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

Grant Agreement no. 101017168

Deliverable D6.3
The SERRANO Integrated Platform

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:	WP6
Responsible Editor:	INTRA
Due date:	30/06/2022
Actual submission date:	01/07/2022

Dissemination level:	Public
Revision:	Final

Revision History

Date	Editor	Status	Version	Changes
03.05.2022	INTRA	Draft	0.1	Initial version with Table of Contents
13.05.2022	INTRA	Draft	0.2	Input provided in Section 4.4.2 (Stream Handler)
18.05.2022	INTRA	Draft	0.2	Added Verification and Validation Plan (Section 6.3)
06.06.2022	ICCS	Draft	0.2	Input provided in Sections 4.1, 4.2
06.06.2022	IDEKO	Draft	0.2	Input provided in Section 4.12.x (UC3)
07.06.2022	UVT	Draft	0.2	Input provided in Section 4.9
08.06.2022	ICCS	Draft	0.2	Input provided in Sections 4.3, 4.4.1
08.06.2022	INNOV	Draft	0.2	Input provided in Section 4.5
14.06.2022	INTRA	Draft	0.3	Consolidated version 0.3
15.06.2022	AUTH	Draft	0.3	Input provided in Section 4.7
16.06.2022	UVT	Draft	0.3	Input provided in Section 6.3
16.06.2022	CC	Draft	0.3	Input provided in Section 6.3
16.06.2022	IDEKO	Draft	0.3	Input provided in Section 4.12
17.06.2022	HLRS	Draft	0.3	Input provided in Sections 4.6, 6.3
17.06.2022	INB	Draft	0.3	Input provided in Section 4.11
18.06.2022	ICCS	Draft	0.3	Input provided in Sections 4.1.3.1, 4.2, 4.3, 4.4.1, 6.3
20.06.2022	INTRA	Draft	0.4	Input provided in Sections 5.1, 5.2, 6.1, 6.2. Consolidated version 0.4
21.06.2022	NBFC	Draft	0.4	Input provided in Section 4.7
21.06.2022	INNOV	Draft	0.4	Input provided in Section 4.5
21.06.2022	INTRA	Draft	0.5	Consolidated version 0.5
21.06.2022	NBFC	Draft	0.5	Input in Sections 4.7.1 and 4.7.2
22.06.2022	IDEKO	Draft	0.5	Updates and input in Sections 4.12.2 and 6.3
22.06.2022	AUTH	Draft	0.5	Input in Section 4.1.10.2
22.06.2022	HLRS	Draft	0.5	Updates and input in Sections 4.1.3, 4.3.3, 4.6.1 and 6.3
23.06.2022	INTRA	Draft	0.6	Consolidated version 0.6
23.06.2022	ICCS	Draft	0.6	Input in Section 3
24.06.2022	MLNX	Draft	0.6	Input in Sections 4.8.2, 4.10.1.2 and 6.3
24.06.2022	INTRA	Draft	0.6	Input in Sections 1, 2, 5, 6 and 7
25.06.2022	INTRA	Draft	0.7	Pre-final version for internal review
30.06.2022	INTRA	Final	1.0	Final Version for submission

Author List

Organization	Author
INTRA	Makis Karadimas, Paraskevas Bourgos
CC	Marton Sipos
ICCS	Aristotelis Kretsis, Panagiotis Kokkinos, Polyzois Soumplis, Emmanouel Varvarigos
IDEKO	Javier Martin
INNOV	Efthymios Chondrogiannis
UVT	Iuhasz Gabriel
AUTH	Kostas Siozios, Dimosthenis Masouros, Dimitris Danopoulos, Argyris Kokkinis, Aggelos Ferikoglou, Ioannis Oroutzoglou
HLRS	Kamil Tokmakov
INB	Maria Oikonomidou, Ferad Zyulkyarov
NBFC	Anastasios Nanos, Charalampos Mainas, George Ntoutsos
MLNX	J.J. Vegas Olmos, Yoray Zack

Internal Reviewers

Andreas Litke, INNOV

Ferad Zyulkyarov, INB

Abstract: This deliverable (D6.3) presents the outcomes of Task 6.1 – "Integration, Verification and Testing" during the first 6 months (M13-M18) of WP6, which aims at unifying the outcomes of the developed components and services in WP3-5 to release the integrated SERRANO platform. The deliverable presents an overview of the SERRANO platform, including the initial release status, the SERRANO platform components and functionalities, the development and integration environment, the software deployment specifications and the verification and validation results on the platform components.

Keywords: SERRANO platform, integrated components, development environment, integration environment, software deployment, verification, validation

Table of Contents

List of Figures

List of Tables

Abbreviations

ACL	Access Control Lost
AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
API	Abstract Programming Interface
ARM	Advanced RISC Machines
ASGI	Asynchronous Server Gateway Interface
AWS	Amazon Web Services
CI/CD	Continuous Integration / Continuous Development
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
CSV	Comma-Separated Values
CUDA	Compute Unified Device Architecture
D	Deliverable
DAST	Dynamic Application Security Testing
DBScan	Density-Based Apatial clustering of applications with noise
DevSecOps	Development, Security, and Operations
DL	Deep Learning
DMM	Digital MultiMeter
DoW	Description of Work
DPO	Dynamic Portfolio Optimization
DPU	Data Processing Unit
DTW	Dynamic Time Warping
EC	European Commission
EDE	Event Detection Engine
EFT	Electronic Funds Transfer
ETL	Extract, Transform, Load
FPGA	Field-Programmable Gate Array
FTT	Fast Fourier transform
GB	GigaByte
GDPR	General Data Protection Regulation
GEMM	GEneral Matrix to Matrix Multiplication
GPS	Global Positioning System
GPU	Graphics Processing Unit
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HW	Hardware
IaC	Infrastructure as Code
ID	IDentification
IDE	Integrated Development Environment
IO	Input Output
IoT	Internet of Things
IP	Internet Protocol

IPsec	Internet Protocol Security
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
K8S	Kubernetes
KNN	K-Nearest Neighbors algorithm
ML	Machine Learning
MPSoC	MultiProcessor System on a Chip
MQTT	MQ (IBM MQ) Telemetry Transport
NBI	North Bound Interfaces
NIC	Network Interface Controller
OS	Operating System
OWASP	Open Web Application Security Project
PBS	Portable Batch System
PCIe	Peripheral Component Interconnect express
PM	Project Manager
PMDS	Persistent Monitoring Data Storage
PO	Project Officer
PyPI	Python Package Index
RDBMS	Relational DataBase Management System
REST	Representational State Transfer
RHEL	Red Hat Enterprise Linux
ROT	Resource Orchestration Toolkit
SAST	Static Application Security Testing
SCA	Source Composition Analysis
SDLC	Software Development Life Cycle
UI	User Interface
YAML	YAML Ain't Markup Language

1 Executive Summary

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

Deliverable 6.3 reports on the work performed in WP6, for developing and providing the SERRANO integrated platform. The WP6 activities related to D6.3 aim at unifying the outcomes of the developed components and services in WP3-5 to release the integrated SERRANO platform.

The deliverable presents an overview of the SERRANO platform, including the initial release status, the SERRANO platform components and functionalities, the development and integration environment, the software deployment specifications and the verification and validation results on the platform components.

The information provided in the present deliverable is expected (a) to support the preliminary evaluation of the use cases, reported at deliverable D6.4 “Business, end user and technical evaluation” (M20), which will provide critical feedback for the second development iteration, (b) to be used as a basis to develop the SERRANO full platform prototype that will include the remaining functionality, which has not been included in the early prototype, and will be provided in M31, and (c) to lead towards the final release of the SERRANO platform that will be integrated and documented as part of deliverables D6.7 “Final version of SERRANO integrated platform” (M36) and D6.8 “Final version of business, end user and technical evaluation” (M36).

2 Introduction

2.1 Purpose of this document

The present deliverable (D6.3) presents the outcomes of Task 6.1 – "Integration, Verification and Testing" during the first 6 months (M13-M18) of WP6, which aims at unifying the outcomes of the developed components and services in WP3-5 to release the integrated SERRANO platform.

The objective of this deliverable is to include the first release of the integrated platform and to report on the outcomes of the initial tested functionalities and interfaces. This document provides a basic functional prototype that will support the core functionalities of the three project use cases. The initial platform prototype will support the preliminary evaluation of the use cases, reported at deliverable D6.4 "Business, end user and technical evaluation" (M20), which will provide critical feedback for the second development iteration.

The initial release of the SERRANO platform includes the components developed during the first iteration (M1-M18) of the project implementation plan, along with a partial integration of them. In this version, each component implements a subset of the envisioned features along with the primary interfaces for inter-component communication. The initial release aims to provide a basic functional prototype that will support the core functionalities of the envisioned SERRANO platform.

The SERRANO full platform prototype will be based on the initial release reported in the current deliverable, and will include the remaining functionality, which has not been included in the early prototype, and will be provided in M31.

The final release of the SERRANO platform will be fully integrated and documented as part of deliverables D6.7 "Final version of SERRANO integrated platform" (M36) and will be delivered at the end of the project. In the final release, the consortium will focus on implementing the feedback from the final evaluation of SERRANO platform through the demonstration of the three project use cases.

2.2 Document structure

The present deliverable is split into seven main sections:

- Executive Summary
- Introduction
- Overview of SERRANO Platform
- SERRANO Platform Components and Functionalities
- Development and Integration Environment
- Software Deployment Specifications and Validation
- Conclusions

2.3 Audience

The deliverable is public and available to anyone interested in the first release of the SERRANO integrated platform unifying the outcomes of the developed components and services. Moreover, this document can also be useful to the general public for obtaining a better understanding of the framework and scope of the SERRANO project.

3 Overview of SERRANO Platform

3.1 SERRANO Architecture

The architecture of SERRANO has been initially presented in deliverable D2.3 “SERRANO architecture” (M09) and updated in its final version in the context of D2.5 “Final version of SERRANO architecture” (M18) based on the feedback from the development activities during the first iteration of the implementation (M1-M18). D2.5 (M18) provides a more comprehensive description of the overall architecture, the SERRANO components, their interfaces and supported workflows. In this section, we provide a short description of the architecture (Figure 1) to facilitate the presentation of the initial version of the SERRANO integrated platform.

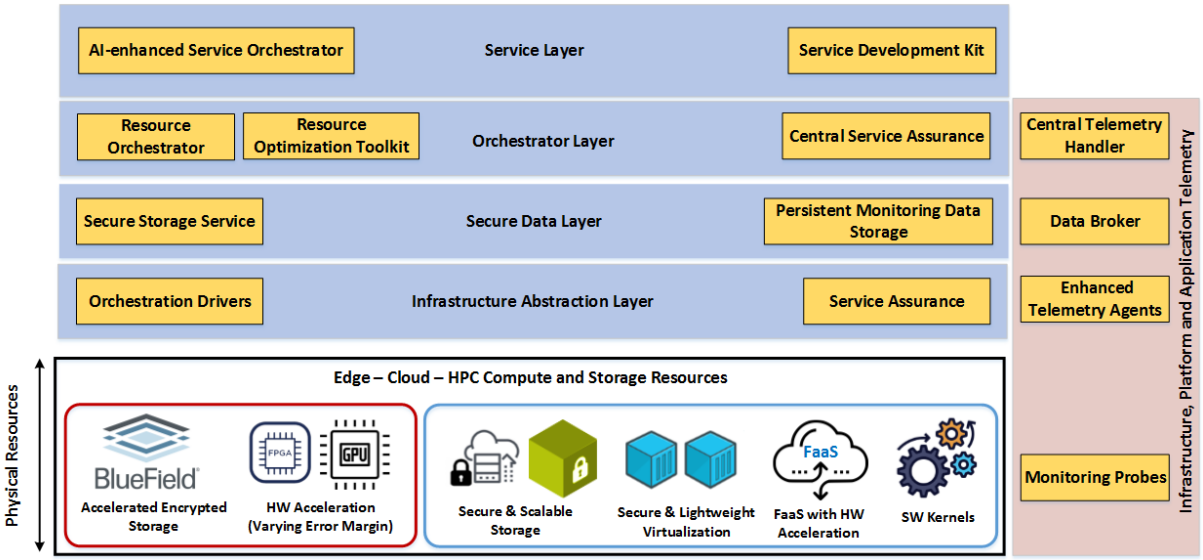


Figure 1: SERRANO high-level architecture

The Resource Layer includes heterogeneous edge, cloud and HPC computational and storage resources encompassing the SERRANO-enhanced resources (Sections 4.7, 4.8.2). Across the SERRANO ecosystem resides the Infrastructure, Platform and Application Telemetry stack (Section 4.3) that collects metrics across the infrastructure and deployed applications. The main components are the Central *Telemetry Handler*, the *Enhanced Telemetry Agents* and *Monitoring Probes*. In addition, the *Data Broker* (Section 4.4) provides the required asynchronous inter-component communication within the SERRANO platform and connects external data sources, making them available to internal services.

The Service Layer contains the *AI-enhanced Service Orchestrator* (Section 4.5) that analyses applications to determine the possible deployment scenarios and translates the given application requirements (high-level requirements) to lower-level ones. The Orchestration Layer ensures efficient service orchestration and resource management through the SERRANO *Resource Orchestrator* (Section 4.1). The *Resource Optimization Toolkit* (Section 4.2) provides

joint computational and storage resource allocation and service placement algorithms, leveraging optimization and AI/MI techniques. The *Central Service Assurance* manages the runtime lifecycle of each application deployment across the SERRANO heterogeneous infrastructure. It receives notifications from the *Service Assurance and Remediation* mechanisms (Section 4.9) at the infrastructure level and triggers proactively and reactively re-optimization actions to maintain the required performance level.

The Secure Data Layer includes the *Secure Storage Service (Section 4.8.1)* that abstracts the required actions for edge and cloud storage resources, operating as a security access broker that guarantees and enforces privacy and security requirements on data. The *Monitoring Data Storage* allows the management of the historical monitoring data, required mainly by the service assurance and remediation system. Finally, the Infrastructure Abstraction Layer facilitates the integration of new hardware and software platforms within the SERRANO platform. The *Orchestration Drivers* (Section 4.1) enable efficient and transparent deployment of services across the heterogeneous infrastructure. The *Service Assurance* (Section 4.9) includes data-driven mechanisms that facilitate the identification of critical situations and activate self-driven adaptations.

In Section 4, the components of these main entities and their subcomponents are listed, explaining the provided functionalities in this first release along with their integration, as part of the platform.

3.2 Initial Release Status

The initial release of the SERRANO platform includes the components developed during the first iteration (M1-M18) of the project implementation plan, along with their partial integration. In this version, each component implements a subset of the envisioned features along with the primary interfaces for inter-component communication. The initial release aims to provide a basic functional prototype that will support the core functionalities of the envisioned SERRANO platform.

To this end, the focus was to support functionalities such as the application description and initial deployment over edge, cloud, and HPC resources. Also, besides the basic deployment functionality, the platform's initial release supports the collection of telemetry data and the provision of asynchronous communication between the SERRANO platform components. Furthermore, the secure and trusted execution on diverse and heterogeneous infrastructure is supported through the integrated, trusted virtualization mechanisms, along with the provision of hardware acceleration abstractions. Moreover, the core components of the Secure Storage Service are successfully integrated. Finally, in the early release, we provide the core functionality of the Service Assurance service within the SERRANO platform through the integration of the Event Detection Engine.

According to the adopted implementation plan, the complete prototype will be provided in M31. It will be based on the initial release and provide the remaining functionality, which has

not been included in this version. The final release of the SERRANO platform will be delivered at the end of the project.

Finally, the following table summarizes the integration status of the various components and interfaces provided by the SERRANO platform components.

Table 1: Integration status of SERRANO interfaces

Name	Involved Components	Status
WP5T1AISO-I: AI-enhanced Service Orchestrator	AI-enhanced Service Orchestrator, Resource Orchestrator, Central Telemetry Handler	Initial version is integrated, improvements are planned for the next releases
WP5T5RO-I: Resource Orchestrator	Resource Orchestrator, AI-enhanced Service Orchestrator, Service Assurance	Initial version is integrated, improvements are planned for the next releases
WP5T5OD-I: Orchestration Drivers	Orchestration Driver, Resource Orchestrator, K8s, HPC Gateway	Initial version is integrated, improvements are planned for the next releases
WP5T2ROT-I: Resource Optimization Toolkit	Resource Optimization Toolkit, Resource Orchestrator	Interface is completed and integrated
WP5T4EMT-I: Energy & Resource Aware Mapping Interface	SERRANO HPC Gateway, Resource Orchestrator	Planned for the next release (full SERRANO platform)
WP4T2HPC-I: Uncertainties Estimation Interface	SERRANO HPC Gateway, Resource Orchestrator	Planned for the next release (full SERRANO platform)
WP3T2DSS-I: Secure Storage API	On-premises Storage Gateway, SERRANO applications and services	Initial version is integrated, improvements are planned for the next releases
WP3T2DSSSLT-I: Storage location telemetry API	On-premises Storage Gateway, Central Telemetry Handler	Interface is completed and integrated
WP5T3CTH-I: Central Telemetry Handler Interface	Central Telemetry Handler, AI-enhanced Service Orchestrator, Resource Optimization Toolkit, Service Assurance	Initial version is integrated, improvements are planned for the next releases
WP5T3ETA-I: Enhanced Telemetry Agent Interface	Enhanced Telemetry Agent, Central Telemetry Handler, Orchestration Drivers, Service, Monitoring Probes	Initial version is integrated, improvements are planned for the next releases
WP5T5PMDS-I: Persistent Monitoring Data Storage Interface	Enhanced Telemetry Agent, AI-enhanced Service Orchestrator, Service Assurance and Remediation	Planned for the next release (full SERRANO platform)
WP5T5MB-I: Message Broker Interface	SERRANO platform components, use cases and applications, external data sources	Initial version is integrated, improvements are planned for the next releases

WP5T5SC-I: Streaming Core Interface	SERRANO platform components, use cases and applications, external data sources	Initial version is integrated, improvements are planned for the next releases
WP5T5SAR-I: Service Assurance Interface	Service Assurance and Remediation, Resource Orchestrator, use cases and applications	Initial version is integrated, improvements are planned for the next releases
WP4T3PC-I: Plug&Chip Interface	SERRANO Plug&Chip framework, use cases and applications	Initial version is integrated, improvements are planned for the next releases
WP4T1HWRT-I: Hardware Accelerators Interface	Local Orchestrators, SERRANO hardware accelerated kernels	Initial version is integrated, improvements are planned for the next releases
WP4T2HPC-I: HPC Services Interface	HPC Gateway, Resource Orchestrator, Orchestration Drivers	Initial version is integrated, improvements are planned for the next releases
WP5T5TLV-I: Trusted and Lightweight Virtualization Interface	Orchestration Drivers, Local Orchestrators, use cases and applications	Initial version is integrated, improvements are planned for the next releases

4 SERRANO Platform Components and Functionalities

4.1 Resource Orchestrator

4.1.1 Description

SERRANO adopts a two-stage approach to assign the applications to the available resources cognitively. The Resource Orchestrator implements the initial stage, coordinates the necessary supplemental actions (e.g., transfer required data), and initiates the second state that includes the actual application deployment.

The Resource Orchestrator assigns the application's microservices to the most appropriate individual platform and also forms the appropriate low-level infrastructure-specific deployment objectives for the Local Orchestrators. This process 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.

Figure 2 depicts the architecture and the main components of the Resource Orchestrator.

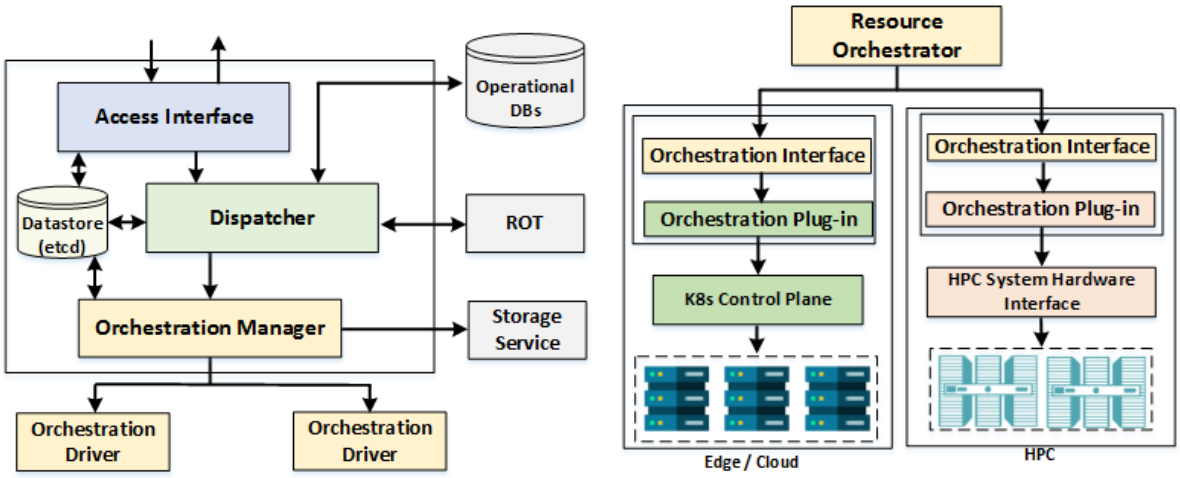


Figure 2: Resource Orchestrator architecture and main components

4.1.2 Inner components

The *Access Interface* exposes the necessary REST interface (Figure 3) to support the required interactions, enabling the exchange of requests, information, and notifications. The *Dispatcher* 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.

The *Datastore* includes configuration and state data for the available platforms and deployed applications. It is based on etcd [1], an open-source distributed key-value store. The Resource Orchestrator components leverage the etcd's "watch" function to keep track of the actual and desired state of the deployed workloads across the overall unified infrastructure. The *Orchestration Manager* prepares the necessary application deployment instructions (declarative approach) based on the ROT's decisions and triggers the application deployment at the selected edge/cloud/HPC platforms.

The *Orchestration Drivers* complete the implementation of the hierarchical resource orchestration. The *Orchestration Interface* provides an infrastructure-agnostic interface with the Local Orchestrators. 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.

4.1.3 Integration details and REST APIs

4.1.3.1 Integration Details

The exposed REST APIs (Figure 3) facilitate all the supported interactions with the Resource Orchestrator either by its internal or other SERRANO components (e.g., AI-enhanced Service Orchestrator).

Table 2: Integration details of Resource Orchestrator

IP(s)/Port(s)	Resource Orchestrator: 147.102.22.140:10100 ROT Controller: 34.90.62.43:10020 Central Telemetry Handler: 147.102.16.113:9090 Orchestration Drivers: • 147.102.16.113:10100 • 34.90.1.58:10100 • hpc-interface.serrano.cs.uvt.ro:8080
Publicly accessible (y/n and other details)	The IPs are publicly accessible, but the access has been restricted though authentication.
Type of API	REST
Associated host names	N/A
API documentation	https://gitlab.com/serranoproject/wp5/resource-orchestrator/-/raw/main/orchestrator_rest.yaml
Location of integration tests	https://gitlab.com/serranoproject/wp5/resource-orchestrator/-/raw/main/Jenkinsfile

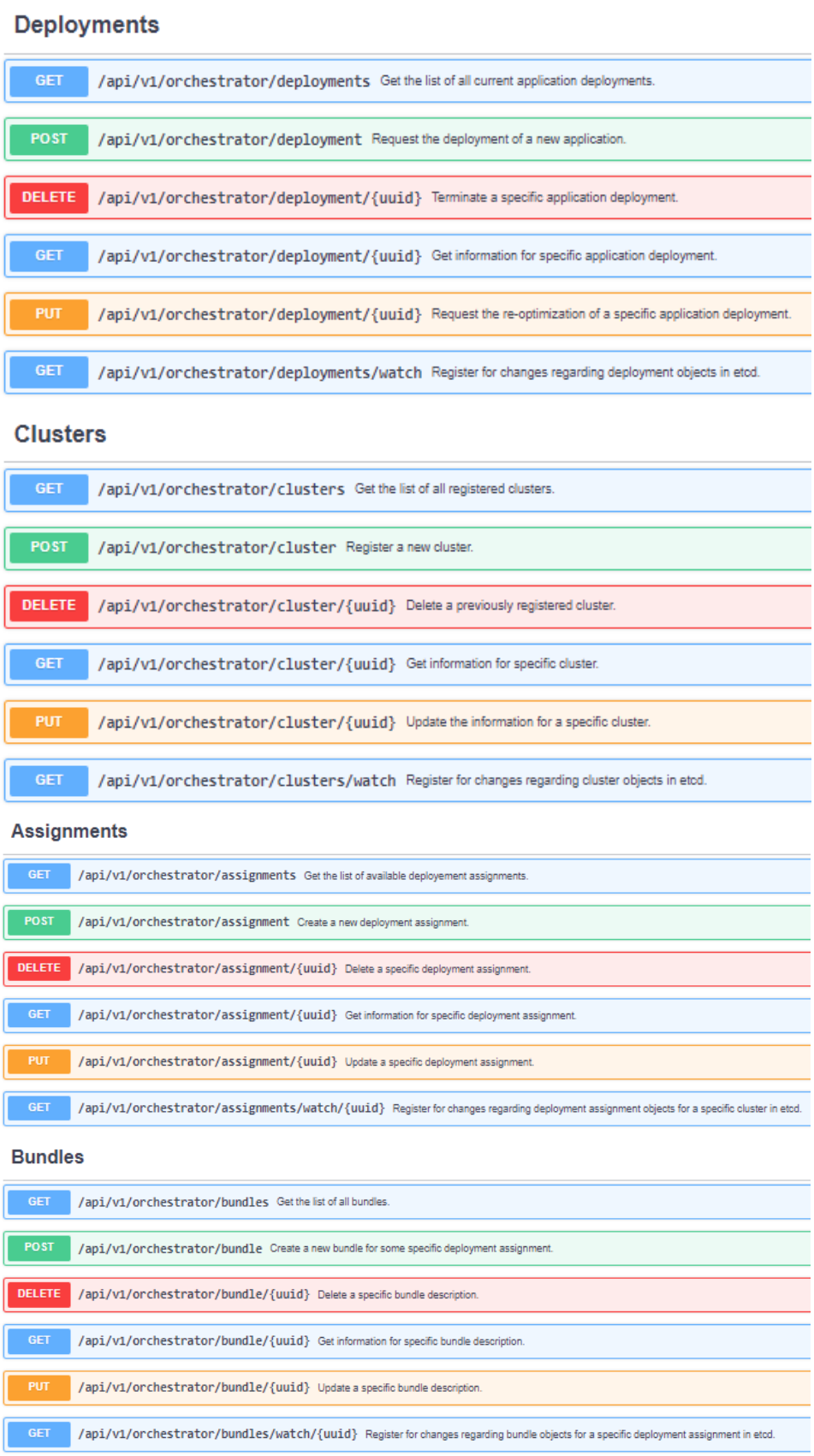


Figure 3: REST Endpoints exposed by Resource Orchestrator (through the Access Interface)

The methods in the above interface are organized based on four entities that facilitate the lifecycle management of deployments by the Resource Orchestrator. These entities also correspond to how the relevant information is organized in the Datastore. The following table provides an overview description.

Cluster	/serrano/orchestrator/clusters/cluster/CLUSTER_UUID
Deployment	/serrano/orchestrator/deployments/deployment/DEPLOYMENT_UUID
Assignment	/serrano/orchestrator/assignments/CLUSTER_UUID/assignment/ASSIGNMENT_UUID
Bundle	/serrano/orchestrator/bundles/bundle/BUNDLE_UUID

Figure 4 presents a detailed workflow for deploying an application within the SERRANO platform, highlighting the roles of the Resource Orchestrator components and their interaction with other components within the SERRANO platform.

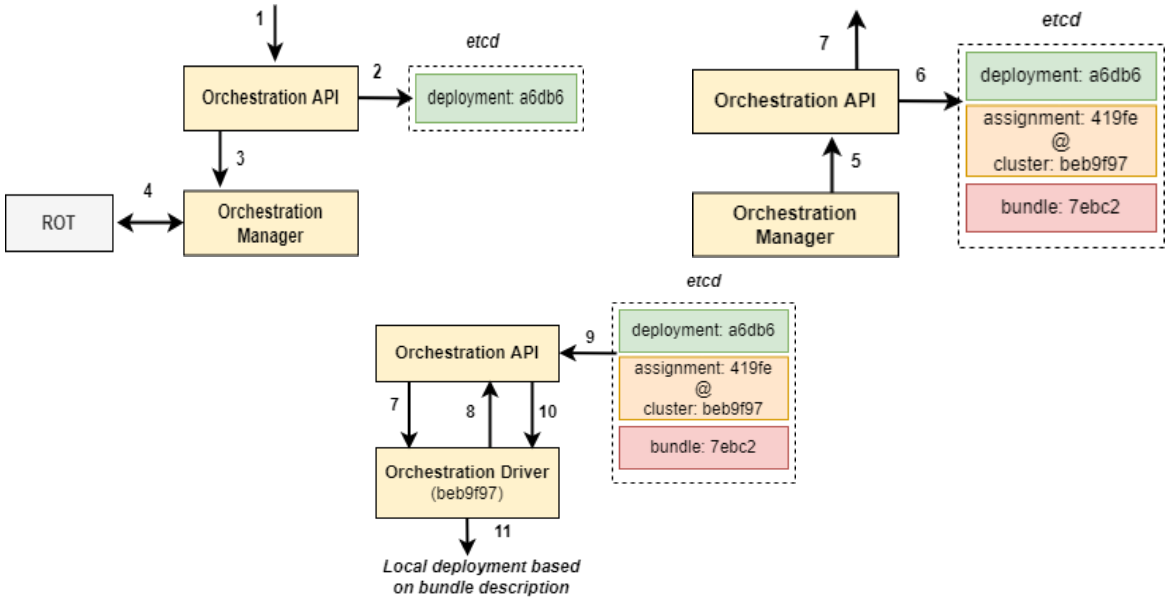


Figure 4: Integration workflow of Resource Orchestrator, Orchestration Drivers and Local Orchestrators

4.1.3.2 Sample requests and responses

Resource Orchestrator and Orchestration Drivers

During their initialization procedure, the Orchestrator Drivers registered to the Resource Orchestrator. The operation is performed through the available POST method in the Resource Orchestrator REST API. A successful registration also leads to appropriate notification of the Orchestration Manager.

```
POST /api/v1/orchestrator/cluster

Parameters:

{'cluster_uuid': '6a1a14fe-d623-4398-ba34-9965f8c25a29', 'type': 'k8s', 'nodes': [
{'name': 'snf-883643', 'public_ip': '83.212.102.89', 'labels': {'beta.kubernetes.i
o/arch': 'amd64', 'beta.kubernetes.io/os': 'linux', 'foo': 'bar', 'kubernetes.io/a
```

```
rch': 'amd64', 'kubernetes.io/hostname': 'snf-883643', 'kubernetes.io/os': 'linux'
, 'node-role.kubernetes.io/worker': 'worker', 'nodeMarker': 'snf-883643'}}, {'name
': 'snf-883644', 'public_ip': '83.212.98.176', 'labels': {'beta.kubernetes.io/arch
': 'amd64', 'beta.kubernetes.io/os': 'linux', 'foo': 'bar', 'kubernetes.io/arch':
'amd64', 'kubernetes.io/hostname': 'snf-883644', 'kubernetes.io/os': 'linux', 'nod
e-role.kubernetes.io/worker': 'worker'}}, {'name': 'telis', 'public_ip': '147.102.
16.113', 'labels': {'beta.kubernetes.io/arch': 'amd64', 'beta.kubernetes.io/os': '
linux', 'foo': 'bar', 'kubernetes.io/arch': 'amd64', 'kubernetes.io/hostname': 'te
lis', 'kubernetes.io/os': 'linux', 'node-role.kubernetes.io/control-plane': '', 'n
ode-role.kubernetes.io/master': '', 'node.kubernetes.io/exclude-from-external-load
-balancers': '', 'role': 'admin'}}]}
```


AI-enhanced Service Orchestrator and Resource Orchestrator

According to the SERRANO architecture, the users submit their application descriptions to the AI-enhanced Service Orchestrator that translates the high-level requirements to the appropriate resource constraints and produces the possible deployment scenarios. Then, it provides the scenarios along with the enhanced application descriptions to the Resource Orchestrator. Section 4.5 provides more information on this component integration within the SERRANO platform. The AI-enhanced Service Orchestrator uses the following POST method to submit a deployment request to the Resource Orchestrator.

```
POST /api/v1/orchestrator/deployment
```

After the initial validation of the request, the Dispatcher creates the corresponding Deployment entry in the Datastore through the provided interface. This operation triggers the Orchestration Manager that requests from the Resource Optimization Toolkit (ROT) to provide the high-level application assignment to the available platforms. Below is an example of storing a deployment description in the Datastore.


```
/serrano/orchestrator/deployments/deployment/d7b35e4a-a626-4012-a53b-2dc978dec7c7
Description:
{"deployment_scenarios":[{"accelerator_fpga": null, "node_exec_capability": "Tier_
1", "node_storage_encryption": null, "platform_type": "CLOUD_PROVIDEER"},{ "accele
rator_fpga": null,"node_exec_capability": "Tier_2","platform_type": "CLOUD_PROVIDE
ER"}],"description": {"description":[{"apiVersion":"apps/v1","kind":"Deployment","
metadata":{"name":"productcatalogservice-e650351fa5a3"},"spec":{"template":{"spec"
:{"containers":[{"name":"server","image":"gcr.io/google-samples/microservices-demo
/productcatalogservice:v0.3.7","ports":[{"containerPort":3550}],"resources":{"requ
ests":{"cpu":"100m","memory":"64Mi"},"limits":{"cpu":"200m","memory":"128Mi"}}}]}}
}},{"apiVersion":"apps/v1","kind":"Deployment","metadata":{"name":"cartservice-e65
0351fa5a3"},"spec":{"template":{"spec":{"containers":[{"name":"server","image":"gc
r.io/google-samples/microservices-demo/cartservice:v0.3.7","ports":[{"containerPor
t":7070}],"resources":{"requests":{"cpu":"200m","memory":"64Mi"},"limits":{"cpu":"
300m","memory":"128Mi"}}}]}}}},{"apiVersion":"apps/v1","kind":"Deployment","metada
ta":{"name":"currencyservice-e650351fa5a3"},"spec":{"template":{"spec":{"containe
rs":[{"name":"server","image":"gcr.io/google-samples/microservices-demo/currencyser
vice:v0.3.7","ports":[{"name":"grpc","containerPort":7000}],"resources":{"requests
":{"cpu":"100m","memory":"64Mi"},"limits":{"cpu":"200m","memory":"128Mi"}}}]}}}}]}
```

Resource Orchestrator and Resource Optimization Toolkit

A description of this interaction is available in Section 4.8.1.3.

Resource Orchestrator and Orchestration Drivers

When the Resource Orchestrator receives the ROT decision, the Orchestration Manager prepares the deployment instructions that will then be forwarded to the Orchestration Drivers of the selected platforms. For example, if the ROT assigns the application's workload to two K8s clusters, there will be two assignment objects, each including the deployment instructions for a specific Orchestration Driver. It follows an example for the assignment "e36ce8e5-5097-40ee-b184-6f3d4a9f8b6a" that includes the instructions, through the bundle object with identifier "a8ef266a-6b8f-48dd-98df-4e8edee718ad" that will handle the Orchestration Driver " b3808eb1-8761-4678-8d2e-722f39540990".

```
/serrano/orchestrator/assignments/b3808eb1-8761-4678-8d2e-722f39540990/assignment/
e36ce8e5-5097-40ee-b184-6f3d4a9f8b6a

{"uuid": "e36ce8e5-5097-40ee-b184-6f3d4a9f8b6a", "cluster_uuid": "b3808eb1-8761-46
78-8d2e-722f39540990", "bundle_uuid": "a8ef266a-6b8f-48dd-98df-4e8edee718ad", "sta
tus": 1, "updated_by": "Orchestration.Manager", "logs": [{"timestamp": 1654932292,
"event": "Cluster assignment"}], "created_at": 1654932292, "updated_at": 165493229
2}
```

Orchestration Drivers and K8s

The Orchestration Driver for the specific K8s cluster is registered to watch for assignments by the Resource Orchestrator. To this end, whenever there is an update in its topic is notified to handle the new deployment request. Then. It retrieves the deployment instructions through the following request.

```
GET /api/v1/orchestrator/assignment/e36ce8e5-5097-40ee-b184-6f3d4a9f8b6a
```

Next, it prepares the appropriate deployment request and submits it to the Kubernetes Orchestrator through the provided Python API. The Kubernetes scheduler makes the final assignment of workload to the available worker nodes based on the instructions received from the Orchestration Driver. Finally, the corresponding pods are created in the selected worker nodes.

```
NAME                                          READY   STATUS    RESTARTS   AGE
cartservice-e650351fa5a3-84479c45dc-67229     1/1     Running   0          103s
currencyservice-e650351fa5a3-6c99bbb975-95rhd 1/1     Running   0          103s
redis-cart-7667674fc7-vvrwv                   1/1     Running   0          103s
shippingservice-e650351fa5a3-56b498954c-t6nrk 1/1     Running   0          103s
```

Orchestration Drivers and HPC

If the application's workload is assigned for execution in the HPC, the Orchestration Manager based on the ROT response will create the necessary deployment description and will trigger the intended Orchestration Driver by writing in the Datastore the appropriate assignment description. For example, for an assignment with unique identifier "834684d7-4293-4bd0-ac18-d4e819d0a4a9" that is indented for the Orchestration Driver "0b150fc9-f296-41e6-a44d-839806b97010" there will be the following entry.

```
/serrano/orchestrator/assignments/0b150fc9-f296-41e6-a44d-839806b97010/assignment/
834684d7-4293-4bd0-ac18-d4e819d0a4a9
```

Then, the corresponding Orchestration Driver in the HPC platform is notified and retrieves the deployment instructions. In this case, it interacts with the SERRANO HPC Gateway through the exposed interface. Section 4.6 provides more information for this integration.

4.2 Resource Optimization Toolkit

4.2.1 Description

The Resource Optimization Toolkit (ROT) implements the developed multi-objective optimization and orchestration algorithms. Moreover, it prepares, coordinates, and manages the appropriate algorithm's execution to facilitate the Resource Orchestrator. Figure 5 presents the architecture of the ROT, its main components, and the interactions with other components within the SERRANO architecture. According to the selected design, there is one ROT Controller but multiple workers. Each worker is composed of the Execution Engine and the library of the decision algorithms.

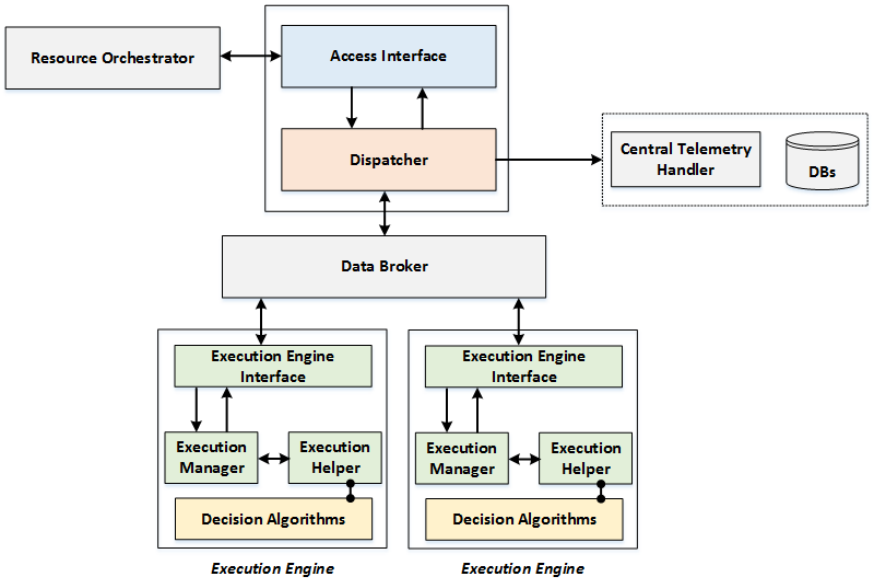


Figure 5: Resource Optimization Toolkit architecture and main components

4.2.2 Inner components

The *Access Interface* exposes the necessary interfaces that allow bidirectional communication for exchanging commands, information and notifications. The *Dispatcher* manages the execution of the requests and handles the interaction with the multiple instances of the Execution Engine. It also interacts with the Central Telemetry Handler and the operational and monitoring databases to retrieve the characteristics of the resources, their current status, and the deployed applications. These are the two components of the ROT controller.

The Execution Engine, through the *Execution Engine Interface, Execution Manager* and *Execution Helper,* receives requests, for starting or terminating algorithm executions, from the ROT Controller and performs all the required actions, including the preparation of the execution environment, the monitoring of the execution progress and the handling of the final results or possible failures. Furthermore, it is responsible for monitoring the node's resources where it is executed and returns related information. In addition, the *Decision Algorithms* include the library of multi-objective optimization and orchestration algorithms.

4.2.3 Integration details and REST APIs

4.2.3.1 Integration Details

There are two main North Bound Interfaces (NBIs), the first one is based on REST APIs (Figure 6) and the second one is an asynchronous messaging interface based on the Advanced Message Queuing Protocol (AMQP). The former exposes control operations to manipulate and inspect the execution of deployment algorithms. The latter offers asynchronous communication between ROT and the Resource Orchestrator to send responses and notification messages.

Table 3: Integration details of Resource Optimization Toolkit

IP(s)/Port(s)	ROT Controller: 34.90.62.43:10020 Resource Orchestrator: 147.102.22.140:10100 Central Telemetry Handler: 147.102.16.113:9090
Publicly accessible (y/n and other details)	The IP is publicly accessible, but the access has been restricted though token authentication.
Type of API	REST and asynchronous (AMQP)
Associated host names	N/A
API documentation	https://gitlab.com/serranoproject/wp5/resource-optimization-toolkit/-/raw/main/rot_rest.yaml
Location of integration tests	https://gitlab.com/serranoproject/wp5/resource-optimization-toolkit/-/raw/main/Jenkinsfile

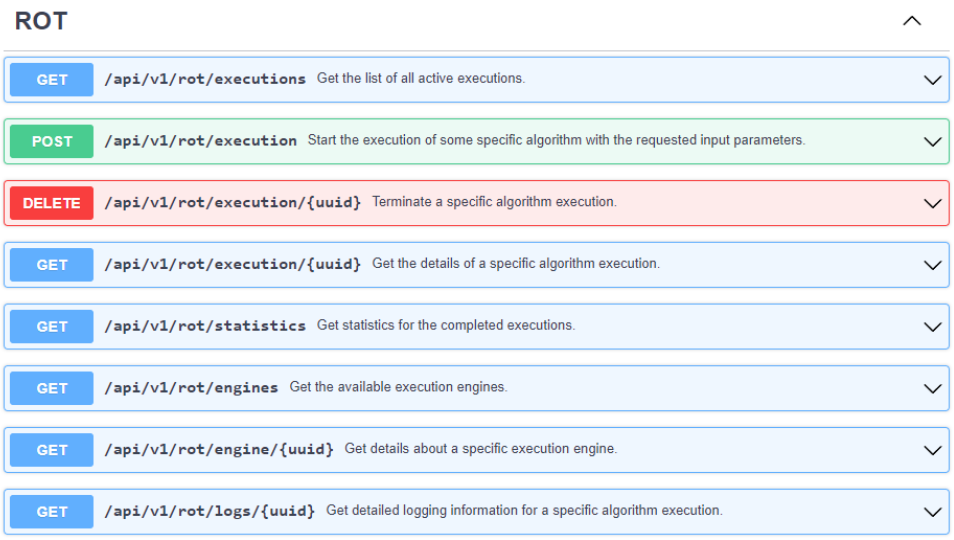


Figure 6: REST Endpoints exposed by ROT (through the Access Interface)

Figure 7 provides the detailed workflow for executing resource allocation algorithms within the SERRANO platform, highlighting the roles of the ROT components and their interaction with other components within the SERRANO platform (Resource Orchestrator, Central Telemetry Handler and Message Broker).

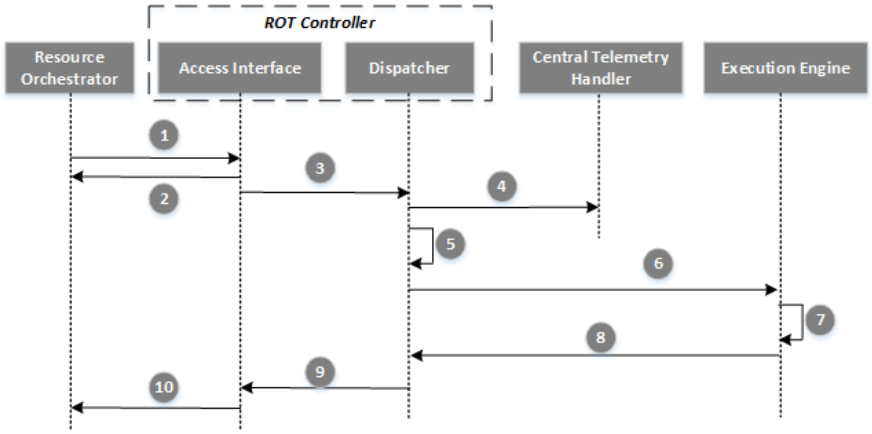


Figure 7: Integration workflow of ROT, Data Broker, Resource Orchestrator and Central Telemetry Handler

4.2.3.2 Sample requests and responses

ROT Controller and Execution Engine

The ROT Controller and Execution Engines use some predefined topics at the Message Broker to exchange the necessary information asynchronously. For example, an Execution Engine sends, during its initialization procedure and periodically, an identification message to the ROT Controller. Similarly, the Execution Engines report the execution responses through a different exchange. Below, there is an example of an identification message.

```
{"engine_id": "7ebef019-1894-4fb0-8ee6-2dd8a096171e", "type": "heartbeat",
 "timestamp": 1654263489, "hostname": "k8s-worker-1"}
```


Moreover, we can use the provided REST method to query the available Execution Engines:

```
GET /api/v1/rot/engines
```


```
{
    "engines": [
        "7ebef019-1894-4fb0-8ee6-2dd8a096171e",
        "6218118f-748d-42ad-b2f2-65e6332435ac"
    ]
}
```


Resource Orchestrator and ROT Controller

The Resource Orchestrator requests the execution of a basic resource allocation algorithm (Step 1 in the above workflow), which is available in the ROT for the integration tests, through the following request.

```
POST /api/v1/rot/execution
```

Parameters:


```
{"execution_plugin": "SimpleMatch","parameters": { "deployment": {"deployment_uuid
": "b9f480d5-3b02-48ed-b3dc-e650351fa5a3", "description": [{"apiVersion": "apps/v1
", "kind": "Deployment", "metadata": {"name": "productcatalogservice"}, "spec": {"
selector": {"matchLabels": {"app": "productcatalogservice"}}, "template": {"metada
ta": {"labels": {"app": "productcatalogservice"}}, "spec": {"containers": [{"name"
: "server", "image": "gcr.io/google-samples/microservices-demo/productcatalogservi
ce:v0.3.7", "ports": [{"containerPort": 3550}], "resources": {"requests": {"cpu":
"100m", "memory": "64Mi"}, "limits": {"cpu": "200m", "memory": "128Mi"}}}]}}}},  {
"apiVersion": "apps/v1", "kind": "Deployment", "metadata": {"name": "currencyservi
ce"}, "spec": {"selector": {"matchLabels": {"app": "currencyservice"}}, "template"
: {"metadata": {"labels": {"app": "currencyservice"}}, "spec": {"containers": [{"n
ame": "server", "image": "gcr.io/google-samples/microservices-demo/currencyservice
:v0.3.7", "ports": [{"name": "grpc", "containerPort": 7000}], "resources": {"reque
sts": {"cpu": "100m", "memory": "64Mi"}, "limits": {"cpu": "200m", "memory": "128M
i"}}}]}}}}]}, "pods_per_cluster": {"6a1a14fe-d623-4398-ba34-9965f8c25a29": 5, "b38
08eb1-8761-4678-8d2e-722f39540990": 9}}
```


The Access Interface assigns a unique identifier in the request, validates it and returns the response to the previous request (Step 2) while forwarding the request to Dispatcher (Step 3).


```
{
    "execution_id": "2b29bb66-45cd-4409-a2fa-23004ca946a1",
    "status": "Accepted"
}
```

ROT Controller and Central Telemetry Handler

The Dispatcher prepares the input parameters, and it also interacts with the Central Telemetry Handler to retrieve the required operational and monitoring data (Steps 4 and 5) through the REST API that provides the SERRANO telemetry framework (Section 4.3.3.1). More specifically, the first request returns the characteristics of the available platforms and the second one detailed information for a specific Kubernetes infrastructure.

```
GET /api/v1/telemetry/central/inventory
```

```
{"uuids": [{"type": "Probe.EdgeStorage","uuid": "ddced532-2c76-4557-9be1-2be622cbd
cee"}, {"type": "Probe.k8s", "uuid": "cb8cce21-d0d1-4638-beee-081dafe63621"},
{"type": "Probe.k8s","uuid": "d5022fba-8dff-41a6-b839-54f045db0e07"}]}
```

```
GET /api/v1/telemetry/central/monitor/cb8cce21-d0d1-4638-beee-081dafe63621
```

```
{ "snf-883643": {"Labels": { "kubernetes.io/arch": "amd64", "node-role.kubernetes.
io/worker": "worker"}, "Node_Capacity": {"cpu": 2, "ephemeral-storage": "61794300K
i", "hugepages-2Mi": 0,"memory": "8148664Ki","pods": 11"}}, "snf-883644": {  "Labe
ls": { "kubernetes.io/arch": "amd64", "node-role.kubernetes.io/worker": "worker"},
"Node_Capacity": { "cpu": "4", "ephemeral-storage": "81794300Ki”, "hugepages-2Mi":
0, "memory": "8148668Ki","pods": 52}}}
```

ROT Controller and Execution Engine

After, the controller assigns the execution request to the Execution Engine with the least workload (Step 6). The selected Execution Engine retrieves the request, executes the selected algorithm with the provided input data (Step 7).

```
DEBUG:SERRANO.ROT.Dispatcher:Execution "c4def23f-6d9b-4ad8-9bf2-2cd3c282f3ff" is a
ssigned to engine: "6218118f-748d-42ad-b2f2-65e6332435ac"
```

Then, it sends the resource allocation decisions to the controller through the Message Broker (Step 8).

```
DEBUG:SERRANO.Orchestrator.ROTInterface: ROT response for execution uuid "c4def23f
-6d9b-4ad8-9bf2-2cd3c282f3ff"
DEBUG:SERRANO.Orchestrator.ROTInterface:{"assignments": [{"cluster_uuid": "6a1a14f
e-d623-4398-ba34-9965f8c25a29", "replicas": 2, "instructions": [{"metadata.name":
"e650351fa5a3"}]}]}
```

ROT Controller and Resource Orchestrator

Finally, the ROT forwards the output to the Resource Orchestrator (Steps 9 and 10).


```
{"uuid": "c4def23f-6d9b-4ad8-9bf2-2cd3c282f3ff", "status": 2, "reason": "", "resul
ts": {"assignments": [{"cluster_uuid": "6a1a14fe-d623-4398-ba34-9965f8c25a29", "re
plicas": 2, "instructions": [{"metadata.name": "e650351fa5a3"}]}]}}
```


4.3 SERRANO Telemetry Framework

4.3.1 Description

The SERRANO platform includes a set of resource monitoring and telemetry mechanisms that provide the sense (detect what is happening) operation in the envisioned closed-loop control. They collect data that is used to improve orchestration decisions, detect problems and trigger proactive or reactive adjustments to SERRANO-enhanced resources and deployed applications. The SERRANO telemetry stack consists of three key building blocks: (a) the Central Telemetry Handler, (b) Enhanced Telemetry Agents and (c) Monitoring Probes.

The heterogeneous and federated edge/cloud/HPC infrastructure, which 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.

4.3.2 Telemetry framework components

4.3.2.1 Central Telemetry Handler and Enhanced Telemetry Agent

They provide the same core functions at different scales and views of the infrastructure resources and deployed applications. 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. Their common architecture is depicted in Figure 8.

The Notification Engine handles the event notifications generated by the Analytic Engine and the Telemetry Controller. The Telemetry Controller manages all the other telemetry entities (i.e., Enhanced Telemetry Agents or Monitoring Probes) under the specific instance's control. The Analytic Engine provides the analytic login in the telemetry framework. The Data Engine undertakes the initial processing of the collected data, which is forwarded by the Data Collector mechanisms. 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.

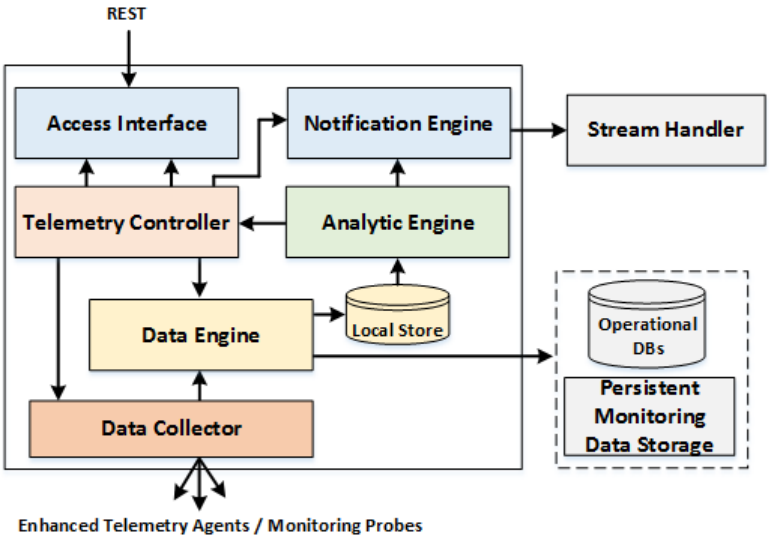


Figure 8: Central Telemetry Handler and Enhanced Telemetry Agent architecture

Moreover, the telemetry framework includes the Operational Database and the Persistent Monitoring Data Storage (PMDS) component. The former stores data related to the available resources, their current state, and details about the applications' deployments. The latter provides long-term storage for the collected telemetry data, providing historical data to the SERRANO orchestration and service assurance mechanisms.

4.3.2.2 Monitoring Probes

They collect information about the characteristics and current status of the infrastructure resources, services, and deployed applications. There is a set of different probes, each one specialized to monitor a specific resource type. SERRANO adopts a common design for the monitoring probes (Figure 9).

The Access Interface provides a common interface for integrating the various probes with the data collection and management services of the Enhanced Telemetry Agents. It also configures the low-level monitoring probes. The Streaming Telemetry enables the collection of measurements based on the streaming telemetry approach, where continuous measurements are sent at a rate much faster than the typical monitoring approach. The Monitoring Probes are the components that collect the monitoring information.

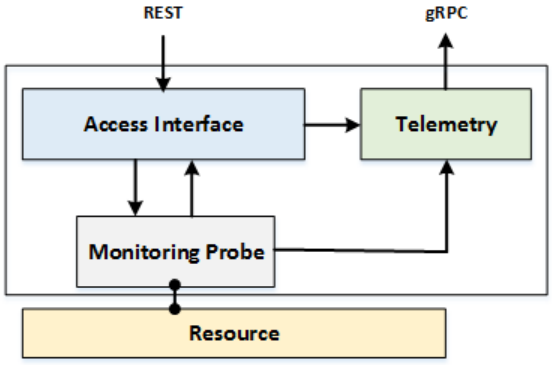


Figure 9: General architecture of SERRANO monitoring probes

Regarding the monitoring of cloud hardware accelerators, a specific monitoring probe has been developed to scrape metrics from the cloud GPUs (i.e., NVIDIA T4) and FPGAs (i.e., Xilinx Alveo U50 and U200). We make use of the Data Center GPU Manager (DCGM) Exporter and an in-house custom FPGA Exporter (FCE) to expose accelerator metrics. Table 4 shows the metrics exposed for the cloud GPUs.

Table 4. DCGM metrics description

DCGM Metric	Description
DCGM_FI_DEV_SM_CLOCK	SM clock frequency (in MHz)
DCGM_FI_DEV_MEM_CLOCK	Memory clock frequency (in MHz)
DCGM_FI_DEV_MEMORY_TEMP	Memory temperature (in C)
DCGM_FI_DEV_GPU_TEMP	GPU temperature (in C)
DCGM_FI_DEV_POWER_USAGE	Power draw (in W)
DCGM_FI_DEV_TOTAL_ENERGY_CONSUMPTION	Total energy consumption since boot (in mJ)
DCGM_FI_DEV_PCIE_REPLAY_COUNTER	Total number of PCIe retries
DCGM_FI_DEV_MEM_COPY_UTIL	Memory utilization (in %)
DCGM_FI_DEV_ENC_UTIL	Encoder utilization (in %)
DCGM_FI_DEV_DEC_UTIL	Decoder utilization (in %)
DCGM_FI_DEV_XID_ERRORS	Value of the last XID error encountered
DCGM_FI_DEV_FB_FREE	Framebuffer memory free (in MiB)
DCGM_FI_DEV_FB_USED	Framebuffer memory used (in MiB)
DCGM_FI_DEV_NVLINK_BANDWIDTH_TOTAL	Total number of NVLink bandwidth counters for all lanes
DCGM_FI_DEV_VGPU_LICENSE_STATUS	vGPU License status
DCGM_FI_DEV_UNCORRECTABLE_REMAPPED_ROWS	Number of remapped rows for uncorrectable errors
DCGM_FI_DEV_CORRECTABLE_REMAPPED_ROWS	Number of remapped rows for correctable errors
DCGM_FI_DEV_ROW_REMAP_FAILURE	Whether remapping of rows has failed
DCGM_FI_PROF_GR_ENGINE_ACTIVE	Ratio of time the graphics engine is active (in %)
DCGM_FI_PROF_PIPE_TENSOR_ACTIVE	Ratio of cycles the tensor (HMMA) pipe is active (in %)
DCGM_FI_PROF_DRAM_ACTIVE	Ratio of cycles the device memory interface is active sending or receiving data (in %)
DCGM_FI_PROF_PCIE_TX_BYTES	The number of bytes of active PCIE TX data including both header and payload
DCGM_FI_PROF_PCIE_RX_BYTES	The number of bytes of active PCIE RX data including both header and payload

Moreover, Table 5 shows the metrics exposed for the cloud FPGAs. In the final version of the SERRANO platform more FPGA metrics (e.g., resources utilization) will be exposed.

Table 5. FCE metric description

FCE Metric	Description
FCE_ FPGA_TEMP	The FPGA temperature (in C)
FCE_POWER	The FPGA power consumption (in W)

4.3.3 Integration details and REST APIs

4.3.3.1 Integration Details

Figure 10 presents the setup for the integration tests of the SERRANO telemetry framework. Moreover, it shows the interactions between all the involved components, which are based on the REST APIs (Figure 11) that expose the components of the SERRANO telemetry framework.

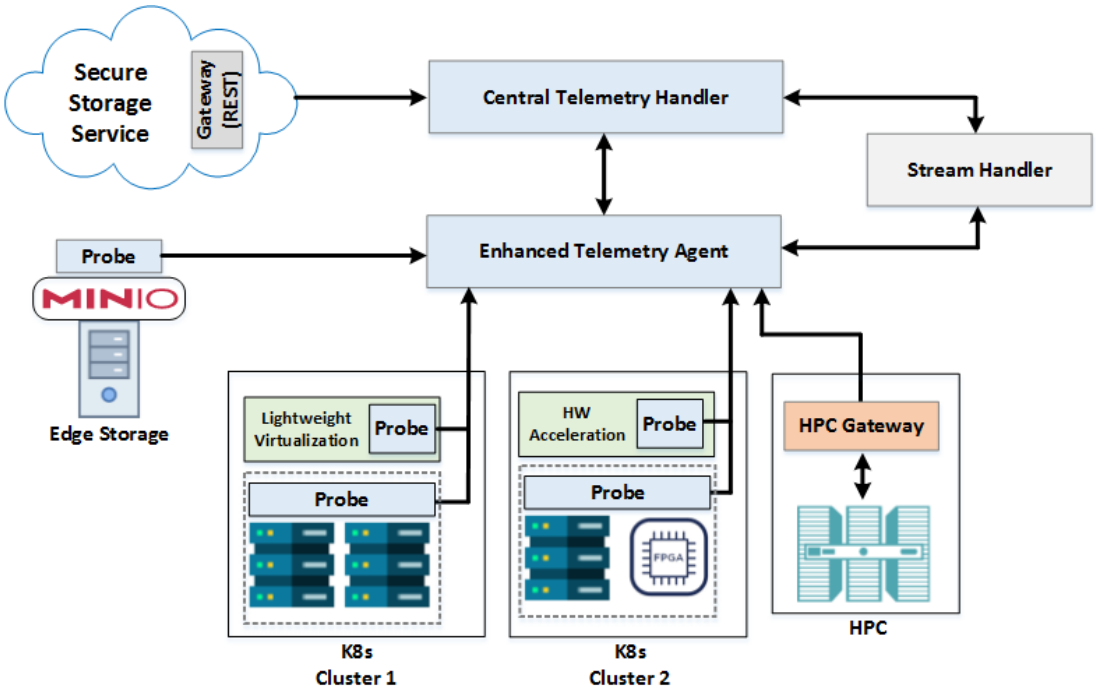
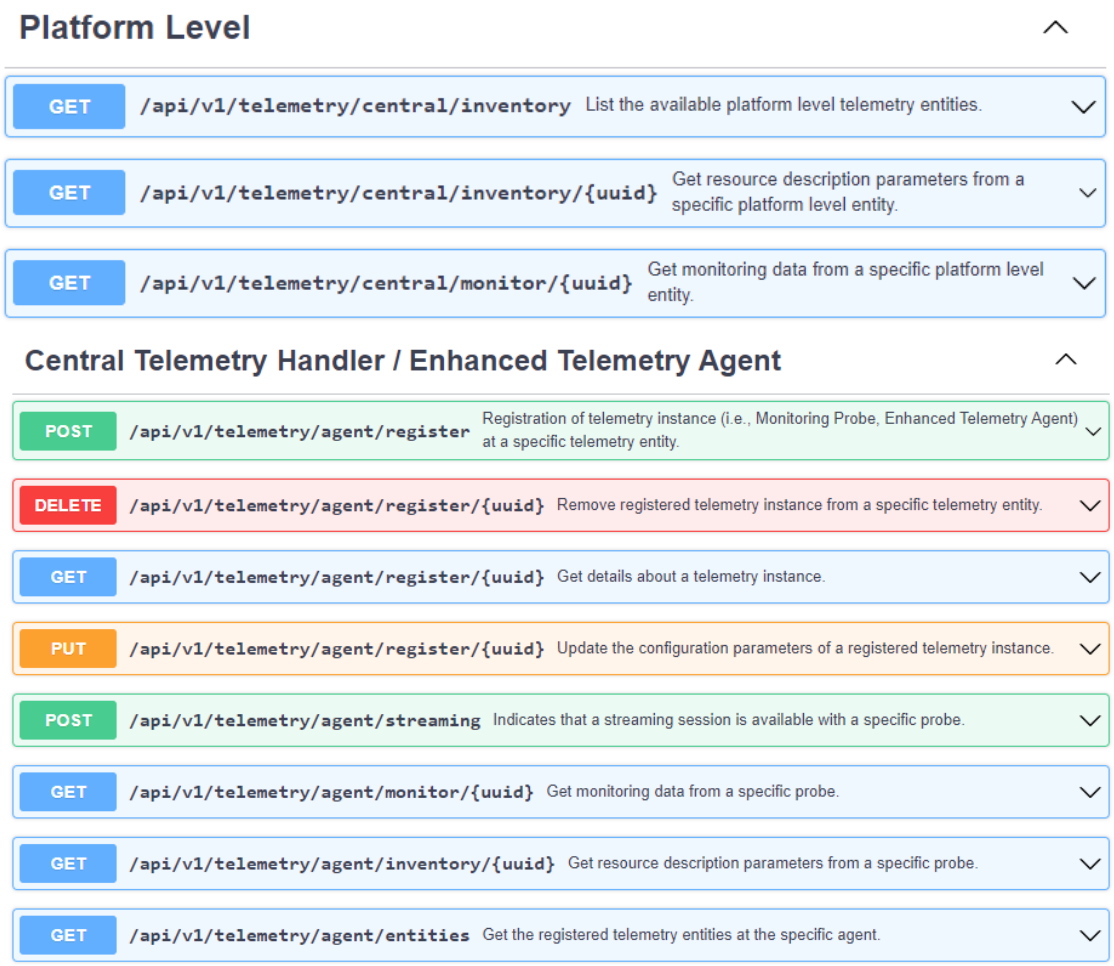


Figure 10: Setup for the integration tests of SERRANO telemetry framework

Table 6: Integration details of SERRANO telemetry framework

IP(s)/Port(s)	Central Telemetry Handler: 147.102.16.113:9090 Enhanced Telemetry Agent: 147.102.22.141:9092 Stream Handler: 88.198.124.99:9092 Monitoring Probes: • Cloud storage locations: Accessible through port 2525 • Edge storage device: 147.102.22.140:9020 • Kubernetes clusters: 147.102.16.113:9080 & 34.90.1.58: 9080

	• Hardware accelerators: http://skylla.physics.auth.gr:30090 • HPC Hardware System Interface: hpc-interface.serrano.cs.uvt.ro:8080
Publicly accessible (y/n and other details)	The IPs are publicly accessible, but the access has been restricted though authentication.
Type of API	REST and asynchronous (Stream Handler)
Associated host names	N/A
API documentation	https://gitlab.com/serranoproject/wp5/telemetry-framework/-/raw/master/telemetry_rest.yaml
Location of integration tests	https://gitlab.com/serranoproject/wp5/telemetry-framework/-/raw/master/Jenkinsfile



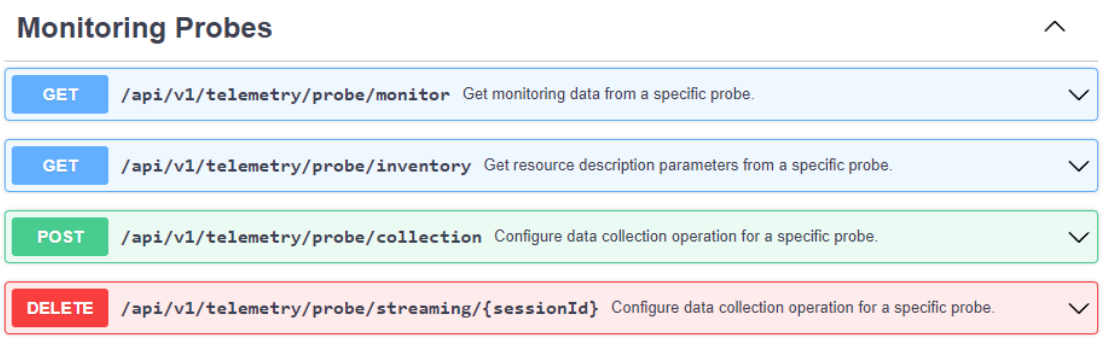


Figure 11: REST Endpoints exposed by telemetry framework components

4.3.3.2 Sample requests and responses

Enhanced Telemetry Agent and Monitoring Probes

The monitoring probes, during their initialization, are automatically registered to a predefined Enhanced Telemetry Agent specified in their configuration file. The registration is performed through the Enhanced Telemetry Agent API. Below, there is an example for registering the monitoring probe for a SERRANO edge storage device.

```
POST /api/v1/telemetry/agent/register
Parameters:

{"uuid": "ddced532-2c76-4557-9be1-2be622cbdcee", "url": "http://147.102.22.140:902
0", "type": "Probe.EdgeStorage", "inventory": {"uuid": "1a558d68-8905-11ec-9b0a-eb
5f7b677049", "hostname": "desktop-140", "location": "", "cores_total": 4, "ram_tot
al": 12448088064, "total_disk_space": 500107862016, "is_k8s_node": "false", "archi
tecture": "x86_64", "os_name": "Ubuntu", "os_version": "20.04.4 LTS (Focal Fossa)"
}}
```

We can get the list of all registered probes at a specific Enhanced Telemetry Agent using the following request:

```
GET /api/v1/telemetry/agent/entities
```

```
{
  "entities": [
   {"type": "Probe.k8s", "url": "http://147.102.16.113:9080","uuid": "cb8cce21-d0d
1-4638-beee-081dafe63621"},
   {"type": "Probe.EdgeStorage", "url": "http://147.102.22.140:9020", "uuid": "ddc
ed532-2c76-4557-9be1-2be622cbdcee"}
  ]
}
```

This response corresponds to the Enhanced Telemetry Agent that manages the first K8s cluster and the SERRANO edge storage device.

SERRANO Edge Storage

This probe collects performance monitoring parameters for the SERRANO edge storage devices. It can track the health of the nodes that host the SERRANO edge storage devices and collect information by the MinIO [2] service that is the basis of the SERRANO edge storage devices. The Enhanced Telemetry Agent, through the following request, can retrieve the collected monitoring data.

```
GET /api/v1/telemetry/probe/monitor
```


```
{
"cpu_usage": 7.75,"disk_space_avail_GB": 155.03, "disk_space_used_GB": 57.21,    "
minio_bucket_usage_object_total": 445.0, "minio_bucket_usage_total_bytes": 1816365
212.0,"minio_node_disk_free_bytes": 166462873600.0, "minio_node_disk_total_bytes":
227894644736.0,"minio_node_disk_used_bytes": 61431771136.0, "minio_node_file_descr
iptor_open_total": 16.0,"minio_node_process_uptime_seconds": 21745.340186411, "min
io_s3_requests": 2.0, "minio_s3_requests_errors": 2.0, "minio_s3_requests_waiting_
total": 0.0, "minio_s3_traffic_received_bytes": 0.0,"minio_s3_traffic_sent_bytes":
226.0, "net_received_kbps": 14.5383512, "net_sent_kbps": 7.065654, "ram_free_MB": 4
010.8096,"ram_used_MB": 4140.6116, "timestamp": 1654610230, "type": "Probe.EdgeSto
rage", "uuid": "ddced532-2c76-4557-9be1-2be622cbdcee"
}
```


Kubernetes Cluster

A probe also monitors a series of parameters regarding the characteristics of the computational and storage resources, their current status, and the deployed applications within a Kubernetes cluster. For example, the following request retrieves information for the available resources through this probe.

```
GET /api/v1/telemetry/probe/inventory
```


```
{"snf-883643": { "Labels": {"kubernetes.io/arch": "amd64", "kubernetes.io/hostname"
: "snf-883643",  "node-role.kubernetes.io/worker": "worker"}, "Node_Capacity": { "
cpu": 4, "ephemeral-storage": "61794300Ki","memory": "8148664Ki","pods": 110}, "No
de_Role": "Worker", "PersistentVolumes": {}}, "snf-883644": { "Labels": { "kuberne
tes.io/arch": "amd64", "kubernetes.io/hostname": "snf-883644","node-role.kubernete
s.io/worker": "worker"},"Node_Capacity": {"cpu": 4,"ephemeral-storage": "81794300K
i","memory": "8148668Ki","pods": 24},"Node_Role": "Worker","PersistentVolumes": {}
},"master-node": {"Labels": {"kubernetes.io/arch": "amd64","kubernetes.io/hostname
": "master-node","node-role.kubernetes.io/master": ""},"Node_Capacity": {"cpu": 2,
"ephemeral-storage": "64135932Ki","memory": "6004348Ki","pods": 43}, "Node_Role":
"Master","PersistentVolumes": {}}}
```

Hardware Accelerators

The developed monitoring probed for the hardware accelerators has been deployed over the Kubernetes and can be accessed by performing PromQL[2] queries to the endpoint: *http://skylla.physics.auth.gr:30090*

The following example demonstrates how end-users can acquire metrics (e.g., the FCE_FPGA_TEMP metric) from the endpoint using the curl command and a basic PromQL query as well as the expected output.

```
curl –X GET http://skylla.physics.auth.gr:30090/api/v1/query?query=FCE_FPGA_TEMP
```

```
{
    "status": "success",
    "data": {
        "resultType": "vector",
        "result": [
            {
                "metric": {
                    "name": "FCE_FPGA_TEMP",
                    "dsa_name": "xilinx_u200_gen3x16_xdma_base_1",
                    "instance": "155.207.169.212:9877",
                    "job": "fpga-custom-exporter"
                },
                "value": [1655285236.858, "29"]
            },
            {
                "metric": {
                    "nam": "FCE_FPGA_TEMP",
                    "dsa_name": "xilinx_u50_gen3x16_xdma_201920_3",
                    "instance": "155.207.169.212:9877",
                    "job": "fpga-custom-exporter"
                },
                "value": [1655285236.858,"44"]
            }
        ]
    }
}
```

HPC Platform

Similarly, the Enhanced Telemetry Agent collects, through the exposed REST interface by the SERRANO HPC Gateway (Section 4.6), monitoring data about the available HPC resources and services. The HPC Gateway exposes telemetry endpoint for a particular HPC infrastructure. The endpoint returns the information about the total and available resources in the partitions of the HPC infrastructure. For the sake of coherence and completeness, the example of the endpoint call, which can be found in Section 4.6.1.2, is outlined below.

```
GET /infrastructure/cluster_name/telemetry
```

```
{
  "host": "A.B.C.D",
  "hostname": "infrastructure.example.com",
  "name": "cluster_name",
  "partitions": [
        {
        "avail_cpus": 158,
        "avail_nodes": 1,
        "name": "profile",
        "queued_jobs": 0,
        "running_jobs": 1,
        "total_cpus": 160,
        "total_nodes": 2
        }
  ],
  "scheduler": "slurm"
}
```


Telemetry framework and Stream Handler

It is critical for the operation of the telemetry framework to support the exchange of events among the various components and services. To this end, the Notification Engine acts as a producer that forwards event notifications through the Stream Handler of the Data Broker to multiple consumers (e.g., other components of the telemetry framework and service assurance mechanisms). For example, there is a dedicated topic in the Stream Handler (*"serrano_telemetry_notifications"*) for exchanging events related to the operational state of the Enhanced Telemetry Agents and Monitoring Probes. We provide an example of an event notification that the Central Telemetry Handler receives through the Stream Handler. The event is triggered by the Enhanced Telemetry Agent that manages the specific probe.


```
{"entity_id": "ddced532-2c76-4557-9be1-2be622cbdcee", "status": "DOWN",
  "type": "Probe", "timestamp": 1654612649}
```


Further information from the perspective of the Stream Handler component integration is available in section 4.4.2.4.1.

Central Telemetry Handler and Enhanced Telemetry Agent

We can verify the availability of the Enhanced Telemetry Agent under the control of the Central Telemetry Handler by using the provided REST API.

```
GET /api/v1/telemetry/agent/entities
```


```
{
  "entities": [
   {"type": "Agent", "url": "147.102.22.141:9092", "uuid": "1e4d651d-e33a-4672-b2d
5-cc43555e3cfe "}
  ]
}
```

Moreover, the Central Telemetry Handler is configured to query the secure storage service to retrieve details about the available cloud storage locations. The Resource Orchestrator can utilize this information during the assignment of the applications' workload at the available resources. Below, it is found an example of this operation and a snippet of the retrieved information.

```
GET /cloud_locations
```


```
{
  "location":"Frankfurt", "country":"Germany", "countrycode":"DE","is_gdpr":true,
  "storage_price":13.0,"download_price":30.0, "upload_price":0,"lat":50.1109221,
  "lng":8.6821267,"cloud_provider_name":"IONOS S3 Object Storage",
  "cloud_provider_jurisdiction":"Germany"
},
{
  "location":"Vienna","country":"Austria","countrycode":"AT", "is_gdpr":true,
  "storage_price":null,"download_price":null,"upload_price":0,"lat":47.9700167,
  "lng":15.4408725, "cloud_provider_name":"Ventus Cloud","cloud_provider_jurisdict
   ion":"Switzerland"
}
```


Telemetry Framework and Service Assurance Mechanisms

A description of this interaction is available in Section 4.9.1**.**

4.4 Data Broker

4.4.1 Message Broker

4.4.1.1 Description

The Message Broker is one of the two components of the SERRANO Data Broker service (Figure 12). Overall, the Data Broker provides the appropriate communication mechanisms to interconnect the individual components and enable the exchange of messages and events within the distributed SERRANO platform.

The Message Broker provides message brokering functionalities enabling the asynchronous communication and data transfer between the SERRANO platform components and the deployed applications. It is based on the RabbitMQ [3] that supports different transport and messaging protocols, such as the different versions of Advanced Message Queuing Protocol (AMQP) and MQ Telemetry Transport (MQTT). For the MQTT, there is a plugin [4] available that is shipped in the core distribution of the RabbitMQ. A simple Python library has been developed in SERRANO that abstracts the interaction with the actual message brokering middleware.

4.4.1.2 Integration details and REST APIs

Table 7: Integration details of Message Broker

IP(s)/Port(s)	Message Broker (AMQP): 147.102.22.241:5672 Message Broker (MQTT): 147.102.22.241:1883
Publicly accessible (y/n and other details)	The service is publicly accessible, but the access has been restricted though user authentication.
Type of API	Python API
Associated host names	N/A
API documentation	https://gitlab.com/serranoproject/wp5/message-broker/docs
Location of integration tests	N/A

Resource Optimization Toolkit

The ROT is one of the SERRANO components that leverages the Message Broker functionalities to enable the asynchronous communication between its components (Section 4.2.2). More details for the usage of the Message Broker in the context of the ROT are available in Section 6.1 of deliverable D5.3 "D5.3 - Resource Orchestration, Telemetry and Lightweight Virtualization Mechanisms" (M15).

For the communication between the multiple Execution Engines and the Dispatcher of the ROT Controller we use the "default" exchange type that contains a predefined queue with the name *"rot_engines_to_controller"*. An example of a response for an execution request that executed successfully follows below.

```
{"engine_id": "6218118f-748d-42ad-b2f2-65e6332435ac", "type": "execution", "execut
ion_id": "c4def23f-6d9b-4ad8-9bf2-2cd3c282f3ff", "status": 2, "timestamp": 1654428
522, "reason": "", "results": {"assignments": [{"cluster_uuid": "6a1a14fe-d623-439
8-ba34-9965f8c25a29", "replicas": 8, "instructions": [{"metadata.name": " e650351f
a5a3"}]}]}}
```

For this operation, we use the *"direct"* exchange type with the name *"rot_dispatcher_requests"* that delivers to queues whose binding key matches exactly the message's routing key, which is the unique identifier of each Execution Engine. Hence, the execution request is received only from the intended engine that handles its execution. An example of an execution request for the engine with the identifier "6218118f-748d-42ad-b2f2-65e6332435ac" follows below.

```
{"engine_id": "6218118f-748d-42ad-b2f2-65e6332435ac","action": "start", "execution
_id": "c4def23f-6d9b-4ad8-9bf2-2cd3c282f3ff", "execution_plugin": "SimpleMatch", "
parameters":{ "deployment": { "deployment_uuid": "b9f480d5-3b02-48ed-b3dc-e650351f
a5a3","description": [{"kind": "Deployment","metadata": { "name": "productcatalogs
ervice"}, "spec": {"containers": [{"image": "gcr.io/google-samples/microservices-d
emo/productcatalogservice:v0.3.7", "resources": {"requests": {"cpu": "100m","memor
y": "64Mi"},"limits": {"cpu": "200m","memory": "128Mi"}}}]}}]},"pods_per_cluster":
{"6a1a14fe-d623-4398-ba34-9965f8c25a29": 17,"b3808eb1-8761-4678-8d2e-722f39540990"
: 10}}}
```


Moreover, we can get the list of the active queues and exchanges where a specific service is subscribed by using the provided methods.


```
[{"auto_delete":false, "message_stats":{"publish_in":61, "publish_out":59},"name":
"rot_dispatcher_requests", "type":"direct", "user_who_performed_action": "serrano_
rot_dev","vhost":"/"},
{"auto_delete":false, "message_stats":{"publish_in":61, "publish_out":59} "name":"
rot_dispatcher_responses","type":"fanout","user_who_performed_action":"serrano_rot
_dev","vhost":"/"}]
```


4.4.2 Stream Handler

4.4.2.1 Description

SERRANO's distributed streaming platform will allow publishing and subscribing 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 allows asynchronous communication between SERRANO platform components as well as deployed applications.

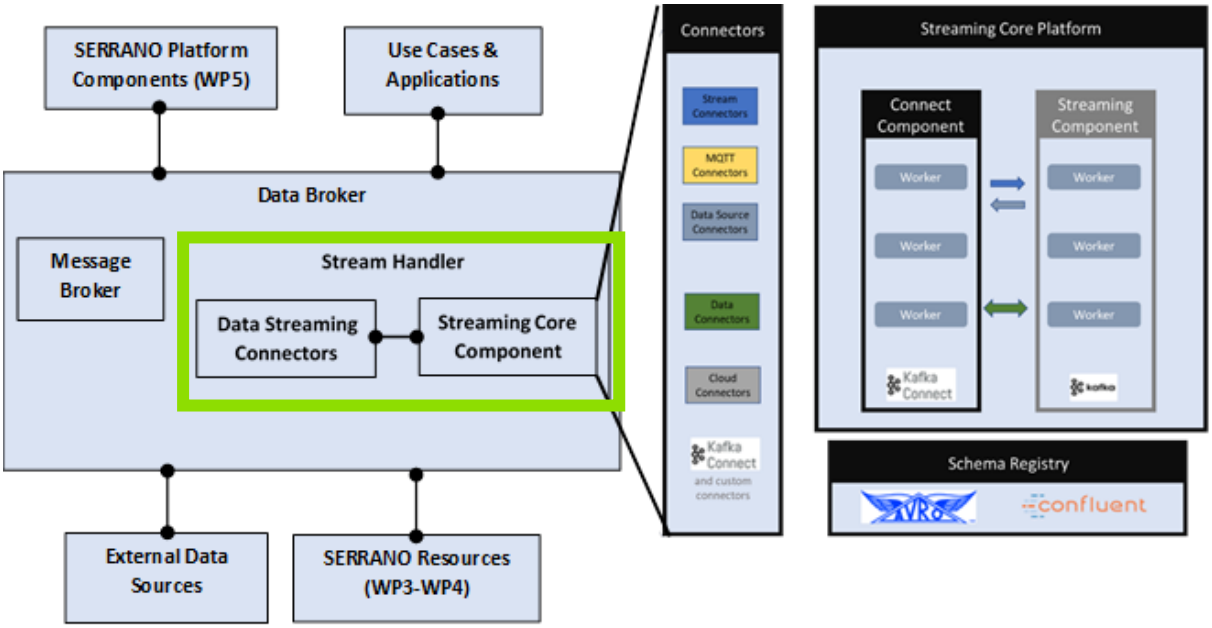


Figure 12 Stream Handler and possible integrations with data sources and other infrastructure

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 12 shows the key building blocks and their interactions with other SERRANO components.

4.4.2.2 Inner components

4.4.2.2.1 Streaming Component (Kafka Cluster)

A Kafka [5] broker is a server running in a Kafka cluster (or, put another way: a Kafka cluster is made up of a number of brokers). Typically, multiple brokers work in concert to form the Kafka cluster and achieve load balancing and reliable redundancy and failover.

Brokers utilize Apache ZooKeeper [6] for the management and coordination of the cluster. Each broker instance is capable of handling read and write quantities reaching to hundreds of thousands each second without any impact on performance. Each broker has a unique ID and can be responsible for partitions of one or more topic logs.

Connecting to any broker will bootstrap a client to the full Kafka cluster. To achieve reliable failover, a minimum of three brokers should be utilized —with greater numbers of brokers comes increased reliability in the Zookeeper Quorum, the number of server nodes that are available for client requests and guarantee a consistent view of the system.

4.4.2.2.2 Connect Component (Kafka Connect)

Kafka connect is built on top of Kafka core components. The Kafka connect includes a bunch of ready to use off the shelf Kafka connectors that one can use to move data between Kafka broker and other applications. For using Kafka connectors, there is no need to write code or make changes to the applications. Kafka connectors are purely based on configurations.

The Kafka Connect also offers a framework that allows developing one's own custom Source and Sink connectors quickly. If there is not a ready to use connector for the system, one can leverage the Kafka connect framework to develop one's own connectors.

4.4.2.2.3 REST Proxy (Connector)

Some applications might want to leverage RESTful HTTP protocol for producing and consuming messages to and from Kafka brokers. The Kafka REST Proxy provides a RESTful interface to a Kafka cluster. It makes 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.

4.4.2.2.4 Schema Registry

The Schema Registry allows the definition and storage of data models describing the data. It stores a versioned history of all schemas, provides multiple compatibility settings and allows evolution of schemas according to the configured compatibility settings. The Schema Registry is implemented through a Kafka add-on, the Confluent Schema Registry that exposes a RESTful interface for storing and retrieving schemas.

4.4.2.3 Supported Integrations

Integration with the following components is supported:

Data sources and Data stores represent data streams and data sources, both in a structured or unstructured format that can be made available and potentially be connected to the Big data platform, generated by any IoT device and/or gateway on the edge. Similarly, and according to the requirements, appropriate persistent storage can be used, as depicted in the input/output data components (Figure 13). The described data sources will be seamlessly integrated with processing components through integration connectors (Connectors). The Big data platform can efficiently interoperate with all the modern data storage technologies of a Big data ecosystem such as RDBMS, NoSQL, HDFS [7], Apache HBASE [8], etc. as well as other persistence approaches such as Mongo [9], MySQL [10], etc.

Data analytics and Data Visualization represent the applications that perform the data processing and analytics. These are dependent on the exact use cases that are implemented through the use of the INTRA's Stream Handler Platform and can be implemented in any programming language typically preferred for data science (such as Python, Java, R and Scala) or any native programming language (e.g., C/C++, Haskel, etc.).

Processing and Machine Learning (ML)/ Deep learning (DL) Infrastructure. The underlying infrastructure spans multiple VMs and provides all the necessary technologies and components that enable the storage and analysis of the data involved and further allow the usage of any technology agnostic algorithms, by providing a distributed computing environment that enables the above. Apache Spark [11], Hadoop [12], Kafka Streams [13], Spark Streaming [14] are included, among others. Moreover ML/DL Infrastructure provides all the necessary components for the analysis of the data in order to build analytics models using open-source frameworks like TensorFlow[15], DeepLearning4J [16], or H2O.ai [17].

4.4.2.4 Integration details and 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.

4.4.2.4.1 Resource Orchestrator and Central Telemetry Handler

The Notification Engine of the Central Telemetry Handler publishes messages related to telemetry events that need to be consumed by other components within the telemetry framework or external services. More specifically, the Notification Engine posts messages to topics having a predefined name, such as "serrano_notification_messages". Other components can subscribe to these topics without limits in the number of subscribers. The content of each notification message is described in JSON format using a common syntax. Table 8 describes the notification messages exposed by the SERRANO telemetry framework.

Table 8: Telemetry notification messages

Notification Type	Event Identifier	Event payload description
General	Information	• ***message***: event related information • ***timestamp***: Unix time stamp
Telemetry	Agent	• ***entity_id***: agent unique identifier • ***status***: "UP" or "DOWN" • ***timestamp***: Unix time stamp
Telemetry	Probe	• ***entity_id***: probe unique identifier • ***status***: "UP" or "DOWN" • ***timestamp***: Unix time stamp
Resources	Status	• ***entity_id***: resource unique identifier • ***event***: Detected event • ***timestamp***: Unix time stamp

An example of a notification message concerning a Telemetry probe status can be found below:

```
{"entity_id": "ddced532-2c76-4557-9be1-2be622cbdcee", "status": "DOWN", "type":
  "Probe", "timestamp": 1654612649}
```

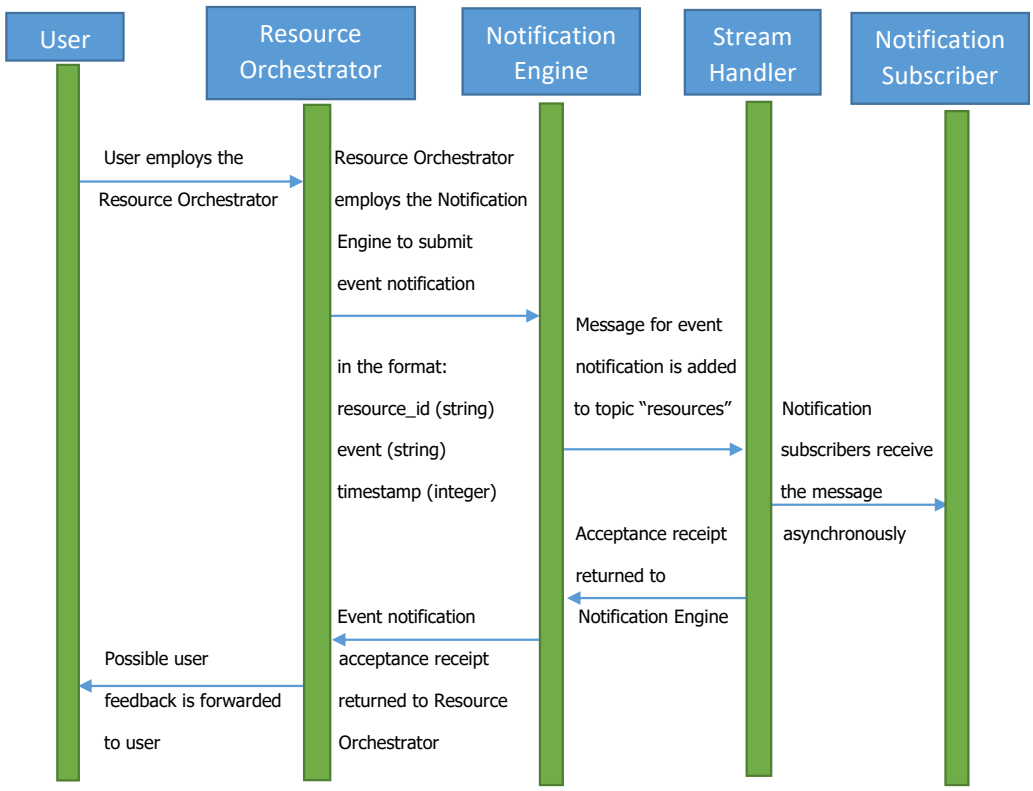


Figure 13 Resource Orchestrator Interaction with Stream Handler

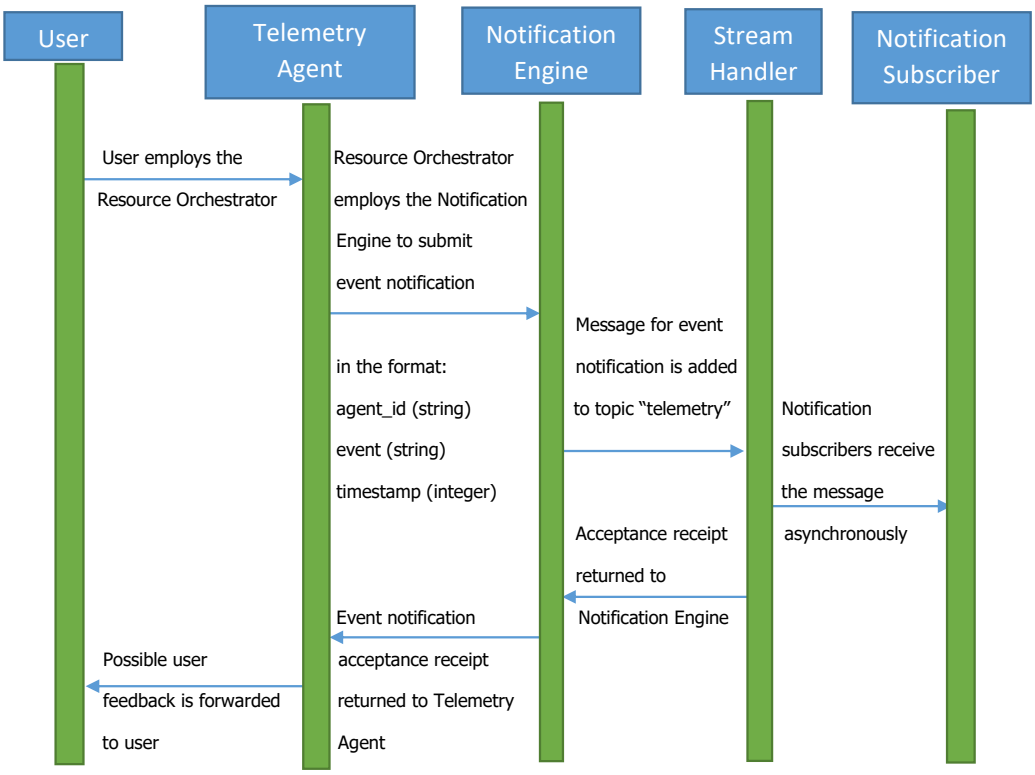


Figure 14 Telemetry Agent Interaction with Stream Handler

4.4.2.4.2 Service Assurance

As described in more detail in Section 4.9.1, when an event anomaly is detected by the Event Detection Engine (EDE) component of service assurance (specifically by its prediction sub-component), it is necessary to notify the SERRANO components in charge of orchestration, scheduling, or remediation. Thus, the service assurance mechanisms publish the detected anomalous event to a particular Kafka topic in the SERRANO Stream Handler to which other SERRANO components can subscribe. This communication is covered in more detail by the sequence diagram contained in Figure 36.

4.4.2.4.3 Integration details

Table 9: Integration details of Stream Handler

IP(s)/Port(s)	Kafka protocol over TCP: 88.198.124.99:9092 Rest-proxy: 88.198.124.99:8082 Schema-registry: 88.198.124.99:8081
Publicly accessible (y/n and other details)	The IP is publicly accessible, but the access has been restricted to specific IPs through the firewall configuration. More IPs that correspond to SEERANO components or partners can be added to this whitelist.
Type of API	Kafka protocol, REST
Associated host names	static.88-198-124-99.clients.your-server.de

API documentation	https://gitlab.com/serranoproject/wp6/streamhandler/-/raw/main/rest_proxy.yaml https://gitlab.com/serranoproject/wp6/streamhandler/-/raw/main/schema_registry.yaml
Location of integration tests	https://gitlab.com/serranoproject/wp6/streamhandler/-/raw/develop/Jenkinsfile

The REST APIs exposed by the REST proxy and the Schema Registry are shown in the following two figures.

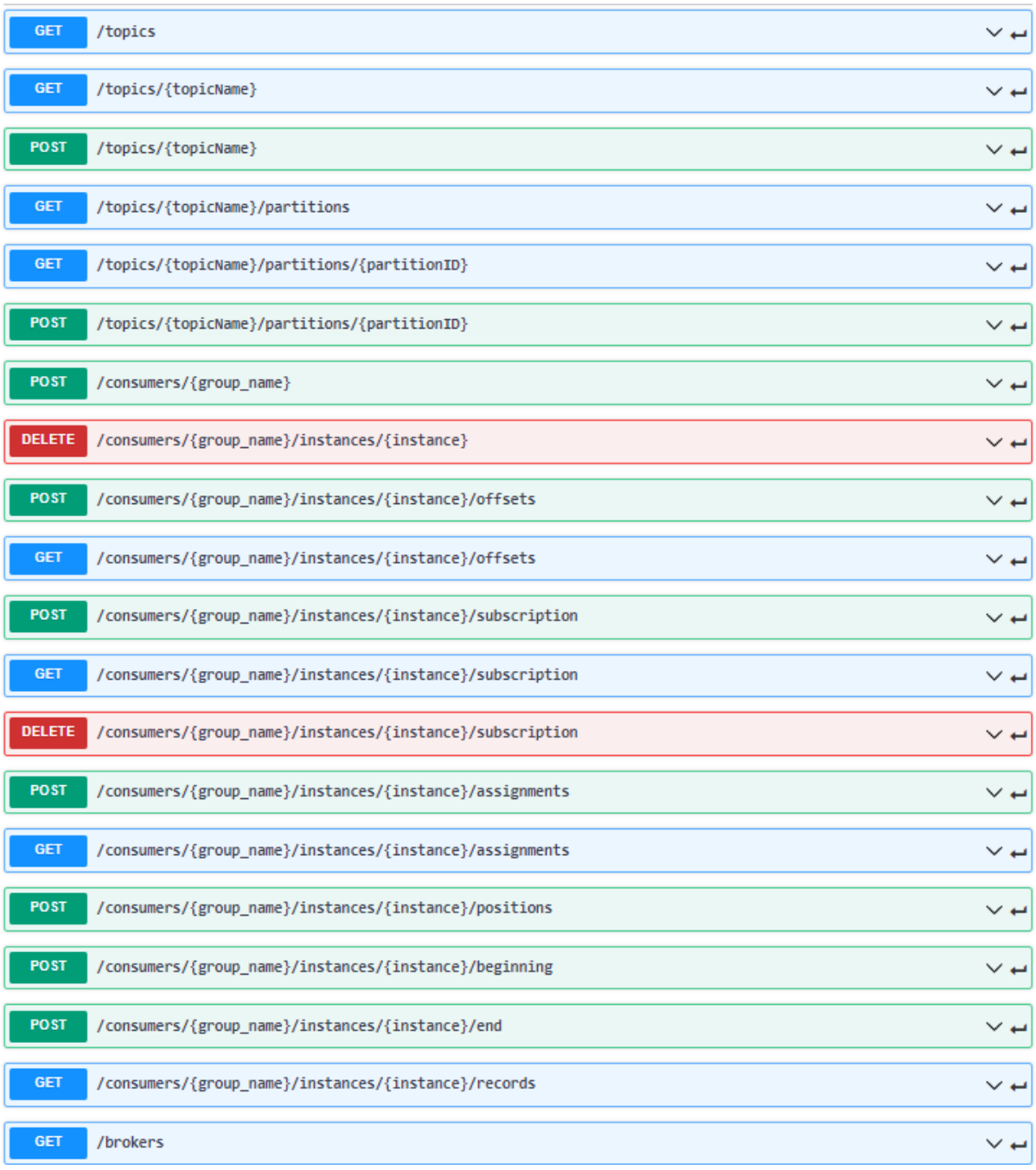


Figure 15: REST Endpoints exposed by Streaming Core Platform (through the REST Proxy)

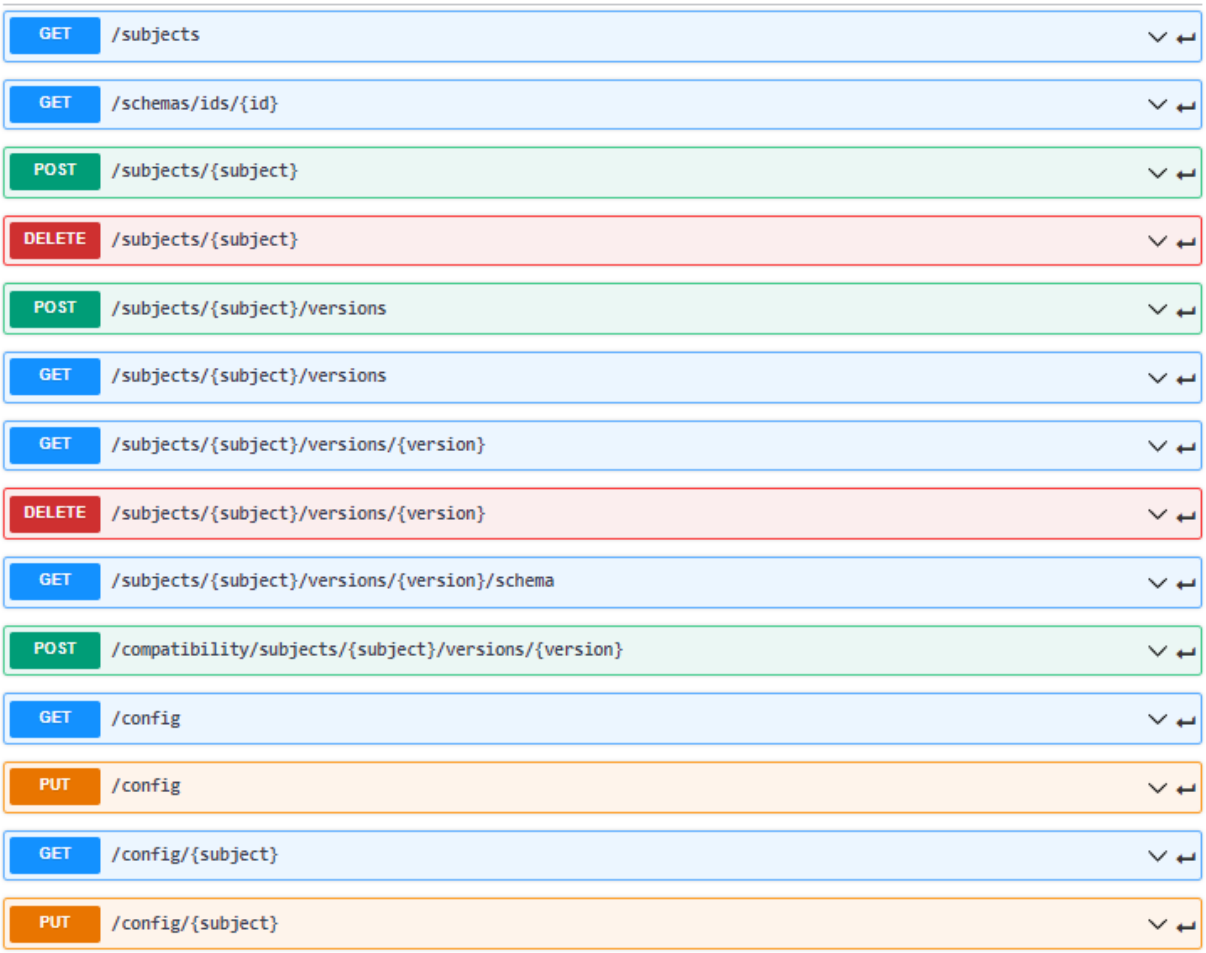


Figure 16: Schema Registry REST API

4.4.2.4.4 Sample requests and responses

Kafka protocol

The integration using the Kafka protocol can be demonstrated using a consumer and producer that publish and subscribe to a Kafka topic, respectively. For simplicity, the example presented in Figure 17 does not involve communication encryption which will be enabled when transferring data that are relevant to the SERRANO platform operations.

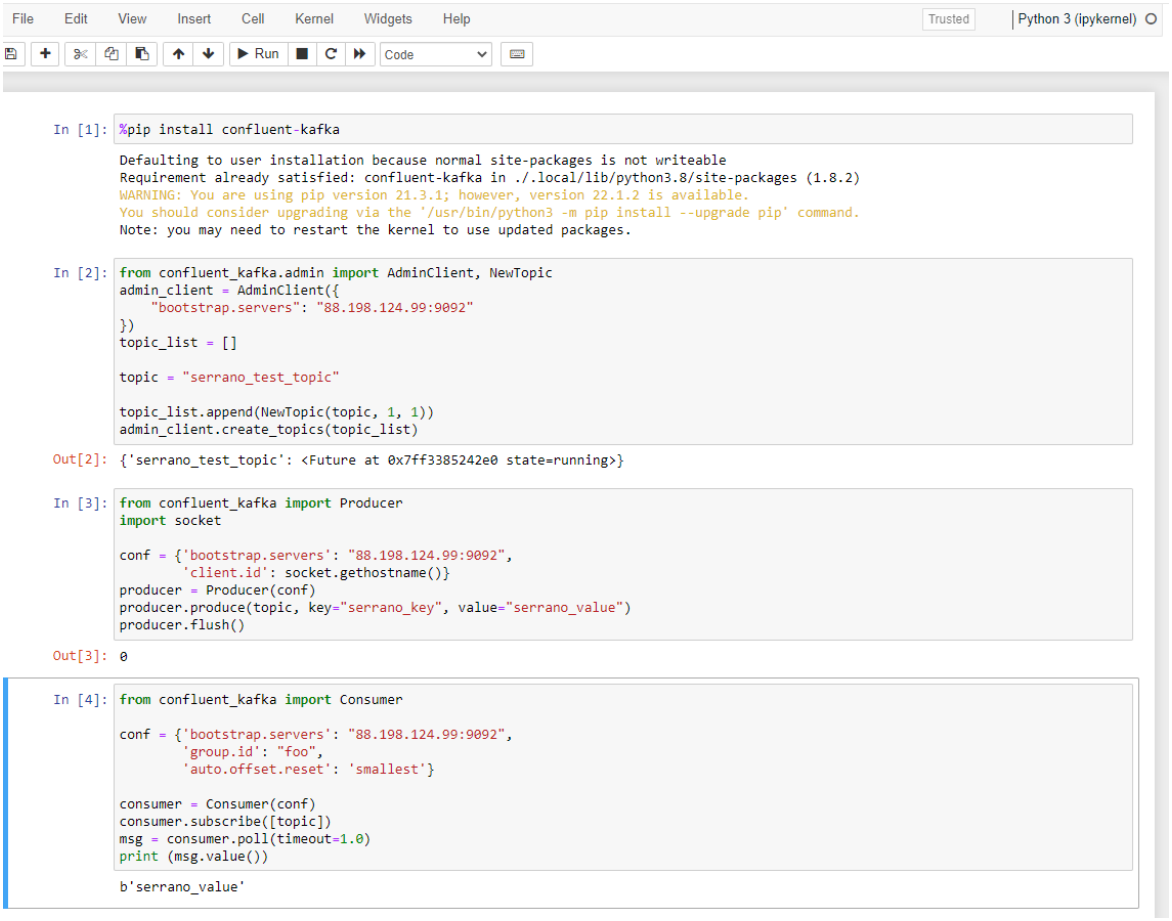


Figure 17: Stream Handler example on a Jupyter notebook

REST

The Rest-proxy API which is described in the Open API Spec YAML that can be found in Table 9 provides full functionality over the Streaming Component. For example, we can create a new Kafka topic ‘test1’ with the following commands:

```
KAFKA_CLUSTER_ID=$(curl -X GET \
     "http://static.88-198-124-99.clients.your-server.de:8080/v3/clusters/" | jq -
r ".data[0].cluster_id")

curl -X POST \
     -H "Content-Type: application/json" \
     -d "{\"topic_name\":\"test1\",\"partitions_count\":6,\"configs\":[]}" \
     "http://static.88-198-124-99.clients.your-server.de:8080/v3/clusters/${KAFKA_
CLUSTER_ID}/topics" | jq .
```

Then we can produce a few messages that will be stored in this topic with the following command:

```
curl -X POST \
     -H "Content-Type: application/vnd.kafka.json.v2+json" \
     -H "Accept: application/vnd.kafka.v2+json" \
     --data '{"records":[{"key":"jane","value":{"count":0}},{"key":"john","value":
{"count":1}}]}' \
     "http:// static.88-198-124-99.clients.your-server.de:8080/topics/test1" | jq .
```

The response should look similar to the below:

```
{
  "offsets": [
    {
      "partition": 0,
      "offset": 0,
      "error_code": null,
      "error": null
    },
    {
      "partition": 0,
      "offset": 1,
      "error_code": null,
      "error": null
    }
  ],
  "key_schema_id": null,
  "value_schema_id": null
}
```

4.5 AI enhanced service orchestrator

4.5.1 Description

The AI-enhanced Service Orchestrator (AI-SO) facilitates the execution of an application taking into account not only the technical details regarding the deployment of the application (aka deployment description) but also additional requirements provided by the users (i.e., the application providers) as well as their ultimate goals and/or their intents (aka application high level description/ constraints). The latter should be expressed using the elements of the Application Model, while the AI-SO is responsible for its translation and expression using the elements of the Resource Model and the production of possible deployment scenarios taking also into account the Telemetry Data collected from the execution of the same or similar applications in the past. The three aforementioned models are part of the ARDIA framework and have already been described in the Deliverable D5.1 [18] along with the functionality of the AI-SO and its architecture.

In this section, particular focus is given to the AI-SO and its interaction with the other components of the SERRANO platform. Initially, the user provides a JSON document with the parameters of their interest based on the elements of the Application Model. In the JSON message, the user should additionally provide the content of the deployment description

using YAML. The AI-SO further processes the given data (especially the description provided using the terms of the Application Model) and expresses them using the elements of the Resource Model. After this, a unique application ID is created for reference purposes and accordingly the data is provided to the Resource Orchestrator so that it can proceed with the deployment, execution and monitoring of the given application. For making the appropriate decisions during the translation of the application high level description using the Resource Model terms, the AI-SO consumes the services provided by the Central Telemetry Handler and the Telemetry Data collected so far.

It should be noted that the source/binary code of a particular application/service should be already available in the appropriate format(s) (e.g., as a Docker image) prior the execution of this service so that it can be accordingly used by the other components of the SERRANO platform.

4.5.2 Inner components

The AI-SO internally uses several components for providing the requested functionality, that is, the Requests Handler, the Translation and the Forecasting Mechanisms.

The Requests Handler is responsible for interacting with the internal components of the AI-SO as well as the other SERRANO components (i.e., Resource Orchestrator and Central Telemetry Handler). The Central Telemetry Handler is used for retrieving information regarding the past deployment and execution of applications in the resources linked with the SERRANO infrastructure, the resource capabilities along with their status. The Resource Orchestrator is used for the actual deployment of the application.

The Requests Handler internally uses the Translation Mechanism for detecting possible deployment scenarios that can take place as well as the particular constraints that the resources should satisfy using the elements of the Resource Model, so that their deployment is aligned with the initial user's constraints. Through this process, the data provided by the user as well as the telemetry data collected from the SERRANO infrastructure are used. The Requests Handler also uses the Forecasting Mechanism in order to foresee additional requirements and accordingly revises and updates the constraints included in each deployment scenario.

The architecture of the AI-SO was also provided in the Deliverable D2.5 [19].

4.5.3 Integration details and REST APIs

Table 10 Integration details of AI-enhanced Service Orchestrator

IP(s)/Port(s)	147.102.19.66:8080/AISO
Publicly accessible (y/n and other details)	YES
Type of API	REST
Associated host names	ponte.grid.ece.ntua.gr
API documentation	https://gitlab.com/serranoproject/wp5/T5.1/-/blob/main/AISO-Swagger/AI-SO-swagger-v.4.yaml
Location of integration tests	N/A

Figure 18 presents the functionality provided by the AI-SO (Open API).

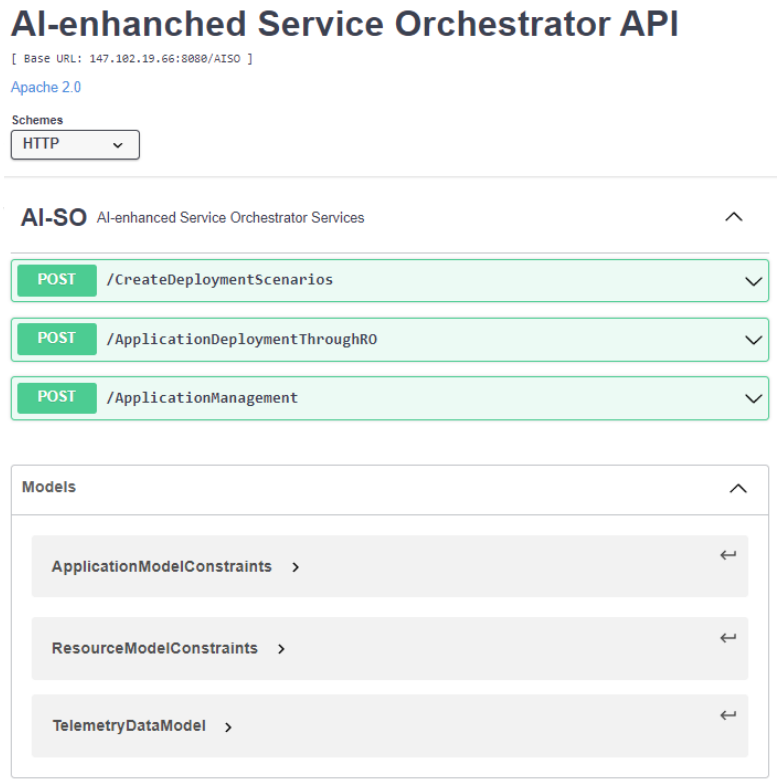


Figure 18: AI-SO Rest Service Open API

The AI-SO provides three services. The first service (mainly used for testing purposes) expresses the application high-level description using the elements of the resource model (taking into account the telemetry data) whereas the second service provides the outcome of the previous process to the Resource Orchestrator for the actual deployment of the

application. The third service enables authorized users to manage their own application such as observing its current status or even un/re-deploy an existing one.

Both input and output of the AI-SO are expressed using the elements included in the ARDIA framework. More precisely, the elements included in the Application and Resource Models are the ones specified in the respective ARDIA abstraction models described in the deliverable D5.1 [18]. The Telemetry Data model is used for the interaction with the Central Telemetry Handler.

In the following three figures (i.e., Figure 19, Figure 20 and Figure 21), there is additional information regarding the input and output of each one of these three services.

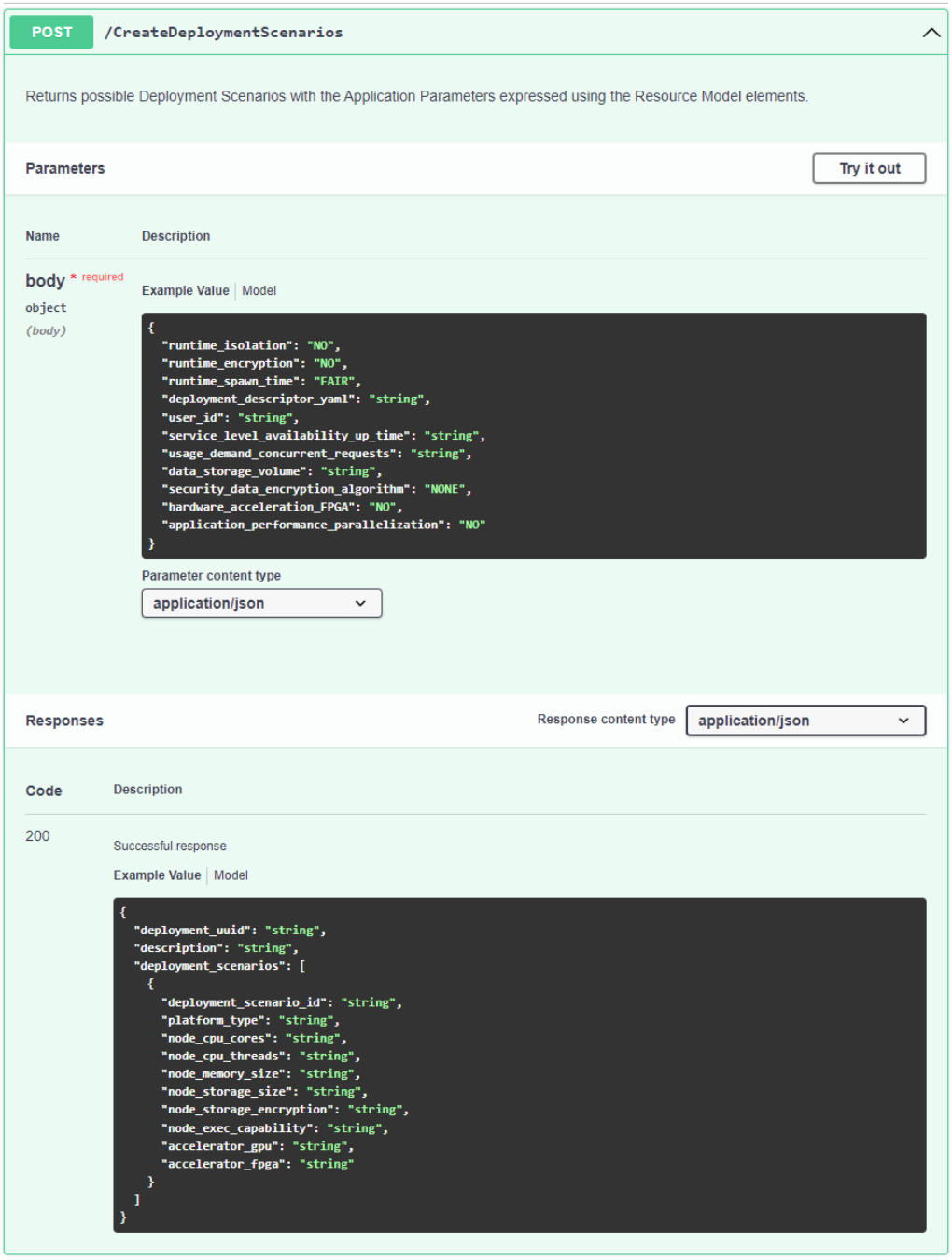


Figure 19: AISO – Service 1: Translate Application Description

The input of the first service is a JSON string with the user constraints expressed using the elements of the Application Model. In this JSON string there is a field regarding the deployment description of this service. The output of this service is another JSON string with the with possible deployment scenarios (in the above image, only one presented) where user constraints are expressed using the elements of the Resource Model.

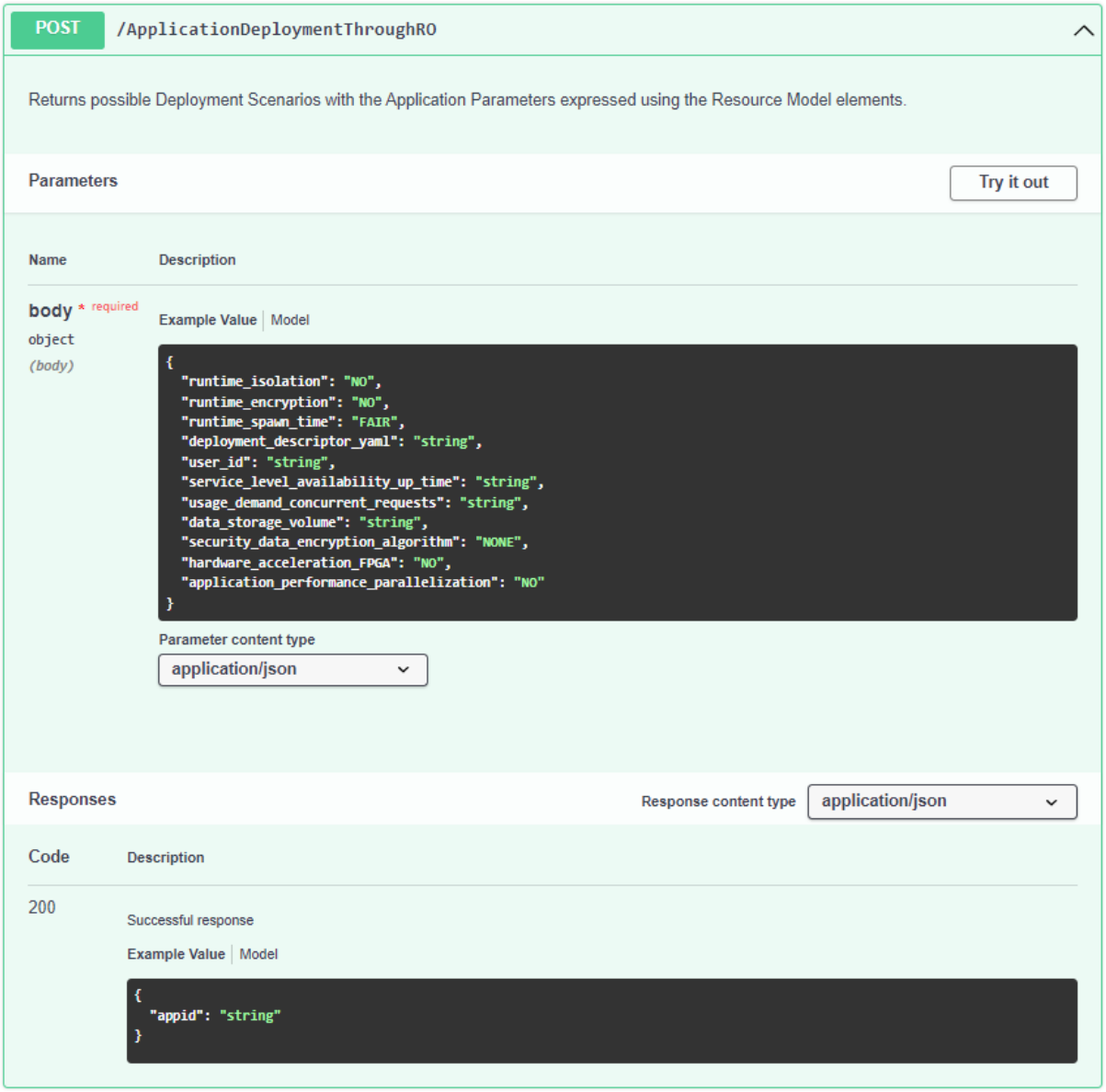


Figure 20: AISO – Service 2: Application Deployment through Resource Orchestrator (RO)

The input of the second service is the same of the first one, i.e., a JSON string with the user constraints expressed using the elements of the Application Model as well as its deployment description. The data provided are initially used by the AI-SO and accordingly by the Resource Orchestrator for the proper application deployment. Hence, the output of this service is a JSON string with the ID of the deployed application.

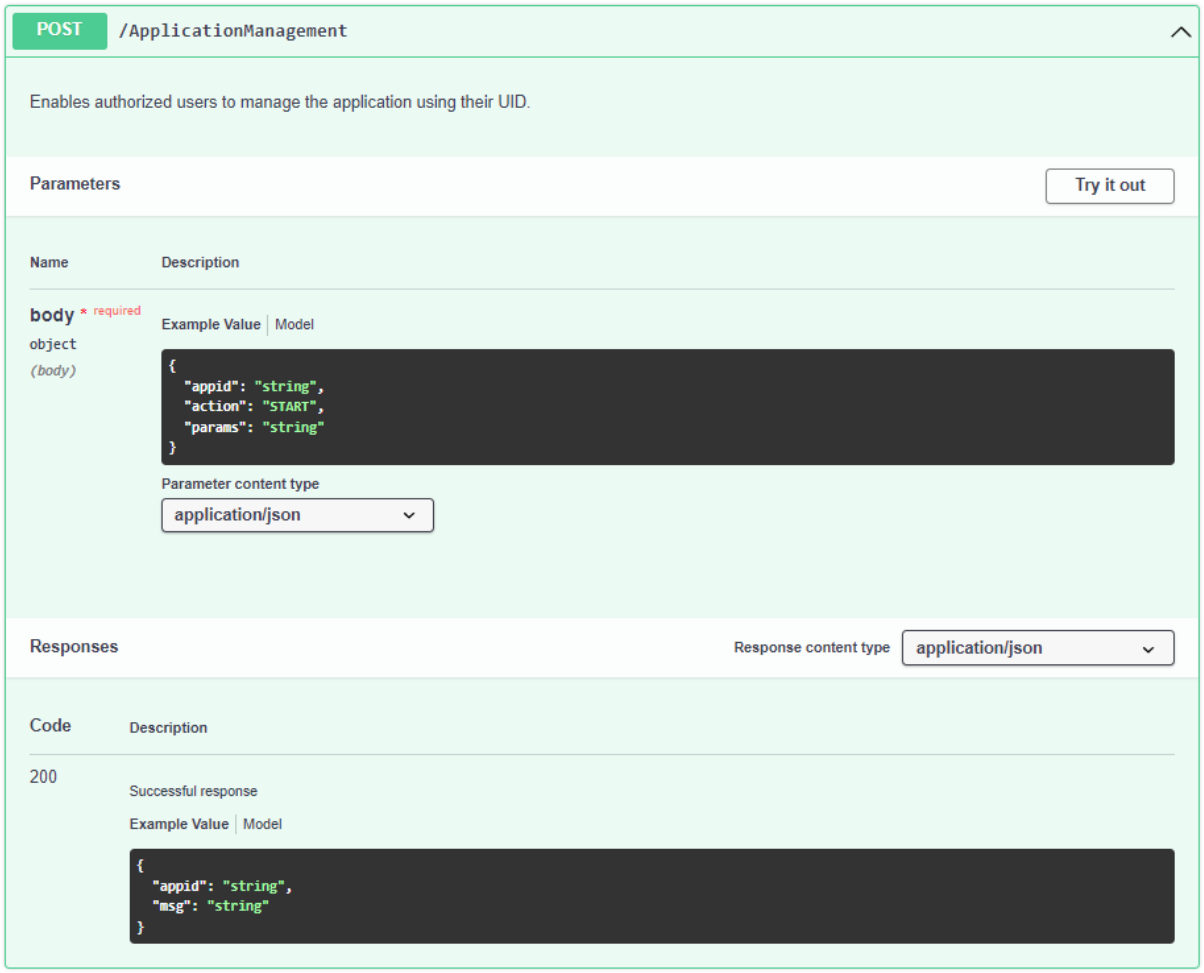


Figure 21 AISO – Service 3: Application Management

The input of the third service is a JSON string with the unique application ID, the actions to be performed (e.g., START, STOP, RESTART, KILL, STATUS) and additional parameters (if any) regarding this action. These actions have to be supported by the Resource Orchestrator and the underlying SERRANO infrastructure. The output is another JSON string with more information regarding the process that took place and its outcome (e.g., the current status of the application).

4.5.3.1 User Constraints

Regarding the expression of the user constraints of each application (and the internal components as well) that is provided as input to the first two services, a simple approach is followed, according to which the desirable set or range of values are specified in the value of a parameter using the following format:

<constraint>: <operator> <term(s)/value(s)>

For example, for specifying that the value of a parameter should be greater than a specific threshold the user should provide “GREATER-THAN ${REAL_VALUE}”. On the other hand, for expressing that the value of a parameter should belong to a particular set, the user should provide “IN-SET ${COMMA_SEPARATED_TERMS}”. The operators used (and the terms as well)

should come from a predefined list to be properly handled. In case the operator is "EQUALS=TO" and is the only one used, it can be omitted. These expressions can be combined using Boolean operators (i.e., AND/OR/NOT) to form more complicated constraints if necessary.

4.5.4 Sample requests and responses

In the following figures, there are two examples. For presentation purposes, Postman [20] application has been used. This software tool provides a GUI so that the existing services can be easily invoked, provided that their URLs along with the format and structure of input messages are already known. For testing and explanation purposes, some indicative constraints that an application should satisfy have been specified using the elements of the Application Model.

In the first example, the deployment scenarios are shown, which are produced by the AI-SO so that the application satisfies certain runtime constraints. It should be noted that only one possible scenario is feasible in this case, due to the particular constraints specified.

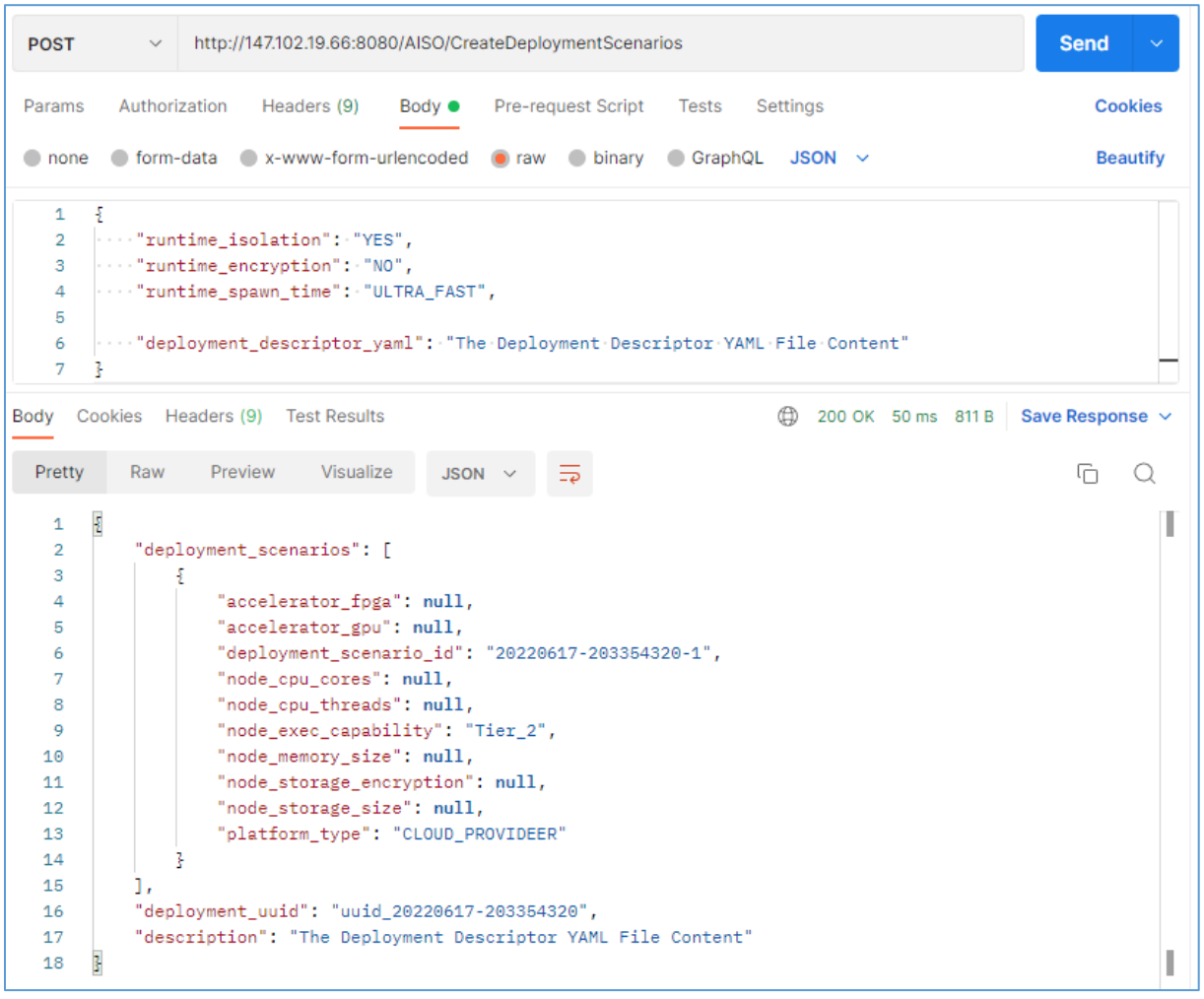


Figure 22: AISO – Request & Response – Example 1

In the above example, the content of the deployment Descriptor YAML file has been deliberately omitted.

In the second example, the response retrieved from the Resource Orchestrator is shown. In this example, the deployment was successful and hence the Resource Orchestrator just provided back the application's unique ID. In case of an error, a JSON message with the issue(s) encountered is returned.

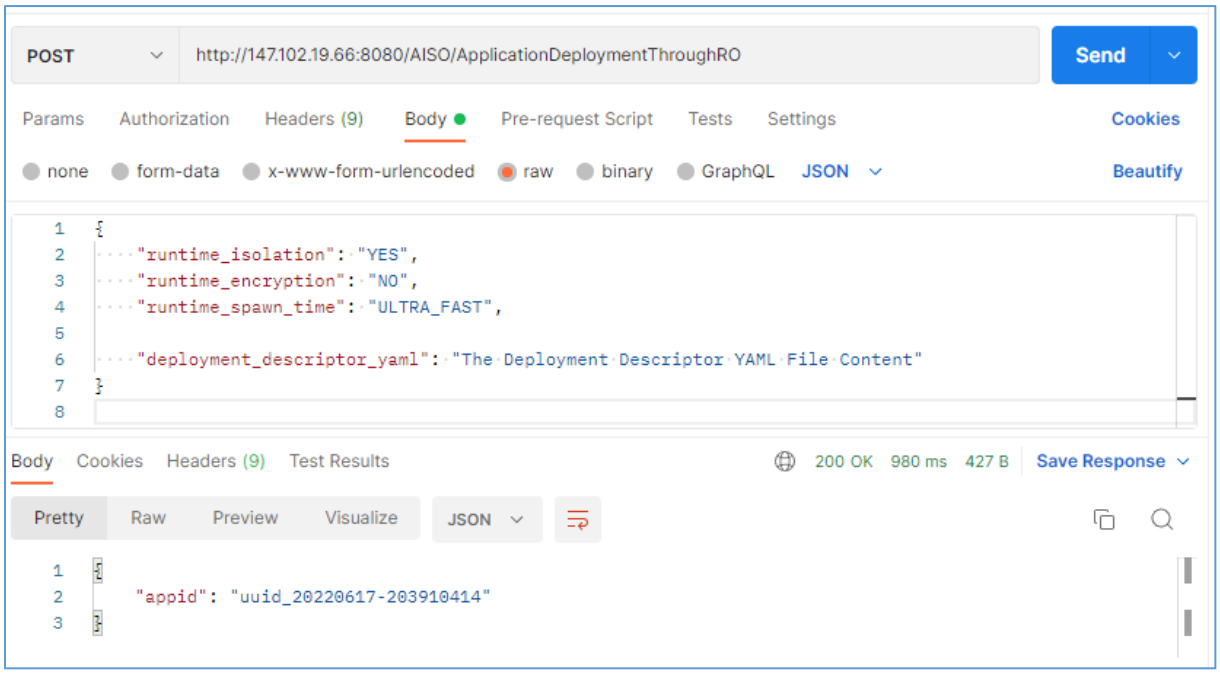


Figure 23: AISO – Request & Response – Example 2

4.6 HPC System Hardware Interface

The HPC System Hardware Interface, or HPC Gateway, is the intermediate component between the SERRANO's HPC services (WP4), the Intelligent Service and Resource Orchestration Layer (WP5) and the HPC infrastructure. The HPC Gateway supports popular batch jobs schedulers, such as Slurm [21] and PBS-based (e.g., TORQUE [22], OpenPBS [23]). Further information can be found in deliverable D4.2 [24].

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

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

Therefore, SERRANO HPC Gateway communicates with the front-end (login) nodes via SSH and uses commands specific to the resource managers under use in order to prepare a batch job script for submission (i.e., to select the appropriate header, see D4.2 [24]), submit the job and monitor the status of the job and the partitions, as shown in Figure 24.

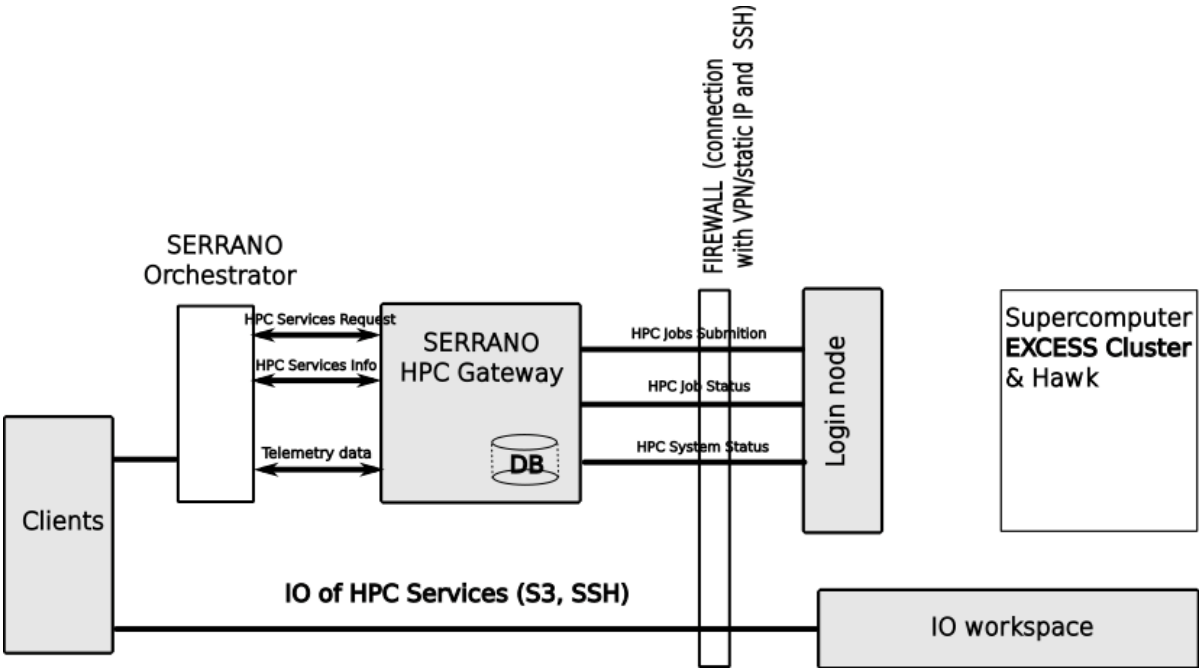


Figure 24: Interaction between HPC Gateway and HPC infrastructure

4.6.1 Integration details and REST APIs

4.6.1.1 Integration details

The HPC System Hardware Interface (HPC Gateway) is integrated with the SERRANO platform and exposes REST API endpoints needed for the Resource Orchestrator and Telemetry Framework for execution of HPC services (or HPC kernels) and monitoring the state of the HPC infrastructure. The HPC Gateway is implemented as a service and interacts with the target HPC infrastructure using SSH protocol (as shown in Figure 24). The administrator maintains SSH keys that will be used for authentication with the infrastructure.

Table 11: Integration details of SERRANO HPC System Hardware Interface

Host name(s)/Port(s)	hpc-interface.serrano.cs.uvt.ro:8080
Publicly accessible (y/n and other details)	The IP is publicly accessible, but the access has been restricted through token authentication.
Type of API	REST API
API documentation	https://gitlab.com/serranoproject/wp5/hpc-interface/-/blob/main/openapi-spec.yaml
Location of integration tests	https://gitlab.com/serranoproject/wp5/hpc-interface/-/blob/main/Jenkinsfile

Figure 25: REST API endpoints exposed by HPC system hardware interface

4.6.1.2 Sample requests and responses

List of available HPC services

Using the following request, the list of available HPC services is returned. The list is being updated, once the service is deployed on the target HPC system. In this sample, Kalman, Min-Max and FFT filters are returned.

```
GET /services
```

```
[
  {
        "id": "75f54ad7-caad-4c70-9227-f0395f30dc5d",
        "name": "kalman_filter",
        "type": "filter",
        "version": "0.0.1"
  },
  {
        "id": "5cb99ee3-36cf-4015-872f-f6227a429202",
        "name": "min_max_filter",
        "type": "filter",
```

```
        "version": "0.0.1"
  },
  {
        "id": "6b0489a9-6e64-4f6c-b4e8-b39f71b18a76",
        "name": "fft_filter",
        "type": "filter",
        "version": "0.0.1"
  }
]
```

HPC infrastructure management and telemetry

An administrator can provide infrastructure details to the HPC Gateway, which include unique name, hostname, scheduler type and SSH authentication. The Gateway will then access the HPC infrastructure using this information.

```
POST /infrastructure
Body:
{
  "host": "A.B.C.D",
  "hostname": "infrastructure.example.com",
  "name": "cluster_name",
  "scheduler": "slurm",
  "ssh_key": {
        "password": "password",
        "path": "/path/to/private/key",
        "type": "ssh-ed25519"
  },
  "username": "username"
}
```

```
{
  "host": "A.B.C.D",
  "hostname": "infrastructure.example.com",
  "name": "cluster_name",
  "scheduler": "slurm"
}
```

The unique name of the infrastructure can then be used to retrieve the telemetry information about the HPC infrastructure and its partitions (also known as queues). Some of the metrics include the total and available number of nodes, CPUs, and number of running and queued jobs in the particular partition.

```
GET /infrastructure/cluster_name/telemetry
```

```
{
  "host": "A.B.C.D",
  "hostname": "infrastructure.example.com",
  "name": "cluster_name",
```

```
  "partitions": [
        {
        "avail_cpus": 158,
        "avail_nodes": 1,
        "name": "profile",
        "queued_jobs": 0,
        "running_jobs": 1,
        "total_cpus": 160,
        "total_nodes": 2
        }
  ],
  "scheduler": "slurm"
}
```

Job submission and retrieval

HPC Gateway exposes endpoints for execution of the HPC services as batch jobs as well as monitoring the status of the job, whether it is still queued, running or completed.

```
POST /job
Body:
{
  "infrastructure": "cluster_name",
  "params": {},
  "services": [
        { "name": "kalman_filter", "version": "0.0.1" },
        { "name": "fft_filter", "version": "0.0.1" }
  ]
}
```

```
{
  "id": "6f67b9b4-1821-41df-991f-c7fbdfc7f959",
  "infrastructure": "cluster_name",
  "scheduler_id": "1732",
  "status": "queued"
}
```

```
GET /job/6f67b9b4-1821-41df-991f-c7fbdfc7f959
```

```
{
  "id": "6f67b9b4-1821-41df-991f-c7fbdfc7f959",
  "infrastructure": "cluster_name",
  "scheduler_id": "1732",
  "status": "running"
}
```

4.7 HW Acceleration Abstractions and Trusted Virtualizations

4.7.1 HW Acceleration abstractions

The SERRANO platform provides several abstractions to facilitate the employment of hardware accelerators both during the development of the accelerated kernels (through the optimization toolkits), as well as during the deployment (transparent deployment on the underlying accelerator device). Below we provide a short description of these abstractions, both regarding the development as well as the deployment phase.

Abstractions for development of fine-tuned FPGA and GPU accelerated kernels

During development, end-users are able to utilize the HW acceleration optimization toolkits (Plug&Chip extensions and DMM4FPGA) to obtain device-aware efficient accelerated kernels. Regarding the Plug&Chip extensions for optimized FPGA and GPU kernels, SERRANO provides a set of Python and Perl scripts that perform the kernel optimizations in an automated manner.

In the case of the Plug&Chip optimization toolkit extension for FPGAs, end-users give as input the source code of a C/C++ kernel annotated with labels in all the action points (i.e., loops and arrays). In addition, information regarding the kernel top level function, the iterations of each loop and the size of the dimensions of each array must be provided. Finally, the end-user specifies the device and the target clock frequency. The proposed methodology provides a set of optimal kernels with added HLS directives as well as information concerning their trade-offs. More information concerning the proposed methodology can be found in deliverable D4.3 [25]. The following example demonstrates how end-users can use this tool to optimize an algorithm (e.g., the Kalman filter) as well as the expected output.

```
$ python3 automatic_optimizer.py --INPUT_SOURCE_PATH ./kalman.cpp --
INPUT_SOURCE_INFO_PATH ./kernel_info.txt
```

n_gen	n_eval	cv (min)	cv (avg)	n_nds	eps	indicator
1	40	0.00000E+00	6.33750E+01	1	-	-
2	80	0.00000E+00	4.075000000	2	1.000000000	ideal
3	120	0.00000E+00	3.575000000	3	0.157894737	ideal
4	160	0.00000E+00	2.775000000	4	0.013167812	ideal
5	200	0.00000E+00	1.775000000	4	0.00000E+00	f
6	240	0.00000E+00	0.00000E+00	6	0.133333333	nadir
7	280	0.00000E+00	0.00000E+00	6	0.333333333	ideal
8	320	0.00000E+00	0.00000E+00	7	0.043636645	f
9	360	0.00000E+00	0.00000E+00	13	0.640000000	nadir
10	400	0.00000E+00	0.00000E+00	11	0.038461538	ideal
11	440	0.00000E+00	0.00000E+00	8	0.111111111	ideal
12	480	0.00000E+00	0.00000E+00	8	2.000000000	nadir
13	520	0.00000E+00	0.00000E+00	8	0.041666667	ideal
14	560	0.00000E+00	0.00000E+00	10	0.881355932	nadir
15	600	0.00000E+00	0.00000E+00	9	0.002525253	f
16	640	0.00000E+00	0.00000E+00	10	0.004918052	ideal
17	680	0.00000E+00	0.00000E+00	13	7.428571429	nadir
18	720	0.00000E+00	0.00000E+00	16	0.020833333	ideal
19	760	0.00000E+00	0.00000E+00	20	0.009817525	f
20	800	0.00000E+00	0.00000E+00	27	0.002936051	ideal
21	840	0.00000E+00	0.00000E+00	28	0.021276596	ideal
22	880	0.00000E+00	0.00000E+00	35	0.175000000	nadir
23	920	0.00000E+00	0.00000E+00	35	0.166666667	nadir
24	960	0.00000E+00	0.00000E+00	31	3.000000000	nadir

Figure 26: Genetic algorithm output

In the optimized directory that is created after the tool's execution, users can find the set of optimal kernels with added HLS directives (e.g., optimized_1.cpp) as well as information concerning their trade-offs (in info.csv file).

In the case of Plug&Chip optimization toolkit extension for GPUs, users provide to the tool their CUDA written application, its workload input size, and the GPU specification that they are interested in using. Then, the tuning model automatically produces and executes the optimal block coarsened version of the given CUDA kernel in terms of performance for the specific device. The autotuning tool mainly consists of a script (bash script) that extracts the static features of the given CUDA kernel through compiling and static analysis, a regression ML model (written in Python) that predicts the execution latency of the kernel on the device and a source-to-source compiler (written in Perl) that performs the application of the block coarsening transformation to the CUDA kernel. The following example demonstrates how end-users can use the proposed autotuning tool to optimize an initial CUDA C application (e.g., GEMM) as well as the expected output.

```
$ ./extract_static_features.sh gemm.cu > gemm_features.csv
$ ./merge_static_and_hw_features.sh gemm_features.csv gpu_features.csv >
features.csv
$ python3 predict.py features.csv > estimated_latencies.csv
$ perl block_coarsen.pl estimated_latencies.csv gemm.cu > gemm_tuned.cu
```

kernel	input size x	input size y	GPU	BC	estimated latency
gemm	256	256	TX1	1	4.2
gemm	256	256	TX1	2	3.5
gemm	256	256	TX1	4	2.7
gemm	256	256	TX1	8	4.1
gemm	256	256	TX1	16	8.6

Figure 27: Output of CUDA auto-tuning framework

Finally, concerning the DMM4FPGA framework, several abstractions have been developed to allow the runtime allocation/de-allocation of on-chip memory, providing functionalities similar to the glibc malloc/free functions. The framework's API consists of **4 functions** that are used by the designer in the accelerator's High-Level Synthesis (HLS) source code for accessing the dynamic memory capabilities (more information regarding the purpose of each function have been reported in D4.3 [25] and will be extended within D4.4 at M30). This framework is implemented on the FPGAs Programmable Logic (PL) along with the accelerators:

Functions for memory allocation/de-allocation
NAME memAlloc(size_t ***size***, uint ***heap_id***, uint_t* ***selected_heap***) **SYNOPSIS** size_t ***size***: Allocation size in bytes. uint ***heap_id***: ID of the heap that shall allocate the requested bytes. uint_t* ***selected_heap***: The ID of the heap that served this allocation request. **DESCRIPTION** This function returns the base address of the newly allocated region. **EXAMPLE** int A[100]; ➔ int* A=(int*)memAlloc(100*sizeof(int),0,&heapA);
NAME memFree(void* ***base_address***, uint_t ***size***, uint_t ***selected_heap***) **SYNOPSIS** **void* *base_address***: The base address of the allocated region. **uint_t *size***: The size in bytes of the allocated region **uint_t *selected_heap***: The ID of the heap where the selected memory region is allocated **DESCRIPTION** This function is used for de-allocating an occupied memory region. **EXAMPLE** memFree((void*) A, 100*sizeof(int), 0)

Functions for memory optimization – defragmentation (optional)
NAME update(void* ***base_address***, uint_t ***size***, uint_t ***selected_heap***) **SYNOPSIS** **void* *base_address***: The base address of the allocated region. **uint_t *size***: The size in bytes of the allocated region **uint_t *selected_heap***: The ID of the heap where the selected memory region is allocated **DESCRIPTION**

```
      This function returns the new address in case that the garbage collection
mechanism has been  executed
EXAMPLE
      update((void*) A, 100*sizeof(int), 0)
```

```
NAME
      checkFlag(uint_t selected_heap)
SYNOPSIS
      uint_t selected_heap: The ID of the heap where the selected memory region
is allocated
DESCRIPTION
      This function is used for checking if the garbage collector is currently
executed
EXAMPLE
      checkFlag(0)
```

Abstractions for seamless deployment HW accelerated kernels

To facilitate hardware-accelerated application deployment in the end-to-end SERRANO platform, the consortium employs a Serverless computing model, where applications are broken down to individual functions that can be accelerated. Additionally, decoupling the hardware-specific part of the application/function from the function call is key to leverage the heterogeneity inherent in a diverse, Cloud-Edge infrastructure, such as the one present in SERRANO.

To achieve this, NBFC develops a hardware acceleration framework, tailored to serverless computing, vAccel. More information about vAccel can be found on D4.3 [25].

Developers for hardware accelerators can easily port their applications to the vAccel programming framework by using the following abstractions:

- Static user API: add an API call for the needed functionality to vAccel, and map it to a relevant plugin that implements this functionality and calls the hardware-specific code (either through a shared library or natively)
- Dynamic user API: port the existing application to the vAccel generic operation.

4.7.1.1 Integration details and APIs

Besides the implemented scripts that end-users can utilize offline on their local machine, the Plug&Chip extensions and DMM4FPGA framework are also integrated on SERRANO and exposed as services over Kubernetes. Figure 28 shows how SERRANO end-users can leverage these services to optimize their C/C++ source codes. Specifically, the communication between the client (end-user) and the server (K8S service) is implemented using TCP socket interfaces. To facilitate the employment of these services, a Python package has been implemented, which contains a set of wrapper functions, that abstract the communication part and deliver a clean programming interface to SERRANO's users. To use this functionality, users need to download the respective python package and install it on their local machine. Once the package is installed, they are able to use the developed functions, as shown in the example below.

```
import serrano_api as serrano

INPUT_KERNEL_PATH="./kernel_default.cpp"
OPTIMIZED_KERNEL_PATH="./kernel_default_received.cpp"

# Open *.cpp file that contains the kernel to be optimized
fp = open(INPUT_KERNEL_PATH,'r')

optimized_kernel = serrano.optimize_kernel(fp)

with open(OPTIMIZED_KERNEL_PATH,'w') as fn:
      fn.write(optimized_kernel)

fn.close()
```

The serrano.optimize_kernel function basically abstracts all the communication related part from the users, thus, simplifying the process.

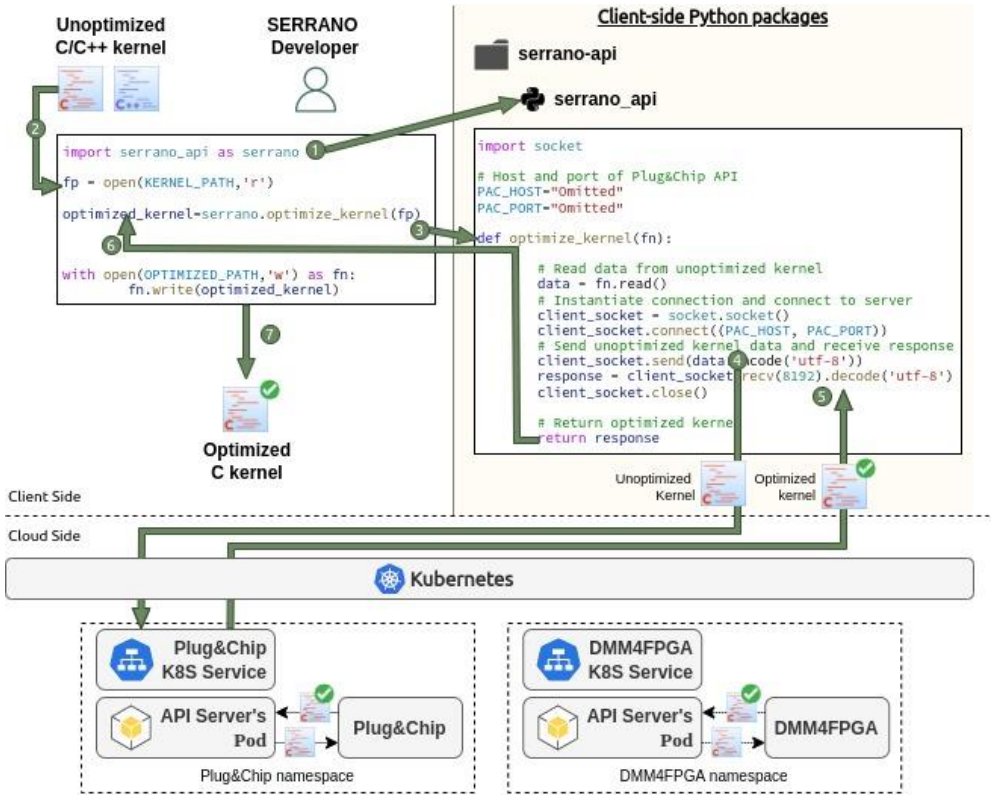


Figure 28. Accessing the exposed services for automatic GPU/FPGA optimizations

To use the vAccel static user API, the user needs to:

- Issue a function call via their application
- Choose (or implement) the respective plugin that implements that function.

An example executing a dummy, no-op function is shown below:

```
#include <stdlib.h>
#include <stdio.h>
#include <vaccel.h>

int main()
{
        int ret;
        struct vaccel_session sess;

        ret = vaccel_sess_init(&sess, 0);
        if (ret != VACCEL_OK) {
                fprintf(stderr, "Could not initialize session\n");
                return 1;
        }

        printf("Initialized session with id: %u\n", sess.session_id);

        ret = vaccel_noop(&sess);
        if (ret)
                fprintf(stderr, "Could not run op: %d\n", ret);

        if (vaccel_sess_free(&sess) != VACCEL_OK) {
                fprintf(stderr, "Could not clear session\n");
                return 1;
        }
        return ret;
}
```

To execute this operation, the vAccel library has to be present, and the relevant plugin needs to be specified. For instance, executing the no-op operation with the no-op plugin (debug example):

```
# VACCEL_DEBUG_LEVEL=4 VACCEL_BACKENDS=./plugins/noop/libvaccel-noop.so ./app
2022.06.21-09:10:03.48 - <debug> Initializing vAccel
2022.06.21-09:10:03.48 - <debug> Registered plugin noop
2022.06.21-09:10:03.48 - <debug> Registered function noop from plugin noop
2022.06.21-09:10:03.48 - <debug> Loaded plugin noop from
./plugins/noop/libvaccel-noop.so
2022.06.21-09:10:03.48 - <debug> session:1 New session
Initialized session with id: 1
2022.06.21-09:10:03.48 - <debug> session:1 Looking for plugin implementing noop
2022.06.21-09:10:03.48 - <debug> Found implementation in noop plugin
Calling no-op for session 1
2022.06.21-09:10:03.48 - <debug> session:1 Free session
2022.06.21-09:10:03.48 - <debug> Shutting down vAccel
2022.06.21-09:10:03.48 - <debug> Cleaning up plugins
2022.06.21-09:10:03.48 - <debug> Unregistered plugin noop
```

To use the dynamic user API with vAccel, the process is as follows:

- tweak the existing application and expose it as a shared library (libify)
- wrap the function call with the vAccel exec call.

For instance, for an example vector_add operation shown below:

```
int vector_add (int *A, int* B, int* C, int dimension)
{

	int i = 0;

	for (i=0;i<dimension;i++)
		C[i]=A[i]+B[i];

	return 0;
}

int main(int argc, char **argv)
{
	int A[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
	int B[] = { 0, 1, 2, 0, 1, 2, 0, 1, 2, 0 };
	int C[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
	int dimension = sizeof(A)/sizeof(int);
	int i = 0;

	int ret = vector_add(A, B, C, dimension);
	if (ret) {
		printf("Error running operation\n");
		goto out;
	}

	return 0;
}
```

We libify the existing implementation by adding an argument unpack function, removing the main function:

```
int vector_add (int *A, int* B, int* C, int dimension)
{

	int i = 0;

	for (i=0;i<dimension;i++)
		C[i]=A[i]+B[i];

	return 0;
}

struct vector_arg {
	uint32_t len;
	uint8_t *buf;
};

int vector_add_unpack(void*out_args, size_t out_nargs, void* in_args, size_t
in_nargs)
{

	int dimension = *(int*)out_arg[2].buf;
	int *A = (int*)out_arg[0].buf;
	int *B = (int*)out_arg[1].buf;
	int *C = (int*)in_arg[0].buf;
```

```
	int ret = vector_add(A, B, C, dimension);
	if (ret) {
		printf("Error running operation\n");
		goto out;
	}

	return 0;
}
```

and building it as a shared object:

```
gcc -Wall -shared -o libvector_add.so -fPIC vector_add.c
```

We then create the user-facing wrapper function that calls the vector_add operation through the vAccel exec API call:

```
int vector_add (int *A, int* B, int* C, int dimension)

struct vector_arg {
	size_t len;
	uint8_t *buf;
}

char *lib = "libvector_add.so";
char *op = "vector_add";
op_arg[0].size = dimension * sizeof(int);
op_arg[0].buf = (uint8_t*)A;
op_arg[1].size = dimension * sizeof(int);
op_arg[1].buf = (uint8_t*)B;
op_arg[2].size = sizeof(int);

op_arg[2].buf = (uint8_t*)&dimension;
op_arg[3].size = dimension * sizeof(int);
op_arg[3].buf = (uint8_t*)C;

ret = vaccel_exec(.., lib, op, &op_arg[0], 3, &op_arg[3], 1);
```

The vAccel API is currently under development. A working prototype is shown below:

```
Image classification

int vaccel_image_classification(struct vaccel_session *sess, const void *img,
	unsigned char *out_text, unsigned char *out_imgname,
	size_t len_img, size_t len_out_text, size_t len_out_imgname);
```

- struct vaccel_session *sess: a pointer to a vAccel session created using vaccel_sess_init();
- const void *img: the buffer holding the data to the input image.
- unsigned char *out_text: the buffer holding the classification tag output

- unsigned char *out_imgname: the name of the processed image, created as a session resource (EXPERIMENTAL).
- size_t len_img: the size of img in bytes.
- size_t len_out_text: the size of out_text in bytes.
- size_t len_out_imgname: the size of out_imagename in bytes.

Image segmentation

```
int vaccel_image_segmentation(struct vaccel_session *sess, const void *img,
        const unsigned char *out_imgname, size_t len_img,
        size_t len_out_imgname);
```

- struct vaccel_session *sess: a pointer to a vAccel session created using vaccel_sess_init();
- const void *img: the buffer holding the data to the input image.
- unsigned char *out_imgname: the name of the processed image, created as a session resource (EXPERIMENTAL).
- size_t len_img: the size of img in bytes.
- size_t len_out_imgname: the size of out_imagename in bytes.

Object detection

```
int vaccel_image_detection(struct vaccel_session *sess, const void *img,
        const unsigned char *out_imgname, size_t len_img,
        size_t len_out_imgname);
```

- struct vaccel_session *sess: a pointer to a vAccel session created using vaccel_sess_init();
- const void *img: the buffer holding the data to the input image.
- unsigned char *out_imgname: the name of the processed image, created as a session resource (EXPERIMENTAL).
- size_t len_img: the size of img in bytes.
- size_t len_out_imgname: the size of out_imagename in bytes.

BLAS library functions

Matrix-to-matrix multiplication

```
int vaccel_sgemm(struct vaccel_session *sess, uint32_t k, uint32_t m,
        uint32_t n, size_t len_a, size_t len_b, size_t len_c,
        float *a, float *b, float *c);
```

- struct vaccel_session *sess: a pointer to a vAccel session created using vaccel_sess_init().
- uint32_t k: first dimension of matrix A & matrix C.
- uint32_t l: second dimension of matrix A & first dimension of matrix B.
- uint32_t n: second dimension of matrix B & matrix C.

```
        • size_t len_a: size of matrix A in bytes.
        • size_t len_b: size of matrix B in bytes.
        • size_t len_c: size of matrix C in bytes.
        • float *a: pointer to matrix A.
        • float *b: pointer to matrix B.
        • float *c: pointer to matrix C.

Generic functions

Noop

int vaccel_noop(struct vaccel_session *sess);

        • struct vaccel_session *sess: a pointer to a vAccel session created
          using vaccel_sess_init().

Exec

int vaccel_exec(struct vaccel_session *sess, const char *library,
                const char *fn_symbol, struct vaccel_arg *read,
                size_t nr_read, struct vaccel_arg *write, size_t nr_write);

        • struct vaccel_session *sess: a pointer to a vAccel session created
          using vaccel_sess_init().
        • const char *library: name of the shared object to open.
        • const char *fn_symbol: name of the symbol to dereference in the
          shared object library.
        • struct vaccel_arg *read: pointer to an array of struct
          vaccel_arg read-only arguments to fn_symbol.
        • size_t nr_read: number of elements for the read array.
        • struct vaccel_arg *write: pointer to an array of struct
          vaccel_arg write-only arguments to fn_symbol.
        • size_t nr_write: number of elements for the write array.
```

4.7.2 Trusted Virtualizations

The SERRANO consortium develops and enhances software and hardware techniques to enable secure and trusted execution on diverse and heterogeneous infrastructure. In SERRANO, we build security tiers that allow the user through the orchestrator to choose the level of trust of the part of the infrastructure their application is going to be deployed.

Integration details

In SERRANO, we build on the confidential computing paradigm to provide end-to-end secure tiers. During the initial design and provisioning phases, we have identified the following security levels that comprise the SERRANO secure infrastructure layer:

- ***Tier-0: No additional security, trustiness or enhanced isolation*** => execution through default containers
- ***Tier-1: More isolated execution environment but no advanced security or trustiness*** => execution through containers in micro-VMs (sandboxing)
- ***Tier-2: More secure execution, better isolation but no advanced trustiness =>*** execution through unikernels that reduce attack surface and provide ultra-fast boot
- ***Tier-3: Advanced security and trustiness, default isolation*** => execution through container with secure boot and trusted execution extensions
- ***Tier-4: Maximum security, trustiness and isolation*** => execution through container with secure boot and trusted execution extensions and within a sandboxed micro-VM

Table 12 summarizes the provided functionality, the different security and trust levels that each deploy method provides and their trade-offs.

Table 12: Security Tiers for the SERRANO platform

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

We implement these tiers using a combination of node labelling and hardware signature footprints on the infrastructure side and descriptor annotation on the orchestration side. Although the final implementation is yet to be delivered (M30), the first iteration of labelling and workload instantiation for Tiers 0-2 is already in place and Tier 3 is expected to be available by M21. More details are available in D3.3 [26].

4.8 Secure Storage Service on-premises gateway and TLS offloading

4.8.1 Secure Storage Service

The Secure Storage Service, also referred to in other deliverables as SERRANO-enhanced Storage Service, is the SERRANO platform's main storage solution. It provides an S3-compatible storage API that is easy to integrate with existing software.

The Secure Storage Service is built around SkyFlok, a file storage and sharing solution created Chocolate Cloud. SkyFlok is an online service that distributes data to several cloud locations of the user's choosing. This gives it better reliability, availability, performance and cost-effectiveness compared to single-cloud solutions. All data is encrypted, and erasure coded before being distributed to the cloud locations. SkyFlok is only accessible through a browser at www.skyflok.com.

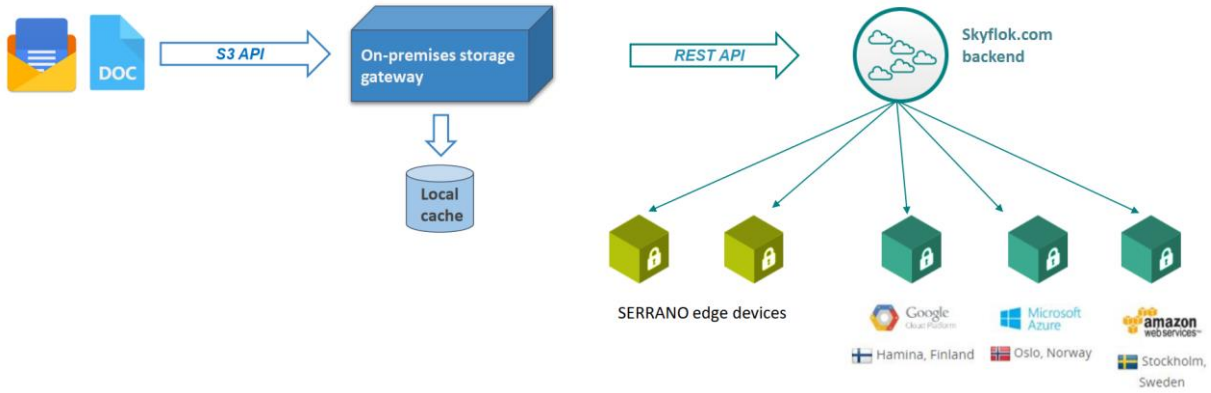


Figure 29: The components of the Secure Storage Service

The Secure Storage Service extends SkyFlok with features aimed at medium-to-large companies. Firstly, it allows the selection of edge storage locations. This has the potential to greatly enhance the service's performance, especially the latency of smaller requests. In practice, this is achieved by taking advantage of the existing on-premises infrastructure enterprise customers can deploy or may already have. Secondly, an S3-compatible interface is introduced that allows for easy integration with existing software. This also makes migration from Amazon's AWS or one of the many smaller solutions that support S3 [27] seamlessly. This also includes on-premises object stores hosted using Ceph [28], Openstack Swift [29] or MinIO [2].

4.8.1.1 Inner components

The Secure Storage Service has been described in greater detail in Deliverable 2.4 [30] and Deliverable 3.2 [31]. Here, we only include a short excerpt. An overview of how the components are connected can be seen on Figure 29.

The **On-premises storage gateway** (Gateway) is the key new development of the Secure Storage Service as well as its most important on-premises component. It is implemented in Python 3.8 using FastAPI [32], a modern ASGI framework. The Gateway runs as a containerized application and can be deployed using the SERRANO orchestration mechanisms. Because it manages no state beyond caching some data for performance reasons, multiple instances can be deployed simultaneously. This makes it possible to tailor the performance of the SERRANO-enhanced Storage Service to the current workload by scaling horizontally. Its statelessness is a key design principle meant to ensure that the Gateway does not become a single point of failure or a performance bottleneck.

An important consideration related to performance is CPU usage. Since all data processing operations are performed by the Gateway, acceleration techniques developed in Work Packages 3 and 4 are used to remove some of the burdens from the CPU. These come in effect if specialised hardware (Nvidia DPU, GPU, FPGA) is available. If they are not available, the Gateway performs these tasks using the CPU. The Gateway features another performance-enhancing feature in the form of a local cache. Thus, files that have been accessed recently or

are very popular are kept in their original, uncompressed, unencrypted, non-erasure-coded form on local, ephemeral storage. Like the other design decisions made when developing the service, the cache aims to improve performance beyond what a purely cloud-based solution can achieve.

The **SERRANO edge devices** provide storage locations at the edge, on the customer's premises. Like the storage service itself, they provide an S3 interface. However, the client applications do not access the devices directly. Instead, the Gateway oversees all file uploads and downloads to both edge and cloud storage locations.

Each SERRANO edge device is a containerized application deployed using the SERRANO orchestration mechanisms. Each is a separate instance of MinIO [2], a high-performance, highly customizable object storage solution. It includes a telemetry agent that is used to provide the Telemetry Service with information regarding the status of the storage resource in use.

When deployed using Kubernetes (K8s) [33], MinIO can make use of a wide range of available storage resources through K8s Persistent Volumes [34]. All information required to run MinIO, as well as all data that it stores can be mounted using this technique. Thus, it is easy to tailor MinIO to the storage resources that are available on the customer's premises.

The Secure Storage Service relies on the software infrastructure behind SkyFlok, the **Skyflok.com backend**, for a wide range of features. These can be grouped as follows:

- File system management
- Storage location management
- Generating pre-signed upload and download links
- Storage policy management
- File and metadata consistency checking
- Authentication and authorization
- User and team management

SkyFlok is a next-generation file sharing and storage solution for users who care deeply about privacy and security. It is a multi-cloud platform that distributes data across a wide range of commercially available clouds. Beyond the big three of Amazon, Google and Microsoft, SkyFlok supports most major EU cloud providers and can be configured to be GDPR compliant (Total of 54+ GDPR-compliant Cloud locations). A key enabler of this is the ability provided to users to select the cloud providers that will store their data as well as the actual locations down to the city level. Internally, SkyFlok's secret sauce is network coding, an erasure code that provides reliable service even if a cloud provider becomes unavailable. It also offers protection from data loss and gives privacy benefits beyond those provided by conventional encryption.

4.8.1.2 Integration details and REST APIs

Deliverable 3.2, Section 4.1 previously described the Secure Storage API. The text in this section is based on this deliverable.

The Secure Storage API provides SERRANO users with a way to store and retrieve files. It is based on what can be considered the industry standard for object storage: Amazon Web Services S3. The decision to use a well-known API brings significant benefits, as it allows users to seamlessly integrate their existing software solutions with the SERRANO platform. There are S3 client libraries for most programming languages along with countless development tools for all common operating systems.

Amazon's S3 service offers object storage. Objects are immutable, versioned entities that have a key as a unique identifier and may have other metadata associated with them. Objects are organized into buckets, which have a name that is unique across the system and may also have metadata associated with them. There are several distinctions between file systems and object storage solutions. However, cloud storage solutions like Dropbox and Google Drive have shown that object storage semantics and characteristics, while being somewhat less permissive than those of file systems, are suitable for file storage. Indeed, SkyFlok also builds on this observation to offer its users a file system that is built on top of object storage.

Host names (s)/Port(s)	on-premises-storage-gateway.serrano.cs.uvt.ro Accessible through port 2525, with s3 prefix
Publicly accessible (y/n and other details)	The IP is not publicly accessible, and authentication is performed using AWS Signature V4. Chocolate Cloud will make credentials available to all partners who wish to access the service.
Type of API	REST, XML responses
API documentation	https://gitlab.com/serranoproject/wp3/on-premise-storage-gateway/-/blob/master/openapi.json
Location of integration tests	https://gitlab.com/serranoproject/wp3/on-premise-storage-gateway/-/tree/master/tests/s3

At the time of writing this document, the Secure Storage API supports all major Create, Read, Update, Delete (CRUD) features of both objects and buckets in their simplest form (support for only the compulsory parameters). An API reference can be found on Amazon's website [35]. Figure 30 shows the list of supported endpoints.

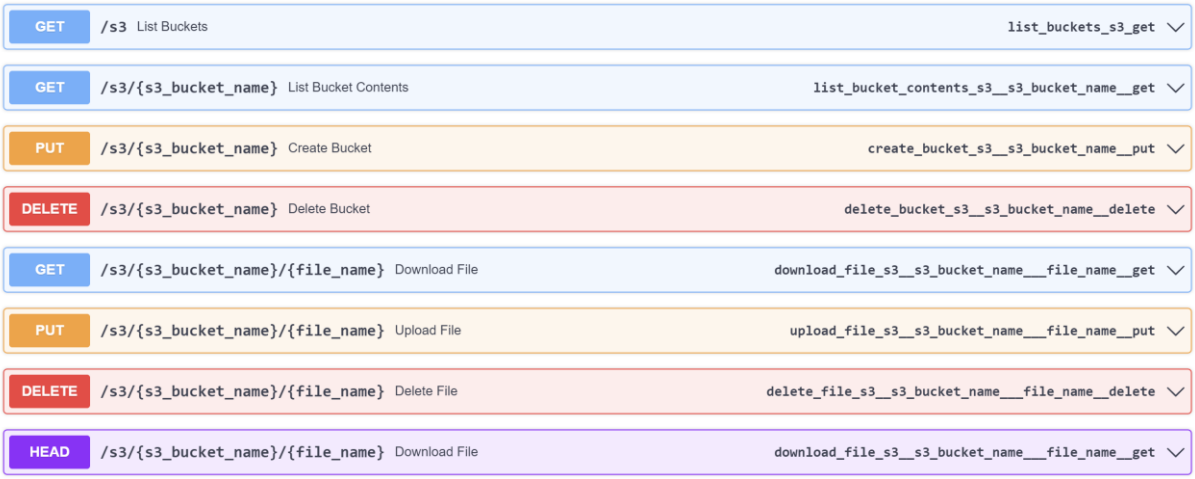


Figure 30: Secure Storage API REST endpoints

Amazon Web Services S3 provides two URL schemas to access buckets and their contents. The Secure Storage API adopts the first one and may potentially be expanded to support the second one at a later stage.

- http://{host:port}/s3/[bucket_name]/
- http://[bucket_name].{host:port} /s3

The Secure Storage API uses the same parameters for each endpoint and maintains the error handling of AWS S3, both in terms of the format of error messages as well as the different codes that identify the underlying causes.

Finally, the Secure Storage API will continue to be expanded with both new options for the existing endpoints as well as new endpoints. These will be focused on features like Access Control Lists (ACL), object versioning and others. In all cases, compatibility with the S3 API will be maintained.

Authentication is handled through AWS Signature Version 4, though support for Version 3 may be added in the future.

The Gateway also exposes two further REST APIs. The Storage Policy API provides integration with the SERRANO orchestration mechanisms, while the Telemetry and Resource API provides integration with the Monitoring service and the Resource Orchestrator of the SERRANO platform. The Gateway is also an integration point with other platform services and functionalities such as TLS-offloading and the acceleration of data processing algorithms. These details are described in Section 4.10, Secure Storage Use Case Integrated Functionality.

4.8.1.3 Sample requests and responses

We provide a sample request-response for listing the objects present in a bucket, which corresponds to the ListObjectsV2 endpoint.

Request:

```
GET /s3/{bucket_name}/
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <Name>{bucket_name}</Name>
    <Prefix/>
    <KeyCount>{number of objects}</KeyCount>
    <Contents>
        <Key>{object key}</Key>
        <LastModified>{date of last modification}</LastModified>
        <Size>{size in bytes}</Size>
    </Contents>
    <Contents>
       ...
    </Contents>
    ...
</ListBucketResult>
```

In the future, we plan to expand the list of supported query parameters as well as the item details in the response for this and other endpoints.

4.8.2 TLS offloading

SERRANO relies on acceleration/offloading engines to perform some of the key functions in establishing links under secure conditions, while ensuring that the enhanced security is not increased at the cost of latency. Internet Protocol Security (IPsec) and Transport Layer Security (TLS) working together allow for full data-path encryption and transport security. SERRANO has investigated autonomous offloads for the TLS layer using dedicated accelerators at the network interface card (i.e., dedicated ARM processors).

TLS accelerators can be implemented at both hardware and software level (through specific SW kernels). However, SW accelerators remain CPU intensive as encryption/decryption processes are heavy on cascaded operations. Hence, in general SW accelerators are unlikely to produce gains in terms of latency. In SERRANO, TLS accelerators run at the HW level within the network interface card, where autonomous offloads are conducted in dedicated processors (i.e., ARM processors). One of the advantages of conducting offloads at the network interface card is also that memory utilization at the host is reduced at a minimum, while memory at the network interface cards is cheap (cost wise) and can be easily increased.

The following figure shows the memory utilization during in-line encryption under various approaches.

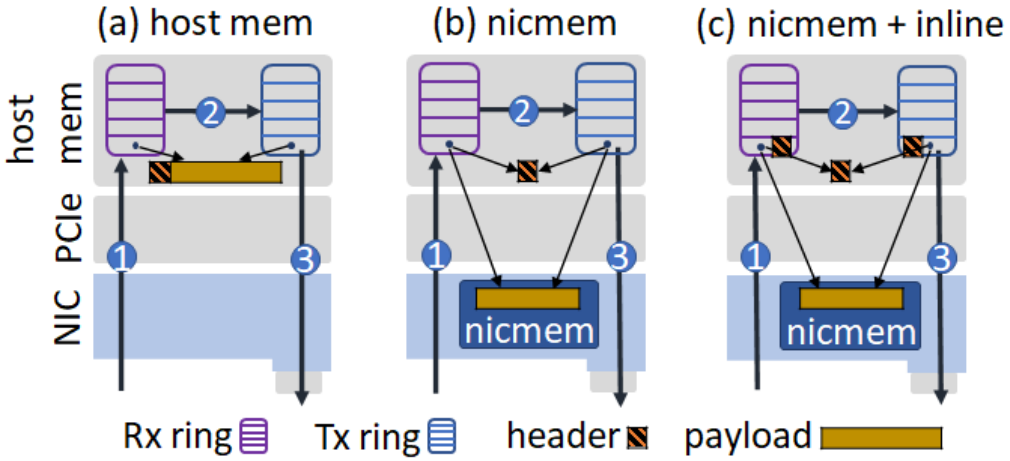


Figure 31 Transitioning from host memory to NIC memory and inline TLS

In a pure host memory utilization approach, packets arriving to the NIC are sent to the host memory, processed, and sent back to the NIC for further network transport. In that case, all processing is conducted by the host, which increases latency. Alternatively, when the NIC memory is employed, the payload of packets arriving to the NIC is placed in the NIC memory and is accessible by both the NIC and the host. In addition, only the header is sent to the host for processing, which upon finalization, it is sent to the NIC again for further network transport. Finally, in SERRANO, the last approach is applied while simultaneously, the encryption and decryption processes of the header are conducted at the NIC level rather than at the host level, which reduces latency and frees up host resources.

Preliminary simulation results for TLS offloading approach showcase the provision of relevant gains in both latency reduction and PCIe throughput. The following figure shows the latency reduction (x3) observed. In addition, host bloating due to CPU utilization is also reduced.

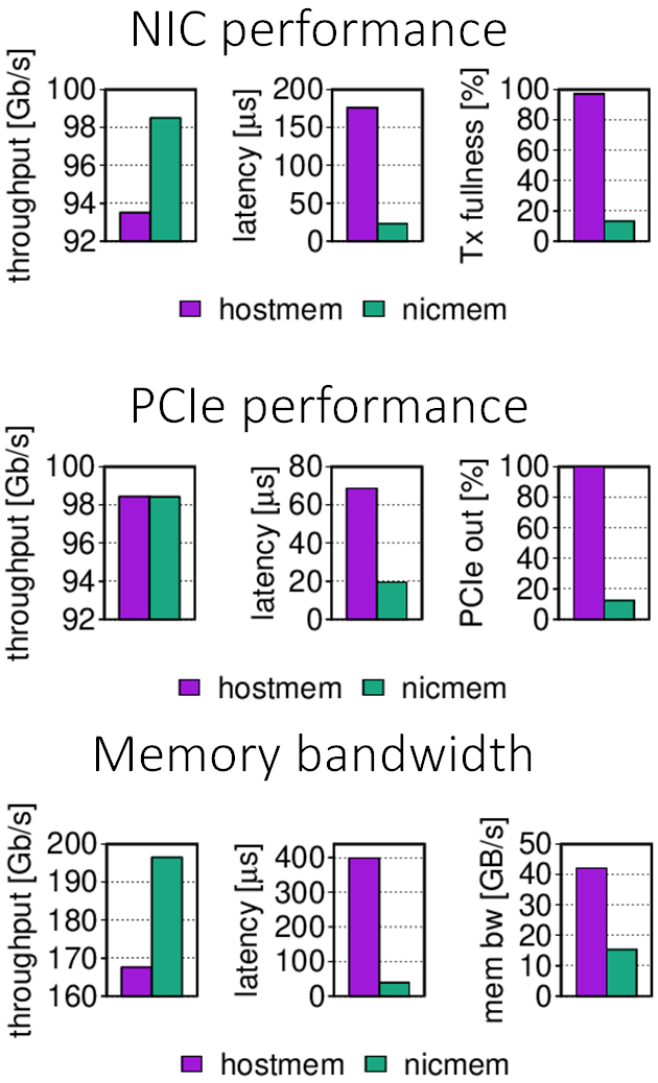


Figure 32 Transitioning from host memory to NIC memory and inline TLS

4.8.2.1 Integration details and REST APIs

The offloading engines are currently integrated into the NICs used for prototyping the SERRANO solution, particularly for the Chocolate Cloud use case on secure storage. APIs for the secure storage use case are described in Section 4.10.

4.9 Service Assurance

The event detection engine (EDE) is a subcomponent of the SERRANO service assurance component. It is designed to detect complex/contextual anomalies during service/application execution from the available monitoring data. Once anomalous events have been detected, they are made available to other components of SERRANO, which can use this information in their decision-making process. For example, anomaly remediation will use the information received from EDE, which will contain the timestamp of the anomalous event and a root cause

analysis. Using this information, appropriate and predetermined remediation actions can be made by other SERRANO components, such as the Resource Orchestrator and AI-enhanced Service Orchestrator.

Contextual anomalies are extremely interesting in the case of complex, distributed systems. These types of anomalies happen when a certain constellation of feature values is encountered. When viewed in isolation, these values are not anomalous but when viewed in a wider context they represent an anomaly. These types of anomalies represent application bottlenecks, imminent hardware failure or software miss-configuration. Another anomaly type which is of interest in Serrano are so called temporal or sometimes sequential anomalies where a certain event takes place out of order or at the incorrect time. These types of anomalies are very important in systems which have a strong spatio-temporal relationship between features, which is very much the case in SERRANO.

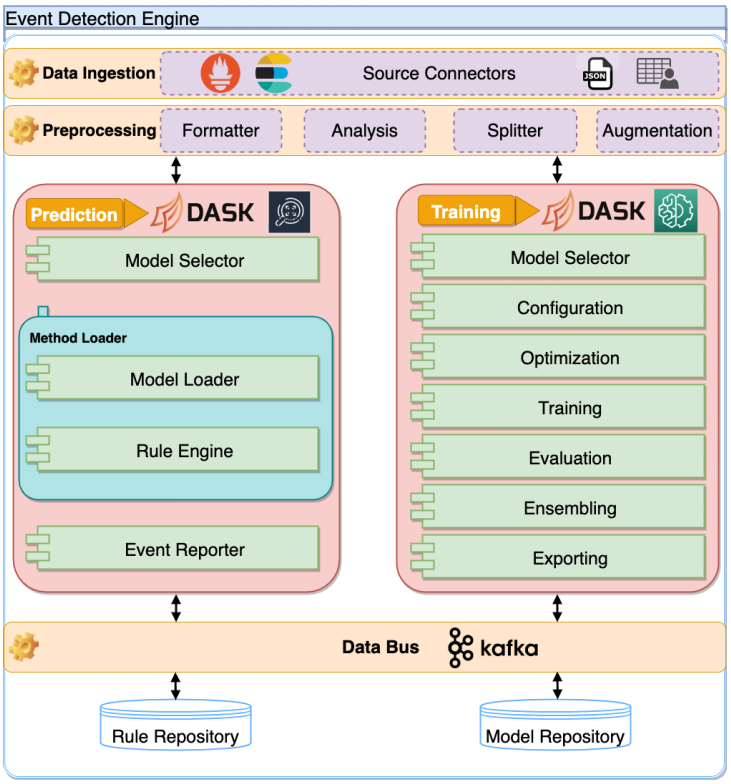


Figure 33 EDE Architecture

EDE has several sub-components, which are based on a lambda-type architecture, having a *speed*, *batch,* and *serving* layer. Because of the heterogeneous nature of most modern computing systems and the substantial variety of solutions that could constitute monitoring services, the *data ingestion component* has to be able to fetch data from a plethora of systems. Therefore, the *Connectors* component is implemented to serve as an adapter for each solution. Furthermore, this component is also able to load data directly from static files (*HDF5*, *CSV*, *JSON*, or even *raw format*).

Data ingestion can be done directly via query from the monitoring solution or *streamed directly from the monitored nodes/services/components* (after ETL if necessary). This ensures that we have the best chance of reducing the time between the occurrence and detection of an event or anomaly.

The *pre-processing component* is in charge of taking the raw data from the data ingestion component and apply several transformations. It handles *data formatting* (i.e., one-hot encoding), *analysis* (i.e., statistical information), *splitter* (i.e. splitting the data into training and validation sets) and finally *augmentation* (i.e. oversampling, undersampling).

The *training component* (batch layer) is used to instantiate and train ML methods (both supervised and unsupervised) that can be used for event and anomaly detection. The end user is able to configure the hyper-parameters of the selected models as well as run automatic optimization on these (i.e., Random Search, Bayesian search etc.). Users are not only able to set the parameters to be optimized but to define the objectives of the optimization, including transprecise objectives if applicable. We should also note that users can specify any ML method which respects the API conventions used in the Scikit-learn[36] Python library. This makes EDE compatible with most state-of-the-art ML methods from the Python ecosystem.

The *prediction component* (speed layer) is in charge of retrieving the predictive model from the model repository and feeding metrics from the monitored system. If and when an event or anomaly is detected, the EDE is responsible for signalling this to other SERRANO components in charge of orchestration, scheduling, or remediation. EDE does this by publishing detected anomalous events to a particular Kafka Topic (SERRANO Stream Handler) as well as storing these events internally for later querying and analysis.

4.9.1 Integration details and REST APIs

EDE relies on a configuration file during execution. This configuration file contains settings for all sub-components mentioned in the previous paragraph (i.e., Connector, pre-processing, training, detection). In the following paragraphs, we will describe the most important configuration settings for EDE integration into the SERRANO solution. The complete API documentation is available at the EDE Gitlab Repository [37]. For the sake of readability, we will use the YAML format instead of the JSON one as it is arguably easier to read. While YAML is a superset of JSON all examples given here are fully compatible with any JSON parser.

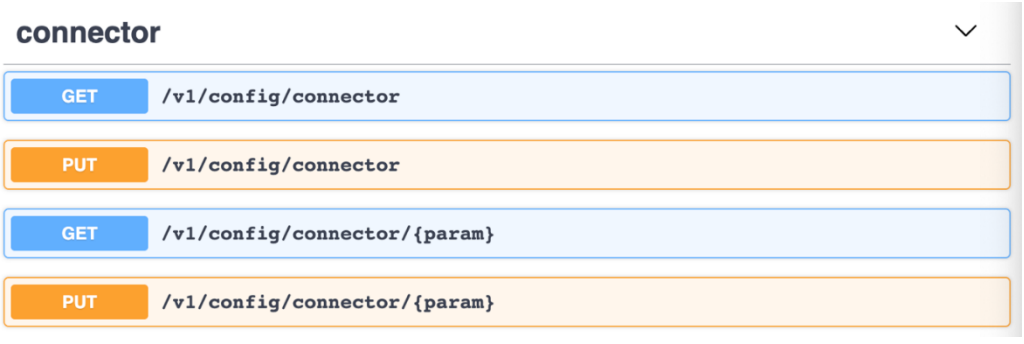


Figure 34 EDE Connector API

Figure 34 shows the Connector API, responsible with setting up monitoring data sources and all connection channels. Bellow you can find a response example:

```
GET /v1/config/connector
```

```
Connector:
  PREndpoint: 192.102.62.155 #Telemetry Endpoint
  Dask:
    SchedulerEndpoint: local # if not local add DASK schedueler endpoint
    Scale: 3 # Number of workers if local otherwise ignored
    SchedulerPort: 8787 # This is the default point
    EnforceCheck: False # Irrelevant for local
  MPort: 9200 # Telemetry port
  KafkaEndpoint: 10.9.8.136
  KafkaPort: 9092
  KafkaTopic: edetopic
  Query: {"query": '{__name__=~"node.+"}[1m]'} # PromQL query example
  MetricsInterval: "1m" # Metrics datapoint interval definition
  QSize: 0
  Index: time
  QDelay: "10s" # Polling period for metrics fetching
  Local: /mnt/data/df_anomaly.csv  # local file location if applicable
```

Using the Connection API, we can specify the telemetry component's location, query, and polling period to be used. The response will be formatted into a Pandas [38] Dataframe which can be used by all ML anomaly detection methods. EDE is, in essence, a technology agnostic tool, meaning it does not need to know the underlying application topology for anomalous event detection. It will try to detect these anomalous events (be it using supervised or unsupervised methods) from the data it receives. However, we can include application topology information by formatting/preprocessing the data before it is fed into the detection methods. This can be achieved in the Connection API by simply adjusting the query for fetching the data or submitting a prelabeled local dataset for training.

Furthermore, we specify the Kafka endpoint (SERRANO Stream Handler) for anomaly reporting and data fetching (if applicable). This parameter is not mandatory for anomaly reporting, EDE stores all detected anomalous events locally, which can later be queried via the Inference API.

For modifying the parameters in this configuration, we can specify the parameter name followed by a JSON payload containing the new value. For example, setting up a Dask [39] local cluster can be done using the following request:

```
PUT /v1/config/connector/dask
```

```
{
        “SchedulerEndpoint”: “local”,
        “Scale”: 3,
        “SchedulerPort”: 8787,
        “EnforceCheck”: False
}
```

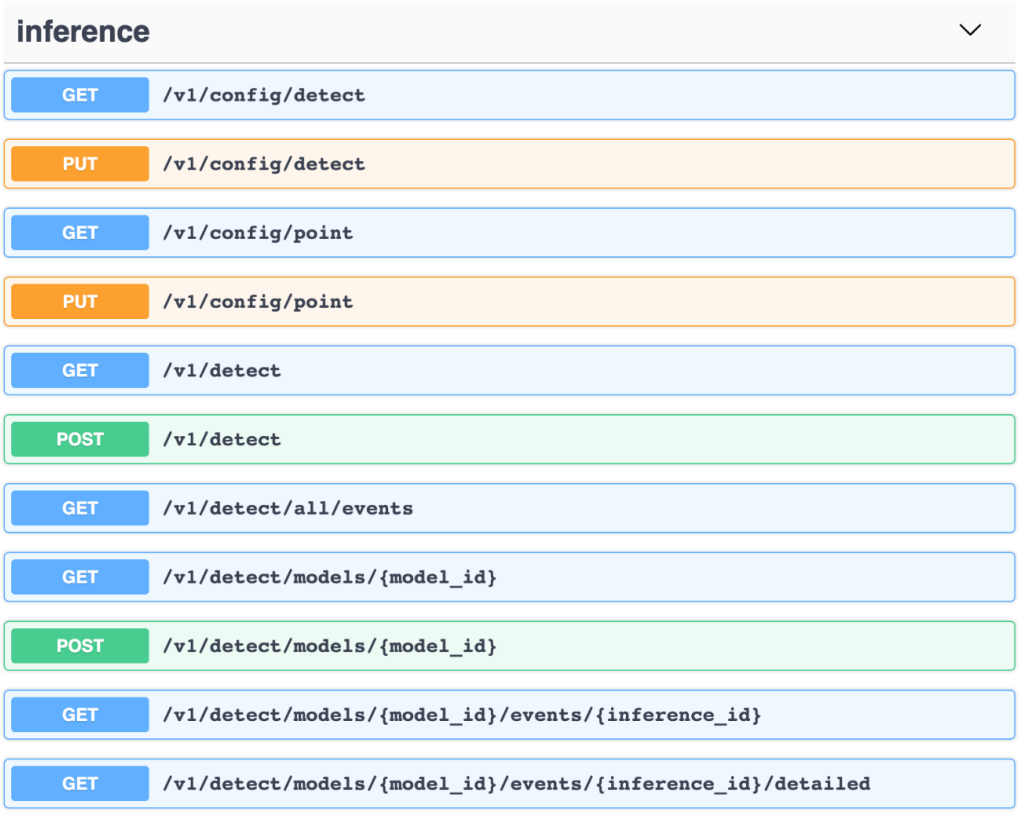


Figure 35 EDE Inference API

The Inference API presented in Figure 35 is used to control the detection capabilities of EDE. For example, the detection config response example can be found bellow:

```
GET /v1/config/detect
```

```
Detect:
  Method: RandomForestClassifier
  Type: classification
  Load: rfc_v2
  Scaler: StandardScaler # Same as for training
#  Analysis: True # Start Shapely value based analysis
  Analysis:
        Plot: True # if plotting of heatmap, summary and feature importance is req
uire, if not set False or use previous example
```

We can see that the inference configuration is fairly simple, we only need to specify the detection method, method type (supervised/unsupervised), and name of the model to be instantiated ("Load"). Additionally, we can specify the scaler used during training, which is considered a separate model, thus predictive models can share scalers. The "Analysis" option gives additional information about the detected events. In the case of supervised methods, this additional data is mostly useful for fine tuning production models; however, in the case of unsupervised methods it is used for root cause analysis via the computation of Shapely values.

The method utilized for setting these parameters is identical to how it is done for the Connector API. One has to specify the parameter and its new value. Because EDE can be distributed via Dask, any number of predictive model instances can run at any given time. Thus, it is important to have a resource in the Inference API to list the available models from the internal model repository and start them, if necessary, as new Dask workers. Bellow, we can see a response example:

```
GET /v1/detect
```

```
{
  models:
        [
                {
                        method: RandomForestClassifier
                        scaler: StandardScaler
                        model_id: rfc_v2
                        analysis: True
                        dask_worker_pid: 3450 # if pid is 0, model inactive
                }
                …
        ]
}
```

If a POST request is issued, the predictive model set in the configuration file will start and use the connection details set in the Connection API. We should note at this point that the Training API has many more parameters that need to be configured and also uses the settings from the Connection API and can run as a separate Dask worker. This means that the cardinality of both training and inference workers is larger than 1.

As mentioned before, EDE will publish any detected anomalous events to a Kafka Topic (Serrano Stream Handler); however, it also stores them locally. These locally stored events can be accessed via the Inference API. For example, to fetch all events we can use:

```
GET /v1/detect/all/events
```

```
{
        events:
        [
                {
                        utc: <timestamp>,
                        hutc: <human_readable_timestamp>,
                        type: <anomaly_class> # if unsupervised type is -1,
                        model_id: <unique_id>,
                        inference_id: <unique_id>,
                        analysis: [
                                        {<feature_1>:<shapley_value_1>},

                                        {<feature_2>:<shapley_value_2>},
```

```
                                ]
                }
        ]
}
```

In case of unsupervised methods, a ranking of which features had a negative or positive impact on the prediction is represented by the analysis filed in the response. This can later be used to correlate what metric is mapped to which component/service/node and gives greater insight into what caused the anomalous event.

If predictions from a specific predictive model are required, we can issue a get request to the following resource:

```
GET /v1/detect/models/<model_id>
```

The response is the same as before, but it will only contain a listing of the predictions of a particular model.

If a particular event has to be checked, we can send the data directly to the model using the following resource:

```
POST /v1/detect/models/<model_id>
```

The payload of this request can be in CSV or JSON format (see Pandas documentation for details). The response will have the same format as before. If the model defined in this request is not currently instantiated, it will be started only for this prediction.

If we want to receive information about a particular anomalous event, we can issue a request to:

```
GET /v1/detect/models/<model_id>/events/<inference_id>
```

Again, the response is the same as before but the response will contain only one anomalous event. If additional details are required a request can be issued to:

```
GET /v1/detect/models/<model_id> events/<inference_id>/details
```

```
{

        events:
        [
                {
                        utc: <timestamp>,
                        hutc: <human_readable_timestamp>,
                        type: <anomaly_class>, # if unsupervised type is -1
                        model_id: <unique_id>,
```

```
                        inference_id: <unique_id>,
                        analysis: [
                                        {<feature_1>:<shapley_value_1>},

                                        {<feature_2>:<shapley_value_2>},
                                ],
                        details: [
                                {<detail_1>:<url_1>},
                                {detail_2>:url_2},
                                …
                        ]
                }
        ]
}
```

The “details” field will contain a list of all of the available information about the current prediction. These details are dependent on the predictive model and can range from visualizations, distance matrices, density graph data etc.

EDE has two main SERRANO components that it relies on and should have some form of integration. The Central Telemetry Handler from which receives up to date monitoring data, and the Stream Handler through which it publishes any detected anomalous events. Once an anomaly has been identified, its remediation is handled by a separate remediation component (the Remediation Engine) and the SERRANO orchestration mechanisms. As EDE is by design a passive component, it will only report anomalous events. Thus, any other components must subscribe to and fetch this data on their own via the Inference API or Stream Handler.

Since, EDE does not know the details of the currently monitored application/service deployments, it has to rely on other SERRANO components to retrieve the required information. The remediation plans should be in the form of concrete actions which can be performed to remediate the impact of the anomalies identified. In essence, these actions will form a remediation plan.

Figure 36 shows the main interactions of EDE with other SERRANO components. First, we have a looping sequence of messages between the Central Telemetry handler and EDE for fetching new, up to date monitoring data using the Connection API. Next, we have a conditional interaction between EDE and the Stream handler; if an anomalous event is detected, it will be forwarded to the predefined topic in the Stream Handler and thus made available to other SERRANO components such as the Resource Orchestrator. Next, both the Orchestrator and the Remediation Engine can query EDE directly for anomalies (using the Inference API). Based on the type of anomaly detected, each component can make certain adjustments.

As this deliverable is an initial version of the SERRANO platform not all components have a complete, testable prototype. The Remediation Engine is one such component. It relies on EDE detected anomalies which it then uses to generate a sequence of tailor-made actions for the remediation of particular anomalies. These sequences of actions in fact form a predetermined remediation plan which is then sent to the SERRANO Orchestrator. Future deliverables will contain more details about the exact interaction and implementation of all components.

As this deliverable is mainly focused on integration into the SERRANO platform some features and APIs from EDE are not fully explored. The Training API is not detailed in this deliverable as users mainly use it during predictive model generation. It is not controlled by or does not interact directly with any SERRANO component.

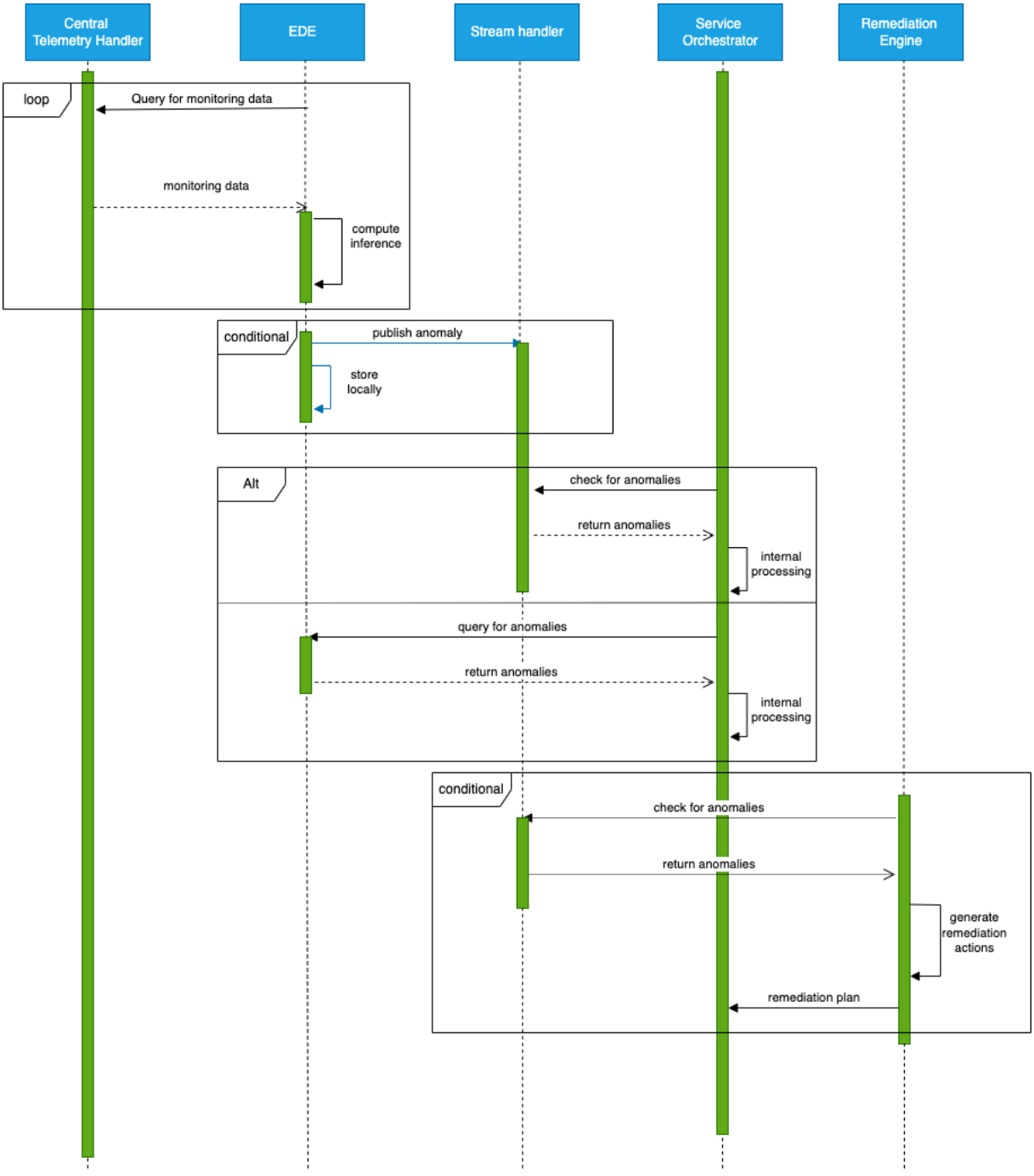


Figure 36 EDE Sequence Diagram (Integration)

4.10 Secure Storage Use Case Integrated Functionality

The Secure Storage Use Case has been described in detail in Deliverable 2.4 [30]. Here, we only included a brief description, extracted from the aforementioned document, focusing on its integration of SERRANO platform features and services.

This UC focuses on providing secure and high-performance storage and sharing of files with lower latency than a purely cloud-based approach. We plan to achieve this goal by extending SkyFlok with on-premises edge devices that can act as storage locations. Most of the features showcased by the UC are provided by the Secure Storage Service (also referred to as the SERRANO-enhanced Storage Service in other deliverables). Section 4.8.1 describes the main S3-compatible API offered by the service and briefly describes its interaction with other platform services. In this section, we focus on the details of these interactions and present two additional REST APIs.

From a high-level perspective, this UC will also involve the SERRANO platform's orchestration and telemetry services in two meaningful ways. First, it will rely on the orchestration mechanisms to deploy the services associated with the edge storage locations and the gateway. Second, the SERRANO telemetry mechanisms will provide information on edge storage locations regarding their status, availability, cost, latency, etc. Hence, this integration will ensure that each storage task is placed on the appropriate storage resources/site, as it enables the automatic creation of storage policies based on the formally described requirements of each storage task. Alternatively, a data-driven mechanism that selects the appropriate policy from a manually created list can also be envisioned.

Finally, there might be benefits if the storage service is aware of the location of the user's application. For example, if an appropriate erasure coding configuration and edge locations are used, some storage tasks may be more efficiently served using nearby locations. This can be accomplished using an optional extension to the standard S3 interface that allows the orchestrator (or the client application with support from the orchestrator) to give hints on the ideal storage locations to access when retrieving a file.

The Gateway is a performance-critical component, given its role in accelerating file operations. Hence, the designed solution will use hardware acceleration for encrypting TLS connections by leveraging Nvidia Bluefield cards, when available, to provide low-latency access to files for a large number of concurrent users. This will reduce some of the load on the CPU and may increase the number of concurrent supported connections. In addition, it will use GPUs and FPGAs to accelerate encryption, erasure coding, and potentially compression using other SERRANO tools and services developed in WP4. This should lead to further CPU offloading as well as a reduction of processing time.

4.10.1 Integration details and REST APIs

The Storage Policy API allows the platform's users as well as the SERRANO Resource Orchestrator to create and retrieve storage policies. These are the recipes used to translate an application's storage task's requirements to a storage resource allocation. The ARDIA framework developed in Work Package 5 contains both the Application and the Unified Resource Model definitions. Storage policies are applied to each storage bucket. Figure 37 presents the list of supported endpoints that allow creating and retrieving storage policies.

IP(s)/Port(s)	on-premises-storage-gateway.serrano.cs.uvt.ro Accessible through port 2525, with storage_policy prefix
Publicly accessible (y/n and other details)	The IP is not publicly accessible, and authentication is performed using the same credentials as the Secure Storage API. Chocolate Cloud will make credentials available to all partners who wish to access the service.
Type of API	REST, JSON requests and responses
API documentation	https://gitlab.com/serranoproject/wp3/on-premise-storage-gateway/-/tree/master/openapi.json
Location of integration tests	https://gitlab.com/serranoproject/wp3/on-premise-storage-gateway/-/tree/master/tests/storage_policy/

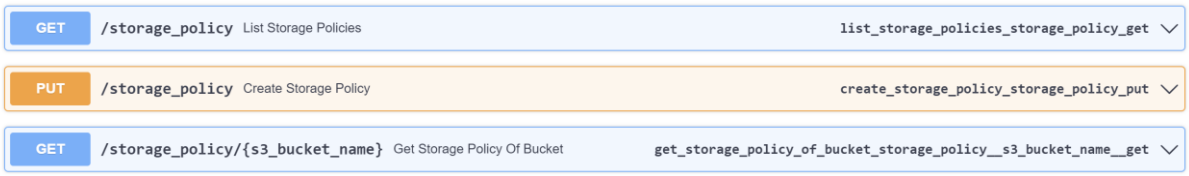


Figure 37: Storage Policy API REST endpoints

Figure 38 presents an example of a storage policy that describes which storage locations to use (referred to as bucket_ids, naming may be updated in the future) and what encryption and erasure coding schemas to apply.

```
default_storage_policy:

 locations:
   bucket_ids: [14, 83, 132, 35, 82, 26]
   # Bucket IDs correspond to storage buckets
   # at cloud-based providers (e.g. AWS S3)
   # or edge storage locations

  encryption:
   enabled: True
   type: "AES-GCM-256"

  erasure_coding:
   type: "RLNC"
   symbols: 5
   generation_size_mb: 120
```

Figure 38: Sample storage policy file

The Resource and Telemetry API is used to expose information about the storage locations. This is used by the SERRANO Resource Orchestrator as input when matching storage tasks with storage policies and as input data for creating new storage policies. It is also used by the Telemetry API to monitor the state of the storage locations as resources of the SERRANO platform.

The parameters exposed for each location include the following static characteristics:

- provider name: Google, Amazon, ….
- geographic location – GPS coordinates
- country/city
- GDPR compliance
- storage cost – typically in $ / GB / month
- ingress cost – typically in $ / GB
- egress cost – typically in $ / GB

Later, several dynamically measured parameters will also be added:

- status: online/offline
- availability over the past year in %
- read latency
- write latency
- read throughput
- write throughput.

Figure 39 shows the single endpoint used to list cloud storage locations. It is possible that in the future, both types of parameters will be exposed through separate endpoints, given that the static characteristics will rarely, if ever, change, while the dynamic characteristics will be retrieved more often by the Telemetry Service.

IP(s)/Port(s)	on-premises-storage-gateway.serrano.cs.uvt.ro Accessible through port 2525, with cloud_locations prefix
Publicly accessible (y/n and other details)	The IP is not publicly accessible. No authentication is needed.
Type of API	REST, JSON responses
API documentation	https://gitlab.com/serranoproject/wp3/on-premise-storage-gateway/-/blob/master/openapi.json
Location of integration tests	https://gitlab.com/serranoproject/wp3/on-premise-storage-gateway/-/tree/master/tests/cloud_locations/

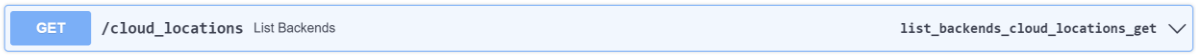


Figure 39: Resource and Telemetry API (exposed by On-premises Storage Gateway) REST endpoints

Later, an endpoint for retrieving the SERRANO edge devices' static characteristics will be added. These will also be monitored by the Telemetry service. However, their dynamic characteristics will be retrieved directly from MinIO's relevant interface, not through this REST API.

4.10.1.1 Sample requests and responses

We provide a sample request-response for listing the storage policies defined for an account.

Request:

```
GET /storage_policy/
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <Name>{bucket_name}</Name>
    <Prefix/>
    <KeyCount>{number of objects}</KeyCount>
   <Contents>
        <Key>{object key}</Key>
        <LastModified>{date of last modification}</LastModified>
        <Size>{size in bytes}</Size>
    </Contents>
    <Contents>
       ...
```

```
    </Contents>
    ...
</ListBucketResult>
```

We also provide a sample for fetching the cloud locations.

Request:

```
GET /cloud_locations/
```

Response:

```
[
   {
      "location":"Iowa",
      "country":"United States",
      "countrycode":"US",
      "is_gdpr":false,
      "storage_price":20.0,
      "download_price":120.0,
      "upload_price":0,
      "lat":41.8780025,
      "lng":-93.097702,
      "cloud_provider_name":"Google Cloud Platform",
      "cloud_provider_jurisdiction":"United States",
      "cloud_provider_url":"https://cloud.google.com/"
   },
 …
   {
      "location":"Marchtrenk",
      "country":"Austria",
      "countrycode":"AT",
      "is_gdpr":true,
      "storage_price":null,
      "download_price":null,
      "upload_price":0,
      "lat":48.1969259,
      "lng":14.0833296,
      "cloud_provider_name":"Ventus Cloud",
      "cloud_provider_jurisdiction":"Switzerland",
      "cloud_provider_url":"https://ventuscloud.eu/"
   }
]
```

4.10.1.2 Integration with acceleration features developed in WP3 and WP4

Whenever a Nvidia DPU is available, the On-premises Storage Gateway will perform the TLS encryption directly on this resource. This CPU-offloading technique is expected to increase the Secure Storage Service's performance. Integration will be achieved through a custom version of the OpenSSL library. This library will be loaded into the container and automatically detect what HW resources are available. If an appropriate DPU is detected, computations related to TLS encryption will be performed on the DPU.

The integration of the TLS encryption is done through the utilization of the DOCA DPU software development kit (SDK), which enables fast and reliable integration of the TLS offloading processes into the DPU. As a result, the specific technical details of the TLS system remain irrelevant for the developer, who simply by employing the provided libraries can leverage the advantages of TLS offloading.

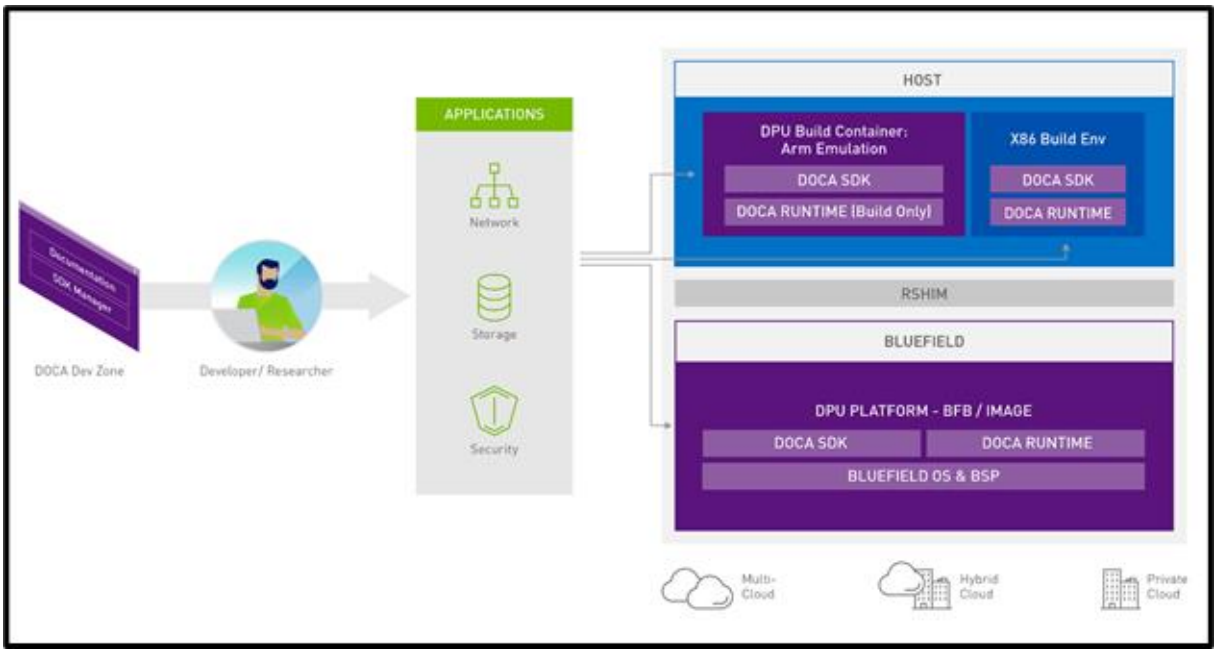


Figure 40: DOCA DPU utilization in SERRANO

The use case will involve the movement of a large amount of data, in the form of uploaded and downloaded files. The Gateway will need to compress, encrypt and erasure code the files to preserve their privacy and reliability in a cost-effective manner. To avoid these data processing tasks becoming a performance bottleneck, the use case will employ the techniques developed as part of Work Package 4. The integration will happen through a series of pluggable dynamically loaded software libraries which will enable the Gateway to perform these tasks using either the CPU or an FPGA or GPU, if these hardware resources are available.

The task of encrypting and decrypting data is performed by the AES-GCM encryption and decryption algorithms. Moreover, these two algorithms are accelerated in cloud (NVIDIA Tesla T4) and edge (NVIDIA AGX Xavier) GPU devices leading to an execution time speedup up to 229x.

The erasure code tasks of encoding and decoding the encrypted and decrypted data have been accelerated in cloud (Xilinx Alveo U50) and edge (Xilinx ZCU102 MPSoC) FPGA devices, leading to an execution time speedup up to 35.9x. The execution of the corresponding algorithm on cloud or edge devices will be determined by the SERRANO orchestration mechanisms based on the user requirements for energy efficiency and performance as well as based on the availability of the devices that are deployed in the SERRANO's infrastructure. Additional details regarding the acceleration of the algorithms of this use case can be found in D4.1.

4.11 Fintech Analysis Use Case Integrated Functionality

InbestMe provides automated and personalized investment portfolios composed of different financial instruments, such as shares, ETFs, funds, bonds, etc. Typically, investment portfolios are managed by an investment manager. The decisions about how to adjust the portfolios are complex and based on past, current, and predicted future market conditions. The investment profiles specify the composition of the investment portfolios, which we call distribution of assets, for a specific time horizon.

Fintech use case (UC) demonstrates automatic optimization of InbestMe's dynamic investment portfolios, which we call Dynamic Portfolio Optimization (DPO). For the portfolio construction, we combine the results from the asset market analysis and the investment strategy, in building investment profiles. Based on this analysis, we determine the distribution of each asset in the investment profiles.

The SERRANO project will contribute to the investment management UC by providing a framework that will simplify the deployment, management, operation, and monitoring of the application of DPO. Additionally, for the provision of the investment management as a service (SaaS), the UC will benefit from the secure storage extension that will keep the data of third parties secure. The UC will also demonstrate the advantages of cloud-based acceleration of various computationally intensive operations.

The SERRANO platform will also be beneficial for InbestMe because it will reduce cloud costs and improve the quality of services. It will be able to easily deploy multiple instances of its investment management platform on local as well as external cloud resources. Furthermore, a more accurate analysis will enable the portfolio constructions with lower risk and higher return. Currently, InbestMe has limitations on performing extensive analysis of all the possible available data. By achieving this objective, InbestMe will be able to analyze more information and implement prediction and forecasting algorithms with higher precision and accuracy.

4.11.1 Integration details and REST APIs

For the Dynamic Portfolio Optimization (DPO), the developed services will be demonstrated in a container application on the SERRANO platform through the SERRANO SDK. For this application, the data needed are investment profile assets and their market data, which are their historical price data. These data will be stored in Skyflok provided by Chocolate Cloud. The SERRANO platform will be triggered through an API request, to start the DPO application.

Through this request, the market and investment profile data will be loaded to the SERRANO platform and the DPO will be executed. When the execution is completed, the resulting asset profile optimizations will be downloaded locally.

In addition, for the DPO, we will leverage the available acceleration mechanism interfaces and HPC system hardware interfaces (described in this deliverable) to enable their access to the SERRANO-enhanced computational resources, to ensure better performance and optimization of the kernels used in the services.

The DPO will have only one API.

```
POST /portfolio_optimization
```

It will take as input an JSON object. The input is described in the textbox below.

```
Input JSON
{
   StartDate: DateTime
   EndDate: DateTime
   Url1ToSkyFlokStorage: String
   Url2ToSkyFlokStorage: String
}

- StartDate:             Historical Asset Data look back date
- EndDate:               Historical Asset Data look up to date
- Url1ToSkyFlokStorage:  URL to Historical Asset Data file stored in SkyFlock
cloud storage
- Url2ToSkyFlokStorage:  URL to Investment Profile Asset Data file stored in
SkyFlock cloud storage
```

The output is defined in the table below.

```
Output JSON
{
   File1Base64: string
   File2Base64: string
}

- File1Base64:   Is the DPO Backtest Result file. Example output in Figure 41.
- File2Base64:   Different asset distributions per investor profile. Example
output in Figure 42.
```

DPO Backtest Results	Specs					
		Start	2020-05-29	MaWindow	10	F
		End	2021-05-31	Momentum	{3: 0.5, 6: 0.25, 9: 0.25}	F
		Period	BM	SelectNum	2	
		Currency				

Metrics											
InvestmentType	Portfolio	InitialBalance	FinalBalance	TotalReturn	CAGR	BestYear	WorstYear	Volatility	Sharpe	MaxDrawdown	MaxUnderwater
Dynamic EUR	Profile 0	100	100.92	0.92	0.91	1.77	-0.83	1.5	0.61	-1.19	5
Dynamic EUR	Profile 1	100	104.14	4.14	4.12	4.11	0.02	2.39	1.72	-1.17	4
Dynamic EUR	Profile 2	100	147.1	47.1	46.83	28.8	14.21	2.93	15.96	0	0
Dynamic EUR	Profile 3	100	170.32	70.32	69.89	41.25	20.59	3.53	19.78	0	0
Dynamic EUR	Profile 4	100	210.01	110.01	109.27	60.79	30.61	4.3	25.43	0	0
Dynamic EUR	Profile 5	100	250.57	150.57	149.47	78.85	40.1	5.42	27.55	0	0
Dynamic EUR	Profile 6	100	260.07	160.07	158.88	82.63	42.4	6.28	25.31	0	0
Dynamic EUR	Profile 7	100	309.02	209.02	207.36	102.47	52.62	7.2	28.81	0	0
Dynamic EUR	Profile 8	100	347.34	247.34	245.28	116.66	60.31	8.43	29.09	0	0
Dynamic EUR	Profile 9	100	374.49	274.49	272.14	126.4	65.41	9.2	29.58	0	0
Dynamic EUR	Profile 10	100	381.99	281.99	279.56	128.67	67.05	10	27.95	0	0
Dynamic USD	Profile 0	100	100.35	0.35	0.35	1.36	-0.99	1.71	0.21	-1.94	10
Dynamic USD	Profile 1	100	104.1	4.1	4.08	4.04	0.06	2.73	1.49	-1.31	4
Dynamic USD	Profile 2	100	147.61	47.61	47.34	29.44	14.04	3.9	12.14	0	0
Dynamic USD	Profile 3	100	178.01	78.01	77.52	45.13	22.66	4.92	15.75	0	0
Dynamic USD	Profile 4	100	213.76	113.76	112.99	62.48	31.56	6.06	18.66	0	0
Dynamic USD	Profile 5	100	256.87	156.87	155.72	82.19	40.99	7.25	21.47	0	0
Dynamic USD	Profile 6	100	265.36	165.36	164.12	85.77	42.84	8.46	19.4	0	0
Dynamic USD	Profile 7	100	296.52	196.52	194.99	98.51	49.37	9.14	21.34	0	0
Dynamic USD	Profile 8	100	314.29	214.29	212.58	105.05	53.27	10.11	21.03	0	0
Dynamic USD	Profile 9	100	337.8	237.8	235.84	114.14	57.74	10.74	21.97	0	0
Dynamic USD	Profile 10	100	357.98	257.98	255.8	121.34	61.73	11.77	21.73	0	0

Figure 41: Dynamic Portfolio Optimization Backtest Result file

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
1	Index	Portfolio	InvestmentType	Date	BSCL US Equi	GLD US Equi	HYG US Equi	VTI US Equi	SPY US EQUI	NEAR US Equi	DBEU US EQUI	Cash	SHY US Equi	VIG US Equi	DBEF US Equi	VWO US EQUI	BNDX US Equi	VEA US Equi	FLOT US Equi	RSP US Equi	LQD US Equi	SMTC LN Equi	EMB US Equi	BSCK US Equi	VNQ US Equi	VPL US EQUI	IEF US Equi
362	360	Profile 10	Dynamic USD	2017-11-01	0	4	0	21	0	0	0	0	20	10	22	5.5			0		0	0	0		12	5.5	0
363	361	Profile 10	Dynamic USD	2017-12-01	0	4	0	21	5.5	0	0	0	20	10	22	0			0		0	0	0		12	5.5	0
364	362	Profile 10	Dynamic USD	2018-01-01	0	4	0	21	0	0	0	0	20	10	22	5.5			0		0	0	0		12	5.5	0
365	363	Profile 10	Dynamic USD	2018-02-01	0	4	0	21	5.5	0	0	0	20	10	22	5.5			0		0	0	0		12	0	0
366	364	Profile 10	Dynamic USD	2018-03-01	0	4	0	21	5.5	0	0	0	20	10	22	5.5			0		0	0	0		12	0	0
367	365	Profile 10	Dynamic USD	2018-04-02	0	4	0	21	0	0	0	0	20	10	22	5.5			0		0	0	0		12	5.5	0
368	366	Profile 10	Dynamic USD	2018-05-01	0	4	0	21	0	0	5.5	0	20	10	22	0			0		0	0	0		12	5.5	0
369	367	Profile 10	Dynamic USD	2018-06-01	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
370	368	Profile 10	Dynamic USD	2018-07-02	0	4	0	21	5.5	0	0	5.5	20	10	22	0			0		0	0	0		12	0	0
371	369	Profile 10	Dynamic USD	2018-08-01	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
372	370	Profile 10	Dynamic USD	2018-09-03	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
373	371	Profile 10	Dynamic USD	2018-10-01	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
374	372	Profile 10	Dynamic USD	2018-11-01	0	4	0	21	0	0	0	11	20	10	22	0			0		0	0	0		12	0	0
375	373	Profile 10	Dynamic USD	2018-12-03	0	4	0	21	2.75	0	0	8.25	20	10	22	0			0		0	0	0		12	0	0
376	374	Profile 10	Dynamic USD	2019-01-01	0	4	0	21	0	0	0	11	20	10	22	0			0		0	0	0		12	0	0
377	375	Profile 10	Dynamic USD	2019-02-01	0	4	0	21	0	0	0	8.25	20	10	22	2.75			0		0	0	0		12	0	0
378	376	Profile 10	Dynamic USD	2019-03-01	0	4	0	21	5.5	0	0	0	20	10	22	5.5			0		0	0	0		12	0	0
379	377	Profile 10	Dynamic USD	2019-04-01	0	4	0	21	5.5	0	0	0	20	10	22	5.5			0		0	0	0		12	0	0
380	378	Profile 10	Dynamic USD	2019-05-01	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
381	379	Profile 10	Dynamic USD	2019-06-03	0	4	0	21	0	0	2.75	8.25	20	10	22	0			0		0	0	0		12	0	0
382	380	Profile 10	Dynamic USD	2019-07-01	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
383	381	Profile 10	Dynamic USD	2019-08-01	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
384	382	Profile 10	Dynamic USD	2019-09-02	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
385	383	Profile 10	Dynamic USD	2019-10-01	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
386	384	Profile 10	Dynamic USD	2019-11-01	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
387	385	Profile 10	Dynamic USD	2019-12-02	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
388	386	Profile 10	Dynamic USD	2020-01-01	0	4	0	21	5.5	0	0	0	20	10	22	5.5			0		0	0	0		12	0	0
389	387	Profile 10	Dynamic USD	2020-02-03	0	4	0	21	5.5	0	5.5	0	20	10	22	0			0		0	0	0		12	0	0
390	388	Profile 10	Dynamic USD	2020-03-02	0	4	0	21	0	0	0	11	20	10	22	0			0		0	0	0		12	0	0
391	389	Profile 10	Dynamic USD	2020-04-01	0	4	0	21	0	0	0	11	20	10	22	0			0		0	0	0		12	0	0
392	390	Profile 10	Dynamic USD	2020-05-01	0	4	0	21	0	0	0	11	20	10	22	0			0		0	0	0		12	0	0
393	391	Profile 10	Dynamic USD	2020-06-01	0	4	0	21	2.75	0	0	8.25	20	10	22	0			0		0	0	0		12	0	0
394	392	Profile 10	Dynamic USD	2020-07-01	0	4	0	21	5.5	0	0	5.5	20	10	22	0			0		0	0	0		12	0	0
395	393	Profile 10	Dynamic USD	2020-08-03	0	4	0	21	5.5	0	0	0	20	10	22	5.5			0		0	0	0		12	0	0
396	394	Profile 10	Dynamic USD	2020-09-01	0	4	0	21	5.5	0	0	0	20	10	22	5.5			0		0	0	0		12	0	0
397	395	Profile 10	Dynamic USD	2020-10-01	0	4	0	21	5.5	0	0	0	20	10	22	5.5			0		0	0	0		12	0	0

Figure 42: Different Asset Distributions per Investor Profile

4.12 Anomaly Detection in Manufacturing Settings Integrated Functionality

This use case has been described in detail in Deliverable 2.4. This section includes a brief description, extracted from the aforementioned document, focusing on its integration of SERRANO platform features and services.

The UC is developing a Data Processing Application to analyse real-time signals from the ball-screw sensors and check for anomalies, detecting anomalous behaviours that may affect the part quality and predicting imminent failures. This application has been divided into two different services that analyse the data coming from the position sensors and the data from the acceleration sensors of the ball screw.

- Position Processor Service: Classifies the difference between expected and the actual position during a time interval as normal or anomalous. The system adapts to the expected degradation of the component during its useful life. Data is gathered from position sensors (linear and angular).
- Acceleration Processor service (not yet fully defined): Classifies the vibration signal as normal or anomalous. The system adapts to the expected degradation of the component during its useful life. Data is gathered from acceleration sensors.

For an effective integration with the SERRANO project, both services are divided into three different microservices.

- Model Inference: Loads the trained classifier and classifies the new incoming stream data, predicting whether the data is anomalous or not.
- Data Manager: Manages the streaming data coming from the ball screw sensors and the predictions that are made by the Model Inference microservice. The application ensures that a limited number of data and predictions are stored as historical data. A moving window of the last **N** number of data and predictions are stored. Apart from that, it computes the distribution of the last predictions and can launch the model re-train that is made by the Classifier Training microservice.
- Classifier Training: Re-trains the model when the ball screw conditions change, which is used for classification by the Model Inference application. This re-train is done using the historical data managed by the Data Manager service.

In addition, in order to obtain streaming data from real machines, at IDEKO's facilities, a test bench has been built with two sensorized ball screws simulating data from machines in a real scenario (Machine ball screw simulator).

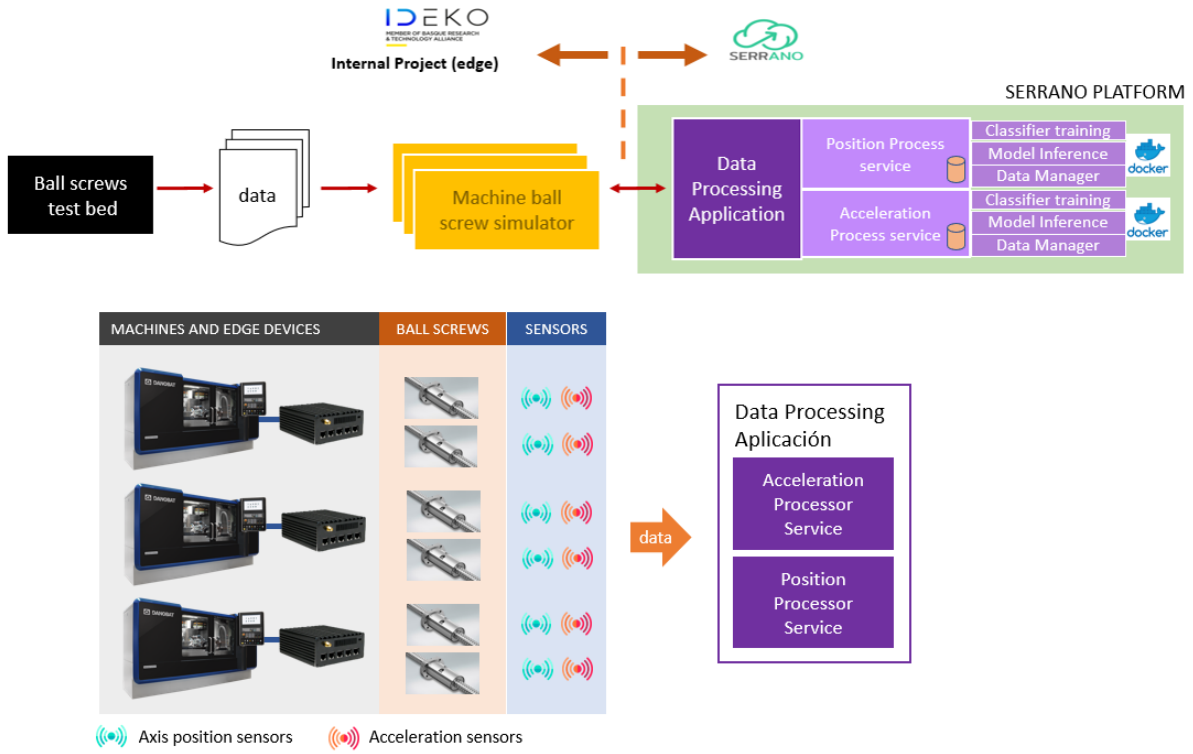


Figure 43: Developed Data Processing application

The following image shows an integration view of the data processing application (purple squares) developed at IDEKO to detect anomalies in the ball screw, with the integration of some of the components available in SERRANO (green squares) to achieve the use case requirements and goals (described in detail in Deliverable 2.4 [30]).

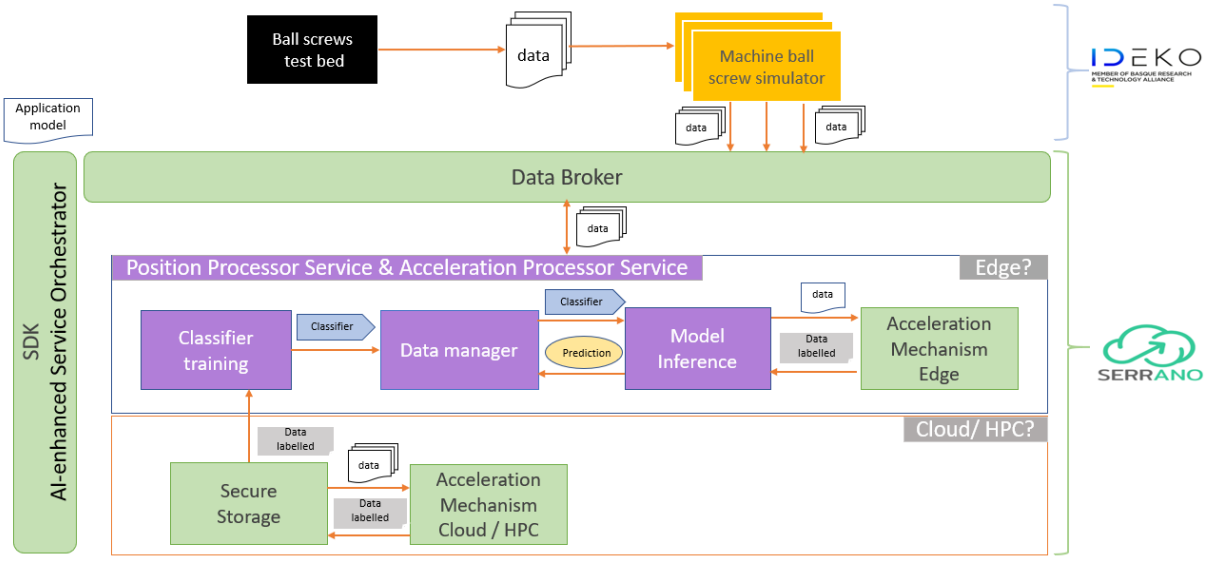


Figure 44: Draft - Defining possible integration with SERRANO components

These anomaly detection services/microservices (Position and Acceleration Processor Services) will be implemented on the SERRANO platform through the SERRANO SDK in appropriate containerized application with the corresponding Jenkins pipeline into accessible Kubernetes server, for example:

```
UVT_KUBERNETES_PUBLIC_ADDRESS= k8s.serrano.cs.uvt.ro'
```

To facilitate the transparent deployment and management of the services/microservices, the appropriate description of the execution requirements should be provided to the AI-enhanced Service Orchestrator using the Application Model, in addition to the deployment description YAML. The SERRANO orchestration mechanisms (AI-enhanced Service Orchestrator and Resource Orchestrator) will then translate these requirements into SERRANO infrastructure-specific runtime parameters and map them to the available resources (Resource Optimization Toolkit).

The streaming data integration with SERRANO platform will be done through the Data Broker component (detailed in Section 4.4). Data Broker will provide an interface based on the MQTT protocol to facilitate the publication and consumption of the data generated from the simulated machines' ball screws to use case applications/services and other SERRANO components.

In addition, the developed anomaly detection services will leverage the available interfaces of the SERRANO acceleration mechanisms in edge/cloud (Section 4.7) and HPC (Section 4.6) to enable their access to the SERRANO-enhanced computational resources. These resources will provide better performance and optimization of the kernels used (e.g., DTW, KMeans, KNN or FTT) by the Model Inference and Classifier Training services. Moreover, the S3-compatible the Secure Storage (detailed in Section 4.8) interface will be used to store the last N streaming data received through the Data Broker. This way, the required data will be stored and accessible by all SERRANO components and the use case services.

The idea is to reduce the classifier training time and the time needed to make a new prediction through the streaming data. This will enable the early detection of possible imminent failures of the ball screw, eliminating also their occurrence. In addition, it will provide greater control of the health status of the ball screw in real time. Since the current techniques and resources available at the edge cannot support the above operations, the SERRANO platform is needed.

4.12.1 Integration details and REST APIs

The Position and Acceleration Processor Services communicate with external agents using a MQTT broker. Not only that, but the microservices inside these services also communicate with each other using this communication protocol. Next, we present the available topics that are built following an established structure for all machines:

- **data** -> All data exchange is under this topic.
    - **{machine1}** -> All data exchange related to {machine1} is under this topic.

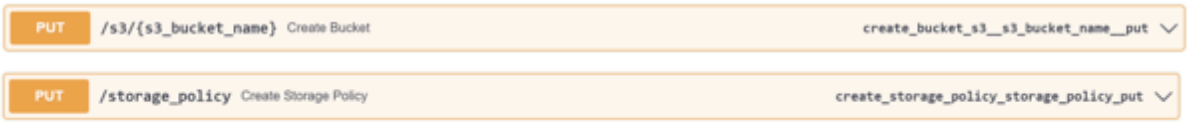
- **position** -> All data exchange from Position Processor Service related to {machine1} is under this topic.
  - **degradation**. Where the simulated degradation is published. This simulated degradation is used to force the classifier model to be retrained more frequently.
  - **cycle**. Where the gathered data from sensors is published.
  - **inference** -> All data exchange from Position Processor Service related to {machine1} that has to do with model inference is under this topic.
    - **prediction**. Where the prediction made by the classifier is published. The model predicts whether the last received data is anomalous (1) or normal (0).
    - **time**. Where the time that has been needed to make a prediction is published.
    - **regular**. Where the percentage of regular predictions in the last ones is published.
    - **anomalous**. Where the percentage of anomalous predictions in the last ones is published.
  - **training** -> All data exchange from Position Processor Service related to {machine1} that has to do with model training is under this topic.
    - **flag**. When a "1" is published on this topic, the Classifier Training microservice retrains the model.
    - **status**. Where the status messages of the training pipeline are published.
    - **time**. Where the time that has been needed to train the classifier is published.
- **acceleration** -> All data exchange from Acceleration Processor Service related to {machine1} is under this topic.

- **{machine2}**

…

Figure 45 shows a detailed workflow for integrating the use case services within the SERRANO platform. Before starting the data streaming, the Position and Acceleration Processor Service microservices must be subscribed to their corresponding topics. The Model Inference Service and the Data Manager must be subscribed to all topics, while the Classifier Training has to be subscribed to the "training flag" of all machines.

Additionally, before subscribing to the data streaming in order to store data in Secure Storage, a storage policy will be created through the provided API, and it will be applied to the new S3 bucket, using the following HTTP PUT methods.



After the S3 bucket and the storage policy are created, all the microservices shall be subscribed to their corresponding topics.

The Machine Ball Screw Simulator that is deployed at IDEKO's facilities collects the sensor data, pre-processes them, and then publishes them into the Data Broker via the MQTT protocol. At this point, all the subscribed elements to the cycle topics (the Model Inference and the Data Manager) will receive this data.

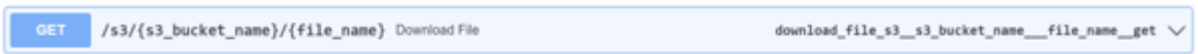
The Data Manager, which is subscribed to the Data Broker, receives the data and saves then as a CSV file in a shared volume. If this file already exists, the previous file is opened, and the latest data received is inserted as a new column. The application ensures that a limited number of data is stored as historical data in columns. This limited number of columns is defined in the microservice's configuration file. A moving window of the last N number of data is stored. So, depending on the available data columns in the file, if necessary, the oldest column is removed before appending the new data. After storing the CSV file in the shared volume, the same file is sent to the secure storage through the following PUT request, replacing the previous file if it exists.

At the same time, the Model Inference microservice, which is also subscribed to the MQTT Broker, reads streaming data and loads the classifier model from a known path. Using that pre-trained model classifies the data as anomalous or normal. Model Inference can classify using the edge device resources, or it could also leverage the SERRANO's acceleration mechanisms through their exposed API to reduce execution time. In this case, a trained KNN will be provided along with the input data in order to accelerate the task of making a prediction using KNN acceleration Kernel.

After making the classification, the prediction is published into the Data Broker and the Data Manager application saves that prediction. As it happens with the data, the Data Manager takes care of storing a moving window of the last N number and the last predictions.

After saving the prediction, the Data Manager checks if the anomalous percentage value is higher than 80 of the last number of predictions. If it is not, it repeats the described process, but if it is, the data manager publishes a retrain flag in the Data Broker.

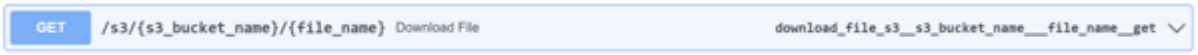
The Classifier Training, which is subscribed to the training flag, receives a "1" as a flag value which triggers the training pipeline. During this training pipeline, the acceleration mechanisms at the selected resources can retrieve the label data through a GET request. Also, the acceleration mechanism makes a GET request to get the last stored filename in S3.



The Secure Storage returns the historical data and the acceleration mechanisms label the retrieved data, training a cluster using DBScan or KMeans based on DTW metric acceleration kernels.

After labelling the data, these labels are saved in the Secure Storage using the same PUT request.

Apart from the file with the stored labels, the process requires the file with the historical data to train the KNN classifier. For that, it makes a GET request to the Secure Storage service.



When the required data are available, the classifier can be trained, and like the Model Inference, the Classifier Training can train the KNN model on its own or it through the acceleration mechanisms (KNN kernel). During this process, the historical data and the corresponding labels (the ones that have been calculated before) will be used.

Once the classifier has been trained, it is stored in the Data Manager, and then the Data Manager make a PUT request to store the file in the Secure Storage.

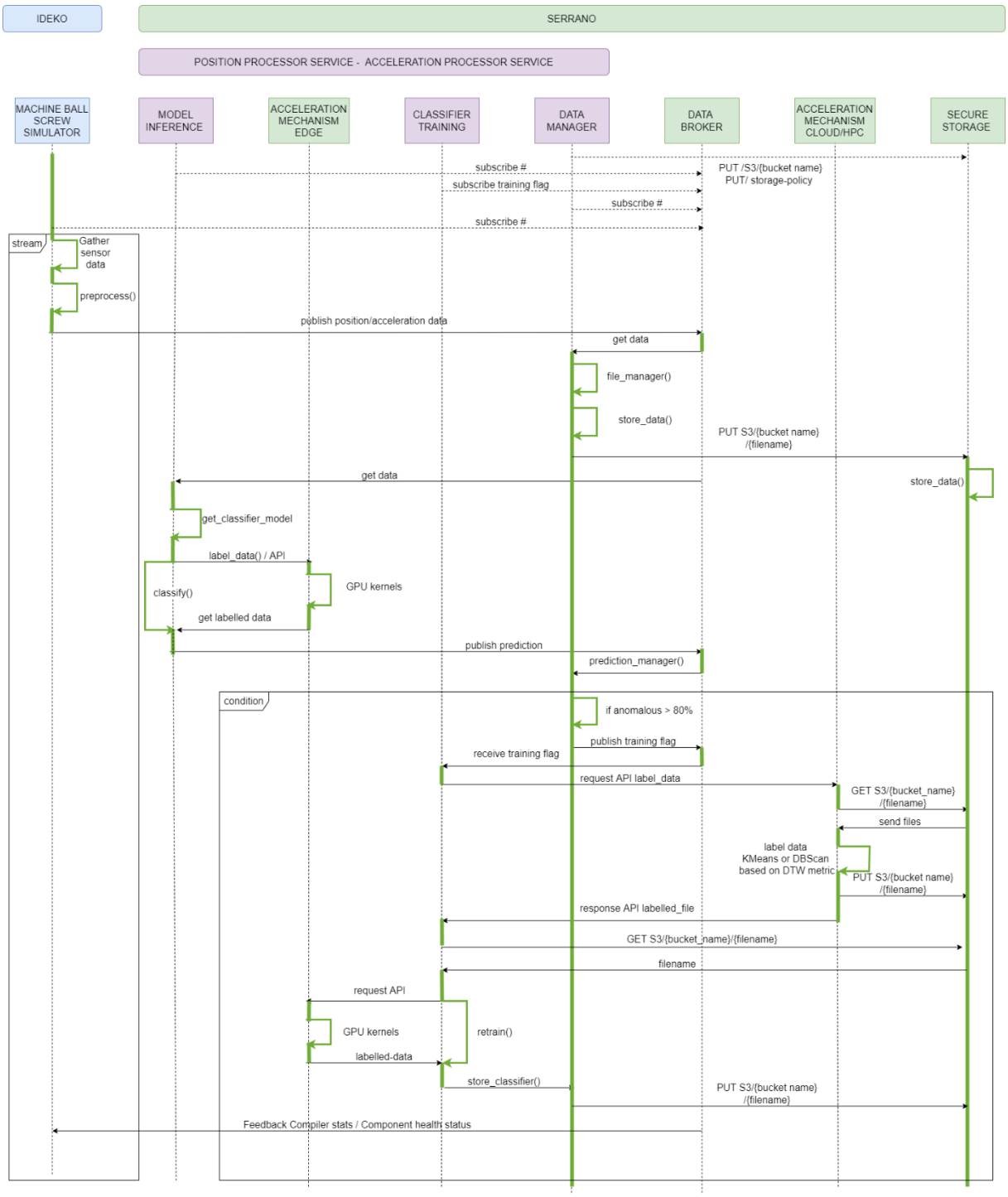


Figure 45: Integration workflow

As an example of the tests carried out internally at IDEKO, the topics generated for communication between the microservices are shown through the MQTT Broker, as well as the published and received data from the streaming data generated in the test bench.

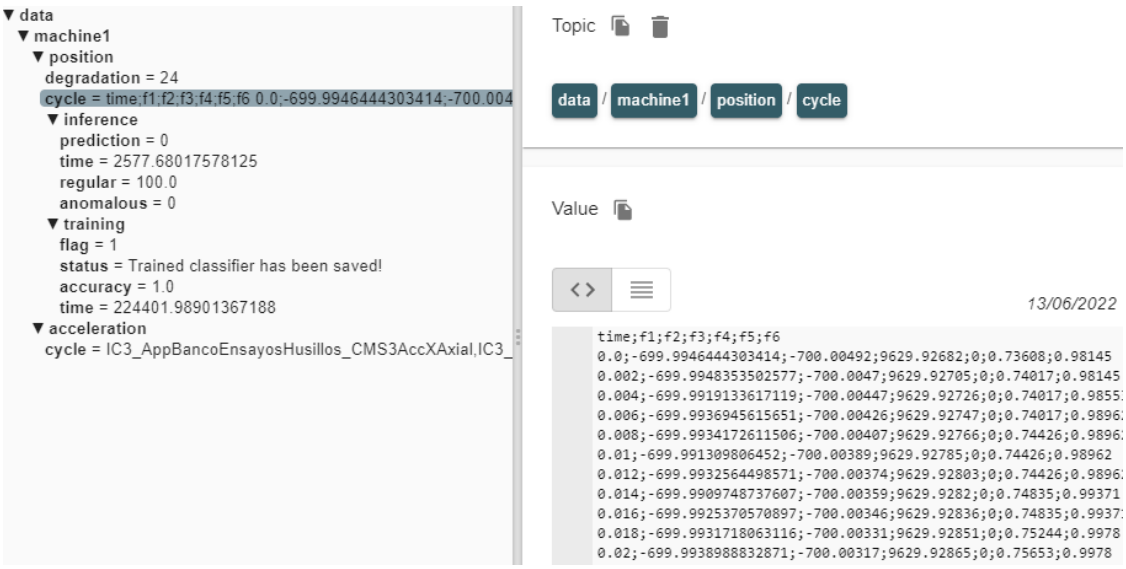


Figure 46: Internal topics generated for communication between the microservices through the MQTT Broker

In addition, a website for aggregated results and statistics (e.g., inference time, classifier time, predictions, classifier training accuracy, etc.) is developed to visualize the status of the ball screw in real time.

Figure 47: Internal website for aggregated results and stats

4.12.2 UC Integration with Data Broker

The use case will make use of the capabilities of the Data Broker component through the exposed publish-subscribe interface, in this case using the MQTT protocol (The Standard for IoT Messaging), to forward the data generated by the use case machines to the services of the SERRANO platform. To this end, a connector will be created in the machine ball screw simulation application that will publish the data in the SERRANO Data Broker component, from where the services/components will consume the data through the corresponding subscriptions, as explained in the previous point.

The results (e.g., predictions, models, data, etc.) could also be exchanged through the MQTT protocol exposed in the Data Broker component.

The image below shows the correlation between the internal tests at IDEKO for sending streaming data with the MQTT protocol and how it will be integrated through the Data Broker component in the SERRANO platform.

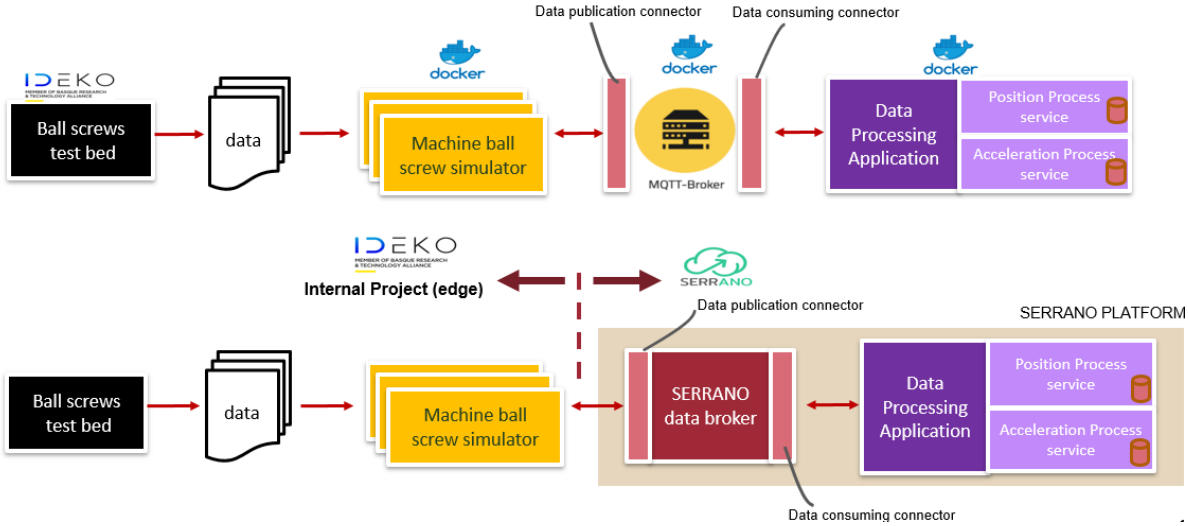


Figure 48 Data Broker component to send data from machine ball screw simulator to SERRANO platform.

Bellow, we present some examples of the subscriptions of the different microservices to the topics generated in the MQTT Broker installed to test the services defined internally in IDEKO. For the internal deployment of these services in container images, a Python 3.7.2 image has been used, while the MQTT Broker was based on the mosquito image.

- MQTT Broker (internally in IDEKO to do tests, which will be replaced by the SERRANO Data Broker (Message Broker – MQTT protocol)):

```
docker run -d --name mqtt-broker -p 1883:1883 -p 9001:9001 -v mqtt-broker-confi
g:/mosquitto/config -v mqtt-broker-data:/mosquitto/data mqtt-broker
```

- Machine ball screw simulator (IDEKO internal service for sending data stream to SERRANO):

```
docker run -d --name machine-simulator -v machine-simulator-dataset:/app/datase
t -v machine-simulator-config:/app/config -v machine-simulator-logs:/app/logs m
achine-simulator
```

```
// Publish machine data from position and acceleration sensors into MQTT broker

mqttc_a = mqtt.Client(config['machine']['id'] + '_acceleration_publisher')
mqttc_p = mqtt.Client(config['machine']['id'] + '_position_publisher')
…
mqttc_p.loop_start()

acceleration=threading.Thread(target=acceleration_publisher,args=(mqttc_a,))
position = threading.Thread(target=position_publisher, args=(mqttc_p,))

position.start()
acceleration.start()
```

- Model Inference (Position Processor Service):

```
docker run -d --name model-inference -v data:/app/data -v model-inference-confi
g:/app/config -v model-inference-logs:/app/logs model-inference
```

```
// Subscribed to the data topic to receive the streaming data and then classify
them as anomalous or not

def on_connect(mqttc, obj, flags, rc):
    mqttc.subscribe("data/+/position/cycle", 0)

def on_message(mqttc, obj, msg):
    classify(msg.topic, msg.payload, mqttc)
```

- Data Manager (Position Processor Service):

```
docker run -d --name data-manager -v data:/app/data -v data-manager-config:/app
/config -v data-manager-logs:/app/logs store-data
```

```
//Subscribed to the data and prediction topics to receive the streaming data
and predictions, managing data and predictions

def on_connect(mqttc, obj, flags, rc):
```

```
    mqttc.subscribe("data/+/position/cycle", 0)
    mqttc.subscribe("data/+/position/inference/prediction")

def on_message(mqttc, obj, msg):
    if msg.topic == "data/machine1/position/cycle":
        file_manager(msg.topic, msg.payload)
    elif msg.topic == "data/machine1/position/inference/prediction":
        prediction_manager(msg.topic, msg.payload)
```

- Classifier Training (Position Processor Service):

```
docker run -d --name classifier-training -v data:/app/data -v classifier-traini
ng-config:/app/config -v classifier-training-logs:/app/logs classifier-training
```

```
// Subscribed to the training flag topic to re-train and create new classifier
model

def on_connect(mqttc, obj, flags, rc):
    mqttc.subscribe("data/+/position/training/flag", 0)

def on_message(mqttc, obj, msg):
    training(msg.topic, msg.payload, mqttc)
```

5 Development and Integration Environment

SERRANO adopts and applies the Continuous Integration and Continuous Delivery/Deployment (CI/CD) practices to set up a standardized process for developing and releasing the software components of the SERRANO platform. In Continuous Integration development environments, team members frequently integrate their new or changed code with the mainline codebase. The CI/CD pipeline, which implements the underlying methodology, includes methods and principles that enable development teams to deliver high-quality code more frequently and reliably. This is accomplished by automating both building and testing and by testing code locally, prior to testing the integration with mainline. Therefore, Continuous Integration facilitates the release process in terms of speed, debugging and development cycle optimization.

Continuous Delivery is the next step in the CI/CD pipeline as an extension of Continuous Integration. Continuous Delivery is the ability to transmit changes of all types including new features, configuration changes, bug fixes and experiments into production safely and quickly in a sustainable way. In the Delivery phase automated build tools are executed to generate an artifact which will be delivered to the end-users. As a result, this step enables frequent releases automatically and accelerates delivering high-quality software while minimizing the risks associated with releasing software.

Continuous Deployment goes one step further than Continuous Delivery. The deployment phase ensures that every change committed is applied in production automatically and distributed to the end-users. This process utilizes the validated features in a staging environment and deploys them into the production environment. The goal of continuous deployment is to release applications to end users faster and more cost-effectively by managing small, incremental software changes which pass through the entire pipeline.

In high level all the described phases are techniques that implement the DevSecOps ideals. DevSecOps is a set of practices that combines software development (Dev), security (Sec) and IT operations (Ops) to improve their communication and collaboration and automate the integration of security at every phase of software development lifecycle. Moreover, it aims to shorten the systems development life cycle and provide continuous delivery with high software quality, security, speed and efficiency based on the Agile methodology.

5.1 DevSecOps and Continuous integration/Continuous Delivery practices

The following sub-sections contain the description of the practices that are part of the usual development process that begins with the code being written on the developer's IDE and ends with the delivery of the application, which is packaged in a container image. The information that is produced by Static Application Security Testing (SAST), Source Composition Analysis

(SCA), and Container Image Scanning in the CI/CD server(s) is used as input by vulnerability management tools that facilitate the security assessment of applications.

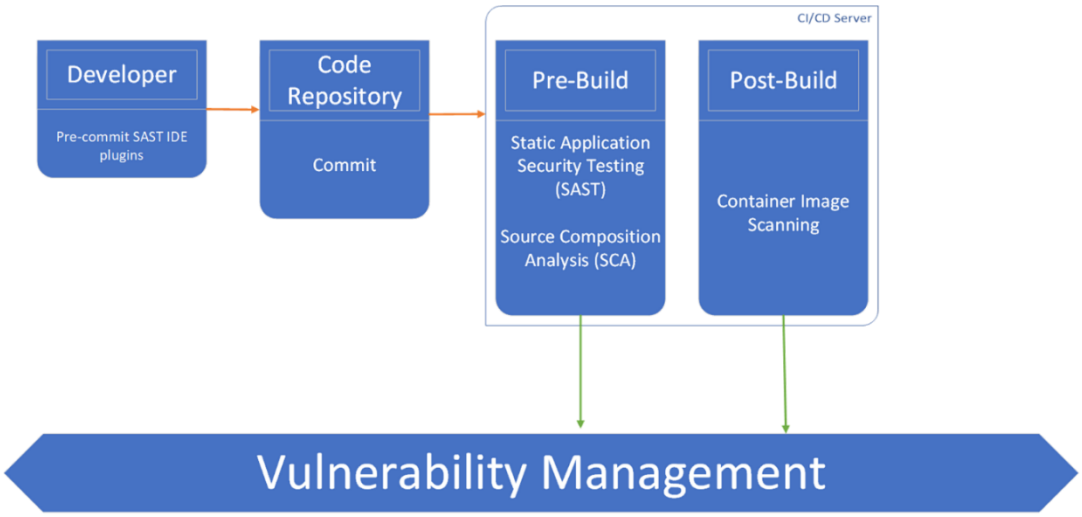


Figure 49: Vulnerability management in CI/CD

5.1.1 Static Application Security Testing (SAST)

Static application security testing (SAST) is used to secure software by reviewing the source code of the software to identify sources of vulnerabilities. SAST tools can check for quality issues and there are cases that they can offer architectural testing as well. The earlier a vulnerability is fixed in the SDLC, the cheaper it is to fix. However, early integration of SAST generates many bugs, which might distract developers from features and delivery.

SAST is particularly useful for weeding out low-hanging fruits like SQL-Injection and Cross-Site Scripting (XSS). Among the results, someone can find many false-positives that need manual oversight to be managed.

Figure 50: SAST by SonarQube

SAST tools such as SonarQube [40] can be deployed locally as well as at a central location, from where the scan results can be obtained. Other SAST tools such as SonarLint [41] can just be installed on the IDE and provide real-time feedback and remediation guidance.

5.1.2 Software Composition Analysis (SCA)

Software Composition Analysis (SCA) is the process of identifying the components that comprise a given piece of software. This is an essential part in identifying and reducing risk in the software supply chain. This identification process begins from the production of a Software Bill of Materials (SBOM), which can follow a standard such as OWASP CycloneDX [42].

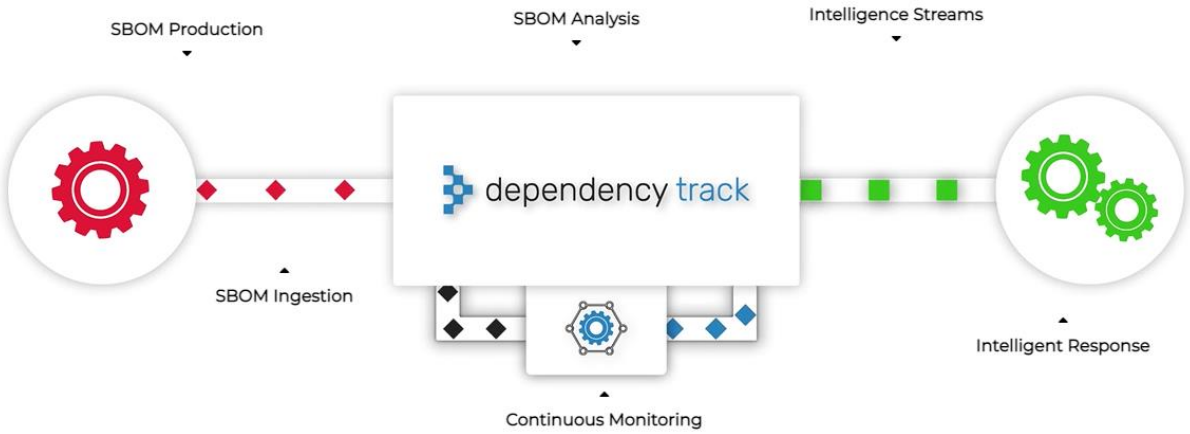


Figure 51: SBOM Operations using Dependency-Track

An example of the operations that the SBOM is involved, is visible in Figure 51. The most important stages and components that appear in this example are the following:

- SBOM Production: CycloneDX Software Bill of Materials created during CI/CD or acquired from suppliers
- SBOM Ingestion: SBOMs published to Dependency-Track [53] via REST, Jenkins plugin, or uploaded through web interface
- SBOM Analysis: Analyzes components for security, operational, and license risk
- Continuous Monitoring: Continuously analyzes portfolio for risk and policy compliance
- Intelligence Streams: Produces real-time analysis and security events delivering actionable findings to external systems
- Intelligent Response: Events delivered via webhooks, or chat-ops and findings published to risk management and vulnerability aggregation platforms

SCA performs checks to identify vulnerable or outdated 3rd party libraries. Most software nowadays is built on frameworks which causes the resulting software to comprise mostly of third-party libraries. For example, software that uses Jackson can inherit the vulnerabilities of this library, which need to be assessed to make important decisions.

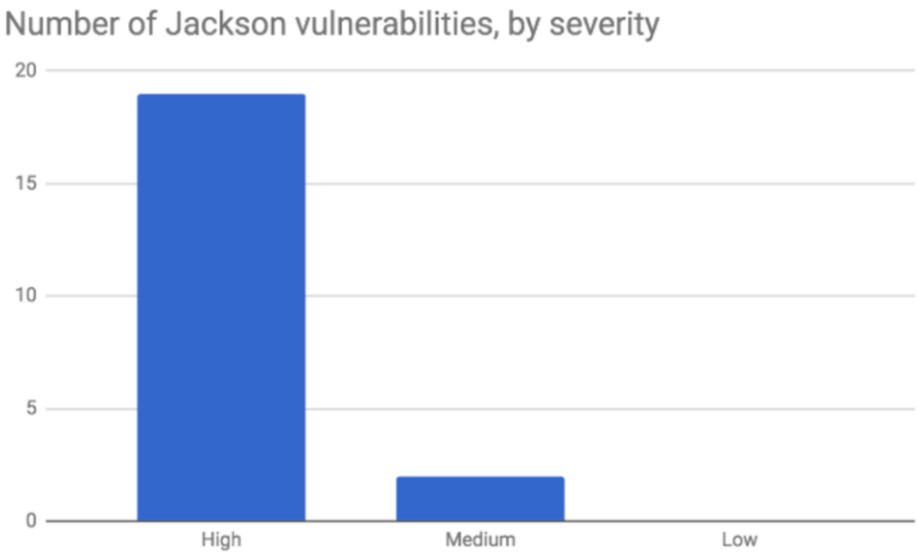


Figure 52: Severity assessment of Jackson vulnerabilities

5.1.3 Container Image Scanning

Image Scanning tools belong to Dynamic Application Security Testing (DAST) category of tools and cannot provide a complete picture of vulnerabilities in an application but can detect critical security vulnerabilities originating from various parts of the containerized service, such as the application itself or the operating system. One such service that is used in SERRANO is Trivy [49], which is described in section 5.2.6.1.

5.1.4 Vulnerability Management

Vulnerability management is the "cyclical practice of identifying, classifying, prioritizing, remediating, and mitigating" software vulnerabilities. Vulnerability management is integral to computer security and network security and must not be confused with vulnerability assessment.

In SERRANO, vulnerability management is facilitated through DefectDojo [54], which is described in section 5.2.8 and Jenkins plugins that integrate the scan results of SonarQube (section 5.2.4) and Dependency-Track (section 5.2.7).

5.2 SERRANO Continuous Integration/Continuous Delivery stack

The SERRANO Continuous Integration/Continuous Delivery stack is a collection of open-source software components, which collectively aim to create an automated build system capable of integrating changes performed by developers working on individual components.

The software components that comprise the SERRANO Continuous Integration/Continuous Deployment platform have been deployed as Kubernetes pods on a Kubernetes cluster, as depicted in Figure 53.

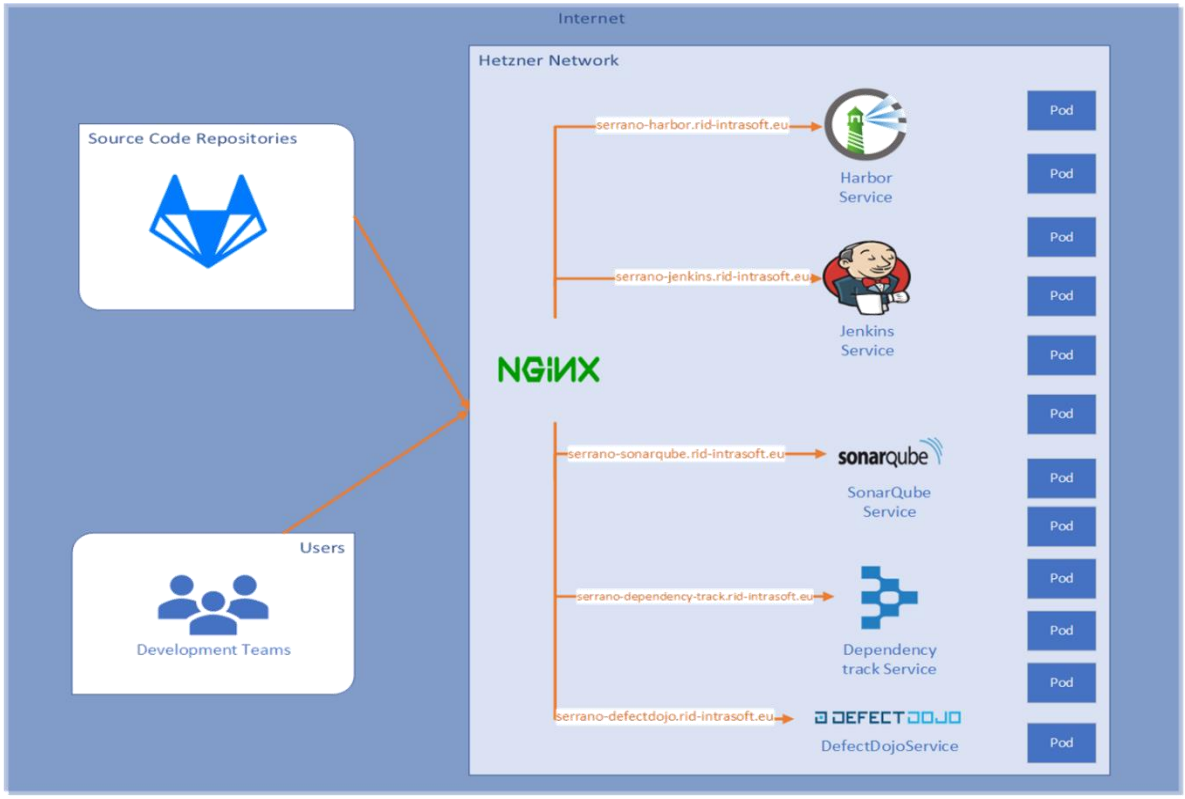


Figure 53: SERRANO CI/CD components

5.2.1 Version Control System - GitLab

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems (VCS) are software tools that help software teams manage changes to source code over time. VCS keeps track of every modification to the code in a special kind of database. It is a remote repository of files that comprise the source code of a software application. If a mistake is made, developers can compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members. Git is one of the most popular free and open-source distributed version control systems.

GitLab [43] is a software that provides remote access to Git repositories. It offers a web-based graphical interface with several built-in features, such as version control, issue tracking, code review, wiki, etc. Multiple developers can concurrently create, merge and delete parts of the code they are working on independently, at their local system before applying the changes to the shared GitLab repository.

For the needs of the SERRANO project, a dedicated and private Gitlab group named "SERRANO" has been built, as depicted in Figure 54. Under this group, whose URL is https://gitlab.com/serranoproject, the code owners can create multiple repositories for the SERRANO components. Each partner has the appropriate access rights, permissions, and restrictions to create repositories, organize users in teams and upload their source code.

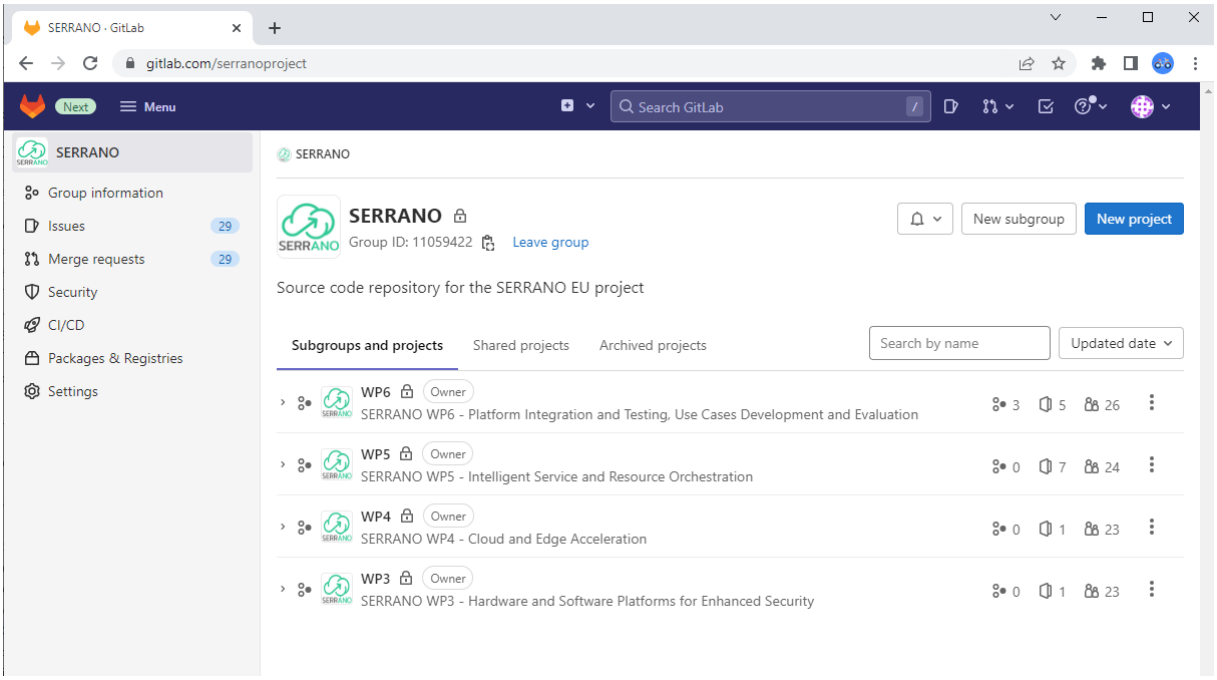


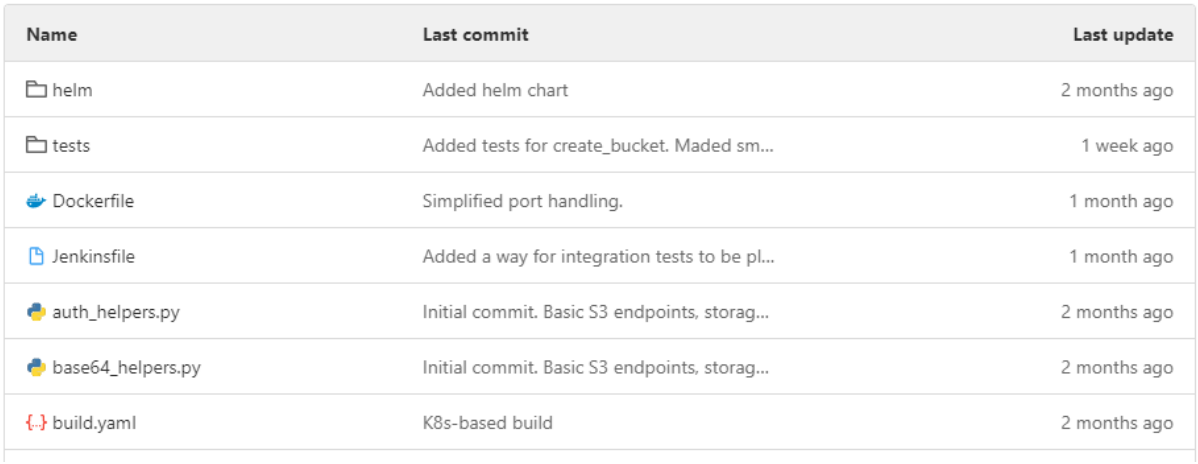
Figure 54: The SERRANO GitLab group and its contents

Several subgroups were generated to store the source code and scripts related to the SERRANO software components. Each WP subgroup includes other subgroups named after the SERRANO individual tools. Each of these subgroups contains files that are used for the execution of the Continuous Integration and Continuous Deployment tests. Such files are the following:

- Jenkinsfile: A text file that contains the definition of a Jenkins Pipeline, including the steps that will be followed during the CI/CD process.
- Dockerfile: A text-based script of instructions that is used to create a container image.
- build.yaml: A YAML file that contains the definition of the Kubernetes Pod that will be used in the Jenkins pipeline in the steps of building the tool.

Additionally, there is usually a folder that contain the helm charts to be used for deploying the component. In case of a tool, such as a library, that can be imported or dynamically linked, a HELM [57] chart is not generally applicable.

GitLab [43] provides native VCS features such as branches. Branching is the practice of creating copies of programs or objects in development to work in parallel versions, retaining the original and working on the branch or making different changes to each. In most circumstances, a repository has one main, or master, branch from which each developer working on a particular feature or bug patch produces a separate, divergent branch. When the developers are finished with their source code changes, they merge their side branch back into the main branch.



Name	Last commit	Last update
helm	Added helm chart	2 months ago
tests	Added tests for create_bucket. Maded sm...	1 week ago
Dockerfile	Simplified port handling.	1 month ago
Jenkinsfile	Added a way for integration tests to be pl...	1 month ago
auth_helpers.py	Initial commit. Basic S3 endpoints, storag...	2 months ago
base64_helpers.py	Initial commit. Basic S3 endpoints, storag...	2 months ago
build.yaml	K8s-based build	2 months ago

Figure 55: Structure of a SERRANO component on GitLab

5.2.2 Continuous Integration - Jenkins

Jenkins [44] was selected as the Continuous Integration server of the CI/CD stack for SERRANO. It operates as a Kubernetes deployment upon Kubernetes workers running on dedicated servers on Hetzner [45] infrastructure and its URL https://serrano-jenkins.rid-intrasoft.eu.

Continuous Integration is a software development practice where developers, as members of a team, regularly merge their code changes into a central repository, leading to multiple integrations per day. Continuous Integration process automates the integration of code changes from multiple contributors into a single software project. Each integration cycle introduces automated builds and unit tests on the latest code changes to immediately surface any errors. The key goals of continuous integration are to find and address bugs quicker, improve software quality and reduce the time it takes to validate and release new software updates.

The steps in the Continuous Integration process are as follows:

- On their local repository, software engineers make changes to the source code.
- They commit the modifications to the shared repository after that.
- As changes are made, a notification is sent to the Continuous Integration server.
- The Continuous Integration server downloads the most recent source code, builds the application, and runs unit and integration tests.
- The server also provides testable deployable artifacts.
- The Continuous Integration server gives the version of code it just built a build tag.
- The Continuous Integration server provides the development team with reports on successful builds and tests, as well as notifications if a build or test fails.
- The issues will be resolved as soon as feasible by the team.
- Throughout the duration of the project, the server continues to integrate and execute tests.

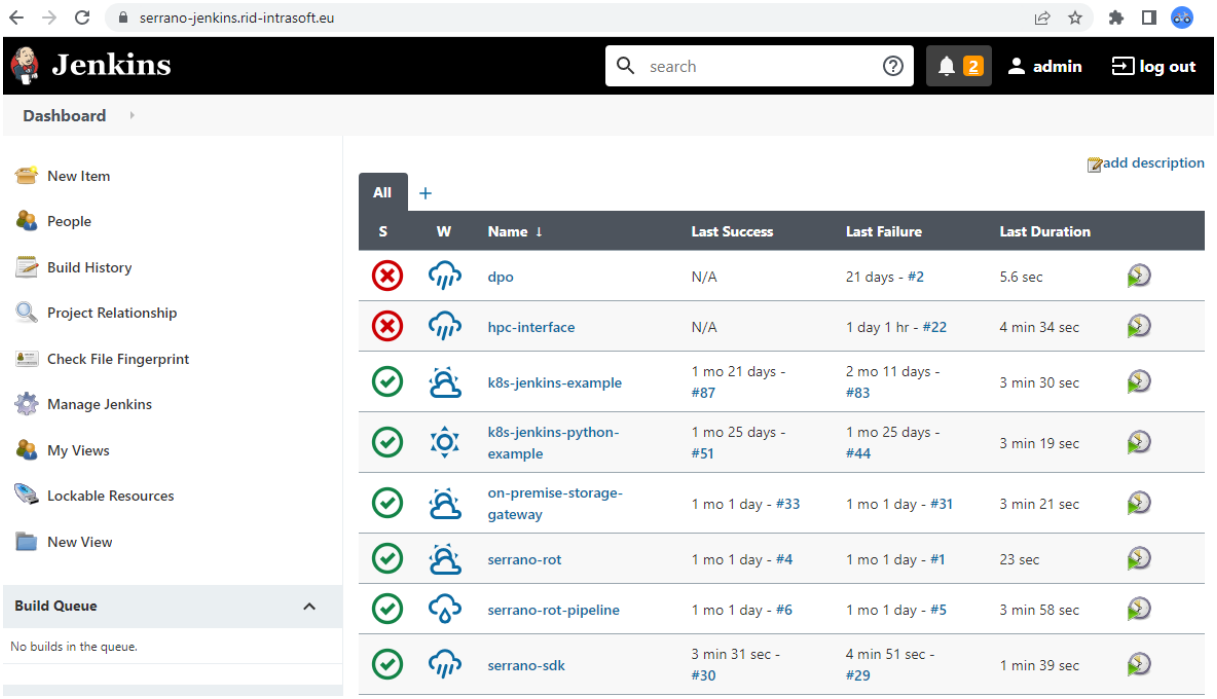


Figure 56: Jenkins Dashboard

A Jenkins pipeline is a set of events, workflows, or jobs that are connected in a certain order within Jenkins Continuous Integration. These actions or tasks are connected to the construction, testing, packaging, deployment, and storage functions. Moreover, Jenkins's pipeline contains a set of modules or plugins that enable the creation and integration of Continuous Delivery pipelines within Jenkins along with other functionalities and integrations with external tools. For the needs of the SERRANO software components several plugins have been installed and configured accordingly such as Kubernetes, SonarQube and OWASP Dependency-Track.

The repositories existing under the SERRANO group in GitLab, will be connected to a Jenkins Pipeline job, like the ones depicted in the Figure 56 above. The Jenkins Dashboard is a list of the folders and individual projects that logged in users have access to view and manage accordingly. Each Pipeline is usually described in a specific file called Jenkinsfile. Every event that will occur as a result of source code changes on the GitLab repositories (e.g., commit, merge, pull request, tag, etc.) triggers a new build to the respective pipeline on Jenkins. The Pipeline includes compilation, build and test stages as shown in the figure above. The stages are the ones declared into the Jenkins file and each build step consists of these stages. A successful stage is coloured with green and a failure with red colour.

5.2.3 Docker

Docker [46] is a free and open platform for building, deploying, and operating applications. Docker allows you to decouple your applications from your infrastructure, allowing you to swiftly release software. You can manage your infrastructure in the same way that you control your applications with Docker. You may drastically minimize the time between writing code

and executing it in production by utilizing Docker's approaches for shipping, testing, and deploying code quickly.

Docker allows to bundle and run an application in a container, which is a loosely isolated environment. Because of the isolation and security, multiple containers may operate on the same host at the same time. Containers are small and include everything needed to operate an application, so there is no need to rely on what's already on the host. Docker provides tools and a respective platform for managing container lifecycles.

Docker is built on a client-server model. The Docker client communicates with the Docker daemon, which handles the construction, execution, and distribution of your Docker containers. Figure 57 represents the basic parts of the Docker architecture.

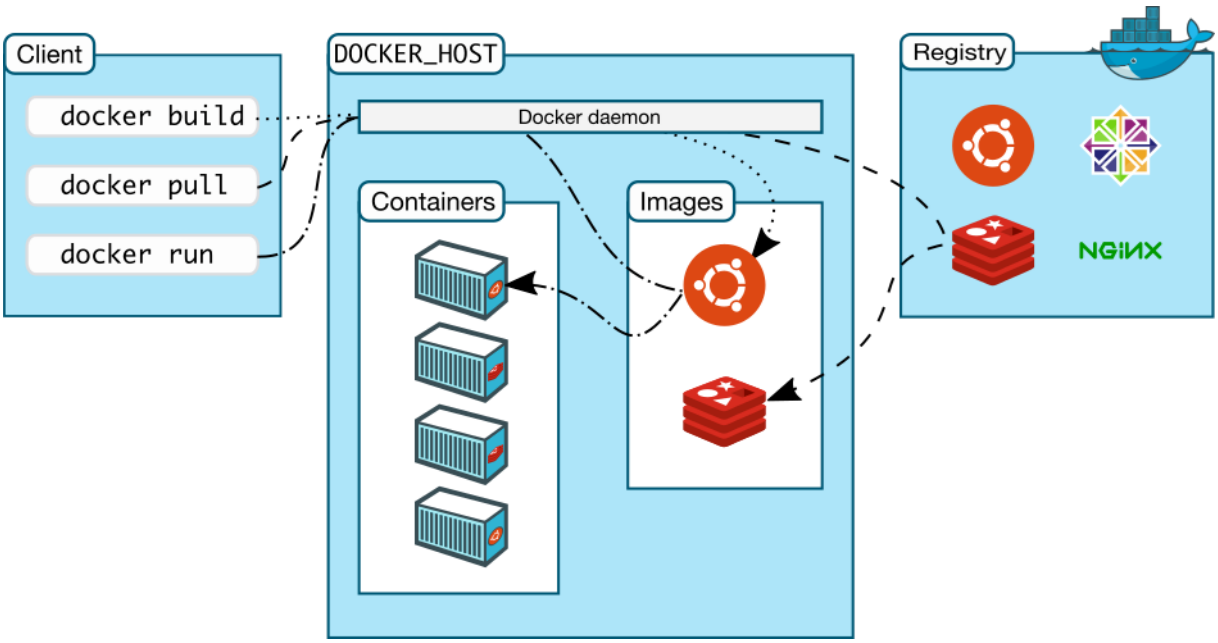


Figure 57: Basic parts of the Docker architecture

The Docker client and daemon can be both executed on the same machine, or a Docker client can be linked to a Docker daemon that is located elsewhere. A REST API, UNIX sockets, or a network interface are used by the Docker client and daemon to communicate. Docker Compose is another Docker client that allows to interact with applications made up of many containers.

5.2.4 SonarQube

SonarQube [40] collects and analyzes source code, measuring quality and providing reports and metrics and information on the key findings. It combines static and dynamic analysis tools and enables quality to be measured continuously over time. Everything that affects the code base, from minor styling details to critical design errors, is inspected and evaluated by SonarQube, thereby enabling developers to access and track code analysis data ranging from

styling errors, potential bugs, and code defects to design inefficiencies, code duplication, lack of test coverage, and excess complexity.

SonarQube supports many languages through built-in rulesets and can also be extended with various plugins. It can be fully integrated into Jenkins pipelines. SonarQube usage in SERRANO will be to perform both security and quality assurance tests. The URL to the SonarQube service in SERRANO is the following:

https://serrano-sonarqube.rid-intrasoft.eu/

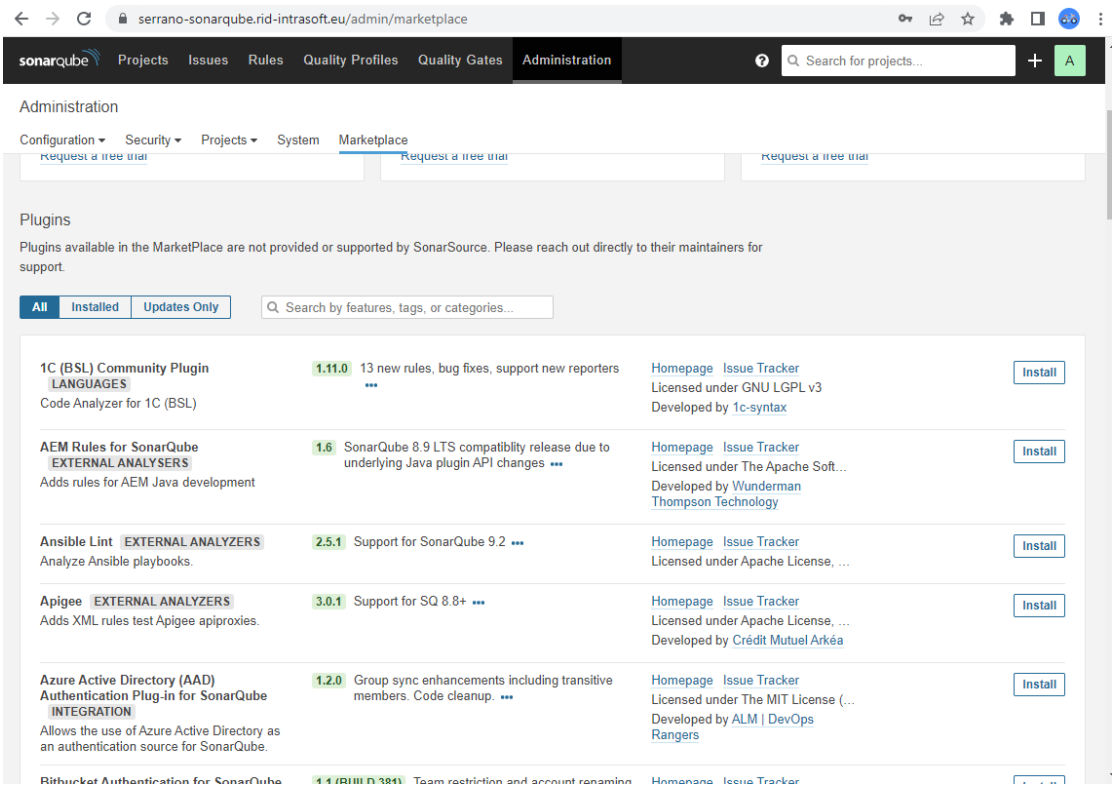


Figure 58: SonarQube in SERRANO CI/CD

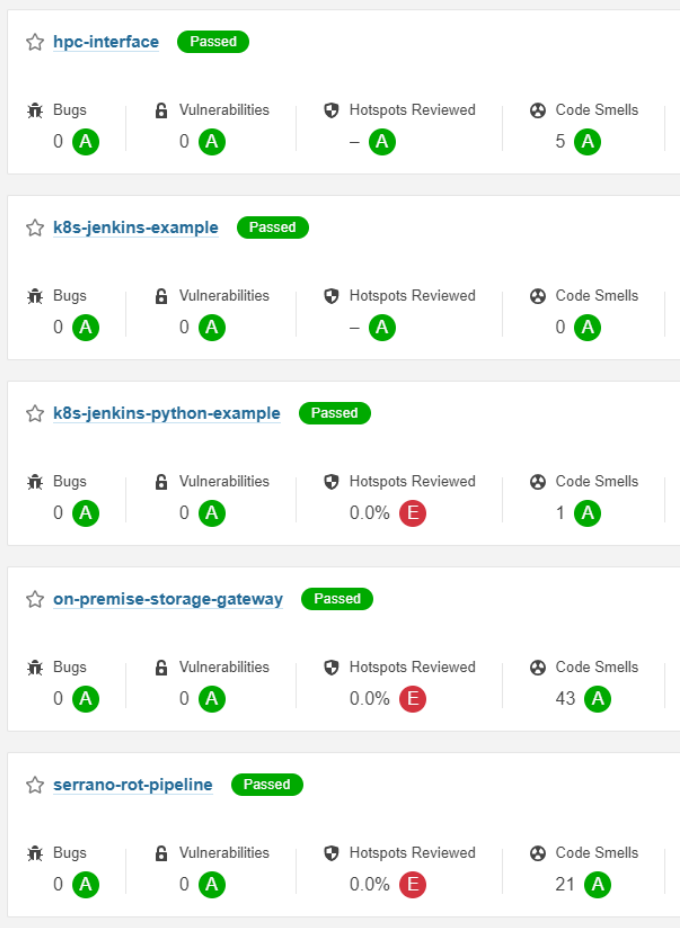


Figure 59: SonarQube scan results overview

5.2.5 NGINX

A reverse proxy server is a type of proxy server that typically sits behind the firewall in a private network and directs client requests to the appropriate backend server. NGINX [48] has been chosen to be deployed as a reverse proxy in front of SERRANO CI/CD services. As depicted in Figure 53, NGINX is located in front of the services that run inside the Kubernetes cluster. Specifically, it runs as part of the Ingress NGINX controller of the cluster.

5.2.6 Harbor

Harbor [47] is an open-source registry that secures artifacts with policies and role-based access control, ensures images are scanned and free from vulnerabilities, and signs images as trusted. Harbor delivers compliance, performance, and interoperability towards securely managing artifacts across cloud native compute platforms like Kubernetes and Docker. The URL to the harbor service in SERRANO is the following:

https://serrano-harbor.rid-intrasoft.eu/

5.2.6.1 Trivy

Within Harbor, Trivy [49] has been configured as the vulnerability scanning engine for Docker containers that are pushed to the Docker registry. Trivy is a simple and comprehensive vulnerability/misconfiguration/secret scanner for containers and other artifacts. Trivy detects vulnerabilities of OS packages (Alpine [50], RHEL [51], CentOS [52], etc.) and language-specific packages (Bundler, Composer, npm, yarn, etc.). In addition, Trivy scans Infrastructure as Code (IaC) files such as Terraform and Kubernetes, to detect potential configuration issues that expose your deployments to the risk of attack. Trivy also scans hardcoded secrets like passwords, API keys and tokens.

As depicted in Figure 60 and Figure 61, Trivy automatically scans the artifacts that are pushed to the registry and produces a report which lists all vulnerabilities detected in the Docker image as well as details for each one. Additionally, the severity of each vulnerability and the version of the package that each vulnerability was fixed are included.

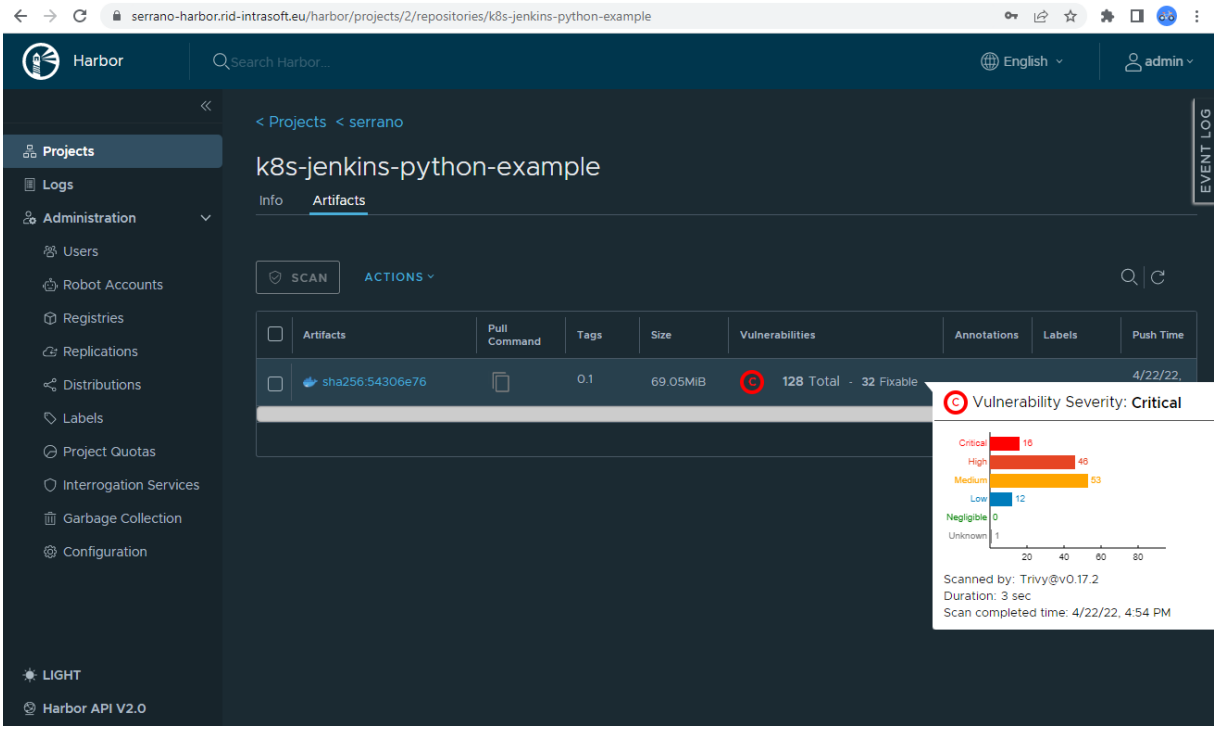


Figure 60: Trivy scan result overview on Harbor

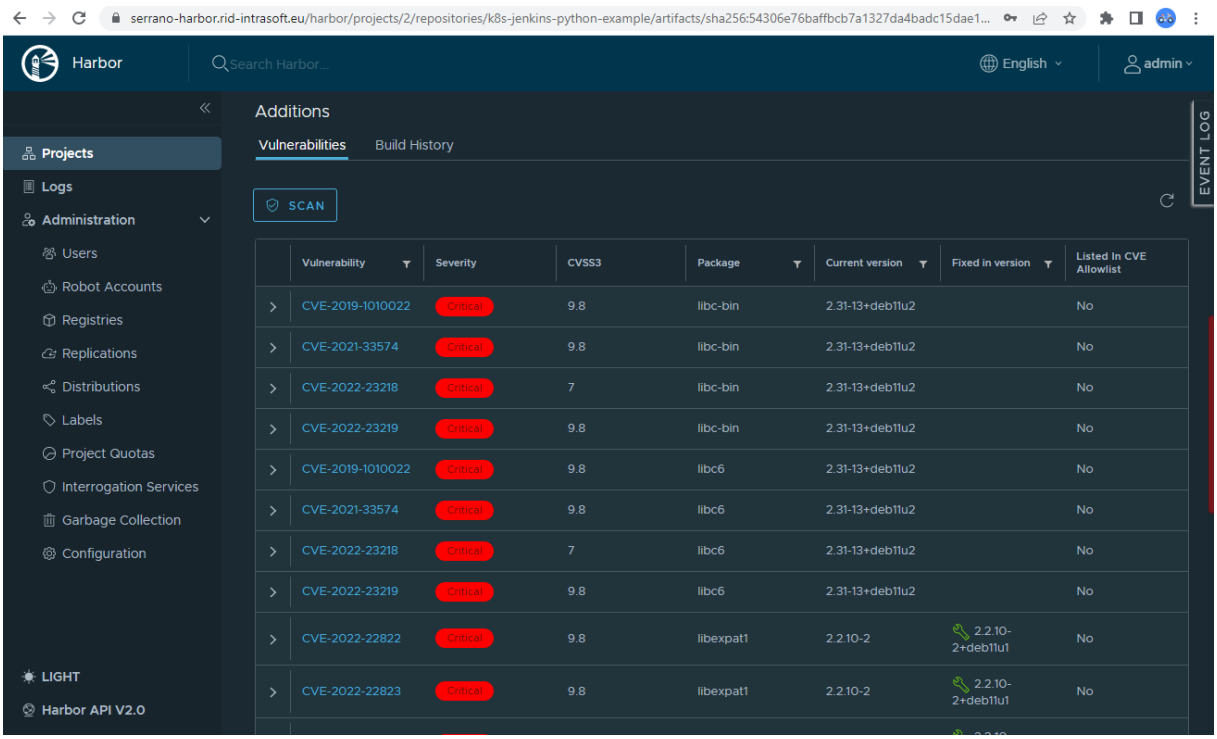


Figure 61: Trivy scan result details on Harbor

5.2.7 Dependency-Track

Dependency-Track [53] is an intelligent Component Analysis platform that allows organizations to identify and reduce risk in the software supply chain. Dependency-Track takes a unique and highly beneficial approach by leveraging the capabilities of Software Bill of Materials (SBOM). This approach provides capabilities that traditional Software Composition Analysis (SCA) solutions cannot achieve.

Dependency-Track monitors component usage across all versions of every application in its portfolio in order to proactively identify risk across an organization. The platform has an API-first design and is ideal for use in CI/CD environments.

The URL to the Dependency-Track service in SERRANO is:

https://serrano-dependency-track.rid-intrasoft.eu/

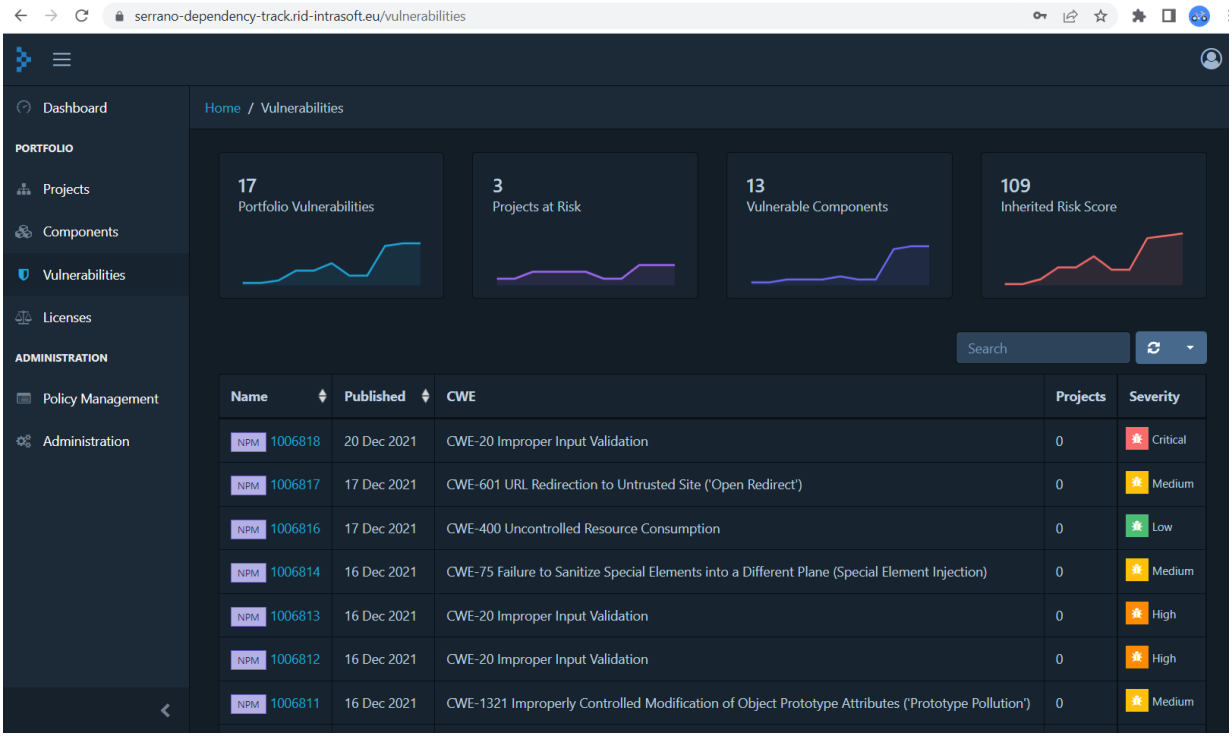


Figure 62: Dependency-Track in SERRANO CI/CD

5.2.8 DefectDojo

DefectDojo [54] is a security orchestration and vulnerability management platform. DefectDojo allows the management of application security programs, the maintenance of product and application information, the triage of vulnerabilities and pushing of findings to systems like JIRA and Slack. DefectDojo enriches and refines vulnerability data using a number of heuristic algorithms that improve with the more you use the platform.

The URL to the DefectDojo service in SERRANO is:

https://serrano-defectdojo.rid-intrasoft.eu/

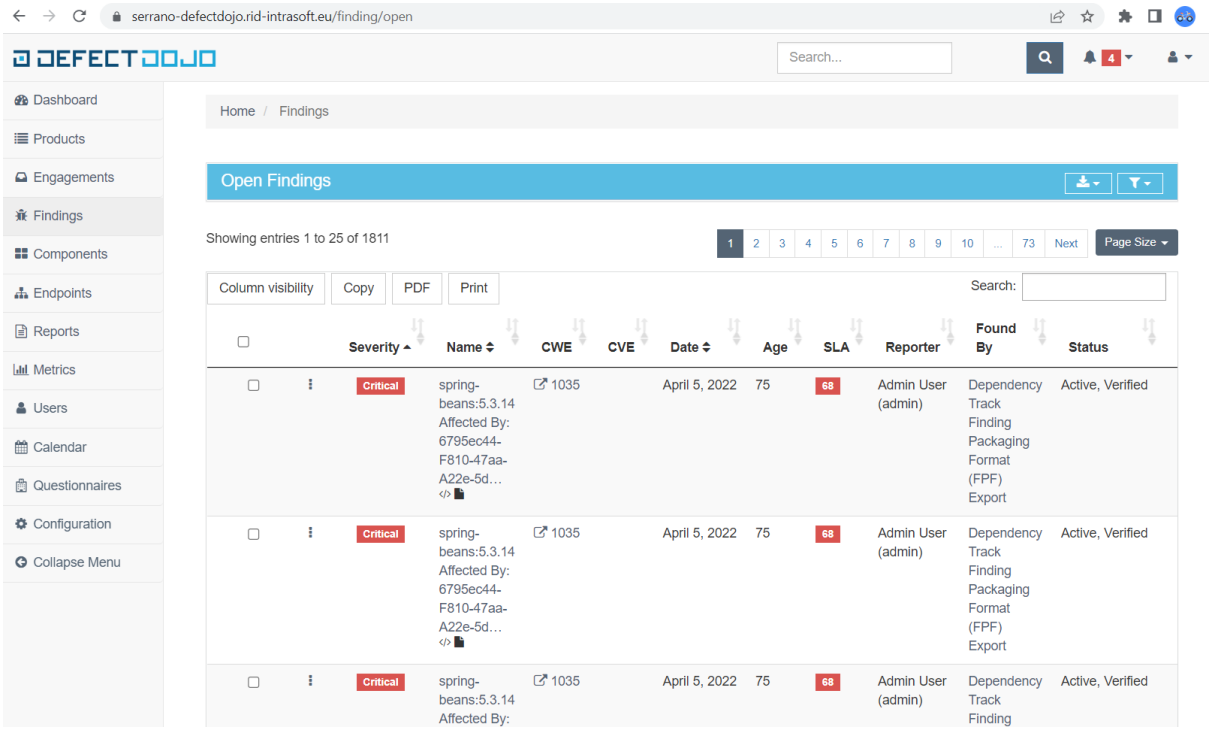


Figure 63: DefectDojo in SERRANO CI/CD

5.2.9 Kubernetes

Kubernetes [55], also known as K8s, is an open-source system for managing containerized applications across multiple hosts. It provides basic mechanisms for deployment, maintenance, and scaling of applications.

Kubernetes builds upon a decade and a half of experience at Google running production workloads at scale using a system called Borg, combined with best-of-breed ideas and practices from the community.

Kubernetes is hosted by the Cloud Native Computing Foundation [56] (CNCF).

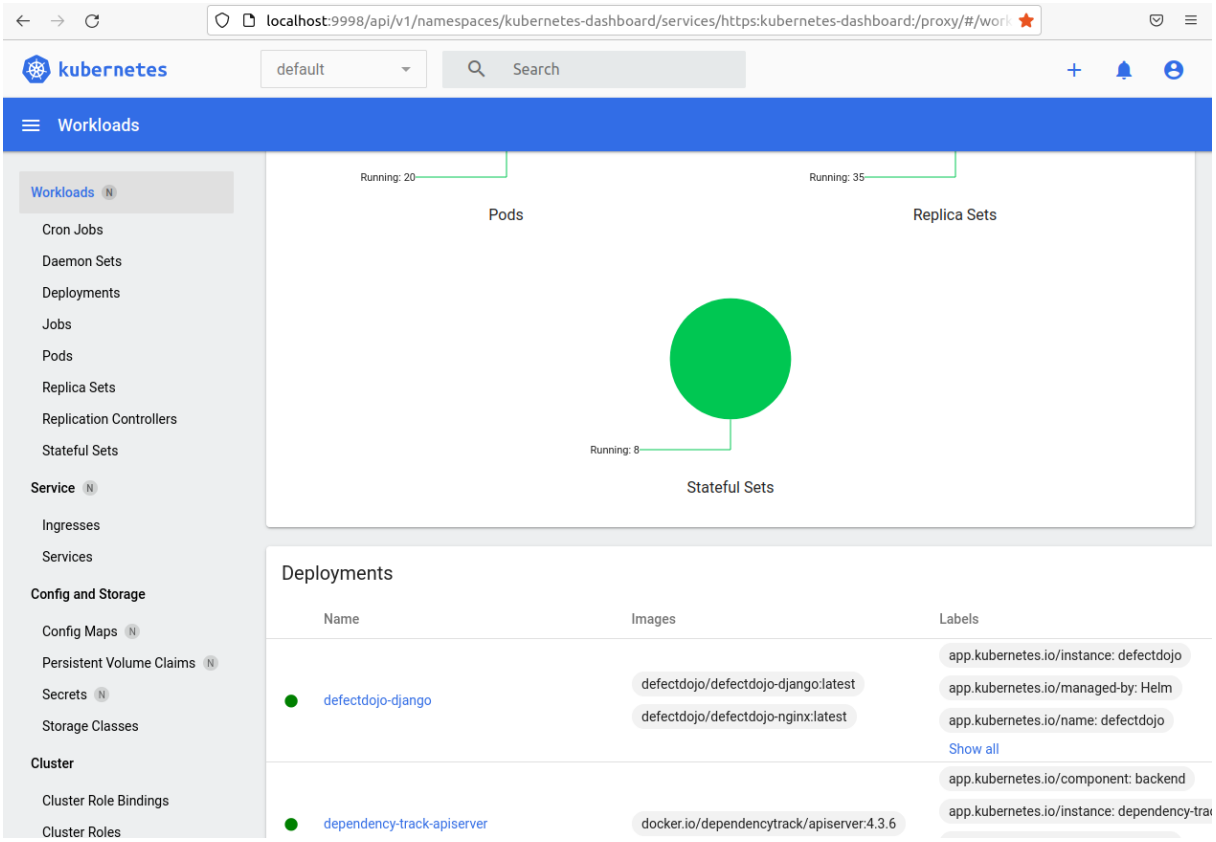


Figure 64: Kubernetes Dashboard in SERRANO CI/CD

5.2.9.1 Helm

Helm [57] is a tool for managing Charts. Charts are packages of pre-configured Kubernetes resources.

Helm can be used to:

- Find and use popular software packaged as Helm Charts to run in Kubernetes.
- Share your own applications as Helm Charts.
- Create reproducible builds of your Kubernetes applications.
- Intelligently manage your Kubernetes manifest files.
- Manage releases of Helm packages.

Helm is a graduated project in the CNCF and is maintained by the Helm community.

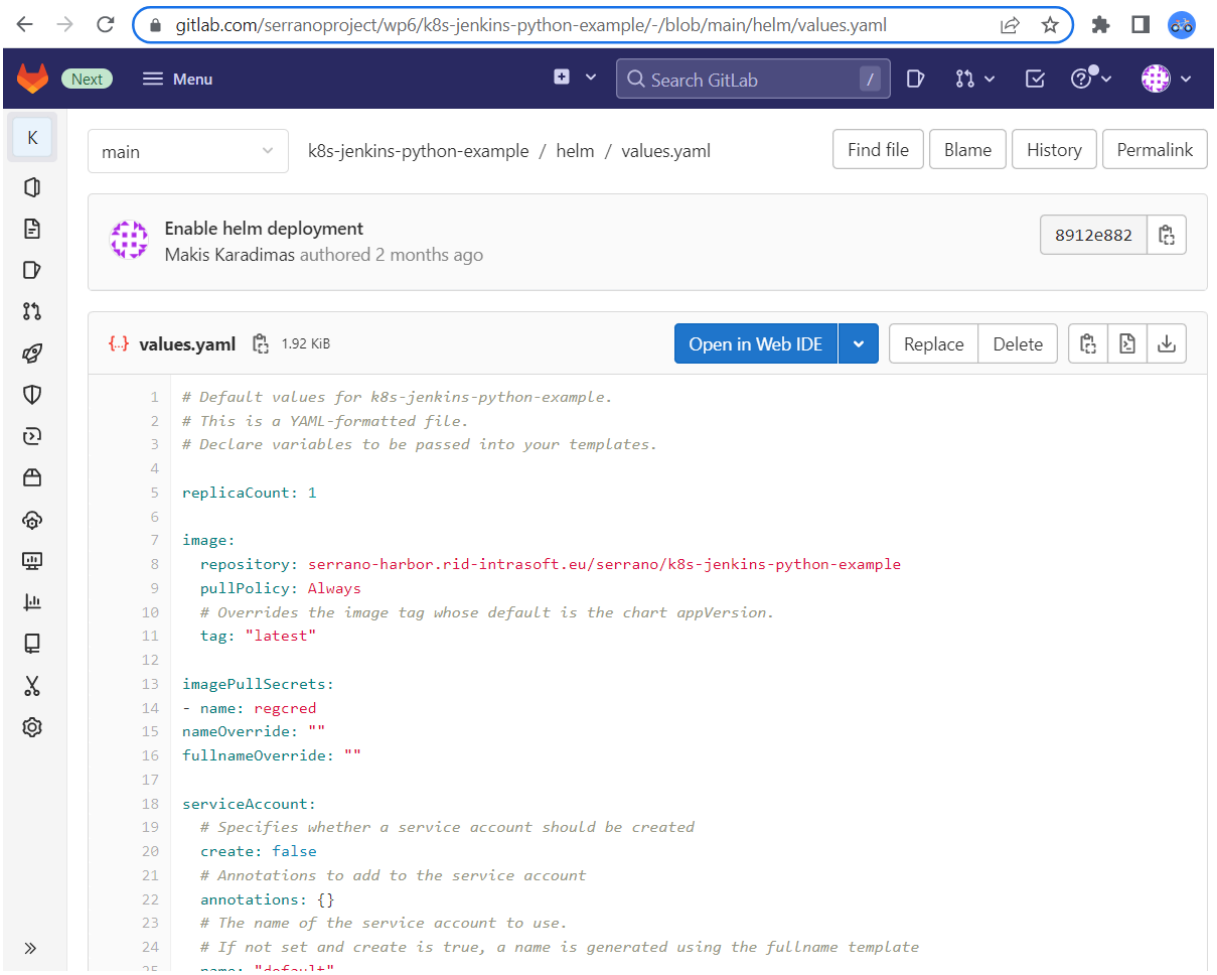


Figure 65: Helm charts in SERRANO on GitLab

6 Software Deployment Specifications and Validation

6.1 Continuous Integration/Continuous Deployment Processes

Figure 66 represents the procedure followed for developing and releasing the software components of the SERRANO platform. The typical CI/CD steps for a consortium partner (represented by one or more developers/software engineers) include committing the owning source code to GitLab, which will trigger the Jenkins server to build the Docker image and push it to the Docker registry in Harbor. Finally, utilizing the Docker images, the generated containers will be deployed to the K8s clusters of the integration and operational environments.

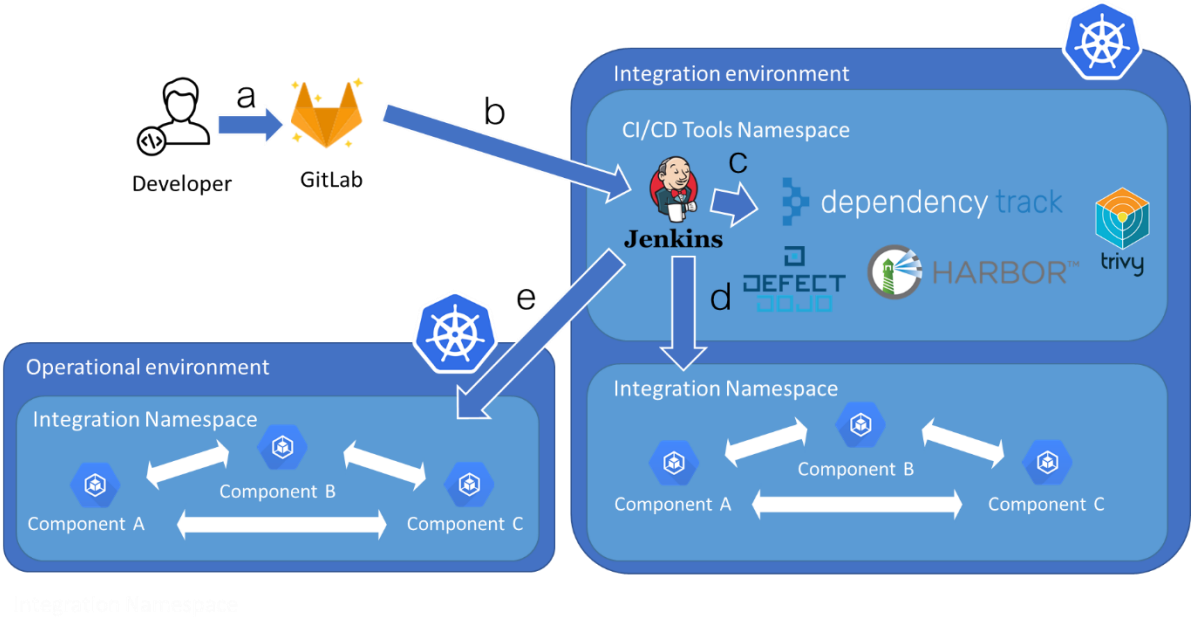


Figure 66: Procedure for developing and releasing the software components

A more extensive description of the steps that a developer of a component of the SERRANO platform should follow to successfully develop, document and deploy it to the operational environment has been shared. This description is the following:

- Step 1: Develop component
- Step 2: Create OpenAPI YAML or JSON and validate on https://editor.swagger.io/
- Step 3: Commit and push code and Swagger to Gitlab https://gitlab.com/serranoproject

- Step 4: Check Unit Test execution and below reports on Jenkins https://serrano-jenkins.rid-intrasoft.eu/
- Step 5: Check SonarQube report for security vulnerabilities on https://serrano-sonarqube.rid-intrasoft.eu/
- Step 6: Check Dependency-Track report for dependency vulnerabilities https://serrano-dependency-track.rid-intrasoft.eu/
- Step 7: Build new docker image
- Step 8: Check image scanning report coming from Trivy on Harbor https://serrano-harbor.rid-intrasoft.eu/ or DefectDojo
- Step 9: Deploy component to integration environment
    - Sample helm charts are available in https://gitlab.com/serranoproject/wp6/k8s-jenkins-python-example/-/tree/main/helm and https://gitlab.com/serranoproject/wp6/k8s-jenkins-example/-/tree/master/helm
    - Helm charts can be converted to equivalent Kubernetes YAML/JSON files that can be applied directly by kubectl using the commands:
    - `$ helm repo add hashicorp https://helm.releases.hashicorp.com`
    - `$ helm template vault hashicorp/vault --output-dir vault-manifests/helm-manifests`
    - Special caution should be taken in properly defining the liveness and readiness probes in deployment.yaml (e.g., https://gitlab.com/serranoproject/wp6/k8s-jenkins-example/-/blob/master/helm/templates/deployment.yaml)
    - Endpoints will be internally exposed at the location http://${COMPONENT_NAME}.integration:${PORT}
- Step 10: Check integration tests execution in integration environment.
    - These can be included in the Jenkinsfile as curl commands (e.g. https://gitlab.com/serranoproject/wp6/k8s-jenkins-example/-/blob/master/Jenkinsfile#L103) or in a separate script/mechanism with corresponding descriptions above each test.
- Step 11: Deploy in the Kubernetes cluster of the operational environment.
    - Similar Helm charts can be used to deploy in another Kubernetes cluster. The two examples on GitLab include a deployment step to the UVT K8s cluster that exposes the component to an internally resolvable domain name.

- Cert-manager for let's encrypt certificates is available and components can be externally exposed to *.serrano.cs.uvt.ro (other domains can be added) using the relevant annotations.

Also, similar instructions have been shared with partners that do not intend to push the source of their solutions to the SERRANO GitLab repository. These can be found below:

- Step 1: Develop component
- Step 2: Build new docker image
- Step 3: Push image to Docker registry
- Step 4: Create OpenAPI YAML or JSON and validate on https://editor.swagger.io/
- Step 5: Commit and push Jenkinsfile and Swagger (and other files) to Gitlab https://gitlab.com/serranoproject
- Step 6: Check image scanning report coming from Trivy on Harbor https://serrano-harbor.rid-intrasoft.eu/ or DefectDojo
- Step 7: Deploy component to integration environment
  - Sample helm charts are available in https://gitlab.com/serranoproject/wp6/k8s-jenkins-python-example/-/tree/main/helm and https://gitlab.com/serranoproject/wp6/k8s-jenkins-example/-/tree/master/helm
  - Helm charts can be converted to equivalent Kubernetes YAML/JSON files that can be applied directly by kubectl using the commands:
  - `$ helm repo add hashicorp https://helm.releases.hashicorp.com`
  - `$ helm template vault hashicorp/vault --output-dir vault-manifests/helm-manifests`
  - Special caution should be taken in properly defining the liveness and readiness probes in deployment.yaml (e.g. https://gitlab.com/serranoproject/wp6/k8s-jenkins-example/-/blob/master/helm/templates/deployment.yaml)
  - Endpoints will be internally exposed at the location http://${COMPONENT_NAME}.integration:${PORT}
- Step 8: Check integration tests execution in integration environment.
  - These can be included in the Jenkinsfile as curl commands (e.g. https://gitlab.com/serranoproject/wp6/k8s-jenkins-example/-/blob/master/Jenkinsfile#L103) or in a separate script/mechanism with corresponding descriptions above each test.

- Step 9: Deploy in UVT (or other) Kubernetes cluster.
    - The same Helm charts can be used to deploy in another Kubernetes cluster. The two examples on GitLab include a deployment step to the UVT K8s cluster that exposes the component to an internally resolvable domain name.
    - Cert-manager for let's encrypt certificates is available and components can be externally exposed to *.serrano.cs.uvt.ro (other domains can be added) using the relevant annotations.

6.2 Integration plan

The integration plan is built around the concepts described in the following sections. In the first stages of the Software Development Life Cycle (SDLC), the common specification approach is based around the OpenAPI specification that is defined as part of the design of applications. In the implementation phase, two important aspects are identified, code version control and containerization. In the next stages of testing, integration and maintenance of the resulting software, a testing approach using unit, integration, and security tests is followed in addition to an approach that verifies code quality and security through automated tools.

6.2.1 Common API specification approach

Established and widely accepted good practices in the field of API specification are used. For example, for REST APIs proper path names, error codes and response types should be used. In addition, the decision in the consortium is to prefer JSON preferred over XML and other formats.

Open-source formats for describing the APIs should be used. For example, OpenAPI 3.0 (3.0.3 is the current latest) is an open-source format for describing and documenting APIs that partners are strongly encouraged to use for the description of REST APIs and the documentation of endpoints and used data schemas. There are many online tools that facilitate the process of creating, extending and updating the Open API specifications, such as Swagger Editor (https://editor.swagger.io/). Also, it can convert various OpenAPI specification versions to the latest one.

Other tools can be used to facilitate the visualization and interactions with the API's resources in a user-friendly way. Swagger UI [58] is one such tool, which can automatically generate a user interface (UI) from the OpenAPI specification.

Developers in the Project can easily communicate through exchanging usage examples of the API in the form of Postman [59] or Insomnia [60] collections can be used to provide full request examples in addition to Swagger, e.g. Figure 22 and Figure 23.

6.2.2 Development using containers

Components developed or used in SERRANO that require a runtime environment for their execution should provide a Docker image for their deployment. Libraries and other components that can be dynamically imported usually would not require the creation of a container, but they can be stored in suitable repositories, such as PyPI [61].

The Docker images can be used in the definition of Kubernetes Pods using Kubernetes YAMLs or Helm. In each pod, Docker images can be used to run one or more containers.

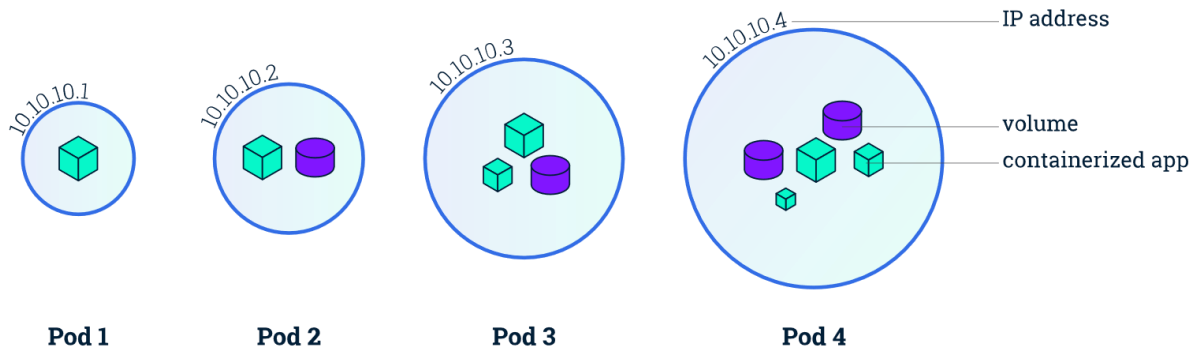


Figure 67: Containers in Kubernetes pods

6.2.3 Code and Deployment configuration on GitLab

Gitlab is the code repository of choice for every component. The code can be placed under the corresponding work package and partner folder, such as the one in Figure 68.

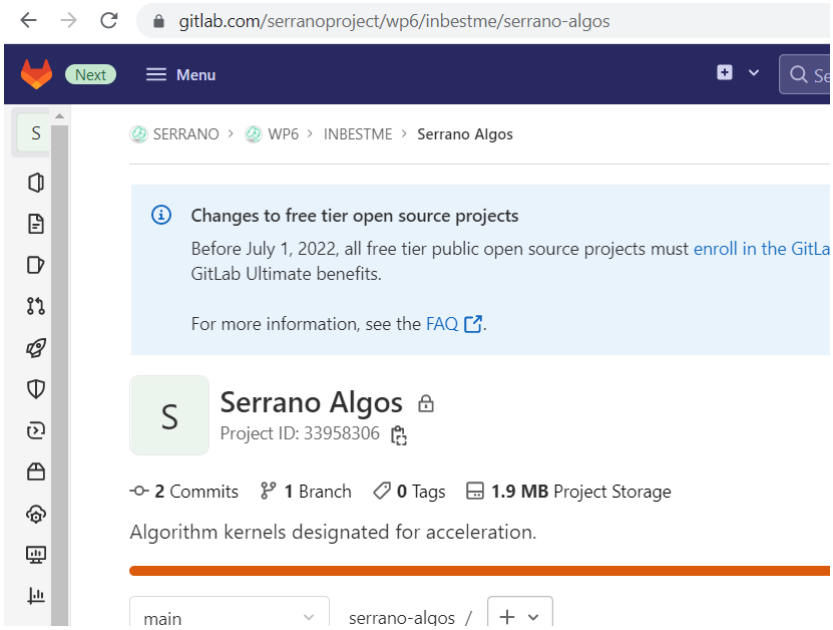


Figure 68: GitLab folder structure

6.2.4 Unit, integration and security tests

Adequate unit, integration and security tests should be available for platform component versions that are considered stable. However, earlier version should also contain a number of unit and integration tests that prevent developers for breaking functionality that worked in the last version, before new changes where introduced.

Automated unit, integration and functional security tests are executed as part of the Jenkins CI/CD pipeline. An example of the build stages that run the unit and integration tests on Jenkins (security tests can be implemented similar to unit or integration tests) is the following:

```
stage('Unit tests') {
  steps {
    container('python') {
      sh 'python -m unittest composeexample.utils'
    }
  }
}
stage('Integration Tests') {
  steps {
    container('java') {
      script {
        echo 'Run your Integration Tests here'
        try {
          String testName = "1. Check that app is running - 200 response code"
          String url =
          "http://${COMPONENT_NAME}.integration:8000/api/v1/puppies"
          String responseCode = sh(label: testName, script: "curl -m 10 -sL -w
'%{http_code}' $url -o /dev/null", returnStdout: true)

          if (responseCode != '200') {
            error("$testName: Returned status code = $responseCode when calling
$url")
          }

          testName = '2. Create record - 201 response code'
          url =
         "http://${COMPONENT_NAME}.integration:8000/api/v1/puppies/"
          responseCode = sh(label: testName, script: """curl -m 10 -s -w
'%{http_code}' --request POST $url --header 'Content-Type: application/json' --
data-raw '{"name":"Jack","age":3,"breed":"shepherd","color":"brown"}' -o
/dev/null""", returnStdout: true)

          if (responseCode != '201') {
            error("$testName: Returned status code = $responseCode when calling
$url")
          }

          testName = '3. Validate stored records'
          url =
          "http://${COMPONENT_NAME}.integration:8000/api/v1/puppies"
          String responseBody = sh(label: testName, script: """curl -m 10 -sL
$url""", returnStdout: true)

          if (responseBody !=
'[{"name":"Jack","age":3,"breed":"shepherd","color":"brown"}]') {
```

```
            error("$testName: Unexpected response body = $responseBody when
calling $url")
          }
        } catch (ignored) {
            currentBuild.result = 'FAILURE'
            echo "Integration Tests failed"
        }
      }
    }
  }
}
```

6.2.5 Code Quality and Security

As described in sections 5.1.1 and 5.2.4, SonarLint and SonarQube are part of the development and integration plan. These tools can ensure code quality and security in the platform components as part of the DevSecOps approach which all partners should adopt. Special focus is put on the capability of SonarQube to analyze source code, ensure code quality and find security vulnerabilities that make SERRANO platform components susceptible to attacks. Therefore, all security vulnerabilities identified by SonarQube should be fixed as early as possible. Other findings, such as code smells and other flagged vulnerabilities, should be checked to ensure no major impact is expected.

6.3 Verification and Validation plan

All SERRANO components that are part of the platform or the use cases should be tested.

In the below table, we have summarized two types of tests for each component.

Component Unit or Functional Tests: Each component has several tests that assess its functionality, which can be unit tests that examine a specific unit of code as part of the development of the component or functional tests that validate the correct behaviour of the component without a specific reference to the source code.

Component Integration Tests: Platform components have points of integration, which might allow integration with other internal or external components, external services, user interfaces, etc. These should all be tested at a level that ensures their correct integration in a way that end-to-end flows can be executed properly.

Table 13: Verification and Validation Results on Platform Components

Platform Components	Component Unit or Functional Tests		Component Integration Tests	
	Tested Functionalities	Number of tests	Tested Interfaces	Number of tests
Resource Orchestrator	• Enable and disable watch functionality over Datastore • Components receive notifications • Setup execution requests for ROT • Handle ROT response • Preparation of deployments • Orchestration Manager and Orchestration Drivers interaction for new deployments • Testing interaction with K8s for the actual deployment	17	• Orchestration Controller REST • Orchestration Manager REST • Orchestration Drivers REST • ROT REST • Telemetry REST	12
Resource Optimization Toolkit (ROT)	• Execution Engine registration to Controller • Valid and invalid execution requests to Controller • Valid and invalid termination requests to Controller • Interaction with telemetry framework • Assignment of requests to Execution Engine • Load and execute selected algorithm • Forward results to Controller Detect performance issues on engines	15	• ROT REST • AMQP publisher • AMQP publisher • Telemetry REST	11
Message Broker	MQTT service: • topic creation • topic subscription	19	• MQTT connector • MQTT publisher • MQTT consumer	19

	• publish messages • receive messages AMQP service: • setup basic publisher, consumer, exchange messages • setup topic publisher, consumer, exchange messages • setup fanout publisher, consumer, exchange messages setup route publisher, consumer, exchange messages		• AMQP connector • AMQP publisher • AMQP consumer	
Telemetry Framework	• Collection of resource characteristics and monitoring data • Monitoring probes registration to agents • Update probes' configuration • Detect events related to operational state of telemetry components • Store collected information to operational databases • Forward telemetry data to Central Handler • Provision of monitoring data to orchestration mechanisms	18	• Monitoring probes REST • Enhanced Telemetry Agent REST • Central Telemetry Handler REST • Stream Handler	14
AI/enhanced service orchestrator	• Can identify application and deployment description • Ensures that Application Model	12	• Resource Orchestrator REST API calls • Central Telemetry Handler REST API calls	8

	Constraints are properly formed • Application Constraints are properly translated to Resource Constraints based on mapping rules specified • Possible Application Deployment Scenarios are being developed • Telemetry Data is being dynamically retrieved • Application is being properly deployed through Resource Orchestrator			
HPC system hardware interface	• List of HPC services • Infrastructure model creation • Infrastructure telemetry • Job submission • Job status retrieval	31	• SSH communication between the Gateway and HPC infrastructure • REST API calls	6
HW Acceleration abstractions	• Static API function call • Dynamic API application port • Generic/debug plugin functionality • Custom FPGA/GPU plugins	6	• VM application execution with hardware accelerator	2
Trusted Virtualizations	• Boot process • K8s integration	2	• Signed kernel booting & node joining k8s cluster	1
Secure storage on-premises gateway	• File caching functionality	5	These features are implemented through integration with the Skyflok.com backend and the SERRANO edge devices:	30

			• Bucket creation, deletion, list contents, retrieve storage policy applied to a bucket • Object creation, deletion, retrieval • Storage policy creation, listing • Retrieve telemetry information	
TLS offloading	• TLS handshake procedure validation • HTTP packet information inclusion • TLS status signalling	10	The features are integrated into libraries that are placed under the DOCA DPU SDK for developer utilization.	10
Service assurance	• SA configuration parameters validation • SA subcomponent distributed cluster configuration validation • SA subcomponent inference functional tests (validation, detection, model selection etc.) • SA subcomponent data bus connector validation (stream handler connection related tests)	12	• SA Connector API • SA Inference API • SA Data bus Connector API	16

Table 14 Verification and Validation Results on Use Case Components

Use Case Components	Component Unit or Functional Test		Component Integration Tests	
	Tested Functionalities	Number of tests	Tested Interfaces	Number of Tests
Secure Storage Use Case Integrated Functionality	• Component unit testing for the Secure Storage Use Case will be performed as part of the integration testing for the Secure storage on-premises gateway.	5	• DPU acceleration • GPU/FPGA acceleration	5
Fintech Analysis Use Case Integrated Functionality	• Activity completion for data retrieval from storage • Activity completion for investment strategy application • Activity completion for the creation of investment profiles • Activity completion for portfolio creation	5	• GPU/FPGA acceleration • Secure Storage	2
Anomaly Detection in Manufacturing Settings Integrated Functionality	• Validate the correct sending of data streaming from the ball screw sensors to the Message Broker through MQTT protocol. • Validate the services and microservices developed for the detection of anomalies in ball screw. • Control and monitor the inference response time in the classification of the new incoming data, as well as the retraining time of the new model to be generated.	7	• Data Broke connection (Message Broker through. MQTT protocol) • GPU/FPGA acceleration • Secure Storage API interface interaction	7

7 Conclusions

Deliverable 6.3 reports on the work performed in WP6, for providing the development of the SERRANO integrated platform. The WP6 activities related to D6.3 aim at unifying the outcomes of the developed components and services in WP3-5 to release the integrated SERRANO platform.

The deliverable presents an overview of the SERRANO platform, including the initial release status, the SERRANO platform components and functionalities, the development and integration environment, the software deployment specifications and the verification and validation results on the platform components.

This document provides the description and documentation of the initial platform release that aims to provide a basic functional prototype that will support the core functionalities of the three project use cases. Moreover, the initial platform prototype will support the preliminary evaluation of the use cases, reported at deliverable D6.4 “Business, end user and technical evaluation” (M20), which will provide critical feedback for the second development iteration.

The SERRANO full platform prototype (M31) will be based on the initial release and will include the remaining functionality, which was not included in the early prototype. The intention is to achieve a fully functional platform resulting from the integration of all project components and provide a prototype suitable for the pilots’ experimentation.

For the final release of the SERRANO platform, delivered at the end of the project, the consortium will focus on implementing the feedback from the final evaluation of the SERRANO platform through the demonstration of the three project use cases. This version will be fully integrated and documented as part of deliverables D6.7 “Final version of SERRANO integrated platform” (M36) and D6.8 “Final version of business, end user and technical evaluation” (M36).

8 References

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

[2] MinIO: https://min.io

[3] Open source message broker: https://www.rabbitmq.com

[4] MQTT Plugin: https://www.rabbitmq.com/mqtt.html

[5] Apache Kafka: https://kafka.apache.org/

[6] Apache Zookeeper: https://zookeeper.apache.org/

[7] HDFS: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[8] Apache HBase: https://hbase.apache.org/

[9] MongoDB: https://www.mongodb.com/

[10] MySQL: https://www.mysql.com/

[11] Apache Spark: https://spark.apache.org/

[12] Apache Hadoop: https://hadoop.apache.org/

[13] Kafka Streams: https://kafka.apache.org/documentation/streams/

[14] Spark Streaming: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[15] TensorFlow: https://www.tensorflow.org/

[16] Deeplearning4j: https://deeplearning4j.konduit.ai/

[17] H2O.ai: https://h2o.ai/

[18] Deliverable D5.1 “Abstraction models and intelligent service orchestration”– SERRANO consortium

[19] Deliverable D2.5 “Final version of SERRANO architecture”– SERRANO consortium

[20] Postman API Platform: https://www.postman.com/

[21] Slurm: https://slurm.schedmd.com/

[22] TORQUE: https://adaptivecomputing.com/cherry-services/torque-resource-manager/

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

[24] Deliverable D4.2 “Performance Maximization under Maximum Affordable Error for the HW and SW IPs” – SERRANO consortium

[25] Deliverable D4.3 “Framework for seamlessly integration of heterogeneous workload-aware performance improvement” – SERRANO consortium

[26] Deliverable D3.3 “Trust and isolated execution on untrusted physical tenders” – SERRANO consortium

[27] Amazon S3: https://aws.amazon.com/s3/

[28] Ceph: https://ceph.io/en/

[29] Openstack Swift: https://docs.openstack.org/swift/latest/

[30] Deliverable D2.4 “Final version of SERRANO use cases, platform requirements and KPIs analysis” – SERRANO consortium

[31] Deliverable D3.2 “Secure Cloud Storage System” – SERRANO consortium

[32] FastAPI framework: https://fastapi.tiangolo.com/

[33] Kubernetes: https://kubernetes.io/

[34] Kubernetes Persistent Volumes: https://kubernetes.io/docs/concepts/storage/persistent-volumes/

[35] S3 API Reference: https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html

[36] Scikit-Learn: https://scikit-learn.org/stable/

[37] EDE GitLab Repository: https://gitlab.dev.info.uvt.ro/serrano/EDE-Serrano.git

[38] Pandas: https://pandas.pydata.org/

[39] Dask: https://dask.org/

[40] SonarQube: https://www.sonarqube.org/

[41] SonarLint: https://www.sonarlint.org/

[42] CycloneDX: https://cyclonedx.org/

[43] GitLab: https://about.gitlab.com/

[44] Jenkins: https://www.jenkins.io/

[45] Hetzner: https://www.hetzner.com/

[46] Docker: https://www.docker.com/

[47] Harbor: https://goharbor.io

[48] NGINX: https://www.nginx.com/

[49] Trivy: https://aquasecurity.github.io/trivy

[50] Alpine Linux: https://www.alpinelinux.org/

[51] Red Hat Enterprise Linux: https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

[52] CentOs: https://www.centos.org/

[53] Dependency-Track: https://docs.dependencytrack.org/

[54] DefectDojo: https://www.defectdojo.org/

[55] Kubernetes: https://kubernetes.io/

[56] CNCF: https://www.cncf.io/about

[57] Helm: https://helm.sh/

[58] Swagger UI: https://swagger.io/tools/swagger-ui/

[59] Postman: https://www.postman.com

[60] Insomnia: https://insomnia.rest

[61] PyPI: https://pypi.org