direct access to hardware accelerator devices.
- Second, we enable vAccel in a multi-tenant Serverless environment using OpenFaaS, K8s, and our custom container runtimes.

The integration of vAccel to kata-containers has been implemented in both runtimes (Go and Rust), as mentioned in Section 10.2. In the Go runtime, we only support AWS Firecracker as the sandboxing mechanism, whereas in the Rust runtime we enable support for all available hypervisors.

Working towards the final version of the SERRANO integration platform, we will showcase the end-to-end serverless instantiation of hardware-accelerated kernels, as well as the containerized mode of deployment with and without vAccel.

11 Conclusions

In this deliverable, we present the work of all tasks in WP5, with a particular focus on the second phase of the work package implementation (M16-31). Specifically, we elaborate on the final design and developments for: (i) the ARDIA (A Resource reference model for Data-Intensive Applications) modelling framework, (ii) AI-Enhanced Service Orchestrator, (iii) multi-objective resource allocation and service orchestration optimization algorithms, (iv) AI/ML-driven service assurance and re-optimization mechanisms, (v) energy and resource-aware flow mappings, (vi) novel network and cloud telemetry framework, (vii) hierarchical resource orchestration, and (viii) lightweight virtualization mechanisms.

The provided developments are integral parts of the cognitive orchestration and transparent deployment mechanisms of the SERRANO complete platform prototype that will be used for the final performance evaluations.

Overall, this document presents the research and development activities of WP5 and builds upon the initial developments reported in M15 at deliverables D5.1, D5.2, and D5.3 to provide the remaining functionality and implement the complete interfaces for inter-component communication.

The above developments will be further enhanced as we move towards the final integration of the related components into the final release of the SERRANO integrated platform.

12 References

[1] Cao, Keyan, et al. "An overview on edge computing research." IEEE access 8 (2020): 85714-85728

[2] Qasaimeh, Murad, et al. "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels." 2019 IEEE international conference on embedded software and systems (ICESS). IEEE, 2019

[3] Alveo U50 Data Center Accelerator Card: https://www.xilinx.com/products/boards-and-kits/alveo/u50.html

[4] García-Martín, Eva, et al. "Estimation of energy consumption in machine learning." Journal of Parallel and Distributed Computing 134 (2019): 75-88

[5] Kubernetes YAML Generator: https://k8syaml.com/

[6] Alien4Cloud: https://alien4cloud.github.io/index.html

[7] TOSCA: https://www.oasis-open.org/committees/tosca/

[8] Pallewatta, S., Kostakos, V., & Buyya, R. (2019). Microservices-based IoT application placement within heterogeneous and resource constrained fog computing environments. UCC 2019 - Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, 71–81. https://doi.org/10.1145/3344341.3368800

[9] Santoro, D., Zozin, D., Pizzolli, D., de Pellegrini, F., & Cretti, S. (2018). Foggy: A Platform for Workload Orchestration in a Fog Computing Environment. https://doi.org/10.1109/CloudCom.2017.62

[10] Mutlag, A. A., Ghani, M. K. A., Mohammed, M. A., Lakhan, A., Mohd, O., Abdulkareem, K. H., & Garcia-Zapirain, B. (2021). Multi-agent systems in fog–cloud computing for critical healthcare task management model (CHTM) used for ECG monitoring. Sensors, 21(20). https://doi.org/10.3390/s21206923

[11] Alfakih, T., Hassan, M. M., Gumaei, A., Savaglio, C., & Fortino, G. (2020). Task offloading and resource allocation for mobile edge computing by deep reinforcement learning based on SARSA. IEEE Access, 8, 54074–54084. https://doi.org/10.1109/ACCESS.2020.2981434

[12] Wang, S., Guo, Y., Zhang, N., Yang, P., Zhou, A., & Shen, X. (2021). Delay-Aware Microservice Coordination in Mobile Edge Computing: A Reinforcement Learning Approach. IEEE Transactions on Mobile Computing, 20(3), 939–951. https://doi.org/10.1109/TMC.2019.2957804

[13] Chen, L., Xu, Y., Lu, Z., Wu, J., Gai, K., Hung, P. C. K., & Qiu, M. (2021). IoT Microservice Deployment in Edge-Cloud Hybrid Environment Using Reinforcement Learning. IEEE Internet of Things Journal, 8(16), 12610–12622. https://doi.org/10.1109/JIOT.2020.3014970

[14] Bertsekas, D. P. (2010). Rollout Algorithms for Discrete Optimization: A Survey.

[15] Bertsekas, D. P., Tsitsiklis, J. N., Wu, C., Bertsekas, D. P., Tsitsiklis, J. N., & Wu, C. (1997). Rollout Algorithm For Combinatorial Optimization ROLLOUT ALGORITHMS FOR COMBINATORIAL OPTIMIZATION

[16] Sallam, G., & Ji, B. (2019). Joint Placement and Allocation of VNF Nodes with Budget and Capacity Constraints. http://arxiv.org/abs/1901.03931

[17] J. Singh, J. Powles, T. Pasquier, and J. Bacon, "Data flow management and compliance in cloud computing," IEEE Cloud Computing, vol. 2, no. 4, pp. 24–32, Jul. 2015.

[18] Kata Containers, "The speed of containers, the security of VMs," Available online: https://katacontainers.io/

[19] A. Madhavapeddy et al., "Unikernels," ACM SIGARCH Computer Architecture News, vol. 41, no. 1, pp. 461–472, 2013.

[20] Xiong and H. Chen, "Challenges for Building a Cloud Native Scalable and Trustable Multi-tenant AIoT Platform," in IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD, Nov. 2020.

[21] S. Meng et al., "Security-Aware Dynamic Scheduling for Real-Time Optimization in Cloud-Based Industrial Applications," IEEE Trans Industr Inform, vol. 17, no. 6, pp. 4219–4228, Jun. 2021.

[22] Y. Wang, W. Zhang, H. Deng, and X. Li, “Efficient Resource Allocation for Security-Aware Task Offloading in MEC System Using DVS,” Electronics (Switzerland), vol. 11, no. 19, Oct. 2022.

[23] Z. Li, V. Chang, H. Hu, D. Yu, J. Ge, and B. Huang, “Profit maximization for security-aware task offloading in edge-cloud environment,” J Parallel Distrib Comput, vol. 157, pp. 43–55, Nov. 2021.

[24] M. Sabt, M. Achemlal and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," 2015 IEEE Trustcom/BigDataSE/ISPA, Helsinki, Finland, 2015, pp. 57-64

[25] S. Kuenzer et al., “Unikraft: Fast, Specialized Unikernels the Easy Way,” arXiv.org, Apr. 21, 2021.

[26] A. Alexandru, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D. M. Popa. "Firecracker: Lightweight Virtualization for Serverless Applications." In NSDI, vol. 20, pp. 419-434. 2020.

[27] Kretsis, A., et al.: SERRANO: transparent application deployment in a secure, accelerated and cognitive cloud continuum. In: 2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom). pp. 55–60. IEEE, Athens, Greece (2021).

[28] Kokkinos, P., Margaris, D., Spiliotopoulos, D.: A Quality of Experience Illustrator User Interface for Cloud Provider Recommendations. In: HCI International 2022 Posters. HCII 2022. Communications in Computer and Information Science, vol 1580. Springer, Cham. (2022).

[29] Clemm, A., Ciavaglia, L., Granville, L. Z., Tantsura, J. (2020). Intent-based networking-concepts and definitions. IRTF draft work-in-progress.: "Intent-based networking-concepts and definitions." IRTF draft work-in-progress (2020).

[30] Hong, C. H., Varghese, B.: Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms, ACM Computing Surveys (CSUR), 52(5), 1-37, (2019).

[31] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction 2nd edn. The MIT Press Cambridge, Massachusetts, USA (2018).

[32] Aihara,N., Adachi, K., Takyu, O., Ohta, M., Fujii, T.: Q-Learning Aided Resource Allocation and Environment Recognition in LoRaWAN With CSMA/CA, in IEEE Access, vol. 7, pp. 152126-152137, (2019). doi: 10.1109/ACCESS.2019.2948111

[33] Rezwan, S., Choi, W.: Priority-Based Joint Resource Allocation With Deep Q-Learning for Heterogeneous NOMA Systems, in IEEE Access, vol. 9, pp. 41468-41481, (2021). doi: 10.1109/ACCESS.2021.3065314

[34] Dab, B., Aitsaadi, N.,Langar, R.: Q-Learning Algorithm for Joint Computation Offloading and Resource Allocation in Edge Cloud, IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Arlington, VA, USA, pp. 45-52, (2019).

[35] Ning, Z., Wang, X., Rodrigues, J. J. P. C., Xia, F.: Joint computation offloading power allocation and channel assignment for 5G-enabled traffic management systems, IEEE Trans. Ind. Informat., vol. 15, no. 5, pp. 3058-3067, (May 2019).

[36] J. Kong, J., Wu, Z. -Y., Ismail, M., Serpedin, E., Qaraqe, K. A.: Q-Learning Based Two-Timescale Power Allocation for Multi-Homing Hybrid RF/VLC Networks, in IEEE Wireless Communications Letters, vol. 9, no. 4, pp. 443-447, (April 2020), doi: 10.1109/LWC.2019.2958121

[37] Qiu,C., Yao, H., Yu, F. R., Xu, F., Zhao, C.: Deep Q-Learning Aided Networking, Caching, and Computing Resources Allocation in Software-Defined Satellite-Terrestrial Networks in IEEE Transactions on Vehicular Technology, vol. 68, no. 6, pp. 5871-5883, (June 2019), doi: 10.1109/TVT.2019.2907682

[38] Valkanis, A., Beletsioti, G. A., Nicopolitidis, P., Papadimitriou, G., Varvarigos, E.: Reinforcement learning in traffic prediction of core optical networks using learning automata, IEEE International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI) (pp. 1-6), (2020).

[39] AlQerm, I., Pan, J.: Enhanced Online Q-Learning Scheme for Resource Allocation with Maximum Utility and Fairness in Edge-IoT Networks, in IEEE Transactions on Network Science and Engineering, vol. 7, no. 4, pp. 3074-3086, (1 Oct.-Dec. 2020), \doi: 10.1109/TNSE.2020.3015689

[40] Eshratifar, A. E., Pedram, M.: Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment. In: Proceedings on Great Lakes Symp. VLSI (GLSVLSI), pp. 111-116. Chicago, IL, USA (2018), https://doi.org/10.1145/3194554.3194565

[41] Zheng, T., Wan, J., Zhang, J., Jiang, C.: Deep reinforcement learning-based workload scheduling for edge computing. Journal of Cloud Computing, 11(1), 3. (2022).

[42] Zeng, D.,Gu, L.,Pan, S.,Cai., J, Guo, S,: Resource Management at the Network Edge: A Deep Reinforcement Learning Approach, in IEEE Network, vol. 33, no. 3, pp. 26-33, May/June (2019), doi: 10.1109/MNET.2019.1800386.

[43] Pang, L., Yang, C., Chen, D., Song, Y., Guizani, M.:A survey on intent-driven networks, IEEE Access, 8, 22862-22873, (2020).

[44] Abbas, K., Afaq, M., Ahmed Khan, T., Rafiq, A., Song, W. C.: Slicing the core network and radio access network domains through intent-based networking for 5g networks. Electronics, 9(10), 1710. (2020).

[45] Mehmood, K., Kralevska, K., Palma, D.: Intent-driven Autonomous Network and Service Management in Future Networks: A Structured Literature Review", (2021).

[46] Chao, W., Horiuchi, S. Intent-based cloud service management. In 2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN) (pp. 1-5). IEEE. (February 2018).

[47] Kang, J. M., Lee, J., Nagendra, V., Banerjee, S. LMS: Label management service for intent-driven cloud management. In 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM) (pp. 177-185). IEEE (May 2017).

[48] Liao, H., Zhou, Z., Kong, W., Chen, Y., Wang, X., Wang, Z.,Al Otaibi, S.:Learning-based intent-aware task offloading for air-ground integrated vehicular edge computing. IEEE Transactions on Intelligent Transportation Systems, 22(8), 5127-5139, (2020).

[49] Wu, C., Horiuchi, S., Murase, K., Kikushima, H. and Tayama, K. Intent-driven cloud resource design framework to meet cloud performance requirements and its application to a cloud-sensor system. Journal of Cloud Computing, 10(1), 1-22, (2021).

[50] He, L., Qian, Z.: Intent-based resource matching strategy in cloud. Information Sciences, 538, 1-18,(2020).

[51] Leivadeas, A.,Falkner, M.: VNF placement problem: a multi-tenant intent-based networking approach. In 2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN) (pp. 143-150). IEEE. (March 2021).

[52] Amazon Instance Types, https://aws.amazon.com/ec2/instance-types/

[53] M. Barika, S. Garg, A. Y. Zomaya, L. Wang, A. V. Moorsel, and R. Ranjan, “Orchestrating big data analysis workflows in the cloud: research challenges, survey, and future directions,” ACM Computing Surveys (CSUR), vol. 52, no. 5, pp. 1–41, 2019.

[54] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” Advances in neural information processing systems, vol. 30, 2017.

[55] M. Eisen and A. Ribeiro, “Optimal wireless resource allocation with random edge graph neural networks,” ieee transactions on signal processing, vol. 68, pp. 2977–2991, 2020.

[56] W. Li, H. Wang, X. Zhang, D. Li, L. Yan, Q. Fan, Y. Jiang, and R. Yao, “Security service function chain based on graph neural network,” Information, vol. 13, no. 2, p. 78, 2022.

[57] X. Deng, J. Sun, and J. Lu, “Graph neural network-based efficient subgraph embedding method for link prediction in mobile edge computing,” Sensors, vol. 23, no. 10, 2023.

[58] K. Yang, H. Ma, and S. Dou, “Fog intelligence for network anomaly detection,” IEEE Network, vol. 34, no. 2, pp. 78–82, 2020.

[59] O. Ibidunmoye, F. Hernandez-Rodriguez, and E. Elmroth, “Performance anomaly detection and bottleneck identification,” ACM Comput. Surv., vol. 48, no. 1, jul 2015.

[60] C. Sauvanaud, M. Kaaniche, K. Kanoun, K. Lazri, and G. Da Silva Silvestre, “Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned,” Journal of Systems and Software, vol. 139, pp. 84–106, 2018.

[61] [A. Hogan, E. Blomqvist, M. Cochez, C. d’Amato, G. d. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier et al., “Knowledge graphs,” Synthesis Lectures on Data, Semantics, and Knowledge, vol. 12, no. 2, pp. 1–257, 2021.

[62] M. Barshan, H. Moens, S. Latre, B. Volckaert, and F. De Turck, “Algorithms for network-aware application component placement for cloud resource allocation,” Journal of Communications and Networks, vol. 19, no. 5, pp. 493–508, 2017.
[63] W. Tarneberg, A. Mehta, E. Wadbro, J. Tordsson, J. Eker, M. Kihl, and E. Elmroth, “Dynamic application placement in the mobile cloud network,” Future Generation Computer Systems, vol. 70, pp. 163–177, 2017.
[64] G. Sun, D. Liao, V. Anand, D. Zhao, and H. Yu, “A new technique for efficient live migration of multiple virtual machines,” Future Generation Computer Systems, vol. 55, pp. 74–86, 2016.
[65] T. Miyazawa, V. P. Kafle, and H. Harai, “Reinforcement learning based dynamic resource migration for virtual networks,” in 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2017, pp. 428–434.
[66] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba, “Topology-aware prediction of virtual network function resource requirements,” IEEE Transactions on Network and Service Management, vol. 14, no. 1, pp. 106–120, 2017.
[67] I. Robinson, J. Webber, and E. Eifrem, Graph databases: new opportunities for connected data. ” O’Reilly Media, Inc.”, 2015.
[68] “Cypher query language - developer guides.” [Online]. Available: https://neo4j.com/developer/cypher/
[69] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” Advances in neural information processing systems, vol. 30, 2017.
[70] M. Fey and J. E. Lenssen, “Fast graph representation learning with pytorch geometric,” ArXiv, vol. abs/1903.02428, 2019.
[71] M. Zhang and Y. Chen, “Link prediction based on graph neural networks,” Advances in neural information processing systems, vol. 31, 2018.
[72] K. Yang, Y. Liu, Z. Zhao, X. Zhou, and P. Ding, “Graph attention network via node similarity for link prediction,” The European Physical Journal B, vol. 96, no. 3, p. 27, Mar 2023.
[73] “Networkx documentation.” [Online]. Available: https://networkx.org/
[74] “Neo4j python driver documentation.” [Online]. Available: https://neo4j.com/docs/api/python-driver/current/
[75] “Neo4j documentation.” [Online]. Available: https://neo4j.com/
[76] S. Narayan, “The generalized sigmoid activation function: Competitive supervised learning,” Inf. Sci., vol. 99, no. 1–2, p. 69–82, jun 1997.
[77] Dask Python parallel computing: https://www.dask.org
[78] Prometheus - Monitoring system & time series database: https://prometheus.io
[79] Elasticsearch Platform: https://www.elastic.co
[80] pandas - Python Data Analysis Library: https://pandas.pydata.org/
[81] scikit-learn - machine learning in Python: https://scikit-learn.org/stable/
[82] Joblib: joblib.readthedocs.io
[83] Open Neural Network Exchange: https://onnx.ai/
[84] Apache Kafka: kafka.apache.org/
[85] L. Shapley, A Value for n-Person Games, Princeton University Press, 2016
[86] N. Takeishi, Y. Kawahara, On Anomaly Interpretation via Shapley Values, 2020
[87] F. T. Liu, Isolation fores. Eighth IEEE International Conference on Data Mining, (pp. 413-422), 2008
[88] F. T. Liu, Isolation-based anomaly detection. ACM Transactions on Knowledge Discovery from Data, 2012
[89] Z. X. He, Discovering cluster-based local outliers. Pattern Recognition Letters, 24(9), 1641-1650, 2003
[90] R. L. Yeo, Unsupervised anomaly detection using variational auto- encoder based feature extraction. IEEE International Conference on Prognostics and Health Management (ICPHM), 2019
[91] Cerdà-Alabern Llorenç, Gabriel Iuhasz, Gabriele Gemmi, Anomaly detection for fault detection in wireless community networks using machine learning. Computer Communications, 202, 191-203, 2023
[92] Flask 2.0: https://flask.palletsprojects.com/en/2.0.x/

[93] Pika: https://pika.readthedocs.io/en/stable/

[94] PyQt: https://riverbankcomputing.com/software/pyqt/intro

[95] SERRANO GitHub repository: https://github.com/ict-serrano

[96] SERRANO Harbor container registry: https://serrano-harbor.rid-intrasoft.eu

[97] Kubernetes kube-state-metrics: https://github.com/kubernetes/kube-state-metrics

[98] Kubernetes metrics-server: https://github.com/kubernetes-sigs/metrics-server

[99] Prometheus node exporter: https://github.com/prometheus/node_exporter

[100] Grafana: The open observability platform: https://grafana.com

[101] InfluxDB: Open Source Time Series Database: https://www.influxdata.com/developers/

[102] MongoDB: https://docs.mongodb.com

[103] Minio - High Performance Object Storage: https://min.io

[104] etcd: https://etcd.io

[105] Slurm workload manager: https://slurm.schedmd.com/

[106] OpenPBS: https://www.openpbs.org/

[107] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, Today, and Tomorrow, pages 195–216. Springer International Publishing, Cham, 2017

[108] J. Thönes. Microservices. IEEE Software, 32(1):116–116, Jan 2015.

[109] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust. Mobile-edge computing architecture: The role of mec in the in- ternet of things. IEEE Consumer Electronics Magazine, 5:84–91, 10 2016

[110] Cloud Native Computing Foundation, “Frequently Asked Questions”, https://www.cncf.io/about/faq

[111] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou. A measurement study on linux container security: Attacks and countermeasures. In Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC ’18, pages 418–429, New York, NY, USA, 2018. ACM

[112] AWS Lambda. https://aws.amazon.com/lambda. Accessed: 2022-02-01

[113] Firecracker: Lightweight Virtualization for Serverless Computing. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing

[114] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019

[115] http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html

[116] D. Williams, R. Koller, M. Lucina, and N. Prakash. Unikernels as processes. In Proceedings of the ACM Symposium on Cloud Computing, SoCC ’18, pages 199–211, New York, NY, USA, 2018. ACM

[117] Modern Hypervisor for the Cloud. https://github.com/intel/nemu

[118] crosvm VMM: https://google.github.io/crosvm/

[119] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, J. Crowcroft, Unikernels: Library Operating Systems for the Cloud, ASPLOS, 2013

[120] The Solo5 Unikernel: https://github.com/solo5/solo5

[121] F. Bellar, QEMU a Fast and Portable Dynamic Translator, Conference: Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, Anaheim, CA, USA, 2005

[122] A. Agache, M. Brooker, A. Florescu; A Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D. Popa, Firecracker: Lightweight Virtualization for Serverless Applications, 17th USENIX Symposium on Networked Systems Design and Implementation, 2020

[123] S. Kuenzer, V. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, F. Huici, Unikraft: fast, specialized unikernels the easy way. Sixteenth European Conference on Computer Systems EuroSys '21. New York, NY, USA: Association for Computing Machinery. doi:https://doi.org/10.1145/3447786.3456248

[124] A. Kantee, A., J. Cormack, Rump Kernels: No OS? No Problem! login Usenix Mag, 2014

[125] Kata Containers, an open source container runtime: https://katacontainers.io