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

Grant Agreement no. 101017168

Deliverable D5.3
Resource Orchestration, Telemetry and Lightweight Virtualization 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:	ICCS
Due date:	31/03/2022
Actual submission date:	06/04/2022

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
16.11.21	Panagiotis Kokkinos	Draft	0.1	Initial ToC
28.02.22	Aristotelis Kretsis	Draft	0.2	Add contribution in Section 3
09.03.22	Anastasios Nanos	Draft	0.3	Add contribution in Section 5
11.03.22	Panagiotis Kokkinos	Draft	0.4	Add contribution in Section 4
14.03.22	Anastasios Nanos	Draft	0.5	Expand section 4.3
14.03.22	Panagiotis Kokkinos	Draft	0.6	Integrate INTRA contribution by INTRA in Section 6
15.03.22	Aristotelis Kretsis	Draft	0.7	Corrections, Section 1, Section 7 Ready for internal review
21.03.22	Aristotelis Kretsis	Draft	0.8	Integrate updates by HLRS in Section 4
29.03.22	Aristotelis Kretsis	Draft	0.9	Integrate post review contributions
06.04.22	ICCS	Final	1.0	Final version for submission

Author List

Organization	Author
ICCS	Aristotelis Kretsis, Panagiotis Kokkinos, Polyzois Soumplis, Athina Kyriakou, Emmanouel Varvarigos
NBFC	Anastassios Nanos, Charalampos Mainas, George Ntoutsos
INTRA	Paraskevas Bourgos, Makis Karadimas
MLNX	Yoray Zack, Juan Jose Vegas Olmos, Sandra Starck
INNOV	Stelios Pantelopoulos, Filia Filippou, Andreas Litke

Internal Reviewers

INTRA, INB

Abstract: The deliverable D5.3 presents results of the activities that took place in the context of Task 5.3 “AI/ML-assisted Network and Cloud Telemetry” and Task 5.5 “Resource Orchestration and Lightweight Virtualization Mechanisms” during the first iteration of the incremental implementation plan (M07-M15). These tasks aim to design and develop: (i) novel network and cloud telemetry framework, (ii) hierarchical resource orchestration, and (iii) lightweight virtualization mechanisms.

Keywords: Cloud and Network Telemetry, Resource Orchestration, Lightweight Virtualization, Containers, Unikernels, vAccel, Message Broker, Stream Handler.

Table of Contents

1 Executive Summary 9

2 Introduction 10
2.1 Purpose of this document 10
2.2 Document structure 11
2.3 Audience 11

3 Network and Cloud Telemetry Framework 12
3.1 SERRANO telemetry architecture 12
3.1.1 Central Telemetry Handler and Enhanced Telemetry Agent 13
3.1.2 Monitoring Probes 15
3.2 Resource description and monitoring parameters 17
3.3 Telemetry interfaces 18
3.4 AI/ML-assisted Network and Cloud Telemetry 21
3.4.1 ML Network Tomography 21
3.5 Anomaly Detection in Networks 31
3.5.1 Introduction to Anomaly Detection 31

4 Resource Orchestration Mechanisms 36
4.1 SERRANO Resource Orchestrator 37
4.2 Orchestration Drivers 39
4.3 Integration with Kubernetes 40
4.3.1 Kubernetes overview 40
4.3.2 Container runtimes 41
4.3.3 Workload variants (containers, unikernels, VMs) 42
4.3.4 Hardware acceleration 43
4.4 Integration with HPC platforms 44
4.4.1 HPC infrastructure overview 44
4.4.2 SERRANO HPC Gateway Interface 44

5 Lightweight Virtualization Mechanisms 46
5.1 Overview 46
5.2 Unikernels 48
5.3 QEMU / KVM 49
5.4 AWS Firecracker 49
5.5 KVMM 49
5.5.1 Virtual Machine Monitor 50
5.5.2 Initial performance results 52

6 SERRANO Data Broker 54
6.1 Message Broker 54
6.2 Stream Handler 57

7 Conclusions 61

8 References 62

List of Figures

Figure 1: SERRANO high-level architecture 10
Figure 2: SERRANO hierarchical telemetry architecture 13
Figure 3: Central Telemetry Handler and Enhanced Telemetry Agent architecture 14
Figure 4: General architecture of SERRANO monitoring probes 16
Figure 5: Monitoring probe for SERRANO edge storage devices 17
Figure 6: Monitoring data collected by SERRANO monitoring probe from a SERRANO edge storage device 17
Figure 7: Summary of collected resource description and monitoring parameters in the SERRANO platform 18
Figure 8: Telemetry access interface – Swagger documentation of REST APIs 20
Figure 9: A network with 2 established monitored paths and one candidate (or unmonitored), sharing one known link and one origin node. Parts of the topology may be unknown .. 25
Figure 10: The DT topology with the link lengths in Km 27
Figure 11: Accuracy (for the delay metric) of three NN architectures for different number of paths and a) k=1, b) k=2, c) k=3 28
Figure 12: Accuracy (for the delay metric) of the algorithms for different number of paths and a) k=1, b) k=2, c) k=3 29
Figure 13: Accuracy (for the delay metric) of the algorithms for different number of unknown links and a) k=1, b) k=2, c) k=3 29
Figure 14: Accuracy (for the bandwidth metric) of the algorithms for different number of paths and a) k=1, b) k=2, c) k=3 30
Figure 15: Accuracy (for the bandwidth metric) of the algorithms for different number of paths and a) k=1, b) k=2, c) k=3 31
Figure 16: SERRANO distributed and cognitive resource orchestration 36
Figure 17: SERRANO Resource Orchestrator 38
Figure 18: Resource Orchestrator access interface – Swagger documentation of REST APIs . 38
Figure 19: Orchestration Drivers 39
Figure 20: Kubernetes architecture 40
Figure 21: Kubernetes CRI Plugin Enabling containerd 41
Figure 22: Kata Containers and containerd integration with shimv2 42

Figure 23: vAccel framework 43
Figure 24: Interaction between HPC Gateway and HPC infrastructure 45
Figure 25: A Virtual Machine running on a generic user-space VMM on top of KVM 47
Figure 26: A simple unikernel stack (rumprun over solo5) 49
Figure 27: A unikernel running as a VM on KVMM 50
Figure 28: Execution time breakdown of the basic VM setup and spawn operations for a short-lived guest over KVMM and solo5-hvt 52
Figure 29: RTT latency measured with ping between an external Linux host and a generic Linux VM running on QEMU/KVM (with and without VHOST), a microVMM (firecracker), a unikernel running on QEMU/KVM, on Nabla (solo5-spt), Solo5-HVT, and on KVMM. ... 53
Figure 30: RabbitMQ basic elements [54] 54
Figure 31: Overview of RabbitMQ exchange types and queues 55
Figure 32: ROT asynchronous communication over SERRANO Message Broker 56
Figure 33: Stream Handler building blocks and interactions 57
Figure 34: REST Endpoints exposed by Streaming Core Platform 58
Figure 35: Schemas used in Streaming Core Platform REST API 59
Figure 36: Schema Registry REST API 60
Figure 37: Schemas used in Schema Registry API 60

List of Tables

Table 1: Telemetry REST API 19
Table 2: Telemetry notification messages 20
Table 3: Important Notations 26
Table 4: UNSW-NB15 dataset statistics 34
Table 5: Supervised Learning results 35
Table 6: ROT asynchronous communication over SERRANO Message Broker 56

Abbreviations

AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CRI	Container Runtime Interface
D	Deliverable
DoW	Description of Work
EC	European Commission
FaaS	Function as a Service
HW	Hardware
ICT	Information and Communication Technology
IaaS	Infrastructure-as-a-Service
JSON	JavaScript Object Notation
KVM	Kernel-based Virtual Machine
KVVM	in-kernel Virtual Machine Monitor,
MAE	Mean Absolute Error
ML	Machine Learning
MQTT	MQ Telemetry Transport
NFV	Network Function Virtualization
NN	Neural Network
NT	Network tomography
PM	Project Manager
PMDS	Persistent Monitoring Data Storage
PO	Project Officer
QoS	Quality of Service
REST	Representational State Transfer
ROT	Resource Optimization Toolkit
SDN	Software-Defined Networking
SW	Software
STOMP	Simple (or Streaming) Text Orientated Messaging Protocol
VM	Virtual Machine
VMM	Virtual Machine Manager
gRPC	Google Remote Procedure Call

1 Executive Summary

The deliverable is divided in seven main sections. Section 1 is this executive summary. Section 2 serves as an introduction to the document. It provides information about the document's purpose, structure, and audience to which it is mainly addressed.

Section 3 presents the design and the implementation of the initial release of the SERRANO Network and Cloud Telemetry framework. AI/ML-based network and cloud telemetry is also discussed and a novel network tomography mechanism is presented. An introduction to anomaly detection mechanisms is provided, while a number of unsupervised and unsupervised learning algorithms are evaluated.

Section 4 describes the architecture and the initial implementation activities for the SERRANO Resource Orchestrator, including the high-level central orchestrator and the local orchestrators and the orchestration drivers that interact with the edge, cloud and HPC resources. This section also elaborates on the integration with the Kubernetes container orchestration system and on the extensions performed so as to support hardware acceleration platforms through vAccel. In addition, Section 4 describes the integration of the SERRANO platform with a HPC system using the HPC Gateway component.

Section 5 presents the lightweight virtualization mechanisms used in the system’s software stack of the SERRANO platform: Unikernels, QEMU, Firecracker and KVMM. For the latter, a number of micro-benchmarks are performed to analyse its behaviour.

Section 6 describes the architecture and implementation of the Data Broker component, responsible to provide message brokering and distributed steaming capabilities to the SERRANO, through the appropriate interfaces.

Section 7 concludes the deliverable. It discusses the main developments and future work to be undertaken in the SERRANO project.

2 Introduction

2.1 Purpose of this document

The present deliverable (D5.3) presents the outcomes of Task 5.3 “AI/ML-assisted Network and Cloud Telemetry” and Task 5.5 “Resource Orchestration and Lightweight Virtualization Mechanisms”, during the first iteration of the incremental implementation plan (M07-M15). T5.3 is associated with the design of the autonomous and data-driven network and cloud telemetry framework within the SERRANO platform. This framework will be able to autonomously collect telemetry data from multiple resources and the deployed applications. T5.5 is focused on the development of the SERRANO hierarchical resource orchestration mechanisms that have to interface with well-established orchestration solutions at edge, cloud and HPC platforms. Moreover, the task develops the necessary software components to enable the transparent execution of workloads in various lightweight virtualization solutions, hypervisors and unikernels frameworks.

The objective of D5.3 is to build on the initial architecture of the SERRANO platform (Figure 1), as reported in the deliverable D2.3 in M9, towards the provision of the initial release of the SERRANO telemetry mechanisms (i.e., Central Telemetry Handler, Enhanced Telemetry Agent, Monitoring Probes), resource orchestration mechanisms (i.e., Resource Orchestrator, Orchestration Drivers), advanced virtualization mechanisms (i.e., Lightweight Virtualization, FaaS with HW acceleration) and data broker mechanisms (i.e., Data Broker).

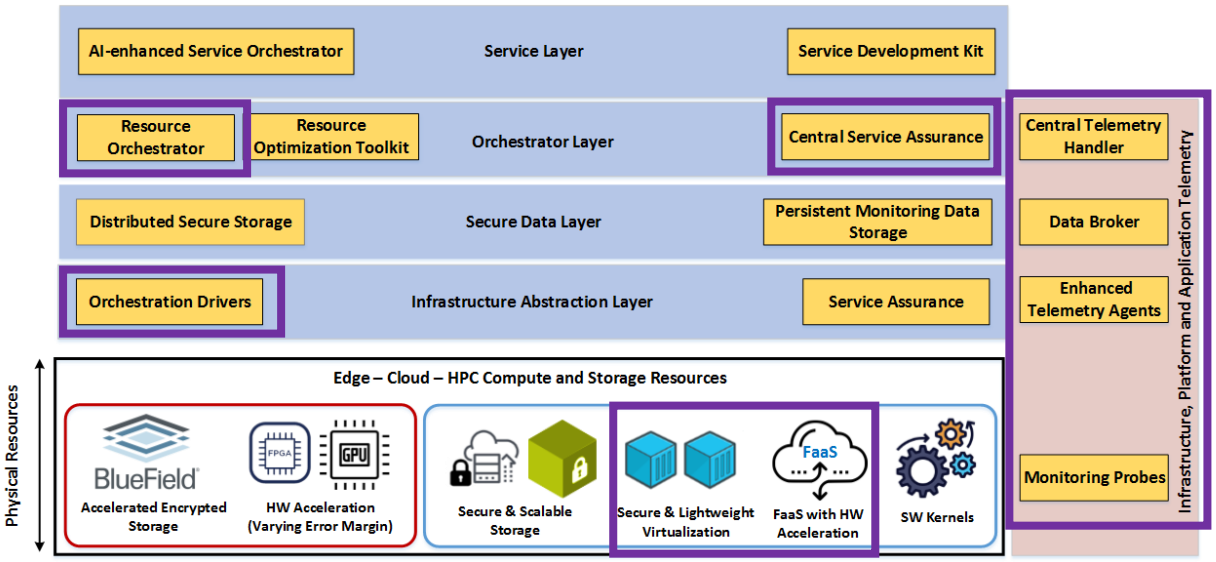


Figure 1: SERRANO high-level architecture

D5.4 “Intelligent Service and Resource Orchestration” in M31 will include the final release of the SERRANO components and mechanism that are developed in the context of T5.3 and T5.5.

2.2 Document structure

The present deliverable is split into seven chapters:

- Executive Summary
- Introduction
- Network and Cloud Telemetry Framework
- Resource Orchestration Mechanisms
- Lightweight Virtualization Mechanisms
- SERRANO Data Broker
- Conclusions

2.3 Audience

This document is publicly available and should be of use to anyone interested in the initial description of the SERRANO mechanisms and algorithms related to cloud and network telemetry, resource orchestration over edge/cloud/HPC resources and lightweight virtualization mechanisms.

3 Network and Cloud Telemetry Framework

A heterogeneous and distributed infrastructure, like any system, has to be observable before it can become subject to optimization, and this is the main capability that provides the SERRANO telemetry framework. To this end, the SERRANO platform includes a set of autonomous and scalable mechanisms that provide the sense (detect what is happening) and discern (interpret senses) operations in the envisioned closed-loop control.

SERRANO's increased observability relies on information provided by the resource monitoring and telemetry mechanisms that are deployed across the infrastructure. This collected information is transferred to decision modules (e.g., Analytics Engines, Service Assurance, Resource Orchestrator) over SERRANO's hierarchical monitoring infrastructure, with the aim to improve orchestration decisions, detect problems and trigger proactive or reactive adjustments to SERRANO-enhanced resources and deployed applications. Moreover, the designed telemetry framework includes the appropriate AI/ML methods to enable the dynamic adaptation of the telemetry infrastructure and the correlation of collected information to infer appropriate metrics, identify general performance problems, predict the future infrastructure state and localise failures.

Next, we provide more details regarding the collected information, design, and implementation of the initial release of the SERRANO network and cloud telemetry framework.

3.1 SERRANO telemetry architecture

SERRANO aims to develop a holistic and data-driven telemetry framework for heterogenous infrastructures (cloud, edge and HPC) based on a hierarchical telemetry infrastructure coupled with AI/ML algorithms to provide platform and application performance monitoring and observability. The SERRANO telemetry stack consists of three key building blocks: (a) the Central Telemetry Handler, (b) Enhanced Telemetry Agents and (c) Monitoring Probes.

Figure 2 shows the architecture of the telemetry framework, its main components, and the interactions with other components within the SERRANO architecture. The heterogeneous and federated edge/cloud/HPC infrastructure, that is under the control of the SERRANO platform, is considered to be equipped with the appropriate Monitoring Probes. These resource-specific entities collect the telemetry data and are controlled by the management components of the telemetry framework.

Each management entity provides the same core functions: autonomous management of Monitoring Probes, collection of resource monitoring data, initial correlation of gathered information, and activation of additional actions for resolving performance issues. The various Enhanced Telemetry Agents are responsible for a specific set of Monitoring Probes in the SERRANO architecture. The collection and exchange of monitored information is performed periodically while the granularity and streaming telemetry operations can be activated based on detected events or explicitly by entities at upper layers. Each layer correlates and filters

the received information efficiently sending less amount of monitoring information to an upper layer toward the root element of this hierarchical telemetry infrastructure, the Central Telemetry Handler. Hence, 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. Moreover, the root component s called Central Telemetry Handler.

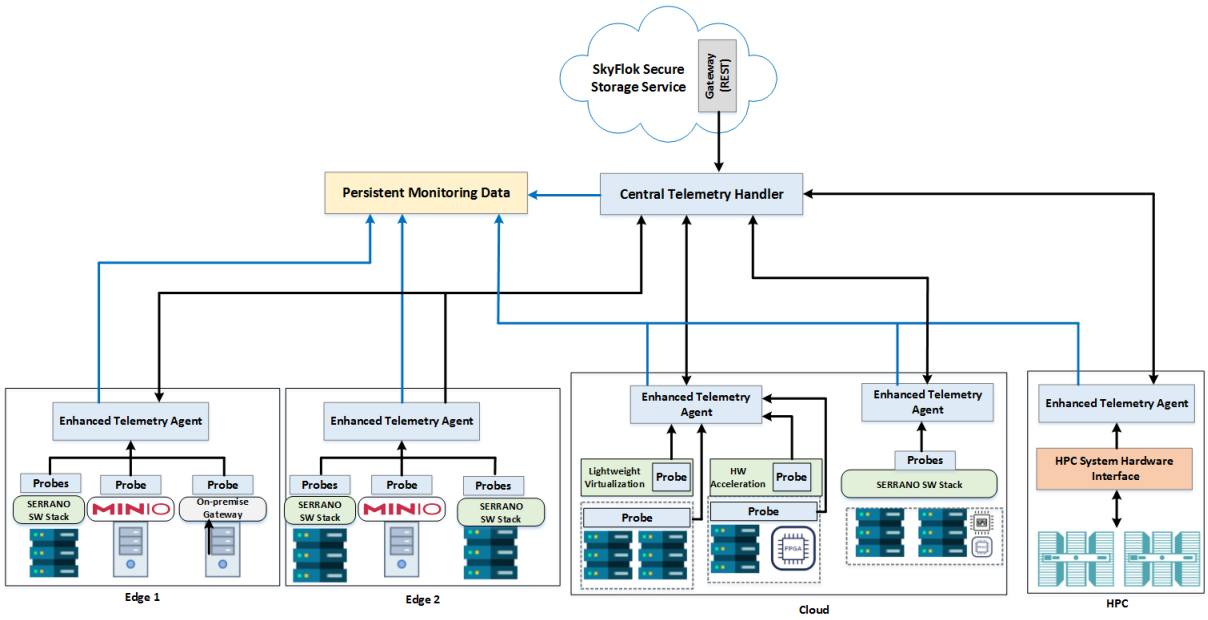


Figure 2: SERRANO hierarchical telemetry architecture

3.1.1 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 a common design and a joint implementation for their components. To this end, their actual role in SERRANO hierarchical infrastructure depends on their configuration, which determines the specific entities under their control and management. The Central Telemetry Handler manages multiple instances of Enhance Telemetry Agents, while each agent controls a specific set of Monitoring Probes. The Enhancement Telemetry Agent and Central Telemetry Handler architecture is shown in Figure 3. They are implemented in Python using frameworks like Flask 2.0 [24], Pika [25] and PyQt [26].

The ***Access Interface*** provides loose coupling between the entities of the telemetry framework and enables external services to retrieve information and monitoring data. It is basically a REST controller that exposes APIs for exchanging commands and information. More details for the provided methods are available in section 3.3.

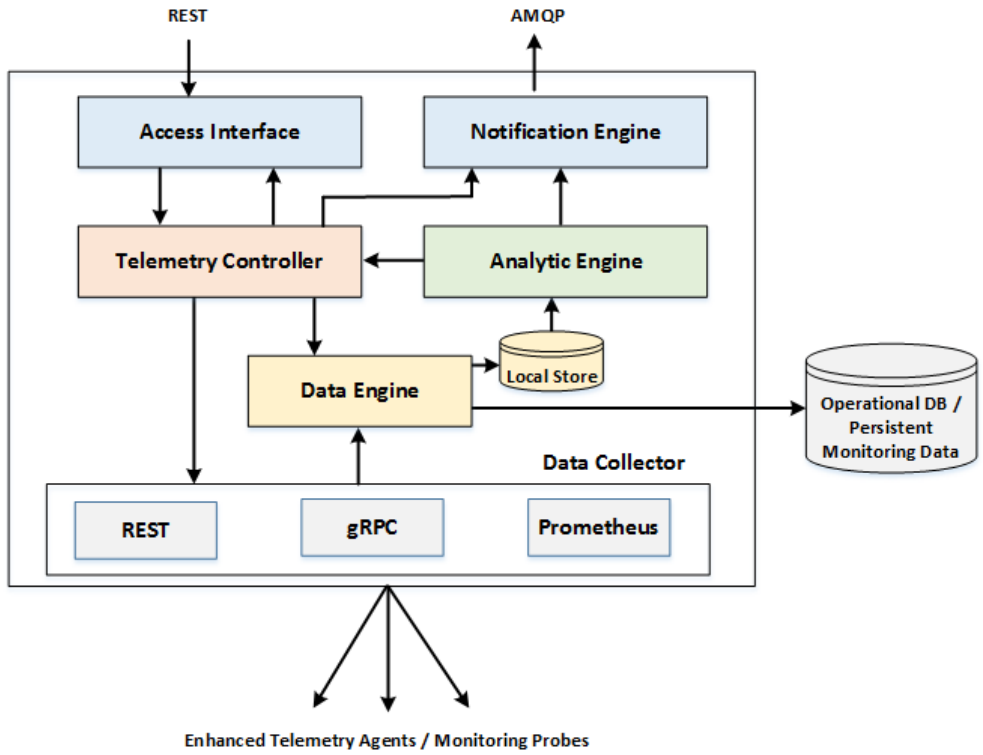


Figure 3: Central Telemetry Handler and Enhanced Telemetry Agent architecture

The ***Notification Engine*** handles the event notifications generated by the Analytic Engine and the Telemetry Controller. This component enables other services within the SERRANO platform, primarily the orchestration and service assurance mechanisms, and external services to register for various performance-related events. The Notification Engine dispatches these events through the Data Broker service in a scalable and efficient way to the subscribed services. Then, these services analyze each event to trigger the required actions to either proactively or reactively readjust the current configuration of the resources and deployed applications. More details for the supported events are available in section 3.3.

The ***Telemetry Controller*** is the core component within the telemetry entity and provides the logic for managing all the other telemetry entities (i.e., Enhanced Telemetry Agents or Monitoring Probes) under the specific instance's control. It coordinates their operation based on the requests of the upper layers and the feedback by the Analytic Engine. The Telemetry Controller also prepares the appropriate commands to the Monitoring Probes, including the provisioning of streaming telemetry data and adaptation of the monitoring rate.

The ***Analytic Engine*** provides the analytic login in the telemetry framework. This component will integrate the AI/ML algorithms (section 3.4) that will rely on collected data to: (i) enable the autonomous operations of the telemetry mechanisms, (ii) estimate additional metrics for different parts of the infrastructure by correlating telemetry measurements and (iii) detect performance degradations and failures. Moreover, it provides the required feedback to Telemetry Controller to readjust the telemetry infrastructure and to Notification Engine to trigger the SERRANO orchestration and service assurance mechanisms.

The ***Data Engine*** undertakes the initial processing of the collected data, which is forwarded by the Data Collector mechanisms. Then, it fuels the Analytic Engine with monitoring and streaming telemetry data, enabling the execution of the data-driven automation mechanisms at the SERRANO platform. The Data Engine is also responsible for properly updating the stored information at the operational databases and the Persistent Monitoring Data Storage component.

The ***Data Collector*** handles the collection of monitoring and streaming telemetry data either directly from the Monitoring Probes or other Enhanced Telemetry Agents. It feeds the Data Engine with all collected information and informs the Telemetry Controller of any internal operational issues. The design supports the ingestion of multiple collection methods (i.e., pull and push) as well as types of monitored data (i.e., metrics, events, streams) from the various telemetry entities based on different protocols. To this end, each collection mechanism is implemented as a custom plug-in. The implementation of the initial prototype supports: (i) REST client to manage the periodic and on-demand collection of monitoring information through the default exchange interface in SERRANO, (ii) gRPC module to handle the streaming telemetry data, and (iii) Prometheus client to collect and handle monitoring data based on the OpenMetrics format that are stored in Prometheus server.

A fundamental piece of supporting an effective monitoring pipeline is the availability of a central repository with up-to-date information for the current state of the heterogeneous resources as well as retention of historical analytical data to feed the various AI/ML-based decision mechanisms. To this end, the SERRANO platform includes the Operational Database that stores all the accessible information related to the characteristics of the available resources, the most up-to-date information for their current state, and details about the applications' deployments. This database is based on the MongoDB [27], an open-source document-oriented database that stores data in flexible format JSON-like documents. In addition, the SERRANO architecture includes the Persistent Monitoring Data Storage (PMDS) component that is based on the InfluxDB [28], 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 [29], an open-source analytics and interactive visualization web application. Furthermore, the PMDS acts as long-term storage for the collected timestamped monitoring and streaming telemetry data that provides historical data to the SERRANO orchestration and service assurance mechanisms.

3.1.2 Monitoring Probes

Monitoring probes collect information about the characteristics and the current status of the infrastructure resources, services and deployed applications. Due to the heterogeneity of the targeted information sources (Section 3.2) in the SERRANO platform, the telemetry framework uses a set of different probes, each one specialized to monitor a specific resource type.

Moreover, the telemetry framework should be able to dynamically adjust the granularity of the monitoring information, support both periodic and on-demand monitoring and even activate streaming telemetry operations to collect additional monitoring data. Hence, we

adopted a common design for the SERRANO monitoring probes (Figure 4) to enable the autonomous operation of the telemetry components and also provide transparent integration of the different monitoring probes with the Data Collector component of the Enhanced Telemetry Agents and Central Telemetry Handler.

The general architecture of SERRANO monitoring probes includes the following components:

- ***Access Interface:*** Provides a common interface for the integration of the various probes with the data collection and management services of the Enhanced Telemetry Agents. It serves the requests for periodic and on-demand retrieval of monitoring data. Moreover, it configures the low-level monitoring probes based on the instructions by the upper layer components.
- ***Streaming Telemetry***: Enables the collection of measurements based on the streaming telemetry approach, where continuous measurements are sent at a rate much shorter than the typical monitoring approach. The operation is triggered by components of the upper layers in the telemetry framework and is implemented through the gRPC [30], an open source remote procedure call (RPC) framework that supports also streaming RPCs.
- ***Monitoring Probes:*** The probes that collect information from the supported resource types. Some probes are out of the shelf (e.g., Netdata [31], kube-state-metrics [32], metrics-server [33]), while others are implemented in the context of SERRANO (e.g., HPC platforms, on-premise storage gateway).

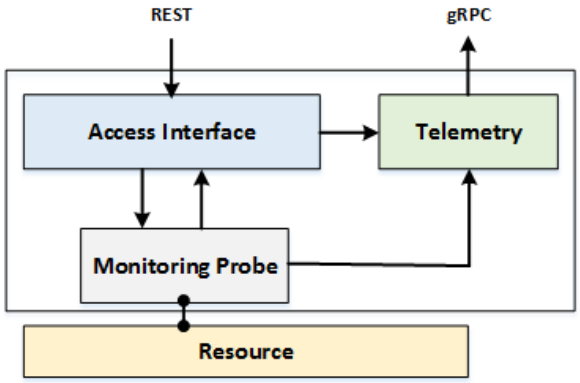


Figure 4: General architecture of SERRANO monitoring probes

Figure 5 presents the probe's implementation details that provide performance monitoring parameters for the SERRANO edge storage devices. More details for these SERRANO-enhanced devices are available in D3.2 “Secure Cloud Storage System” (M15). In this case, we use Netdata to track the health of the nodes that host the SERRANO edge storage devices. Netdata is a popular open-source monitoring agent that runs across physical systems, virtual machines, applications, and IoT devices to collect real-time metrics (e.g., CPU usage, disk activity, bandwidth usage, etc.). Netdata consists of a lightweight daemon that is responsible for collecting information. Moreover, we leverage the ability of the MinIO [34] server to expose monitoring data over Prometheus-compatible endpoints. Our probe uses these endpoints to collect information about the current state of the SERRANO edge storage

devices. The Access Interface abstracts the interaction with those tools and automatically collects the required telemetry data according to the requests of the specific Enhanced Telemetry Agent that manages the respective Monitoring Probe.

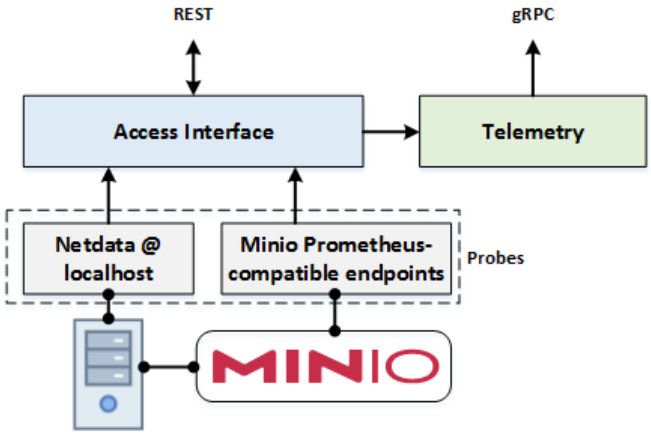


Figure 5: Monitoring probe for SERRANO edge storage devices

Additional implementations according to the respective general architecture for the Monitoring Probes within the SERRANO platform provide the autonomous monitoring of other SERRANO-enhanced resources (e.g., on-premise gateway), general edge/cloud computational resources, and HPC platforms. Regarding the monitoring of the HPC resources, a key component is the HPC System Hardware Interface, developed in the SERRANO context, which provides the actual integration of the HPC management and monitoring systems with the core services of the SERRANO platform.

In Figure 6, we present an example of collected resource description and performance monitoring parameters as reported by the initial prototype of the SERRANO monitoring probe for the SERRANO edge storage devices.

```
INFO:SERRANO.Telemetry.Probe: Edge Storage Monitoring Probe is running ...
DEBUG:SERRANO.Telemetry.Probe: Registration to Enhanced Telemetry Aget at '147.102.22.140:9200'
DEBUG:SERRANO.Telemetry.Probe: Inventory and monitoring information - timestamp '1644685040'
DEBUG:SERRANO.Telemetry.Probe: Inventory information ....
{
  "uid":"1a558d68-8905-11ec-9b0a-eb5f7b677049", "hostname": "edge_storage_dev", "cores_total": "2", "ram_total": "12448092160",
  "total_disk_space": "500107862016","is_k8s_node": "false", "architecture": "x86_64", "os_nane": "Ubuntu",
  "os_version": "20.04.3 LTS (Focal Fossa)"
}
DEBUG:SERRANO.Telemetry.Probe: Monitoring information ....
{
  "timestamp": "1644685040", "cpu_usage": 5.98, "ram_free_MB": 306.642998, "ram_used_MB": 3941.6852, "disk_space_avail_GB": 164.31,
  "disk_space_used_GB": 47.94, "minio_s3_requests_total": 30.0, "minio_s3_requests_errors_total": 9.0, "minio_s3_requests_waiting_total":0,
  "minio_s3_traffic_received_bytes_total": 71.0, "net_received_kbps": 12.745, "net_sent_kbps": 6.412, "minio_node_process_uptime_seconds": 81355,
  "minio_node_disk_total_bytes": 2147483648, "minio_s3_traffic_send_bytes_total": 8094.0, "minio_node_disk_used_bytes": 7200000000,
  "minio_node_disk_free_bytes": 14340000000, "minio_node_file_descriptor_open_total": 15
}
```

Figure 6: Monitoring data collected by SERRANO monitoring probe from a SERRANO edge storage device

3.2 Resource description and monitoring parameters

The SERRANO telemetry framework should automatically maintain a catalogue with the available computational and storage resources within the individual edge, cloud and HPC platforms that constitute the SERRANO platform. Moreover, it has to monitor the current state of the available resources and deployed applications. 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., hardware acceleration for encrypted storage 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.

Figure 7 summarizes the resource description and monitoring parameters that currently collect the Monitoring Probes of the SERRANO telemetry framework.

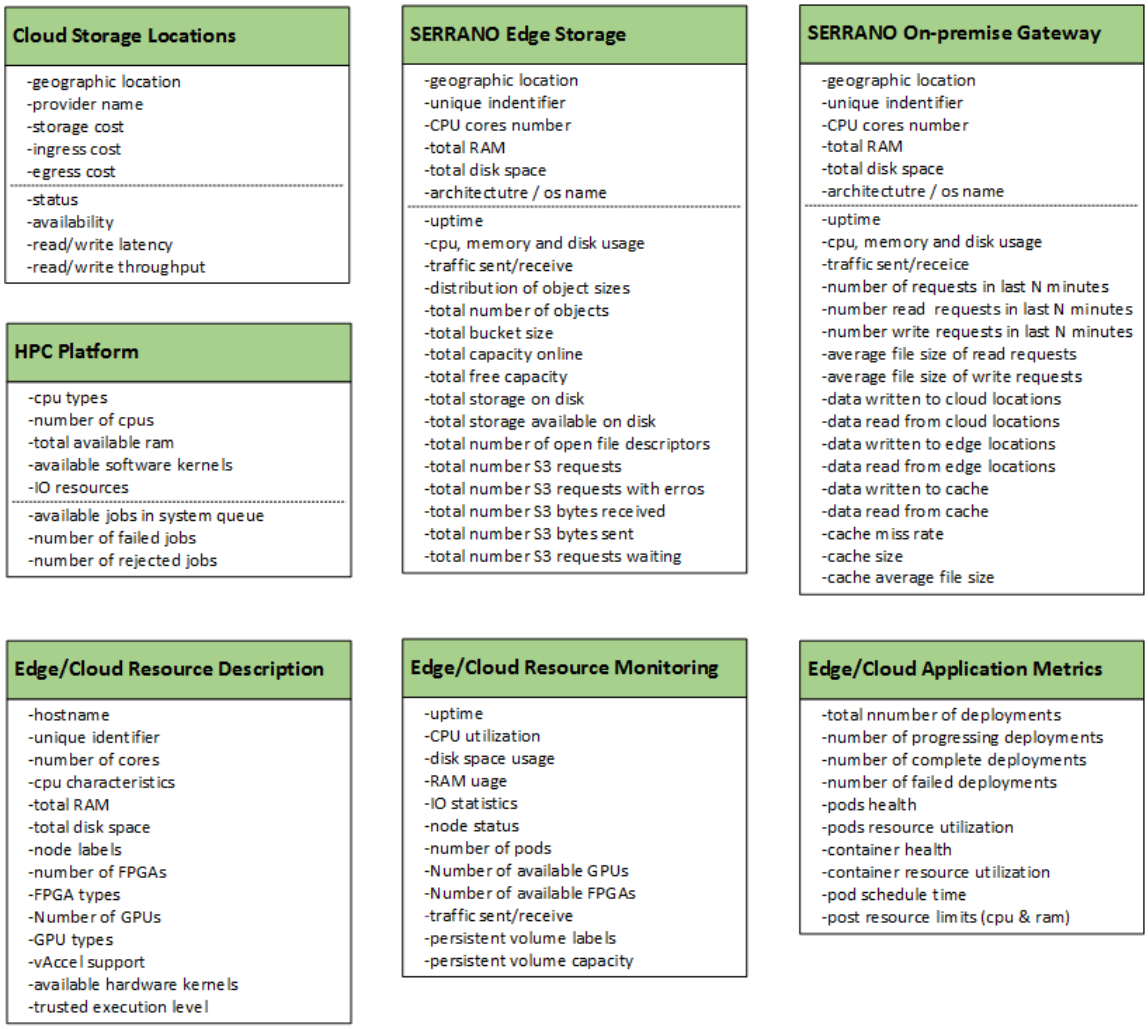


Figure 7: Summary of collected resource description and monitoring parameters in the SERRANO platform

3.3 Telemetry interfaces

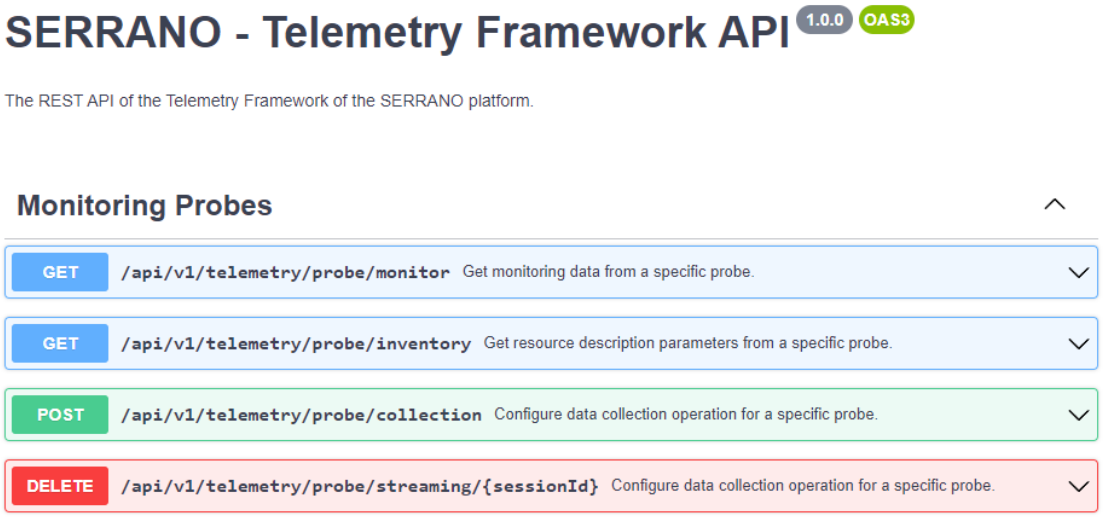
The Central Telemetry Handler and the Enhanced Telemetry Agent expose control and management operations through a RESTful API. They also provide an asynchronous messaging interface based on the Advanced Message Queuing Protocol (AMQP) to forward notification

messages at subscribed services (either from the SERRANO platform or external ones) through a set of predefined exchange topics.

Table 1 summarizes all the supported methods for the REST-based interface which is implemented in Python based on the Flask 2.0 micro web framework, with the async module enabled. In addition, all the REST APIs include Swagger documentation (Figure 8).

Table 1: Telemetry REST API

HTTP Verb	URI	Description
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 in probe.
POST	/api/v1/telemetry/probe/streaming	Activate streaming session at a probe.
DELETE	/api/v1/telemetry/probe/streaming/{ID}	Terminate streaming session at a probe.
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/{ID}	Remove registered telemetry instance from a specific telemetry entity.
GET	/api/v1/telemetry/agent/register/{ID}	Get details about a telemetry instance.
PUT	/api/v1/telemetry/agent/register/{ID}	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/{ID}	Get monitoring data from a specific probe.
GET	/api/v1/telemetry/ agent/inventory/{ID}	Get resource description parameters from a specific probe.



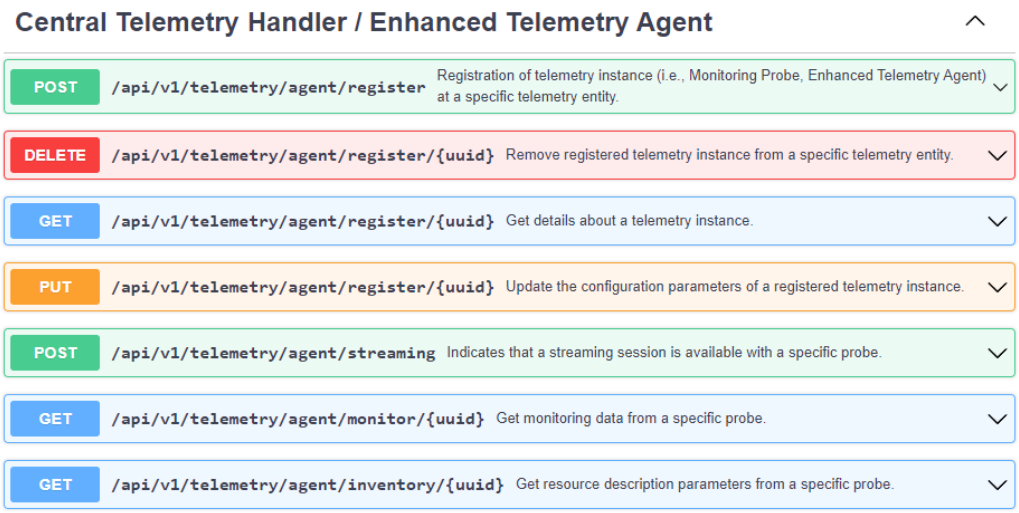


Figure 8: Telemetry access interface – Swagger documentation of REST APIs

The SERRANO platform relies on a message broker-based interface to collect and forward asynchronously the appropriate messages and events from the various distributed components. This interface is provided by the Data Broker (Section 6). For the telemetry framework operation there is a predefined topic exchange, an exchange is a messaging routing mechanism that can support different routing logics, where the messages published by the Notification Engine are tagged with a routing key composed by a list of words delimited by dots. These words usually specify some feature connected to the messages; consumers bind their queues to specific routing keys according to the messages they want to receive.

The Notification Handler posts messages to a predefined topic exchange using a routing key following the format: ***<notification-category>.<event-identifier>***. The first parameter indicates the category of the notification and can be: *(a) Telemetry, (b) Resources, (c) General*. This approach allows the subscribed components to filter the notification messages using any combination of the two parameters. The content of each notification message is described JavaScript Object Notation (JSON) format using a common syntax. Table 2 provides the structure of the notification messages exposed by the SERRANO telemetry framework.

Table 2: Telemetry notification messages

Notification Type	Event Identifier	Parameters
General	Information	• ***message*** (*string*): event related information • ***timestamp*** **(***integer)*: Unix time stamp
Telemetry	Agent	• ***agent_id*** *(string)*: Enhanced Telemetry Agent unique identifier • ***status*** *(string)*: “UP” or “DOWN” • ***timestamp*** *(integer)*: Unix time stamp
Telemetry	Probe	• ***probe_id*** *(string)*: Monitoring probe unique identifier • ***status*** *(string)*: “UP” or “DOWN” • ***timestamp*** *(integer)*: Unix time stamp
Resources	Status	• ***resource_id*** *(string)*: Resource unique identifier • ***event*** *(string)*: Detected event • ***timestamp*** *(integer)*: Unix time stamp

3.4 AI/ML-assisted Network and Cloud Telemetry

3.4.1 ML Network Tomography

Networks are always progressing to support the evolving and diverse applications and the needs for improved capacity, latency and security. To this end, monitoring is key to ensuring the uninterrupted network operation and the QoS of the applications. Network Tomography uses a subset of monitoring information, corresponding to partial view of the network state, to estimate wide-sense network performance, including unmonitored parameters. In this work, we present a novel Machine Learning (ML) formulation for Network Tomography. The proposed formulation accounts for realistic scenarios where: i) the existence of certain links of the network is not known (e.g., due to security reasons), ii) the routing is dynamic (non-deterministic), i.e., for the same origin-destination node pair, a different route may be selected depending on the state of certain links. Our simulations indicate that our proposal has better estimation accuracy compared to traditional algebraic or other ML approaches that cannot or do not take into account these two assumptions.

3.4.1.1 Introduction

Cloud and edge computing infrastructures play an essential part in today's economies. They offer a whole range of processing services, for example, online commerce, smartphone applications, video streaming, gaming, etc. At the same time their heavy use and rapid deployment makes them increasingly complex, heterogeneous and difficult to manage. The level of performance of the edge-cloud continuum determines the efficiency of the whole Information and Communication Technology (ICT) infrastructure, and the Quality of Service (QoS) perceived by the deployed applications. The heterogeneity of the networks coupled with the increased traffic volumes and rates, make their real time (dynamic) management both challenging and necessary. Multiple factors can affect network performance, such as: packet loss, abnormal delay, delay variance, bad load distribution and poor behavior of network operating systems or user applications. These factors can result in network soft (network performance degradation) or hard (network downtime) failures, causing significant costs. Therefore, it is critical to provide advanced network monitoring tools able to reflect changes in the network state and analyze them.

As is the case with any system, a network has to be observable in order to be manageable and stable. Network tomography (NT) [1] is an indispensable tool for this purpose. NT refers to large-scale network inference. It involves estimating network (path) performance parameters based on traffic measurements at a limited subset of network nodes and links. Indicative monitoring data set includes (but is not limited to): throughput, delay, jitter, packet delivery ratio, congestion, bandwidth for specific paths. Some of these metrics are additive per link (e.g., delay, jitter), or can be transformed to an additive form (e.g., use the log function in the case of the packet delivery ratio or reliability), or they are non-additive (e.g., the congestion level or the bandwidth, depends on the worst link of a path, involving a min operator).

NT reduces the monitoring needs for the whole network. Thus, it increases efficiency, reduces equipment and operating costs and can help verify service level agreements. The more accurate is the knowledge of the network state obtained through NT, the better positioned is the Orchestration layer in maximizing the efficiency of resource utilization (including network, compute, memory and storage resources) in mixed cloud/fog-edge/HPC environments. For example, in real time communication it is important to estimate in advance the quality of a connection before actually establishing it over a specific path [2]. NT can provide the means for this purpose, as an unestablished connection cannot really be monitored. Moreover, certain security events and anomalies can be promptly detected and dealt with, before they significantly affect the operation of the network.

Monitoring can be passive or active. In active monitoring, certain probes (that create extra traffic) are purposefully deployed to measure the required network parts. In passive monitoring, (part of) the existing traffic is monitored. In all cases, the need to keep the monitoring cost low implies that the number of probes that are deployed or of existing connections that are measured, is kept as small as possible. A major point of concern is the complexity of the monitoring solution. In large scale networks, vast amounts of monitoring data are generated, making the correlation of the data a very difficult task. Therefore, efficient algorithms need to be developed to infer the appropriate metrics from the data. Another point of concern is that in complex networks a path may cross different and heterogeneous subnetworks. Under these circumstances, the complete network topology is usually not completely known by a single actor as it is not fully advertised. So, a path that is monitored or whose performance we want to estimate may cross different networks and contain an unknown number of links. This makes difficult or even impossible the use of traditional NT algebraic methods. These methods assume complete knowledge of the network topology, where the links of the network are typically represented in a matrix. Moreover, in modern networks, the routing between an origin and a destination node may be dynamic (non-deterministic), as it may change based on the state of congestion in the network. This should also be taken into account when designing a NT algorithm, to be able to provide accurate estimations.

In this work we develop a novel ML formulation for NT. The features of the ML algorithm are designed to take into account the peculiarities of modern networks where the topology may or may not be completely known and the routing may be static or dynamic. Moreover, the algorithm can estimate additive as well as certain non-additive metrics. The features of the ML formulation can provide the required information to estimate both additive metrics, and non-linear metrics. We demonstrate through simulation experiments that the accuracy of our proposal is excellent in a variety of scenarios, and is in many cases far better than that of other ML or pure algebraic approaches. The improved accuracy can help the network operator have a better view of the network conditions and make better optimization decisions. Thus, the overall network efficiency and the application QoS are improved.

3.4.1.2 Related Work

The term Network Tomography was first mentioned in [1]. The initial problem statement was to estimate node-to-node network traffic intensity from link measurements. As the subsequent research is vast and ongoing, we will provide a short summary of it here. More information can be found in the references of the mentioned work and surveys. Note that when we refer to delay in this work, we assume that it comprises of propagation delay and queueing delay. The path delay is the sum of the delays of the links that are contained. In [3], the authors used unicast end-to-end traffic measurements and developed techniques to estimate link delay distribution. Passive network tomography was first introduced in [4]. The authors assumed the measurement of end-to-end performance metrics of existing traffic, and researched the problem of identifying lossy links. Reference [5] focused on characterizing end-to-end additive metrics. The authors find a minimal set of *k* linearly independent paths that can describe the metrics of all the other paths. The authors in [6] used multiple and simple one-way measurements among pairs of nodes. Then they estimated the one-way delay between network nodes. In doing so they used a global objective function that is affected by the network topology and not just by individual measurements. Reference [7] focused on the problem of identifying link level metrics from end-to-end metrics of selected paths. The authors developed a low-complexity algorithm to construct linearly independent, cycle-free paths between monitors without examining all candidate paths. The authors of [8] assumed a similar problem statement. They investigated the conditions under which the link level metrics could be acquired, depending on the network topology and the number and location of the monitors. In [9] various variations are surveyed, such as link delay inference through multicast end-to-end measurements, origin-destination matrix inference and topology identification. Survey [10] describes subsequent developments in NT. It also presents network coding and compressed sensing to improve estimation accuracy, computational complexity and probing and operational cost.

Another topic is that of failure detection using network tomography. In this scenario, network tomography pertains to identifying whether a network node has failed given binary (normal or failed) end-to-end path metrics. In [11] the authors researched the conditions that need to be satisfied in order to identify a bounded number of node failures. They also quantified the maximum number of identifiable node failures and the largest node set within which failures can be localized for a given number of failures. In [12] the authors provided upper bounds on the maximum number of identifiable failed nodes, considering a certain number of monitored paths, constraints on the network topology, the routing scheme, and the maximum path length.

Recently, ML has been applied to the field of NT. The authors of [13] proposed a NT approach for non-deterministic routing where the measured flows for a given origin-destination node pair may cross different paths. In [14] a Neural Network is trained to infer additive metrics in an SDN/NFV environment. The authors in [15] also propose the use of neural network to infer metrics based only on the origin and destination node pair, and also to reconstruct the network topology. In [16], assuming in-vehicle network monitoring, neural networks are again

applied to estimate the performance of an unmonitored part of a network. Finally, in [17] the same principles are applied to the domain of network slicing.

In this work we consider that the network topology information may be incomplete as in [15], and at the same time, the routing may be dynamic. We also assume that the specific paths that are monitored are a given, and that we want to make the most of the available information. We select a set of ML features that can be used to improve the accuracy of the estimations under these assumptions. Our contributions are:

- We present a network tomography approach based on ML, which accounts for incomplete knowledge of the underlying network topology and for dynamic routing.
- Using end-to-end measurements, we infer link-level metrics (both additive and certain non-additive) for known links, or a subset of a path if it crosses unknown parts of the topology.
- Using the above knowledge, we estimate performance metrics for unestablished or unmonitored paths, even with both incomplete topology knowledge (i.e., their exact routing is not known) and dynamic routing.

Our proposal is different to [15] in that we employ additional ML features to account for the unknown part of the network. The features are particularly useful in cases as in [13], where the routing is dynamic. This provides better accuracy for certain scenarios as we will demonstrate in the simulations. To the best of our knowledge, there is no previous NT formulation that considers both dynamic (non-deterministic) routing and partial network topology knowledge.

3.4.1.3 Network Tomography

In this section we will first describe the network scenario and the notation. Next, we will provide details on the algebraic methods that can be used to solve a basic NT problem. Afterwards, we will describe our proposed ML formulation.

3.4.1.3.1 Network Notation

We consider a network $N=(V,L)$ where V denotes the set of nodes and L the set of known directed links (thus, both (i,j) and (j,i) are present in L if nodes i and j are connected through a unidirectional link). We assume a set P of paths already established in the network. The routing matrix of the established paths is defined as the binary matrix $G_P \in \{0,1\}^{|P|\times|E|}$, where $G_P[p,l]=1$ when path p contains link l, and is 0, otherwise. Consider the end-to-end vector of parameters $\mathbf{y}_P \in \mathbb{R}^{|P|}$, with the different entries of $\mathbf{y}_P$ representing the different performance parameters (e.g., delay or jitter, etc.) that we are measuring.

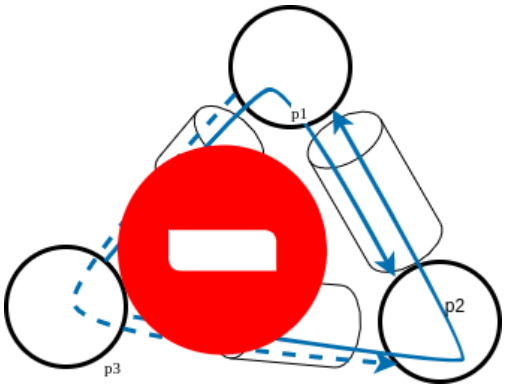


Figure 9: A network with 2 established monitored paths and one candidate (or unmonitored), sharing one known link and one origin node. Parts of the topology may be unknown

If the link-level vector parameters $\mathbf{x} \in \mathbb{R}^{|L|}$ are additive per link, vector $\mathbf{y}_P$ can be written as a linear combination of link-level vector parameters **x,** so that $\mathbf{y}_P = G_P\mathbf{x}$. We assume that we want to estimate the end-to-end parameters of a set *M* of paths (either new or unmonitored ones), denoted by vector $\mathbf{y}_M \in \mathbb{R}^M$, assuming that we know their routing $G_M \in \{0,1\}^{|M|\times|L|}$. Then, we have:

$$\boldsymbol{y} = \begin{bmatrix} y_P \\ y_M \end{bmatrix} = \begin{bmatrix} G_P \\ G_M \end{bmatrix} \boldsymbol{x} \qquad (1)$$

Consider, for example, the network of Figure 9, where a set of $P = \{p_1, p_2\}$ paths are already established and correspond to submatrix G_P and the known end-to-end QoS values $\mathbf{y}_P = \{y_1, y_2\}$. The values in vector $\mathbf{y}_P$ can be different for different applications and use cases. The path to be established is denoted by $M = \{m_3\}$ whose end-to-end value y_3 we want to estimate. We assume that all the (three) links the paths use are known. The routing can be described as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad (2)$$

Since the new path m_3 contains links that are already in use by other paths, it is possible to estimate its end-to-end value. This can be achieved using the Moore-Penrose inverse $(\,.\,)^+$ of G_P, that is:

$\hat{\boldsymbol{y}}_M = G_M G_P^+ y_P.$

3.4.1.3.2 ML network tomography

The above algebraic formulation is normally used when the network topology is completely known. It is not expected to work well when certain links are not known. As we have already mentioned, this is the case in many of the modern networks. In this section we present a ML formulation that takes this into account. The features are chosen to summarize in a heuristic way the important characteristics of the paths with respect to the estimation problem. We organize the feature matrix *F* so that each row corresponds to a path $p \in P$, while the columns

represent (link or node, feature) pairs, for the links or nodes of the path and the values of the chosen features. In particular, we choose two kinds of features. The first set of features consists of all the known links of the topology (we define it as a set $|L|$). The second set of features consists of all the path nodes that can be origin or destination. More specifically, we define *S* as a $|P| \times |L|$ link-level feature matrix designed to take into account the known links that a path contains. Element S_{pl}, corresponds to path *p* and link *l,* and is set equal to 1 if path *p* contains link *l*, and equal to zero, otherwise. We also define the node-level feature matrix *A* to represent information regarding the origin and destination node of a path. In particular, *A* is a $|P| \times |V|$ node-level feature matrix. Element A_{pv} is equal to 1 if path *p* starts or ends at node *v*, and is set to zero, otherwise. Note that this formulation does not specifically take into account which node is origin and which one is destination. However, the information is indirectly captured by the ML algorithm as the set L contains both directions of a link (directed links). We also consider an additional feature to represent the bias term (denoted by BT). The bias term can account for monitoring errors or noise that cannot be reduced by any other means.

Table 3: Important Notations

Symbol	Description
N	The network in study
V	The set of network nodes
E	The set of links
G	Binary routing matrix
y	Vector of end-to-end monitoring metrics
x	Vector of link-level metrics
P	Set of monitored paths
M	Set of new or unmonitored paths
F	Complete ML feature matrix
S	ML feature matrix for the links of the paths
A	ML feature matrix for the origin and destination node of the paths
BT	ML bias term
k	Number of k-shortest paths between two nodes

We concatenate all the link-level feature matrices into one feature matrix defined as:

$$F = [BT \quad S \quad A]_{|P|x(1+|L|+|V|)}$$

The features are designed to capture as much information as possible to evaluate the metric at hand. The absence of knowledge of certain links through which the path passes is counterbalanced by the knowledge of the origin and destination node of a path. Moreover, this formulation is better than assuming only the origin and destination node of a path as features. The knowledge of even a small subset of the links that a path contains can greatly increase the accuracy of the estimation. After the features have been defined, an appropriate ML algorithm can be used to solve the problem at hand. In this work we evaluated various architectures of neural networks towards this end. Neural networks have the potential to

represent complex functions and thus estimate both additive and certain non-additive metrics. The assessment of other algorithms was left a topic of future work.

3.4.1.3.3 Neural Network Architectures

A neural network (NN) is a set of connected functions that interpret a set of inputs into a desired kind of output. A NN consists of an input layer, one (or more) hidden layer(s), and an output layer. Each node of one layer is fully connected to all the nodes of the next layer. This enables data propagation from one layer to another. Each layer multiplies the input by a weight matrix and adds a bias term. The hidden layers can be wider (have more nodes) than the input and the output. In these cases, the neural network may be able to learn more complex relationships between the features and thus have better accuracy. In the next section we present the evaluation of different architectures and compare our proposed tomography framework to alternatives.

3.4.1.4 Results

To evaluate our proposed formulation, we performed a number of simulation experiments. We assumed the Deutsche Telecom topology of Figure 10 with 12 nodes and 20 unidirectional links.

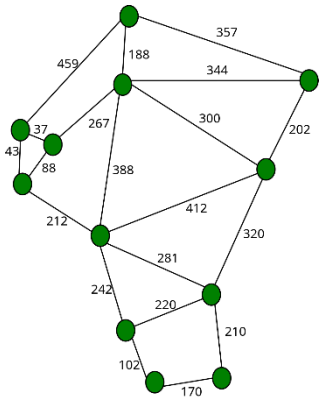


Figure 10: The DT topology with the link lengths in Km

In the figure, the length of each link in km is also depicted. We considered that the required estimations metrics are: i) the delay, which is additive per link, and ii) the bandwidth, which is non-additive and depends on the worst link of the path (min operation over the links of the path). We set the delay of each link to be numerically equal to its length. So, the delay of a path is equal to the sum of the delays of its links. We set the bandwidth of each link to be numerically equal to its length. The bandwidth of a path equals to the minimum bandwidth of the links that it contains. We assumed various different loads of 100, 200, 300, 400, 500 connections with uniformly chosen source-destination nodes. For each source-destination pair we used a *k*-shortest path algorithm with *k*=1, 2, 3 to decide the routing. When *k*>1 the specific route for each source-destination pair was chosen uniformly over these *k* paths. We also assumed an increasing number of unknown links, where, for a given link that is considered unknown, we removed its related value from the routing matrix *G*, and from the feature matrix *S*. Simulations were performed in MATLAB. The training of the NN was performed using 2000

epochs. We used 90% of the established paths for training, and 10% for testing. We exclude from the testing set any path that contains a link that is not used at all in the training set. We run 200 independent iterations and averaged the results.

3.4.1.4.1 Additive metrics evaluation

First, we evaluated several different NN architectures that employed the features described in section III.B.

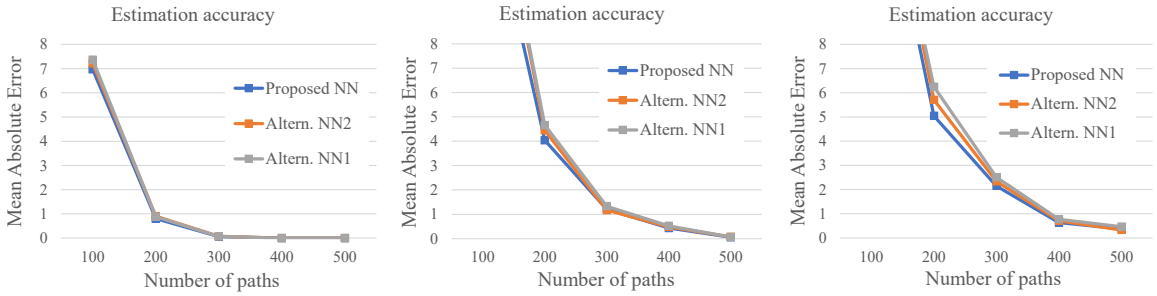


Figure 11: Accuracy (for the delay metric) of three NN architectures for different number of paths and a) k=1, b) k=2, c) k=3

In Figure 11 we present the estimation accuracy of three architectures that achieved the best results for the estimation of the delay. We plot the accuracy in terms of Mean Absolute Error (MAE) as a function of the number of established paths in the network. The proposed NN had three fully connected layers. The number of outputs of the first, second and third layer was four, two and one time(s) the number of features (32), respectively. The alternative NN1 also had three fully connected layers. The number of outputs of the first and second layer was equal to the number of features. The outputs of the third layer were equal to half the number of features. The alternative NN2 had two fully connected layers. The number of outputs of both layers was equal to the number of features. In Figure 11a, k=1, so that each source-destination pair can only be served by one path (static routing). The accuracy of the three architectures is very similar and increases as the number of paths increases, since the NNs are better trained, with more data, in that case. As k increases (Figure 11b and Figure 11c), a source-destination pair may be routed over different paths (non-deterministic or dynamic routing). This means that the neural network should be able to correlate a larger amount of information to provide an accurate output (i.e., the origin and destination pair by themselves do not contain the required information for an accurate estimation). The proposed NN has marginally better accuracy than the other architectures. It seems that the additional outputs of the layers allow the neural network to learn better the relationships between the features, at least for smaller number of established paths. We also compared the three architectures for the case of the bandwidth estimation (non-additive metric) and the results were similar (not shown due to space limitations).

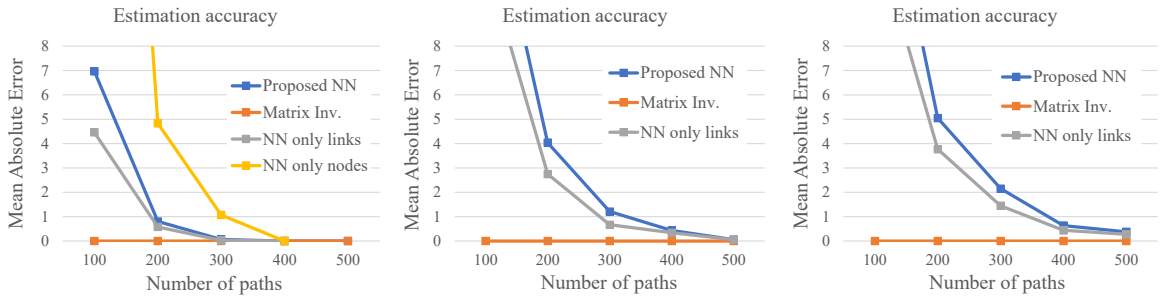


Figure 12: Accuracy (for the delay metric) of the algorithms for different number of paths and a) k=1, b) k=2, c) k=3

We then compared the accuracy of four algorithms: i) our proposed NN formulation, ii) a NN formulation where the features are only the links of the paths, iii) a NN formulation where the features are only the origin and destination nodes of the paths, and iv) a traditional algebraic matrix inversion solution. Figure 12 presents the MAE of the four examined algorithms for different number of measured paths. In Figure 12a, where k=1, we notice that a simple matrix inversion can achieve good accuracy even with a relatively small number of measured paths. The algorithms that are based on NN require a larger number of established paths to be trained so as to have good accuracy. The NN that uses only the links as features, achieves slightly better accuracy earlier than the proposed NN (in this scenario). The reason is that the proposed NN has more features thus requiring a larger training set. The NN that only uses the origin and destination node requires the largest training set. The reason is that the NN needs to be trained with all possible combinations of origin and destination nodes in order to be able to provide an accurate estimate. For 500 established paths, all algorithms have MAE less than 10^{-4} and the maximum error is less than 10^{-3}. In Figure 12b and Figure 12c, where we have k>1, the simple matrix inversion method still achieves the best accuracy in all cases (the MAE and the maximum error are similar to the case where k=1). The NN that relies only on the origin and destination node does not have enough information to provide an accurate estimate. Its accuracy is not depicted in these figures, but is examined in Figure 13. The other NNs require larger number of established paths to achieve comparable accuracy, since the non-deterministic routing requires more training data in order for the NN to understand it. In the case of 500 paths, the MAE of both algorithms are approximately the same, and it is in the order of $5*10^{-2}$ for k=2 and $3*10^{-1}$ for k=3. The maximum error is approximately 150 and 220 delay units for k=2 and k=3, respectively.

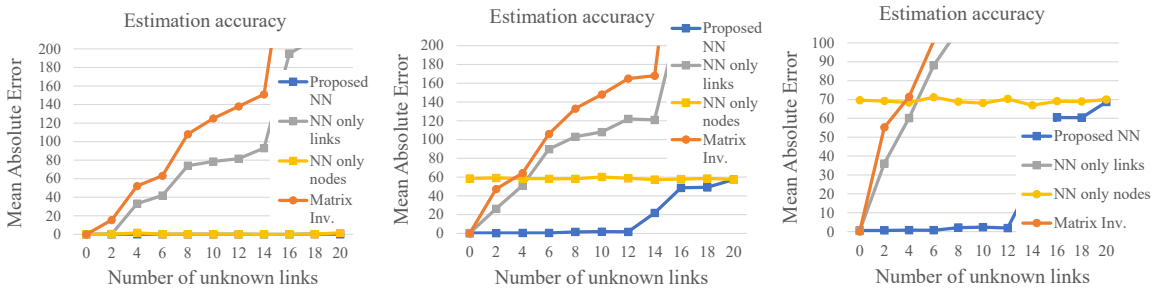


Figure 13: Accuracy (for the delay metric) of the algorithms for different number of unknown links and a) k=1, b) k=2, c) k=3

In Figure 13 we assumed 400 established paths and we examine the MAE of the algorithms for an increasing number of unknown links. In Figure 13a, *k* is equal to 1. Note that the blue line of the proposed NN is under the yellow one of the NN that employs as features only nodes. The matrix inversion and the neural network that relies only on the links contained on a path, exhibit the worst accuracy as the number of unknown links increases. The reason is that these algorithms do not have the appropriate information to compensate for the missing topology knowledge. The proposed NN has a bit better MAE than the NN that uses only the origin and destination node as features. The MAE of our proposal is approximately 10^{-2} for most cases (until the number of unknown links is 16), and is approximately 0.2 for the other cases. The maximum error ranges from 10^{-3} for small number of unknown links, to approximately 100 and 700 when the unknown links are more. As we can see these algorithms are able to compensate for the unknown network links by using the origin and destination node to estimate the delay of each path. In Figure 13b and Figure 13c we have $k>1$. We notice that the proposed NN still achieves good accuracy even with 12 unknown links. After that, its accuracy deteriorates rapidly, and with 20 unknown links (all the links of the network), its accuracy is (as expected) equal to the NN whose features are only the origin and destination node. For both $k=1$ and $k=2$, the MAE is approximately 2 until the unknown links are 12. The maximum error is above 200 in almost all cases. As we can see, the combination of the features of the proposed NN work really well in this scenario. The two kinds of features can work together and provide estimates even for a large number of unknown links, and under non-deterministic routing ($k>1$). This is particularly useful in modern network scenarios where the topology of the network may not be completely known due to security reasons and also connections with the same origin and destination nodes may be routed differently over the physical topology.

3.4.1.4.2 Non-additive metrics evaluation

In this subsection we compare the accuracy of the three different ML algorithms for the case of the bandwidth estimation, which is a non-additive metric. The matrix inversion obviously cannot be evaluated in this case.

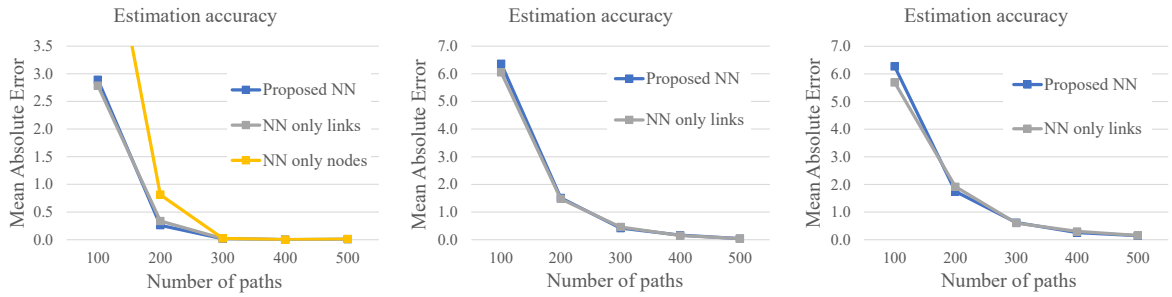


Figure 14: Accuracy (for the bandwidth metric) of the algorithms for different number of paths and a) k=1, b) k=2, c) k=3

In Figure 14, we compare the accuracy of the algorithms again in terms of MAE and for different number of measured paths. In Figure 14a, where $k=1$, we can see that all algorithms achieve excellent accuracy with a relatively low number of measured paths. The MAE for 500 paths is in the order of 10^{-3} for all the algorithms, and the maximum error is approximately 30.

In Figure 14b, Figure 14c where k>1, the accuracy of the NN that uses only the origin and destination nodes is again not good, since it still lacks the necessary information to provide an accurate estimate. The other two algorithms have similar performance. Their MAE is in the order of 10^{-2} and 10^{-1}, and the maximum error is 40 and 100 for k=2 and k=3 respectively. These numbers are slightly worse than the case of the delay estimation. This indicates that the examined NNs may not be the best fit for this kind of non-linear estimation.

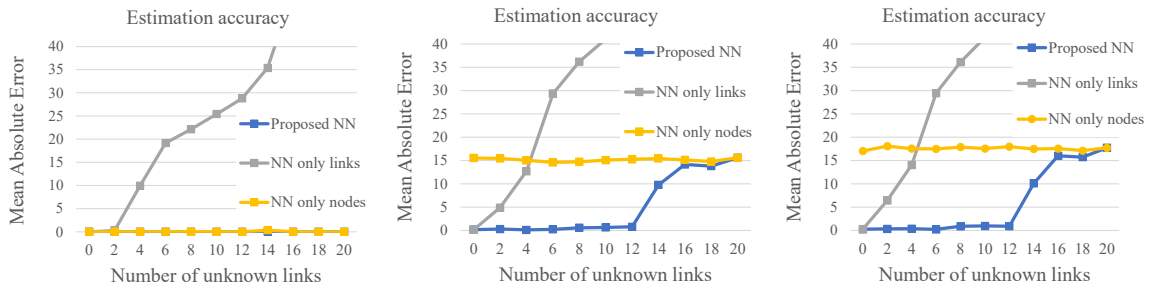


Figure 15: Accuracy (for the bandwidth metric) of the algorithms for different number of paths and a) k=1, b) k=2, c) k=3

Finally, in Figure 15 we considered 400 established paths and we again examined the MAE of the three algorithms for an increasing number of unknown links. For all the algorithms we notice behavior similar to the case of Figure 13 (and again slightly worse MAE), indicating that our proposed NN can achieve good estimation accuracy for both additive and the non-additive metrics examined.

3.4.1.5 Conclusions

In this work we presented a novel ML formulation for Network Tomography under the assumptions of incomplete topology knowledge and dynamic routing. We designed suitable features that work well under these assumptions that are present in modern networks. The results indicate significant improvement in estimation accuracy when compared to other approaches. Future work includes the evaluation of other algorithms, and the localization of failures.

3.5 Anomaly Detection in Networks

3.5.1 Introduction to Anomaly Detection

Anomaly detection or outlier detection refers to the problem of finding patterns in the data that significantly deviate from the expected normal behaviour of the data points in the rest of the data. In other words, as defined by Hawkins [19], an outlier is “an observation which deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism”. Pattern finding uses techniques and methods from the fields of statistics, data mining, machine learning, information theory, spectral theory to apply them to a specific problem formulation.

Anomalies can occur in data due to many different reasons such as malicious activity or systems failures. Anomaly detection differ from *noise removal* and *noise accommodation* but is relatively close to another topic, *novelty detection.* Novelty detection aims at detecting previously unobserved patterns in data not known during training. The difference between anomaly detection and novelty detection is that novelty patterns are incorporated into the normal model after they are found.

Anomaly detection algorithms are used in many application domains such as astronomy, medical, social computing, and many more. Among all these application domains, synonyms are often used such as *outlier detection*, *novelty detection*, *fraud detection*, *misuse detection*, *intrusion detection* and *behavioural analysis*. The techniques used in previous mentioned problems can be used for anomaly detection and vice versa. *Network intrusion detection system* (NIDS) is probably the most well-known use case of anomaly detection. NIDS in a nutshell is the identification of potential intrusion attacks and exploits by monitoring network traffic and server applications. Another use case is *fraud detection* which analyses log data to detect misuses of the data or suspicious events indicating fraud in credit card payments or financial transactions. *Data leakage prevention* is another case like fraud detection which analyses logs of databases access, file servers and other information sources to detect abnormal access patterns in real time.

To define what is perceived as normal behaviour, we can use terms from statistics such as mean, median to capture the *norms* associated with distributions. These single scalar or multidimensional vectors along with the distance of a point from these values are used to assess which point is abnormal. But with *non-normal* distributions this straight forward interpretation no longer applies. In this case we define *norm* as a set of points rather than a single point[20]. For example, if the data set is characterized by variations in the density over clusters in the data space, such variations must also be considered to determine if the point is an outlier. So, a larger distance near a less dense cluster may represent an anomaly whereas a smaller distance near a too much denser cluster may also be an outlier too.

3.5.1.1 Anomaly detection approaches

To adopt a suitable anomaly detection approach and applicable algorithm, we have to take into account the nature of the data and the data labels.

Nature of data: Each data instance is described using a set of attributes. The attributes can be of different data type such as binary, categorical, or continuous. Also, each data instance might consist of only one attribute (univariate) or multiple attributes (multivariate) of the same type or a mixture of different data types. The nature of attributes determines the applicability of the anomaly detection technique[21].

Labels: Data instances with labels denote whether that instance is normal or outlier. Based on these labels there are three broad categories of anomaly detection techniques:

- **Unsupervised**: Here the test dataset is unlabeled, and the assumption is that the majority of the instances in the data set are normal. Anomalies are detected by looking for instances that seem to fit the least to the remainder of the data set.

- **Supervised**: The data set is labeled as “normal” and “abnormal” and involves training a classifier.
- **Semi supervised**: Here a small set of labeled data is used, to improve the accuracy of the unsupervised learning algorithm.

3.5.1.2 Algorithm Categorization

In the following we categorize the algorithms available for anomaly detection. Among them, Nearest Neighbour and Clustering are the most used categories of algorithms in practice. Here we give a short introduction to algorithms in each category and their main ideas.

Nearest-neighbour based techniques

In this technique normal data instances are located in dense neighbourhoods, while anomalies lie far from their closest neighbours. Nearest neighbour-based anomaly detection techniques need a distance or a similarity measure to define the difference between two data instances. By defining abnormal as the process of identifying anomaly as a data point that is different from others, the primary approaches used for outlier detection can be characterized as:

- Distance based: Points that are farther from others are considered more anomalous.
- Density based: Points that are in relatively low-density regions are considered more anomalous.
- Rank based: The most anomalous points are those whose nearest neighbours have others as nearest neighbours.

Anomaly detection algorithms differ based on how they figure out the distance. Some of the similarity measures include measures such as the Minkowski, Euclidean, Mahalanobis, cosine similarity and Jaccard index. The well-known *kth Nearest Neighbor* anomaly algorithm uses an anomaly score of a data instance as its distance to its kth nearest neighbour.

Classifier based algorithms

Here the space region containing normal data is separated from all other regions. This category of algorithms is mainly based on classification algorithms such as neural networks or support vector machines.

Clustering based methods

Clustering refers to unsupervised learning algorithms that classify patterns (observations, data items, or feature vectors) which do not require pre-labeled data to extract rules for grouping similar data instances. Cluster based methods follow a simple assumption: usually anomalies either belong to a small sparse cluster or do not belong to any cluster, whereas the normal objects are a part of large and dense clusters. Clusters of the data objects can be constructed using numerous methods.

3.5.1.3 Network Traffic and Anomalies

In this subsection we will provide some short information about network traffic attributes, and network anomaly detection. Network traffic features in most datasets can be categorized as: flow, basic, content and time. The *flow* category consists of socket information (i.e., IP address and port number) and types of protocols. Information about protocols connections such as traffic payload, network service, and time-to-live are part of the *basic* category. Features of TCP/IP such as traffic payload are considered as *content* category. Category *traffic* includes connection time such as round-trip time of TCP, arrival time between packets, and start/end time.

3.5.1.3.1 Intrusion Detection Systems

Traditional intrusion detection systems are signature-based, need frequent rule-based updates and are not capable of detecting unknown attacks. In contrast, IDS systems based on anomaly detection methods, model the normal system/network behaviour and can be used to detect known and unknown "zero day" attacks. IDS can be used to monitor and analyse inside and outside network data from different data sources or environments to detect malicious behaviours in system performance, tasks behaviours or network attacks. The drawbacks of IDS systems are high false error rates and inner system complexities.

3.5.1.3.2 Anomaly Detection Algorithms: The case of UNSW-NB15 Dataset

To evaluate different machine learning algorithms, and the effects of labels in IDS, an experiment on a subset of the UNSW-NB15 [22] dataset is presented here. This dataset is comprised of 257,673 realistic normal and abnormal (attacks) network activities. It contains 49 features and attacks are categorized as Analysis, Backdoor, DoS, Exploits, Fuzzers, Generic, Reconnaissance, Shellcode and Worms.

Table 4: UNSW-NB15 dataset statistics

Released Year	Num. of Data	Num. of Features	Number of Attack Categories	Normal Data	Attack Data
2015	257,673	49	9	64%	36%

The dataset contains certain errors, so a **preprocessing** phase is required, and it pertains to:

- Unification of text values by changing the lower/upper/proper cases
- Treatment of nulls and spaces according to each column attribute
- Convert the numbers stored as text into a number type

One-Hot Encoding: The nominal features should be converted to numerical values before fitting the machine learning models. One-Hot Encoding deals with categorical columns by creating new columns that are mapped to the number of distinct values. Each new column represents a single distinct category. It assigns ones matched to the category locations in the original column and the remainder are zeros.

Feature Scaling: There can be some problems when fitting machine learning models if the features with large scales dominate the others. The goal of normalization is to equalize the importance of the features. Z-score or Min-Max scaler are some of the ways for normalization.

Feature Selection: This step is very important to design efficient machine learning model and reduce the high dimensionality problem. Univariant Feature Selection and Important Feature selection methods are used to find most relevant features.

3.5.1.3.3 Initial results

We evaluated a number of unsupervised and unsupervised learning algorithms and we present some initial results. More specifically, we considered the following supervised learning algorithms: Support Vector Machine, Decision Tree, and the following unsupervised algorithms: K nearest neighbours, Local Outlier Factor, Variational Auto Encoder. We used the Pyod [23] software package to implement the algorithms. We used 70% of the samples for training and 30% for testing. In Table 5 we can see the results of the supervised learning algorithms. As we can see, both algorithms manage to identify perfectly all the abnormal data. These results however may not be reproducible in realistic scenarios, where the labelling may not be accurate and certain novel attacks could be present. The unsupervised learning algorithms did not achieve good accuracy. We are in the process of optimisation and debugging to improve the results. We will also evaluate the aforementioned algorithms using different datasets.

Table 5: Supervised Learning results

Algorithm	Accuracy	F1 Score	Precision Score
SVM	1.0	1.0	1.0
Decision Tree	1.0	1.0	1.0

4 Resource Orchestration Mechanisms

The realization of SERRANO's ambition requires the efficient and seamless orchestration of the available edge, cloud, and HPC resources. To this end, SERRANO adopts a hierarchical architecture to enable end-to-end cognitive resource orchestration and support transparent application deployment over heterogeneous resources. The architecture includes one high-level orchestrator, the SERRANO Resource Orchestrator, which is developed in the context of the project, and multiple Local Orchestrators that handle the individual parts of the overall infrastructure. The adopted design is aligned with the current transition towards federations of loosely coupled systems. Moreover, it enables SERRANO Resource Orchestrator to manage the underlying heterogeneous infrastructure in a more abstract and disaggregated manner compared to the Local Orchestrators.

SERRANO considers that Local Orchestrators are based on existing and well-established solutions. More specifically, the orchestration platform for the edge and cloud platforms is the Kubernetes (K8s), whereas for the HPC platforms HPC resource managers and batch jobs schedulers, such as Slurm [55] and PBS-based (e.g. TORQUE [56], OpenPBS [57]) are considered. Hence, the Resource Orchestrator is responsible to provide cognitive resource allocation at high-level and ensure seamless application deployment functionalities over this multi-platform infrastructure. Figure 16 presents the architecture of the SERRANO resource orchestration mechanisms and the interactions with other SERRANO components. In this case, the SERRANO orchestration mechanism coordinate and unify three individual platforms.

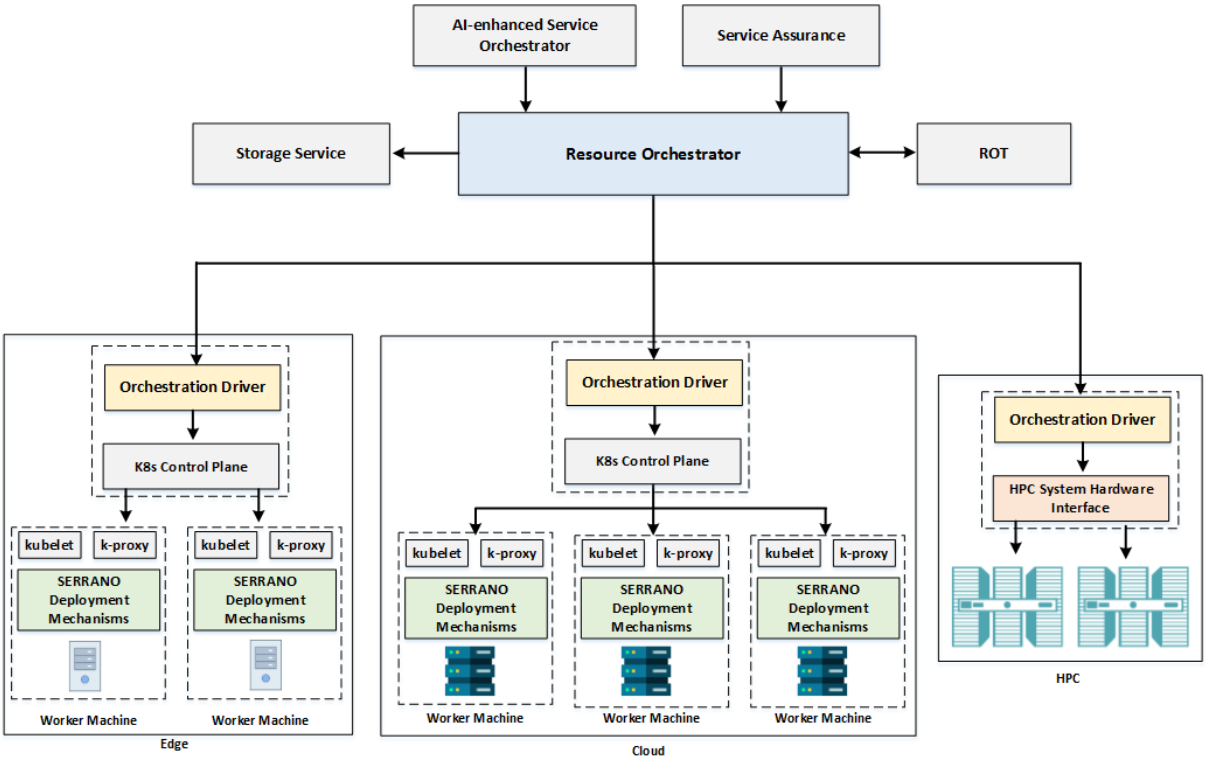


Figure 16: SERRANO distributed and cognitive resource orchestration

4.1 SERRANO Resource Orchestrator

The AI-enhanced Service Orchestrator (details in D5.1 “Abstraction models and intelligent service orchestration”) and the service assurance mechanisms are the two main components that interact with the Resource Orchestrator to request the deployment of cloud-native applications, which follow the microservices architecture design pattern, and the provision of data-driven adjustments.

SERRANO adopts a two-stage approach to cognitively allocate the resources to the application’s microservices and absolve the developers from the burden of resource allocation and scaling. The Resource Orchestrator is responsible for the initial stage, it also initiates the second stage that includes additionally the actual application deployment. Furthermore, it automatically coordinates the necessary supplemental actions (e.g., transfer required data) in order to enable the transparent execution of cloud-native applications across the multiple individual platforms within the SERRANO ecosystem.

Initially, the Resource Orchestrator assigns the application’s microservices to the most appropriate individual platform and forms also the appropriate low-level infrastructure-specific deployment objectives for the Local Orchestrators. 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 of each microservice to the corresponding Local Orchestrators at the selected platforms. To this end, the Resource Orchestrator adopts a declarative approach, instead of an imperative one, to describe the workload requirements to the selected Local Orchestrators. In this way, the control is passed through the Orchestration Drivers to the individual Local Orchestrators that perform the final cognitive resource orchestration that satisfies both the high-level orchestration objectives and ensures optimal use of the platform resources.

The Resource Orchestrator (Figure 17) is implemented in Python and consists of four main components.

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 necessary interfaces that allow bidirectional communication for exchanging commands, information, and notifications. It validates all the requests before forwarding them to the Dispatcher. Figure 18 shows the current draft version of the RESTful API. As the work in T5.1 and T5.5 progresses, a more detailed description of the interface will be available in the upcoming period prior to releasing the initial version of the SERRANO platform (M18).

The ***Datastore*** stores configuration data and state data for the available platforms and deployed applications. It is based on etcd [58], an open-source distributed key-value store. etcd is also among the core Kubernetes components and serves as the primary store for cluster’s data. Kubernetes uses etcd’s “watch” function to monitor this data and reconfigure itself when changes occur. Resource Orchestrator uses etcd similarly to keep track of the actual and desired state of the deployed workloads across the overall unified infrastructure.

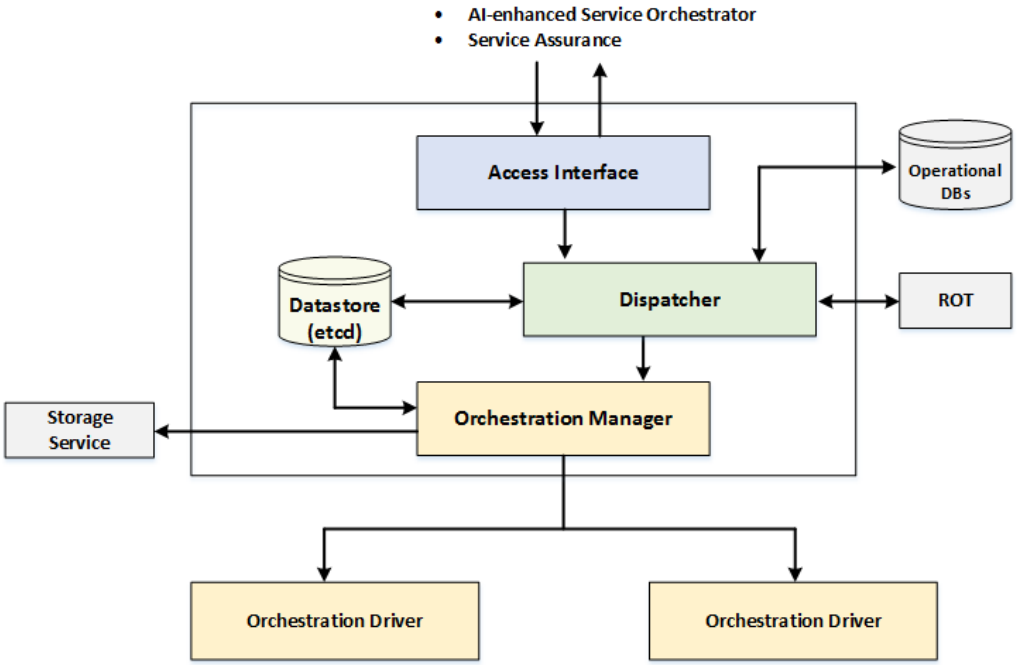


Figure 17: SERRANO Resource Orchestrator

The ***Dispatcher*** implements the application logic, oversees the operation of the other internal components and coordinates the resource allocation and application deployment operations. It interacts with the ROT to retrieve the application deployment instructions and updates accordingly the internal datastore. The Dispatcher also triggers the deployment procedure through the interaction with the Orchestration Manager.

Finally, the ***Orchestration Manager*** prepares the required application deployment instructions (declarative approach) based on the ROT's Moreover, it coordinates the required data movement through the interaction with the Distributed Secure Storage service and triggers the application deployment by interacting with the Orchestration Drivers at the selected edge/cloud/HPC platforms.

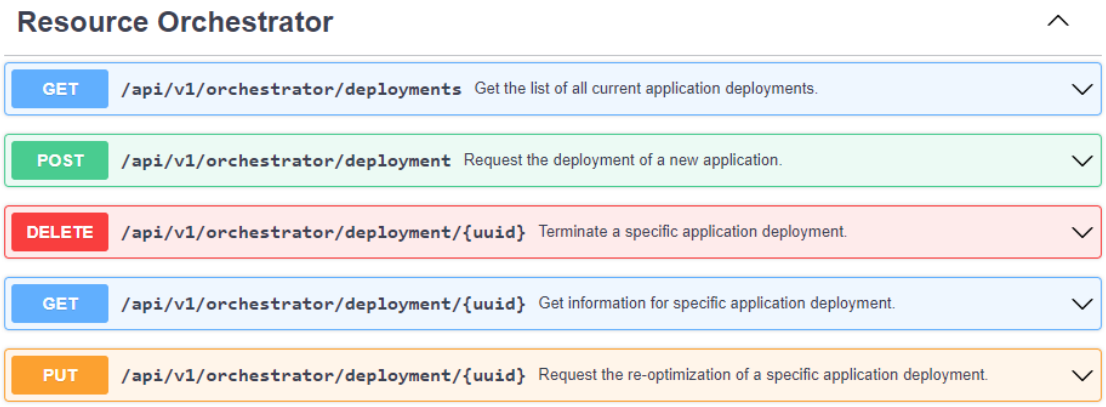


Figure 18: Resource Orchestrator access interface – Swagger documentation of REST APIs

4.2 Orchestration Drivers

According to the SERRANO architecture, the Orchestration Drivers complete the implementation of the hierarchical resource orchestration. An Orchestration Driver provides an abstraction layer for the interaction with the specific edge, cloud, HPC orchestration mechanisms, dealing with the low-level details of the heterogeneous Local Orchestrators at the individual platforms.

Figure 19 shows the design of the Orchestration Drivers. Since SERRANO unifies platforms that use different local orchestration mechanisms (i.e., K8s in edge/cloud and Slurm and PBS in HPC), two types of Orchestration Drivers will also be available.

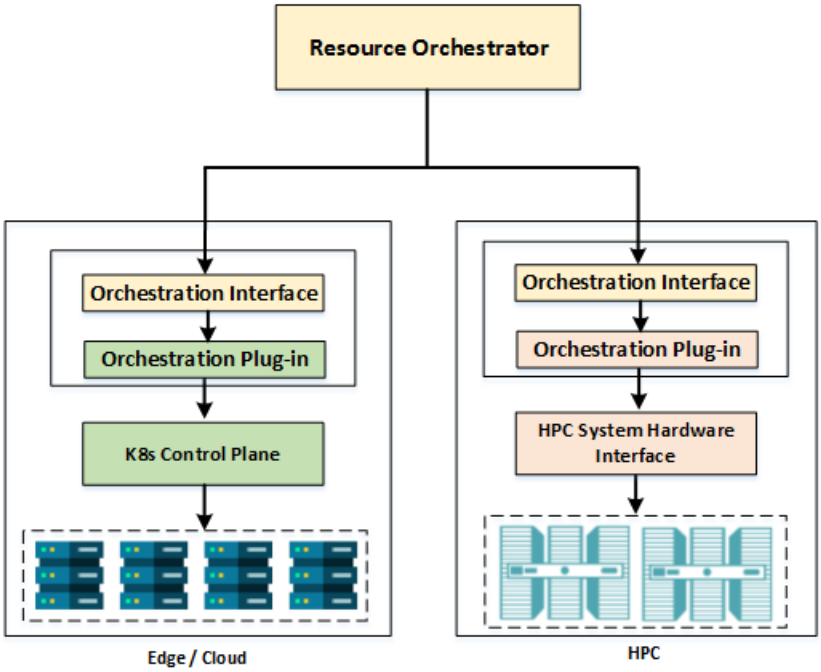


Figure 19: Orchestration Drivers

The ***Orchestration Interface*** provides 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. The ***Orchestration Plug-in*** translates the infrastructure-aware scheduling objectives by the Resource Orchestrator to specific instructions according to the application logic and internal procedures of the Local Orchestrator at each platform. This component is different for each Orchestration Driver type and interfacing with a Local Orchestrator using specific exposed APIs and tools. To this end, the Orchestrator Driver are implemented as plug-ins.

The following sections describe the initial integration of the SERRANO orchestration mechanisms with Kubernetes and HPC platforms.

4.3 Integration with Kubernetes

4.3.1 Kubernetes overview

Kubernetes, whose overall architecture is depicted in Figure 20, is a portable, extensible, open-source platform for managing containerized workloads and services, which facilitates both declarative configuration and automation. It provides a container runtime, container orchestration, container-centric infrastructure orchestration, self-healing mechanisms, service discovery and load balancing.

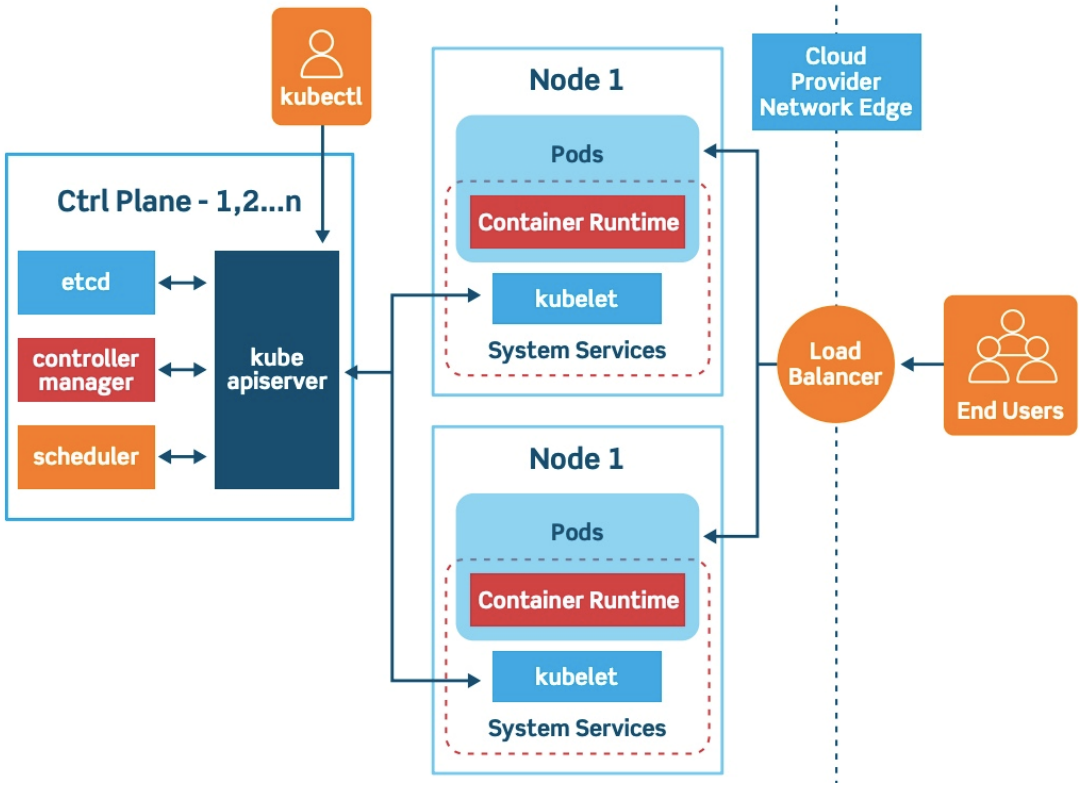


Figure 20: Kubernetes architecture

It aims to reduce the burden of orchestrating underlying compute, network, and storage infrastructure, and enable application operators and developers to focus entirely on container-centric workflows for self-service operation. It allows developers to build customized workflows and higher-level automation to deploy and manage applications composed of multiple containers.

While Kubernetes runs all major categories of workloads, such as monolithic, stateless or stateful applications, microservices, services, batch jobs and everything in between, it is commonly used for the microservice category of workloads.

From a high level, a Kubernetes environment consists of a control plane (master), a distributed storage system for keeping the cluster state consistent (*etcd*), and a number of cluster nodes (*kubelets*).

While in SERRANO we base our orchestration stack for the edge and cloud nodes on K8s, in this section we focus on the low-level components, namely the container runtimes that implement the preparation, spawning and tearing down of workloads on these nodes, since most of the project contributions in terms of resource orchestration, especially for edge nodes, involve this layer.

4.3.2 Container runtimes

The default container runtime for K8s is *containerd*. As of v1.20.0, K8s drops support for Docker, leaving containerd as the dominant container runtime present on most production deployments today. containerd is an industry-standard container runtime with an emphasis on simplicity, robustness and portability. It manages the complete container lifecycle of its host system, including image transfer and storage, container execution and supervision, low-level storage and network attachments, etc. In addition, it fully supports the Open Container Initiative (OCI) runtime specification for running containers [35]. The glue with K8s is CRI, a containerd plugin implementation of the Kubernetes Container Runtime Interface (CRI) (Figure 21), which allows us to use containerd as the container runtime for a Kubernetes cluster.

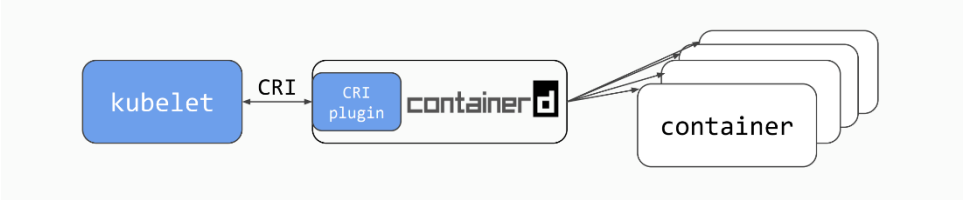


Figure 21: Kubernetes CRI Plugin Enabling containerd

However, since in SERRANO we aim to provide support for more than just containers (e.g., VMs, unikernels etc.) we enhance K8s support for alternative container runtimes that are able to execute traditional VMs, lightweight VMs and unikernels. A runtime environment that supports this kind of functionality is Kata Containers (formerly known as clear containers) [36].

The Kata Containers runtime is compatible with the OCI runtime specification and therefore works seamlessly with the K8s CRI. Kata Containers create a virtual machine for each pod that a kubelet (Kubernetes) creates. The containerd-shim-kata-v2 (referred to as shimv2 from this point onwards) is the Kata Containers entry point, which implements the Containerd Runtime V2 (Shim API) for Kata. Using shimV2, K8s can launch Pod and OCI compatible containers (Figure 22).

The container process is then spawned by kata-agent, an agent process running as a daemon inside the virtual machine. kata-agent runs a ttRPC server in the guest which the Virtual Machine Manager (VMM) exposes to the host. shimv2 uses a ttRPC protocol to communicate with the agent. This protocol allows the runtime to send container management commands to the agent. The protocol is also used to carry the I/O streams (stdout, stderr, stdin) between the containers and the managed engines (containerd).

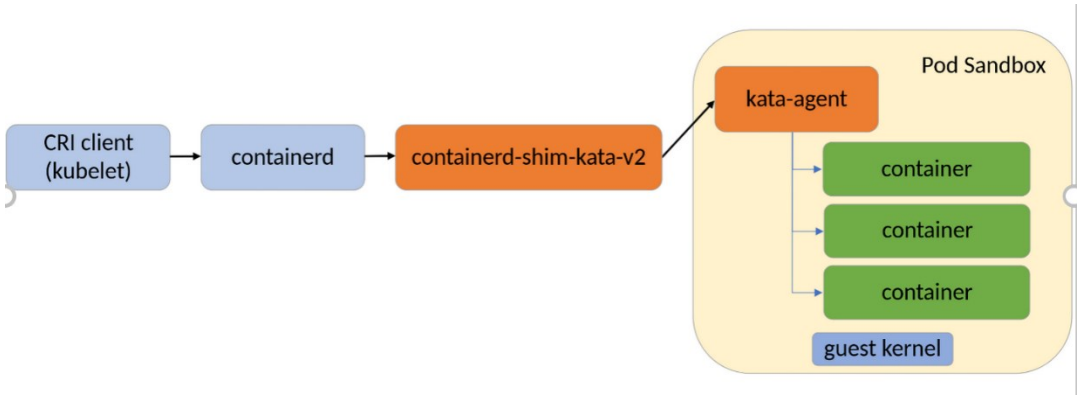


Figure 22: Kata Containers and containerd integration with shimv2

For any given container, both the *init* process and all potentially executed commands within that container, along with their related I/O streams, need to go through the interface exported by the VMM. The container workload, i.e., the actual OCI bundle rootfs, is exported from the host to the virtual machine. kata-agent uses this mount point as the root filesystem for the container process.

In SERRANO, we enhance Kata Containers to support vAccel, the hardware acceleration framework we introduce in the next section, which enables access to hardware acceleration resources without the need for direct hardware/device access.

4.3.3 Workload variants (containers, unikernels, VMs)

As mentioned, in SERRANO we aim to support a hybrid workload deployment framework, providing users with the opportunity to spawn containers, VMs, unikernels depending on the task requirements in networking functionality, computational efficiency etc. To this end, we employ a custom system software stack that provides support for three modes of application spawning: (i) containers, (ii) VMs and (iii) unikernels. All modes present benefits and deficiencies, which we enumerate below. To facilitate application bundling and packaging, we use container images for all three modes (provisioning).

Containers are the simplest form of application spawning. They base their functionality on the Linux kernel's namespaces concept, thus providing minimum support for isolation. Spawning a container on any system works almost out of the box, except for minor issues due to architecture incompatibilities (e.g., aarch64 worker nodes vs x86_64 controller nodes).

VMs is one of the oldest forms of workload/application isolation on the same physical host. The management overhead for spawning and running an application in a Virtual Machine is significant, and that is the reason that the community shifted towards containers. However, their strong isolation properties make them ideal for cloud providers and, in general, for multi-tenant environments.

Containers provide speed and simplicity, but they lack strong isolation between the various containers in the same machine. For this reason, during the first period of their adoption, when used in production in cloud environments, the operators used VMs to separate them.

However, this solution introduces significant overhead and leads to inefficient use of the resources. VMs provide strong isolation but need more resources and increase the overall time to spawn a container-based workload since they require significantly greater boot time than a simple container. Hence, new frameworks were introduced to support container execution in a sandbox environment based on micro-VMS. One such solution is the Kata Containers that support the execution of sandboxed containers.

Unikernels are a fairly new concept: a unikernel is a tiny piece of binary, able to run as a conventional operating system bundled with application, containing only the bits and pieces needed for execution. For example, a web server unikernel contains the binary code for the actual web server, the OS layers to provide network capabilities, storage capabilities as well as the layers needed for bootstrapping the application and the interaction with the hypervisor.

4.3.4 Hardware acceleration

The common way of using hardware accelerators in a distributed, cloud-managed environment is by assigning the hardware accelerator directly to the instance where the workload that needs acceleration is running. This method, however, entails cumbersome setups and complicated requirements to the management layer raising at the same time security concerns. In SERRANO, we introduce a lightweight API-remoting framework where workloads can enjoy hardware acceleration functionality in an operation-level granularity, without direct access to the hardware. To this end, we introduce vAccel, a virtual acceleration framework tailored for cloud-native, lightweight virtualisation setups.

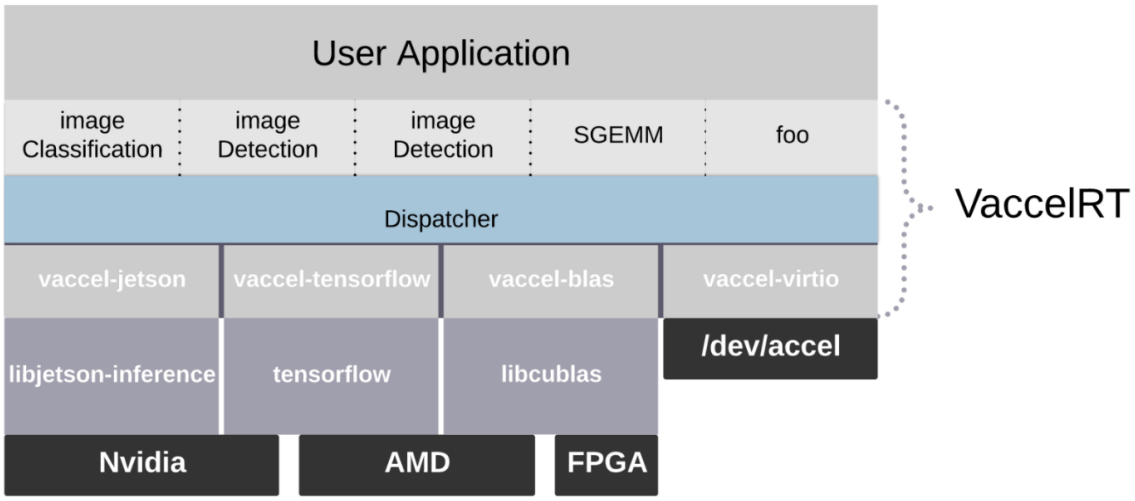


Figure 23: vAccel framework

vAccel, depicted in Figure 23, is a lightweight framework that exposes hardware acceleration functionality to processes or VMs via a thin runtime system, vAccelRT. vAccelRT abstracts away any hardware/vendor-specific code by employing a modular design where backends implement bindings for popular acceleration frameworks and the frontend exposes a function prototype for each available acceleration function. Using an optimized paravirtual interface,

vAccelRT is exposed to a VM's user-space where applications can benefit from hardware acceleration via a simple function call.

We integrate vAccel with the Kata containers runtime, and thus, the higher-level orchestrator is able to annotate tasks with a specific label, so that applications are spawned with the vAccel functionality.

4.4 Integration with HPC platforms

4.4.1 HPC infrastructure overview

There are fundamental differences between cloud and HPC workloads. Unlike typical workloads that run continuously in the cloud, such as services, in HPC the execution time of a workload is limited, either by the walltime (i.e., the maximum time the application can run) set in the user's batch job parameters or default walltime set by the resource manager.

Another difference is the accessibility of the resources. In the cloud, the resources are expected to be accessible on-demand, whereas in HPC, the cluster's resources are shared between the users, therefore job queues are common. When a job is submitted, it is first placed in the queue, where the batch job stays until the requested resources become available. This causes a queue delay between the job submission time and the actual start of the job. It is common to introduce distinct queues for certain resources, e.g., queues dedicated to fast storage (SSDs) or hardware accelerators (GPUs), as well as to dedicate queues to specific job types, such as short queues for small jobs, where test runs can be performed without large delays due to congestion. A user then specifies which queue to use in the job parameters.

HPC clusters are managed by resource managers, which are responsible for the enqueueing and scheduling of users' jobs, as well as the management and access of the cluster resources, such as compute nodes, cores, memory, storage, network, licenses, and hardware accelerators. The most popular resource managers are PBS-based managers (such as Torque and OpenPBS), and Slurm.

4.4.2 SERRANO HPC Gateway Interface

Deliverable D4.2 (Section 9) describes the functionality of the SERRANO interface for HPC Services (SERRANO HPC Gateway). In this deliverable, the interaction between the HPC Gateway and HPC system is explained.

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 *srun* and *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.

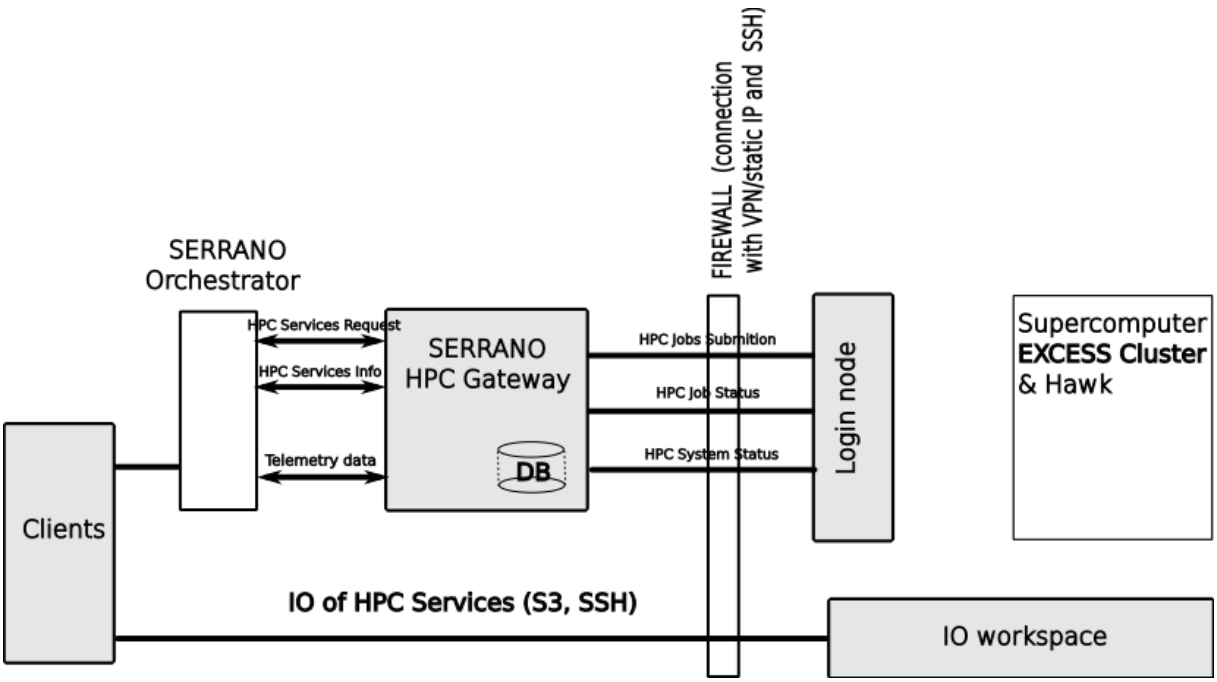


Figure 24: Interaction between HPC Gateway and HPC infrastructure

Therefore, SERRANO HPC Gateway will communicate with the front-end (login) nodes via SSH and use commands specific to the resource managers under use in order to prepare a batch job script for submission (i.e., to select appropriate header, see D4.2), submit the job and monitor the status of the job, as shown in Figure 24. To present a conceptual example of the job management actions, the following snippet for PBS presents a list of shell commands to perform the actions of the HPC Gateway.

```
# qsub returns an ID of the submitted job, which is saved in a variable
export JOB_ID=$(qsub batch-job-script.sh)

# check whether the job is completed, "C" stands for completed
qstat -f $JOB_ID | grep 'job_state' | grep -o '.$'
C

# check whether the job was executed successfully, non-zero exit status means a failed
# job
qstat -f $JOB_ID | grep 'exit_status' | grep -o '.$'
0
```

As the next steps, the HPC Gateway will be extended to support automatic generation of batch scripts based on the selected HPC services, developed in WP4, include invocation of underlying HPC monitoring system, as well as implement interfaces to interact with the orchestrator and provide telemetry data.

5 Lightweight Virtualization Mechanisms

In this section, we describe lightweight virtualization mechanisms used in the systems software stack of the SERRANO platform.

5.1 Overview

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 [37][38] 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 interface of virtual hardware. Additionally, users seek to maximize the number of requests handled, while at the same time minimize 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 [39].

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 [40] 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 [41]. 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 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 [42] per tenant where isolation is provided by the microVM monitor [43].

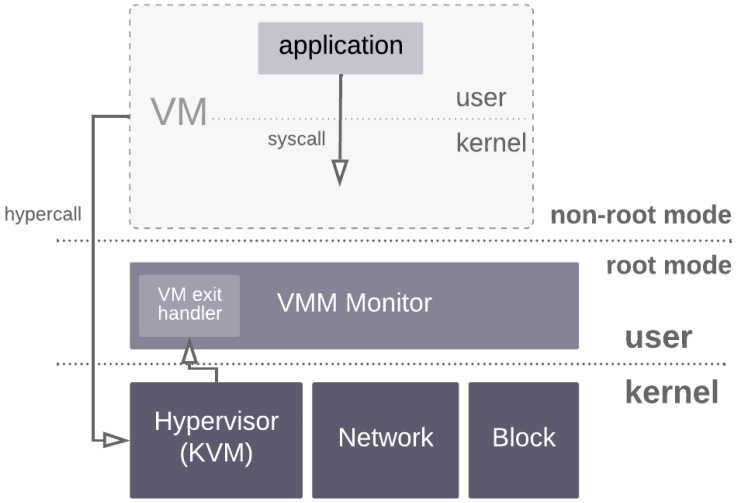


Figure 25: A Virtual Machine running on a generic user-space VMM on top of KVM

Figure 25 captures a snapshot of the traditional mode of execution for a generic VM on Linux/KVM using a standard user-space VMM. The 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 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 a number of architectures / features and makes perfect sense for this code to be in user-space.

On the other hand, in the context of Serverless Computing [44], 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 is probably going to be in the kernel (network, storage I/O etc.)? This is the vhost approach [45], taken to decouple the control path from the data path when performing high-performance I/O.

Apart from optimizing the data path, there has been considerable work by researchers to minimize the overhead of VM spawn and execution. Several minimalistic approaches have been proposed with regards to the systems software stack, in order to facilitate fast and secure application execution in the cloud. For instance, Unikernel as Processes [46] describes a specialized VMM with a number of backends (seccomp, KVM, muen etc.) that minimizes the attack surface by limiting the interface with the underlying layers. NEMU [47] is a stripped down QEMU, specifically built and designed to run modern cloud workloads on x86_64 and ARM CPUs. Amazon's Firecracker [43] is a fork of [48], 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 don't care about 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.

Balancing the trade-offs between lightweight application execution and workload isolation/security, in the SERRANO platform we use a hybrid approach. We enable the deployment of workloads in various execution modes (generic containers, sandboxed containers and unikernels) using the needed virtualization mechanisms for each mode. In the following sections we briefly go through the concept of unikernels, as well as the VMMs available in the SERRANO platform.

5.2 Unikernels

In the last years, a new approach in lightweight virtualization aims to bridge the best from both containers and virtual machines. Unikernels [49] are specialized single address space machine images constructed by using library operating systems. Some of their advantages include, fast boot times, low memory footprint, increased performance while on the same time they provide stronger security and hardware isolation. However, unikernels come with a lot of limitations and running existing application on top of them is not straightforward. While some frameworks try to provide a POSIX like environment some others prefer a clean state approach, requiring the complete refactor of an application to be able to execute on them.

Following the same design principles, Solo5 [50] is a specialized monitor for unikernels. Solo5 can be seen as an abstraction layer, permitting the same unikernel or application to be deployed in different environments (KVM, process, sandbox) without any modifications. One of the supported execution environments of Solo5 is a virtual machine running over KVM. In that scenario, Solo5 works like QEMU in a typical QEMU/KVM setup, i.e. It only provides a very minimalistic hypercall Application Binary Interface (ABI), which the guest can use for I/O requests.

In more details Solo5 provides five hypercalls related to I/O: (a) read from network, (b) write to network, (c) read from block device, (d) write to block device and (e) poll.

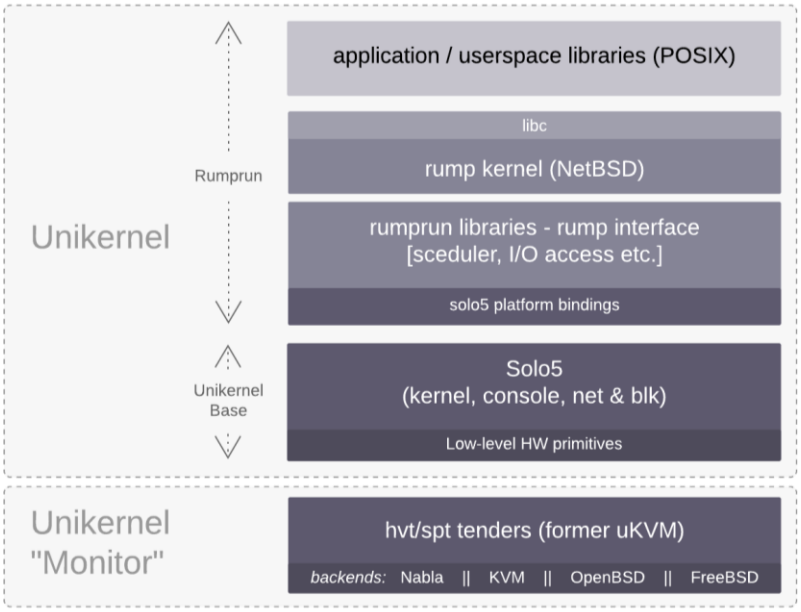


Figure 26: A simple unikernel stack (rumprun over solo5)

Due to the simplicity of both Solo5 and unikernels the initial setup for a guest is limited to the interaction with the KVM API and, as a result, the boot time is limited to some milliseconds.

5.3 QEMU / KVM

QEMU [51] is a free and open-source hypervisor. It emulates the machine's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating system. It can interoperate with Kernel-based Virtual Machine (KVM) to run virtual machines at near-native speed. QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another.

5.4 AWS Firecracker

Firecracker [52] is a virtual machine monitor (VMM) that uses the Linux Kernel-based Virtual Machine (KVM) to create and manage microVMs. Firecracker has a minimalist design. It excludes unnecessary devices and guest functionality to reduce the memory footprint and attack surface area of each microVM. This improves security, decreases the startup time, and increases hardware utilization. Firecracker is generally available on 64-bit Intel, AMD and ARM CPUs with support for hardware virtualization.

5.5 KVMM

In this section, we introduce KVMM, an in-kernel Virtual Machine Monitor, tailored to short-lived, purpose-based workloads. We briefly describe the basic design principles, as well as the management interface (API) of KVMM.

5.5.1 Virtual Machine Monitor

The case of Solo5 demonstrates how simple and minimal a hypervisor can be and it highlights 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 userspace monitor. In its turn the monitor handles the request from the guest and asks KVM to resume the guest execution. While this path makes totally sense in cases such as general-purpose hypervisors where different architectures or devices are emulated, in the case of lightweight virtualization seems more like an unnecessary switch from kernel space to user space. For instance, during a network I/O request the host kernel will return the control to user space monitor in order to handle the guest's request and the userspace monitor will eventually make a system call to transmit or receive the network packet, returning the control back to the host kernel. We would like to explore how big the overhead of these mode switches is and find solutions which can eliminate any overhead.

With the intention to answer the above questions we designed and implemented KVMM, a minimal and simplistic VMM that resides inside the Linux kernel interacting directly with KVM without any intervention from the user space. KVMM is essentially a simple dispatch handler in the kernel that services the needs of a guest. 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.).

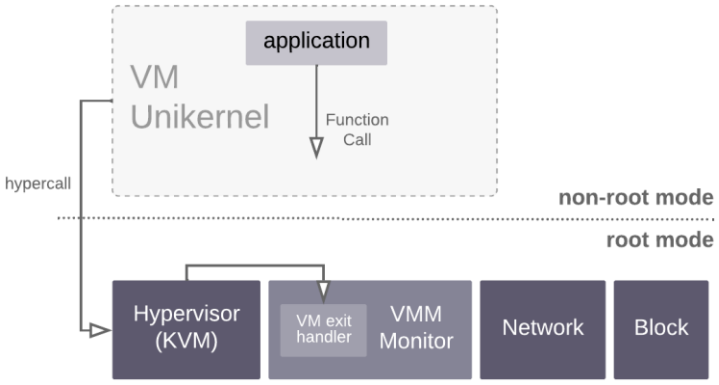


Figure 27: A unikernel running as a VM on KVMM

A major challenge that we faced in our approach was that KVM targets user space processes providing an API through file descriptors. Moreover, it is not possible to use KVM's API from inside the kernel because most needed functions are only used inside KVM. A way around is to create a glue code (which is actually some wrappers of KVM functions) between KVMM and KVM exposing all the needed functionality. For that reason, two small patches are required in order to be able to use KVMM. In all other cases KVMM works in a similar way as 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 a request to span a new VM? is received by the KVMM 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, KVMM allocates a virtual memory which will serve as guest's memory and it maps it to a virtual address of the newly created kernel thread's memory area. Thereby kernel thread mimics a user space process tricking KVM that it gets used from userspace.

An important aspect of KVMM's design is reducing the noise that VMMs enforce to handle I/O requests. Performance is one of the main goals of this project and in order to achieve that the guest needs to run uninterrupted as much as possible. Except of removing the mode switch overhead, KVMM handles I/O requests with the minimum possible overhead. The simple and minimal hypercall ABI from Solo5 helps in that direction. Network packages are formed from the guest and when the I/O request is occurred the job of KVMM 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 KVMM leverages the device mapper (DM) functionality to create a virtual block device, mapped to a physical device. The guest using the block read/write hypercalls from Solo5 ABI 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 a basic functionality of a Linux guest.

As every VMM, KVMM provides its own management interface. For the time being it is a very minimal and is able to handle basic VM operations such as start, stop etc. Someone can easily manage KVMM from both locally (user space) or remotely. In that manner KVMM can be easily managed in cases where user space access is not possible, such as edge nodes. In both cases KVMM 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 at last dump the console output of the guest. Furthermore, a user is also able to access stats such as boot and setup times, I/O operations (both disk and network) and generic stats regarding KVMM such as number of VMs, memory consumption and more. Someone can interact with the management interface locally via a specialized filesystem present in the linux kernel, *procfs*. When KVMM is loaded, two new files and one directory is created under */proc* directory:

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

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

5.5.2 Initial performance results

We employ basic micro-benchmarks to analyse the behaviour of KVMM. The most important design goals of our system are: (a) increase I/O performance, by eliminating the need to do mode-switches and exit to user-space to handle and setup I/O, and (b) minimize spawn & tear-down times, by cutting down management operations to the absolute minimum. Thus, we perform two basic experiments to determine how the system performs with regards to these two goals.

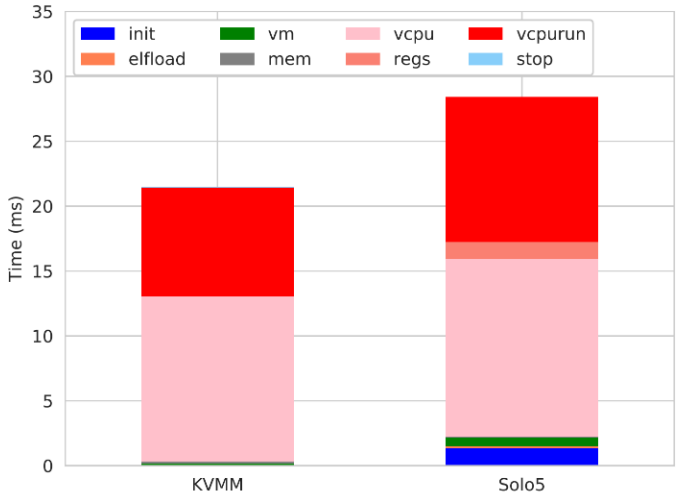


Figure 28: Execution time breakdown of the basic VM setup and spawn operations for a short-lived guest over KVMM and solo5-hvt

To measure the overhead of the operations needed to create, spawn, and shutdown a VM, we annotate the relevant functions and capture the time spent in each of the basic steps of the process.

Figure 28 shows the most important parts of the execution time of a simple hello-world VM broken down into the various sub-tasks. We present numbers for the same binary running on KVMM and on the generic HVT tender of solo5. First, the total execution time of the hello-world VM on KVMM is 24ms whereas on solo5, the time it takes to complete is 31ms. An interesting observation is that the VM initialization (*vm*) along with the vCPU allocation (*vcpu*) and the setup of its initial context (*regs*) are almost identical in the two cases. This shows that the solo5 implementation is optimized towards this direction.

To capture the network performance benefits of KVMM, we deploy a simple ping server[1] running as a solo5 unikernel. We plot the minimum, average and maximum latency numbers obtained using the generic ping utility in Figure 29. We run the test between two identical Atom boards, over two dedicated Intel 1Gbps NICs, connected back-to-back. We invoke ping with the following arguments: *ping -A -i0 -c 100000 IP*. The cases we consider are: a Linux VM running on QEMU/KVM (a) with and (b) without *vhost_net*, (c) a Linux VM running on top of an optimized microVMM (firecracker, including *vhost_net*), a simple ping server running as a unikernel on (d) QEMU/KVM, (e) over solo5-hvt, (f) over nabla (solo5-spt) and (g) over KVMM.

[1] https://github.com/Solo5/solo5/blob/master/tests/test_ping_serve/test_ping_serve.c

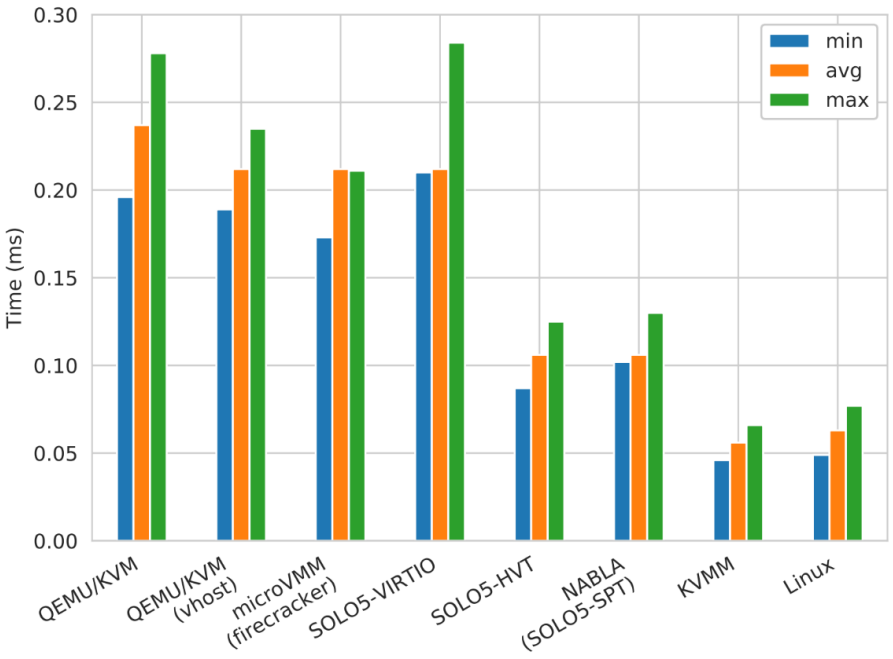


Figure 29: RTT latency measured with ping between an external Linux host and a generic Linux VM running on QEMU/KVM (with and without VHOST), a microVMM (firecracker), a unikernel running on QEMU/KVM, on Nabla (solo5-spt), Solo5-HVT, and on KVMM.

At first, we observe that KVMM outperforms all other cases, being 3.5-4.2x faster than the Linux guest, even over the optimized microVMM using vhost: this is unexpected, since the vhost approach is similar to KVMM. We attribute this difference to overheads associated with virtio and queue handling, pending further investigation. Additionally, KVMM appears almost 2x faster than solo5-hvt and solo5-spt. KVMM exhibits identical latency numbers compared to the native Linux case, which is expected since the Linux kernel network stack is as close as it can be to KVMM's exit handler.

This is a design choice: we leverage the proximity of the exit handler to the rest of the kernel functions providing access to devices, in order to minimize the virtualization overhead imposed by unnecessary layers of abstraction.

6 SERRANO Data Broker

The SERRANO platform is a distributed and event-driven software platform that requires appropriate communication mechanisms to provide the interconnection between the individual components, while ensuring scalability and loose coupling. Moreover, there are cases where a streaming platform is required to enable applications and internal components to handle and process efficiently real-time data streaming. To this end, the overall architecture includes the Data Broker component to support both message brokering, mostly using the Message Broker as described in Section 6.1, and distributed streaming capabilities, mostly using the Stream Handler as described in Section 6.2, through the abstraction and integration of the appropriate message infrastructure.

6.1 Message Broker

The Message Broker component facilitates the asynchronous communication and data transfer between the SERRANO platform components, based on the Message-Oriented Middleware (MOM) architecture. The component uses the publish and subscribe communication pattern to facilitate the loose coupling of the components while simultaneously serving multiple senders and receivers. In addition, it supports message validation (i.e., check if the exchanged messages comply with a specific format) and message transformation (i.e., transforms the messages from the sender's native format to the receiver's native format).

The Message Broker is based on the popular open source message-broker software RabbitMQ [53][54] that supports multiple messaging protocols like Advanced Message Queuing Protocol (AMQP), Simple (or Streaming) Text Orientated Messaging Protocol (STOMP) and MQ Telemetry Transport (MQTT). SERRANO message broker functionalities are based on the AMQP since it supports more advanced messaging services, message delivery guarantees, security at connections and transaction of messages. Moreover, RabbitMQ provides a wide range of developer tools for the most popular languages (e.g., in SERRANO we use pika [25], a pure-Python implementation of the AMQP protocol).

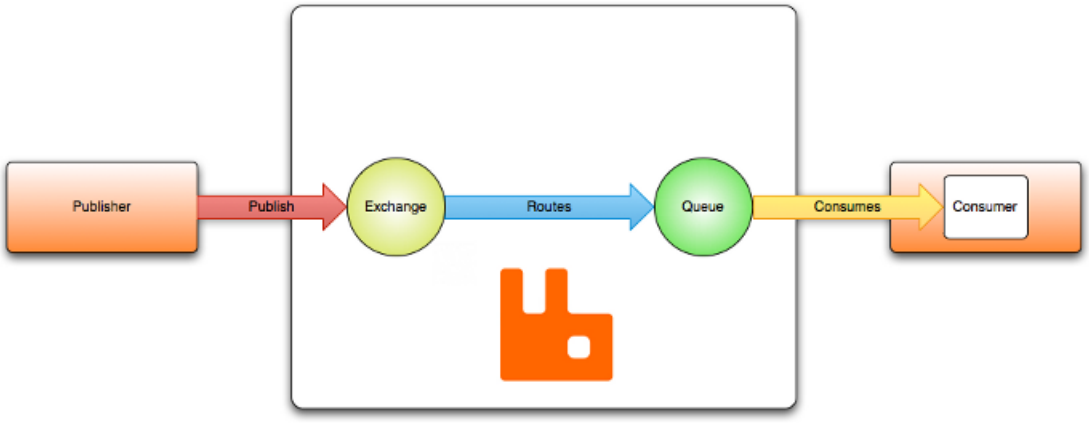


Figure 30: RabbitMQ basic elements [54]

The main functionality of the Message Broker is to provide the necessary message routing and delivery guarantees to the subscribed services. To this end, we build on top of the RabbitMQ concept of queues and exchanges (Figure 30). According to the messaging model in RabbitMQ, applications that publish messages (i.e., producers) never send any messages directly to applications that process them (i.e., consumers). Instead, the messages are sent to an exchange, responsible for routing the messages to the appropriate queues from where the consumer will receive them. The particular queues where the exchange will push a message from a producer is determined by the exchange type and rules called bindings.

The Message Broker component uses the following exchange types (Figure 31) to cover all the asynchronous communication patterns required from the SERRANO components.

The ***direct*** exchange delivers messages only to queues whose binding key exactly matches the message's routing key. It is ideal for the unicast routing of messages. Hence, a direct exchange is often used to distribute tasks between multiple workers. Additionally, the ***default*** exchange is a special direct exchange type with no name pre-declared by the broker. It makes it seem like messages are directly delivered to queues; however, technically, every queue created for those exchanges is automatically bound to it with a routing key that is the same as the queue name. The ***fanout*** exchange routes messages to all of the queues bound to it, and the routing key, if any is specified, is completely ignored. A fanout exchange is ideal for the broadcast routing of messages. Finally, the ***topic*** exchange routes messages to one or many queues based on a wildcard matching between a message routing key and the routing pattern specified in the binding. This type of exchange is widely used to provide multicast routing of messages.

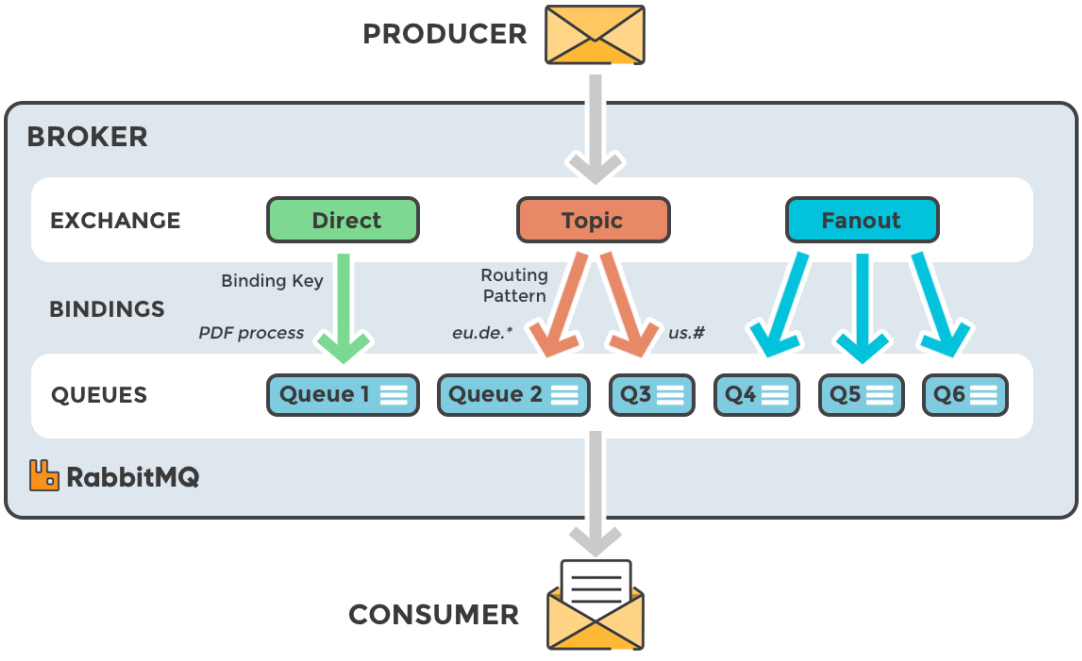


Figure 31: Overview of RabbitMQ exchange types and queues

Finally, Table 6 and Figure 32 describe the usage of the Message Broker within the SERRANO platform for the operation of the Resource Optimization Toolkit (ROT). A detailed description for the design and functionalities of this component is available in D5.2 "Algorithmic Framework, Performance and Power Models".

Table 6: ROT asynchronous communication over SERRANO Message Broker

Involved Components	Exchange Type	Configuration Details
Dispatcher to Execution Engine	***direct*** – Unicast message routing to selected Execution Engine.	• ***exchange*** = *"rot_dispatcher_requests"* • ***exchange_type*** = *"direct"* • ***routing_key*** = *"ENGINE_UUID"*
Execution Engine to Dispatcher	***default*** - Multiple producers (i.e., Execution Engines) to one consumer (i.e., Dispatcher)	• ***exchange*** = *""* • ***routing_key (i.e., queue)*** = *"rot_engines_to_controller"*
Dispatcher to other SERRANO components	***fanout*** – Broadcast message to all subscribed services.	• ***exchange*** = *"rot_dispatcher_events"* • ***exchange_type*** = *"fanout"* • ***routing_key*** = *""*

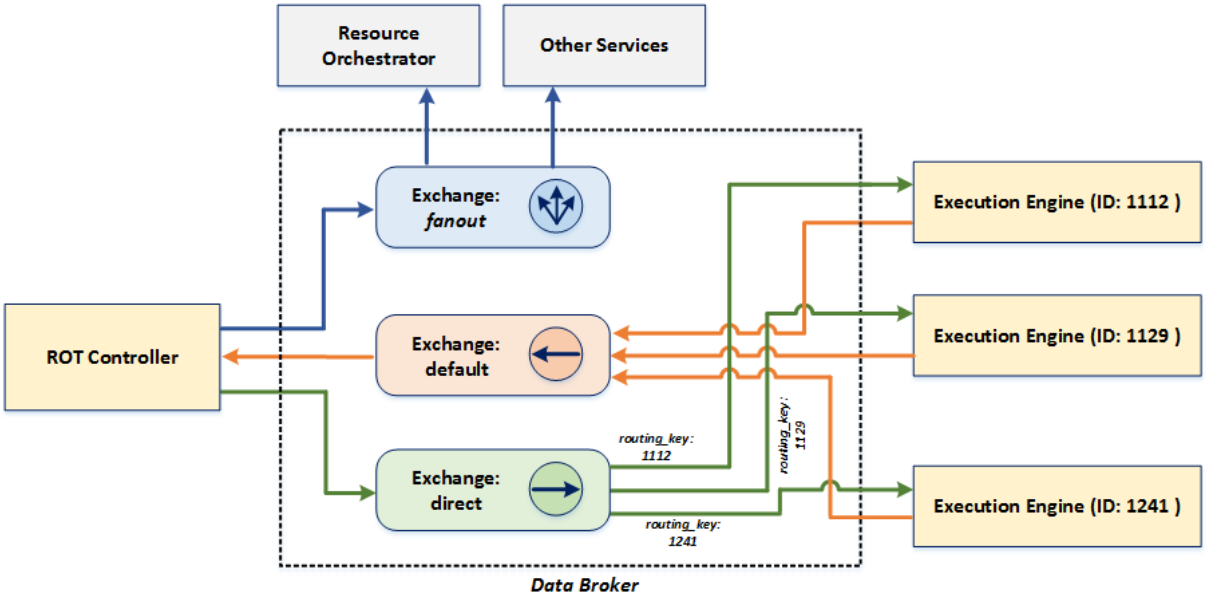


Figure 32: ROT asynchronous communication over SERRANO Message Broker

6.2 Stream Handler

SERRANO's distributed streaming platform will allow to publish and subscribe to streams of records. This part of the messaging infrastructure will support high throughput and high-velocity data streams through a scalable, fault-tolerant communication-efficient framework. This approach provides the ability for asynchronous communication between SERRANO platform components as well as deployed applications.

The implementation of Stream Handler is based on existing and well-known software platforms which support critical features like the loose coupling of components, increased scalability and security. Figure 33 shows the key building blocks and their interactions with other SERRANO components.

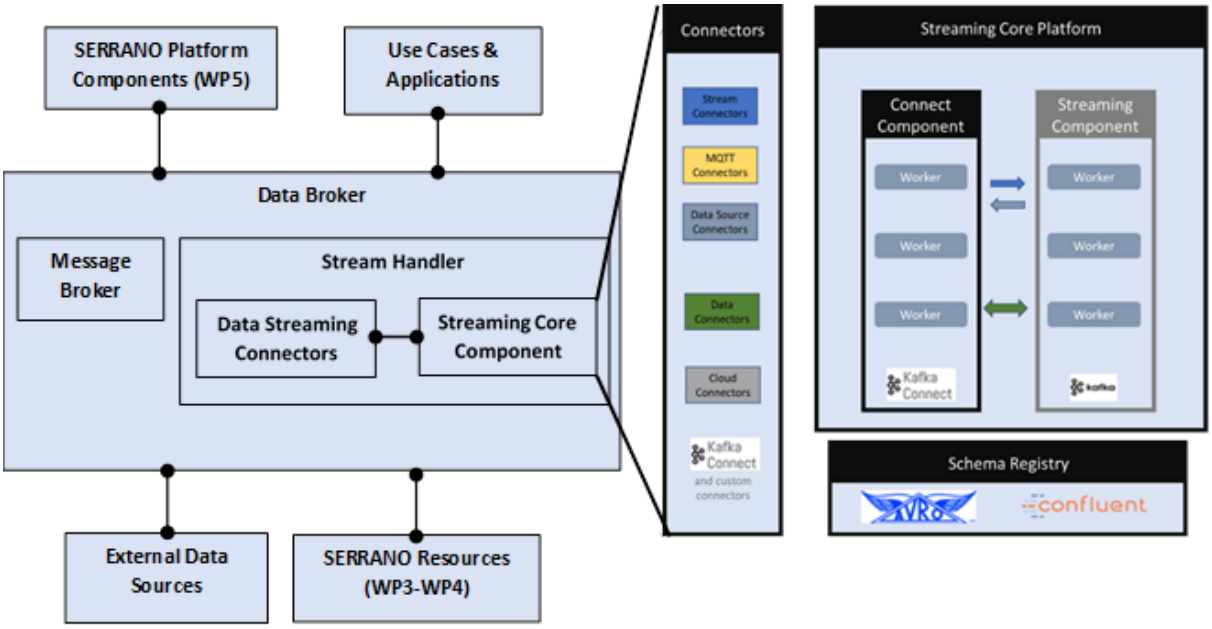


Figure 33: Stream Handler building blocks and interactions

The Streaming Core Platform provides a RESTful interface to the Streaming Component (Figure 34), which is comprised of an Apache Kafka cluster, making it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients.

Some examples use cases are:

- Reporting data from any frontend app built in any language
- Ingesting messages into a stream processing framework that doesn't yet support Kafka
- Automating administrative actions through scripts

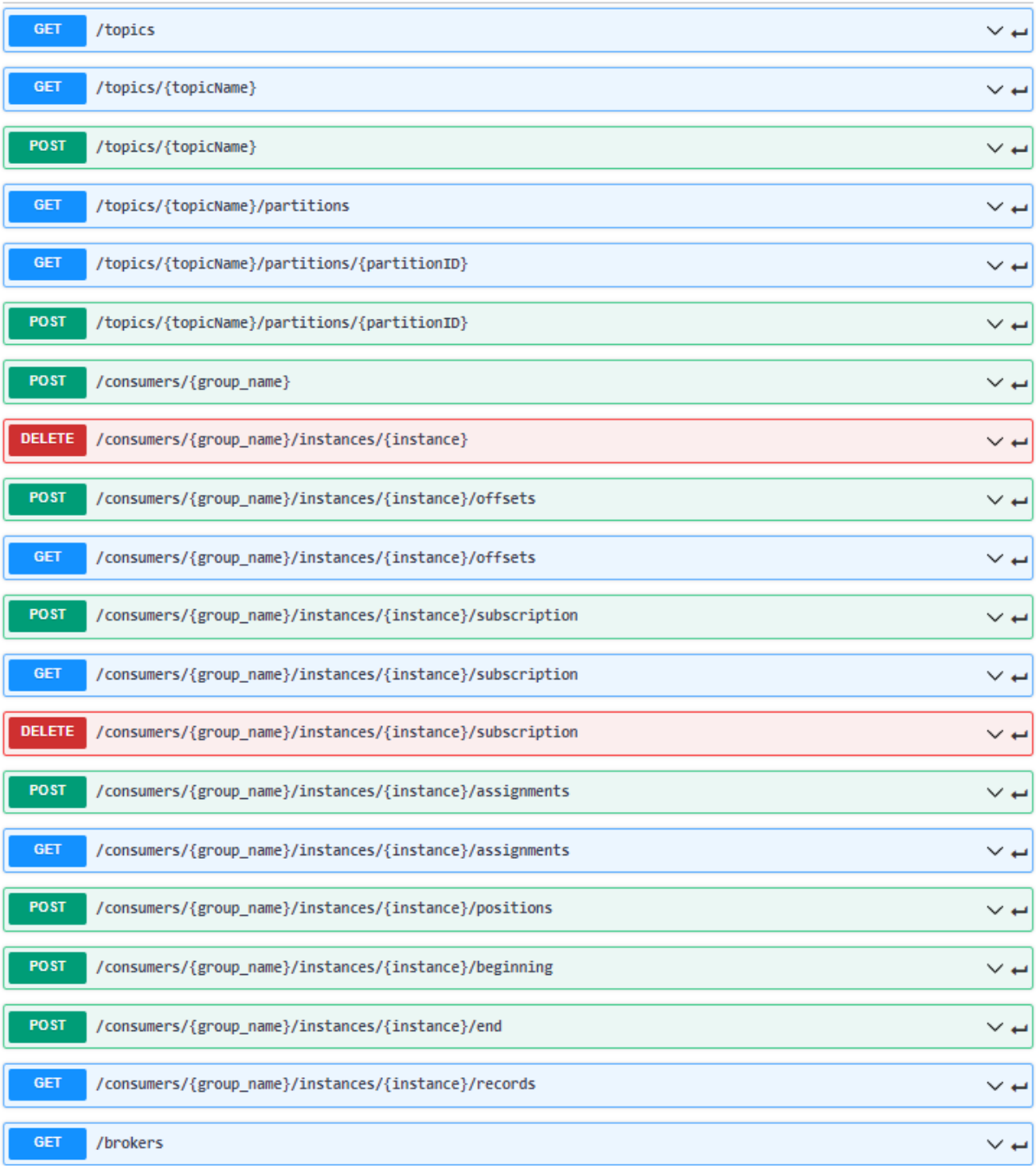


Figure 34: REST Endpoints exposed by Streaming Core Platform

The REST endpoints that appear in Figure 34, use the following schemas (Figure 35), which are also documented in the same document using OpenAPI Spec v3.0.1.

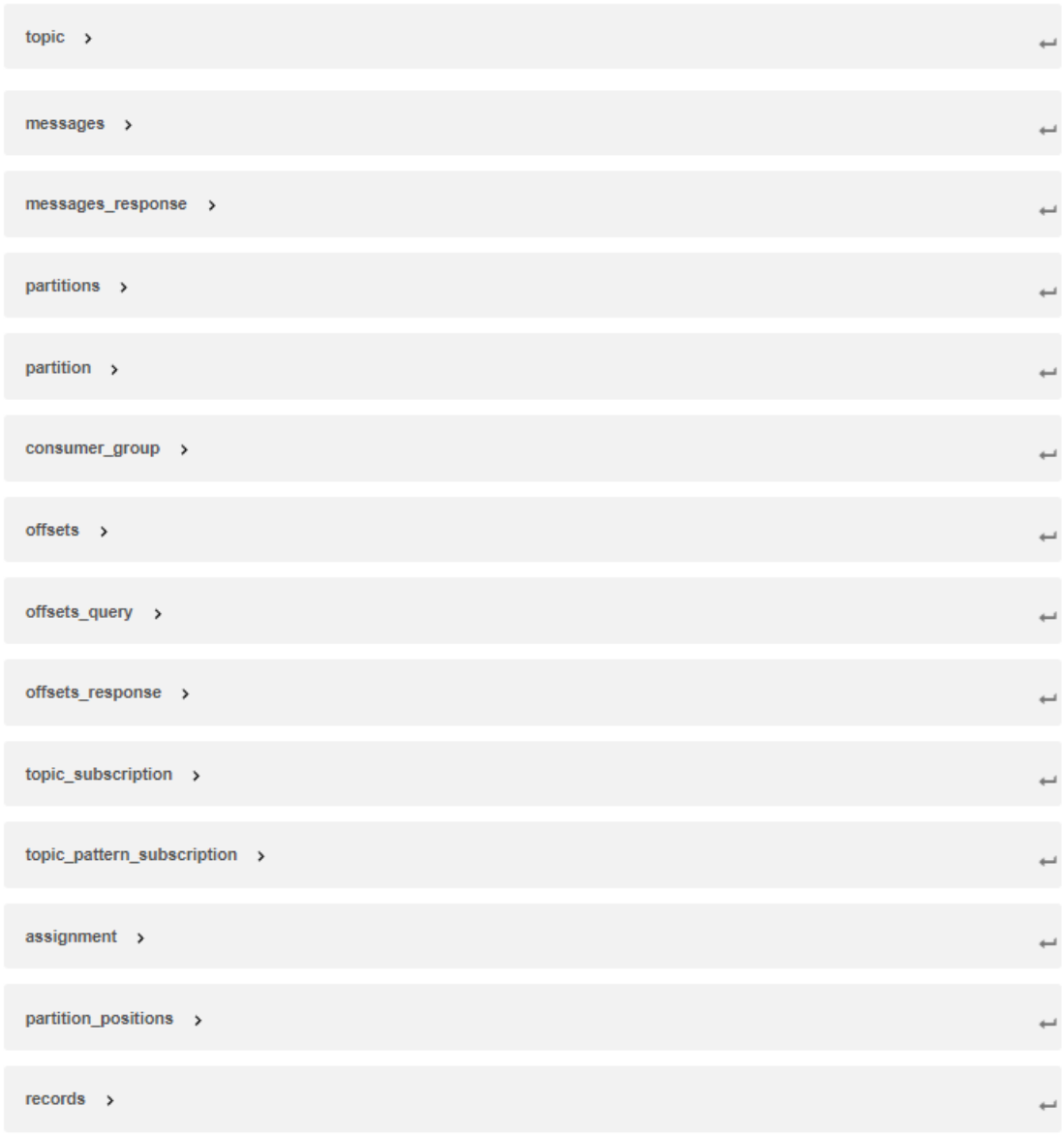


Figure 35: Schemas used in Streaming Core Platform REST API

The Schema Registry provides a RESTful interface for storing and retrieving Avro, JSON and Protobuf schemas (Figure 36). It stores a versioned history of all schemas based on a specified subject name strategy and provides serializers that plug into Apache Kafka clients that handle schema storage and retrieval for Kafka messages that are sent in any of the supported formats.

Schema Registry can be deployed outside of the Streaming Core Platform. Kafka producers and consumers can also talk to Schema Registry to send and retrieve schemas that describe the data models for the messages they public or read from topics.

Figure 36: Schema Registry REST API

The REST endpoints that appear in Figure 36, use the following schemas (Figure 37), which are also documented in the same document using OpenAPI Spec v3.0.1.

Figure 37: Schemas used in Schema Registry API

7 Conclusions

In D5.3 we presented the architecture and the initial implementation activities of two important and related SERRANO components: the SERRANO Network and Cloud Telemetry framework and the SERRANO Resource Orchestrator. The integration and extension of other components and technologies is also discussed (Kubernetes, vAccel, HPC, Prometheus, Netdata, MinIO, virtualization mechanisms). The SERRANO Data Broker component is also presented. In addition, AI/ML based telemetry and anomaly detection methods are discussed and a number of mechanisms and results are presented. The present work will be extended further developing, integrating and evaluating these components and the related algorithms.

8 References

[1] Y. Vardi, “Network tomography: Estimating source-destination traffic intensities from link data,” J. Amer. Statist. Assoc. 91 365–377, 1996.

[2] C. A. Rødbro, P. A. Chou, Ü. Dogan, “Prediction of Bandwidth and Additive Metrics for Large Scale Network Tomography,” Microsoft Research Technical Report MSR-TR-2016-57.

[3] N. G. Duffield, J. Horowitz, F. L. Presti, D. Towsley, “Network delay tomography from end-to-end unicast measurements,” in Springer Thyrrhenian Int. Workshop on Digital Communications, Sept 2001.

[4] V. N. Padmanabhan, L. Qiu, H. J. Wang, “Passive network tomography using bayesian inference,” in Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement, Nov. 2002.

[5] Y. Chen, D. Bindel, R. H. Katz, “Tomography-based overlay network monitoring,” in Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement, Oct. 2003.

[6] O. Gurewitz, I. Cidon, M. Sidi, “One-way delay estimation using network-wide measurements,” IEEE Transactions on Information Theory, 52(6), 2710-2724, 2006.

[7] L. Ma, T. He, K. K. Leung, D. Towsley, A. Swami, “Efficient identification of additive link metrics via network tomography,”
in IEEE 33rd International Conference on Distributed Computing Systems, July 2013.

[8] L. Ma, T. He, K. K. Leung, A. Swami, D. Towsley, “Inferring link metrics from end-to-end path measurements: Identifiability and monitor placement,” IEEE/ACM Transactions on Networking, 22(4), 1351-1368, 2014.

[9] R. Castro, M. Coates, G. Liang, R. Nowak, B. Yu., “Network Tomography: Recent Developments,” Statistical Science, 19(3), 499-517, 2004.

[10] G. Kakkavas, D. Gkatzioura, V. Karyotis, S. Papavassiliou, “A review of advanced algebraic approaches enabling network tomography for future network infrastructures,” Future Internet, 12(2), 20, 2020.

[11] L. Ma, T. He, A. Swami, D. Towsley, K. K. Leung, “Network Capability in Localizing Node Failures via End-to-End Path Measurements,” in IEEE/ACM Transactions on Networking, 25(1), 434-450, Feb. 2017.

[12] N. Bartolini, T. He, H. Khamfroush, “Fundamental limits of failure identifiability by boolean network tomography,” in IEEE INFOCOM May 2017.

[13] R. Tagyo, D. Ikegami, R. Kawahara, “Network Tomography using Routing Probability for Undeterministic Routing,” IEICE Transactions on Communications, 2020EBP3149, 2021.

[14] M. Rahali, J. Sanner, G. Rubino, “TOM: a self-trained Tomography solution for Overlay networks Monitoring,” in IEEE Consumer Communications & Networking Conference (CCNC), Jan. 2020.

[15] L. Ma, Z. Zhang, M. Srivatsa, “Neural network tomography,” arXiv preprint arXiv:2001.02942, 2020.

[16] A. Ibraheem, Z. Sheng, G. Parisis, D. Tian, “Neural Network based Partial Tomography for In-Vehicle Network Monitoring,” in IEEE International Conference on Communications Workshops, June 2021.

[17] A. Rkhami, Y. Hadjadj-Aoul, G. Rubino, A. Outtagarts, “On the use of machine learning and network tomography for network slices monitoring,” in IEEE 22nd International Conference on High Performance Switching and Routing (HPSR), Jun. 2021.

[18] D. B. Chua, E. D. Kolaczyk, M. Crovella, “Network kriging,” IEEE J. Select. Areas Commun., 24, (12), 2263–2272, Dec. 2006.

[19] D. M. Hawkins, “Identification of Outliers,” Springer 1980.

[20] K. G. Mehrotra, C. K. Mohan, H. Huang, “Anomaly Detection Principles and Algorithms,” Springer, 2017.

[21] M. Goldstein, S. Uchida, “A Comparative Evaluation of Unsupervised Anomaly Detection Algorithms for Multivariate Data,” PLoS ONE 11(4), 2016.

[22] N. Moustafa, J. Slay, “UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set),” Military Communications and Information Systems Conference (MilCIS), 2015.

[23] Y. Zhao, Z. Nasrullah, Z. Li, “PyOD: A Python Toolbox for Scalable Outlier Detection,” Journal of machine learning research (JMLR), 20(96), 2019.

[24] Flask 2.0: https://flask.palletsprojects.com/en/2.0.x/

[25] Pika: https://pika.readthedocs.io/en/stable/
[26] PyQt: https://riverbankcomputing.com/software/pyqt/intro
[27] MongoDB: https://docs.mongodb.com
[28] InfluxDB: Open Source Time Series Database: https://www.influxdata.com/developers/
[29] Grafana -The open observability platform: https://grafana.com/oss/grafana/
[30] gRPC- high performance, open source universal RPC framework: https://grpc.io
[31] Netdata: https://learn.netdata.cloud/docs/get-started
[32] kube-state-metrics: https://github.com/kubernetes/kube-state-metrics
[33] metrics-server: https://github.com/kubernetes-sigs/metrics-server
[34] Minio - High Performance Object Storage: https://min.io
[35] Open Container Initiative: https://opencontainers.org/
[36] Kata Containers, an open source container runtime: https://katacontainers.io
[37] 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
[38] J. Thönes. Microservices. IEEE Software, 32(1):116–116, Jan 2015.
[39] 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
[40] Cloud Native Computing Foundation, “Frequently Asked Questions” . https://www.cncf.io/about/faq/. Accessed: 2019-06-10.
[41] 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
[42] AWS Lambda. https://aws.amazon.com/lambda. Accessed: 2022-02-01
[43] Firecracker: Lightweight Virtualization for Serverless Computing. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing
[44] 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
[45] http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html
[46] 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
[47] Modern Hypervisor for the Cloud. https://github.com/intel/nemu
[48] crosvm VMM: https://google.github.io/crosvm/
[49] 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
[50] The Solo5 Unikernel: https://github.com/solo5/solo5
[51] 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
[52] 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
[53] Open source message broker: https://www.rabbitmq.com
[54] AMQP 0-9-1 Model Explained: https://www.rabbitmq.com/tutorials/amqp-concepts.html
[55] Slurm workload manager: https://slurm.schedmd.com/
[56] TORQUE resource manager: https://adaptivecomputing.com/cherry-services/torque-resource-manager/
[57] OpenPBS: https://www.openpbs.org/
[58] etcd: https://etcd.io