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

*Grant Agreement no. 101017168*

# Deliverable D6.7
# Final version of 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:** | 31/12/2023 |
| **Actual submission date:** | 30/12/2023 |

| | |
|---|---|
| **Dissemination level:** | Public |
| **Revision:** | FINAL |

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

## Revision History

| Date | Editor | Status | Version | Changes |
|---|---|---|---|---|
| 09.06.23 | INTRA | Draft | 0.1 | Table of Contents |
| 20.11.23 | INTRA, USTUTT | Draft | 0.2 | Contributions in Sections 4.5, 4.6, 5 and 6 |
| 27.11.23 | ICCS, INB, IDEKO, UVT | Draft | 0.3 | Update in ToC; Contributions in Sections 4.2, 4.3, 4.5, 4.10, 4.11 |
| 04.12.23 | IDEKO | Draft | 0.3 | Table 2 section 6.3 |
| 14.12.23 | INNOV, UVT, ICCS, IDEKO, NBFC | Draft | 0.4 | Contributions in Sections 4.1, 4.7, 4.12.1, 4.12.2 and 6.3 (table 2) |
| 18.12.23 | INB, MLNX, INTRA, NBFC, CC | Draft | 0.5 | Updates in Sections 2, 4.7.1, 4.8, 4.10, 6.3 and 7 |
| 19.12.23 | AUTH, INTRA | Pre-final | 0.6 | Consolidated version for internal review. |
| 22.12.23 | INTRA | Revised | 0.7 | Integrate review changes |
| 28.12.23 | ICCS | Final | 1.0 | |

## Author List

| Organization | Author |
|---|---|
| INTRA | Makis Karadimas, Paraskevas Bourgos |
| MLNX | J.J. Vegas Olmos, Yoray Zack, Amelia Pakouline-Navarro |
| INB | Maria Oikonomidou, Ferad Zyulkyarov |
| IDEKO | Aitor Fernández, Javier Martin |
| ICCS | Aristotelis Kretsis, Panagiotis Kokkinos, Emmanouel Varvarigos, Dimitris Vergados, V. Kosmatos, T. Iliadis |
| INNOV | Andreas Litke, Efstathios Karanastasis, Efthymios Chondrogiannis, Filia Filippou, Kassie Papasotiriou, Stelios Pantelopoulos |
| UVT | Adrian Spataru, Gabriel Luhasz |
| USTUTT | Kamil Tokmakov, Dennis Hoppe |
| CC | Marton Sipos, Daniel E. Lucani, Marcell Fehér |
| AUTH | Argyris Kokkinis, George Zervakis, Dimosthenis Masouros, Vaggelis Argyropoulos, Dimitrios Mitsas, George Margaritis, Ioannis Sofianidis, Stelios Siskos, Dimitris, Danopoulos, Kostas Siozios |
| NBFC | Anastassios Nanos, Charalampos Mainas |

## Internal Reviewers

Kostas Siozios, Dimitrios Danopoulos, AUTH
Ferad Zyulkyarov, INB

**Abstract:** This deliverable (D6.7) presents the outcomes of Task 6.1 – “Integration, Verification and Testing” covering the total duration of WP6, which aims at unifying the outcomes of the developed components and services in WP3-5 to release the versions of the integrated SERRANO platform. The deliverable presents the overview and the details of the SERRANO platform, including the final 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

| | |
|---|---|
| **A4C** | Alien4Cloud |
| **A4COP** | Alien4Cloud Orchestrator Plugin |
| **ACL** | Access Control Lost |
| **AI** | Artificial Intelligence |
| **AISO** | AI-enhanced Service Orchestrator |
| **AMQP** | Advanced Message Queuing Protocol |
| **API** | Abstract Programming Interface |
| **ARDIA** | A Resource reference model for Data-Intensive Applications |
| **ARM** | Advanced RISC Machines |
| **ASGI** | Asynchronous Server Gateway Interface |
| **AWS** | Amazon Web Services |
| **CI/CD** | Continuous Integration / Continuous Development |
| **CNCF** | Cloud Native Computing Foundation |
| **CoT** | Chain of Trust |
| **CPU** | Central Processing Unit |
| **CRUD** | Create, Read, Update and Delete |
| **CSV** | Comma-Separated Values |
| **CT** | Certificate Transparency |
| **CTH** | Central Telemetry Handler |
| **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 |
| **ETA** | Enhanced Telemetry Agent |
| **ETL** | Extract, Transform, Load |
| **FaaS** | Function as a Service |
| **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 |
| **GRAA** | Greedy Resource Allocation Algorithm |
| **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** | Message Queue Telemetry Transport |
| **NBI** | North Bound Interfaces |
| **NIC** | Network Interface Controller |
| **OCI** | Open Container Inititative |
| **OIDC** | OpenID Connect |
| **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 |
| **RO** | Resource Orchestrator |
| **ROT** | Resource Orchestration Toolkit |
| **RoT** | Root of Trust |
| **RQ** | Redis Queue |
| **SAR** | Service Assurance and Remediation |
| **SAST** | Static Application Security Testing |
| **SCA** | Source Composition Analysis |
| **SDK** | Service Development Kit |
| **SDLC** | Software Development Life Cycle |
| **TEE** | Trusted Execution Environment |
| **TPM** | Trusted Platform Module |
| **UI** | User Interface |
| **UUID** | Universal Unique Identifier |
| **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 to be 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 6.7 reports on the work performed in WP6 for developing, integrating, testing, and releasing the final release of the SERRANO platform. The WP6 activities related to D6.7 aim to unify the outcomes of the developed components and services in WP3-5 to release the final integrated SERRANO platform.

The deliverable presents an overview of the SERRANO platform, including the final release status, the final 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 has supported the final evaluation of the use cases, reported in deliverable D6.8 “Final version of business, end user and technical evaluation” (M36).

# 2 Introduction

## 2.1 Purpose of this document

The present deliverable (D6.7) consolidates the outcomes of Task 6.1 – "Integration, Verification and Testing" covering the whole duration of WP6, which aimed at unifying the outcomes of the developed components and services in WP3-5, to release the final integrated SERRANO platform.

In particular, this deliverable aims to present the integrated platform's final release and report on the outcomes of the supported tested functionalities and interfaces. The final release of the SERRANO platform includes the final updates on the components development that synthesize the SERRANO platform, providing the fully fletched SERRANO functionality. Towards this end, each component implements the totality of the envisioned features along with the final inter-component communication interfaces.

The final release of the SERRANO platform has been used as a basis for the evaluation of the SERRANO developments through the demonstration of the three project use cases, reported on deliverable D6.8 "Final version of business, end user and technical evaluation" (M36).

## 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 final release of the SERRANO integrated platform, unifying the outcomes of the developed components and services. Moreover, this document can also be helpful to the general public in obtaining a better understanding of the framework and scope of the SERRANO project.

# 3 Overview of SERRANO Platform

## 3.1 SERRANO Architecture

The SERRANO architecture 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). These deliverables comprehensively describe the overall architecture, SERRANO components, interfaces, and supported workflows. In this section, we provide a short description of the architecture (Figure 1) to facilitate the presentation of the final version of the SERRANO integrated platform.

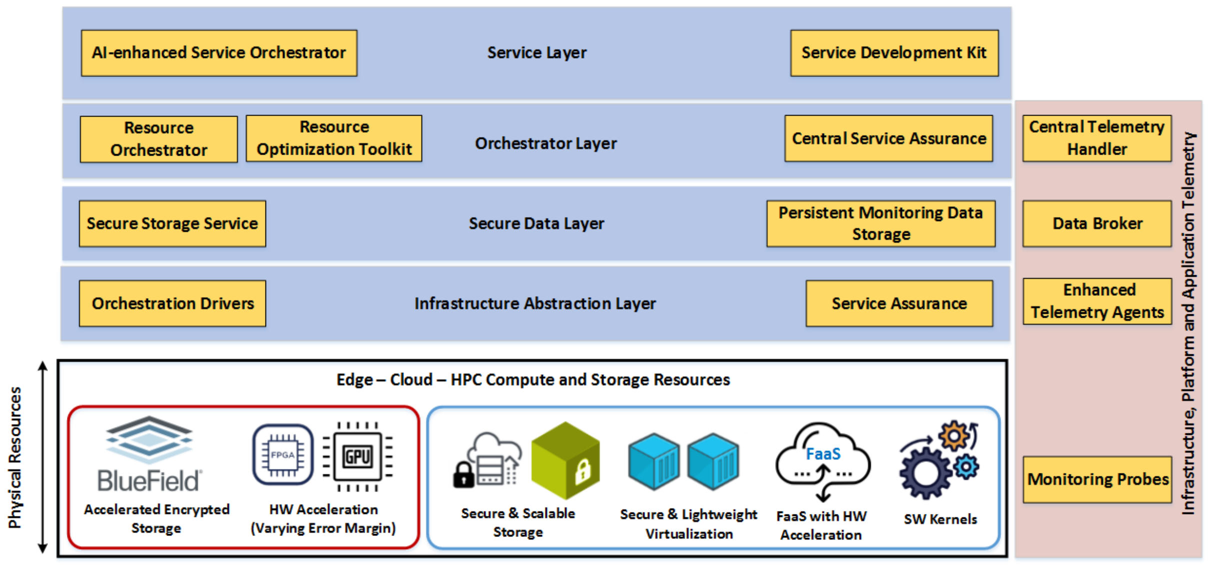

AI-enhanced Service Orchestrator
Service Layer
Service Development Kit
Resource Orchestrator
Resource Optimization Toolkit
Orchestrator Layer
Central Service Assurance
Central Telemetry Handler
Secure Storage Service
Secure Data Layer
Persistent Monitoring Data Storage
Data Broker
Orchestration Drivers
Infrastructure Abstraction Layer
Service Assurance
Enhanced Telemetry Agents
Infrastructure, Platform and Application Telemetry
Physical Resources
Edge – Cloud – HPC Compute and Storage Resources
BlueField
Accelerated Encrypted Storage
FPGA
GPU
HW Acceleration (Varying Error Margin)
Secure & Scalable Storage
Secure & Lightweight Virtualization
FaaS
FaaS with HW Acceleration
SW Kernels
Monitoring Probes

**Figure 1: SERRANO high-level architecture**

The Service Layer contains the *AI-enhanced Service Orchestrator* (Section 4.1) 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 4.2).

The *Resource Optimization Toolkit* (Section 4.3) 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) 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 *Persistent*

*Monitoring Data Storage* (Section 4.4.2) allows the management of the historical monitoring data, which is mainly required by the service assurance and remediation system.

The Infrastructure Abstraction Layer facilitates the integration of hardware and software platforms within the SERRANO platform. The *Orchestration Drivers* (Section 4.2.2) 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.

The Resource Layer includes heterogeneous edge, cloud, and HPC computational and storage resources encompassing the SERRANO-enhanced resources (Sections 4.6, 4.7, 4.8). Across the SERRANO ecosystem resides the Infrastructure, Platform, and Application Telemetry stack (Section 4.4) that collects metrics across the infrastructure and deployed applications. The main components are the Central *Telemetry Handler*, the *Enhanced Telemetry Agents,* and the *Monitoring Probes*. In addition, the *Data Broker* (Section 4.5) provides the required asynchronous inter-component communication within the SERRANO platform and connects external data sources, making them available to internal services.

Section 4 lists the components of these main entities and their subcomponents, highlighting the provided functionalities, along with their integration, as part of the final release of the SERRANO platform.

## 3.2 Final Release Status

SERRANO adopted an iterative approach to implement and evaluate the individual technological developments and overall platform integration. The design and implementation activities were implemented using a spiral model with two iterations (M01-M18, M19-M36). Based on the selected development strategy, there are three main releases for the integrated SERRANO platform: the initial platform prototype, the complete platform prototype, and the final platform prototype.

The initial release of the SERRANO platform was reported on deliverable D6.3 (M18). The initial platform prototype was the outcome of the first development iteration (M1-M18), providing a subset of the envisioned features along with the primary interfaces for inter-component communication. Based on the initial release, the complete platform prototype provided the remaining functionality and was available in M33. This release corresponds to a fully functional platform that integrates all SERRANO components and provides a prototype suitable for the platform and use cases' evaluation.

This deliverable reports the SERRANO platform's final release based on the complete platform prototype. The final platform is fully integrated and includes improvements based on the feedback from the final evaluation of the SERRANO platform through the demonstration of the three project use cases. Deliverable D6.8 "Final version of business, end user and technical evaluation" (M36) includes the performance evaluation analysis and results.

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, ARDIA Framework | Interface is fully implemented, integrated and tested |
| WP5T5RO-I: Resource Orchestrator | Resource Orchestrator, AI-enhanced Service Orchestrator, Service Assurance, ARDIA Framework | Interface is fully implemented, integrated and tested |
| WP5T5OD-I: Orchestration Drivers | Orchestration Driver, Resource Orchestrator, K8s, HPC Gateway | Interface is fully implemented, integrated and tested |
| WP5T2ROT-I: Resource Optimization Toolkit | Resource Optimization Toolkit, Resource Orchestrator | Interface is fully implemented, integrated and tested |
| WP5T4EMT-I: Energy & Resource Aware Mapping Interface | SERRANO HPC Gateway, Resource Orchestrator | Interface is fully implemented, integrated and tested |
| WP4T2HPC-I: Uncertainties Estimation Interface | SERRANO HPC Gateway, Resource Orchestrator | Interface is implemented, integrated and tested |
| WP3T2DSS-I: Secure Storage API | On-premises Storage Gateway, SERRANO applications and services | Interface is fully implemented, integrated and tested |
| WP3T2DSSSLT-I: Storage location telemetry API | On-premises Storage Gateway, Central Telemetry Handler | Interface is fully implemented, integrated and tested |
| WP5T3CTH-I: Central Telemetry Handler Interface | Central Telemetry Handler, AI-enhanced Service Orchestrator, Resource Optimization Toolkit, Service Assurance | Interface is fully implemented, integrated and tested |
| WP5T3ETA-I: Enhanced Telemetry Agent Interface | Enhanced Telemetry Agent, Central Telemetry Handler, Orchestration Drivers, Service, Monitoring Probes | Interface is fully implemented, integrated and tested |
| WP5T5PMDS-I: Persistent Monitoring Data Storage Interface | Enhanced Telemetry Agent**,** AI-enhanced Service Orchestrator, Service Assurance and Remediation | Interface is fully implemented, integrated and tested |
| WP5T5MB-I: Message Broker Interface | SERRANO platform components, use cases and applications, external data sources | Interface is fully implemented, integrated and tested |
| WP5T5SC-I: Streaming Core Interface | SERRANO platform components, use cases and applications, external data sources | Interface is fully implemented, integrated and tested |

| | | |
|---|---|---|
| WP5T5SAR-I: Service Assurance Interface | Service Assurance and Remediation, Resource Orchestrator, use cases and applications | Interface is fully implemented, integrated and tested |
| WP4T3PC-I: Plug&Chip Interface | SERRANO Plug&Chip framework, use cases and applications | Interface is fully implemented, integrated and tested |
| WP4T1HWRT-I: Hardware Accelerators Interface | Local Orchestrators, SERRANO hardware accelerated kernels | Interface is fully implemented, integrated and tested |
| WP4T2HPC-I: HPC Services Interface | HPC Gateway, Resource Orchestrator, Orchestration Drivers | Interface is fully implemented, integrated and tested |
| WP5T5TLV-I: Trusted and Lightweight Virtualization Interface | Orchestration Drivers, Local Orchestrators, use cases and applications | Interface is fully implemented, integrated and tested |

# 4 SERRANO Platform Components and Functionalities

## 4.1 AI-enhanced Service Orchestrator

### 4.1.1 Description

The AI-enhanced Service Orchestrator (AISO) facilitates the deployment and execution of applications (and their internal components/micro-services), taking into account a considerable amount of knowledge that may directly or indirectly come from the end user and the SERRANO platform (e.g., user-specified requirements, user intent, and telemetry data collected for a particular application or micro-service). The application owners initially specify the application requirements and their particular goals/intents based on the elements included in the Application Model, which are consequently used by the AISO. Based on these, the AISO detects potential deployment scenarios/objectives, which it accordingly expresses using the elements of the Resource Model and then communicates to the Resource Orchestrator (RO). These models are part of the ARDIA Framework and were analytically described in the deliverable D5.1 [90]. The Telemetry Data model (also part of the ARDIA Framework) is used to express the data from the Central Telemetry Handler (CTH). The latter was used in the background for the collection and analysis of relevant application data, based on which several mapping rules (which are enforced by the AISO) were specified.

### 4.1.2 Inner components

The architecture of the AISO (analytically described in the deliverable D2.5 [85]) is presented in Figure 2. Its internal components are responsible for expressing the given application requirements and user's goals/intents into the appropriate resource constraints (Translation Mechanism) and preparing the appropriate deployment scenarios/objectives (Deployment Scenarios Preparation).

The functionality provided by the AISO, and in particular the Translation Mechanism, is driven by the specified mapping rules. The latter express the relation among the elements included in the three abstraction models (part of the ARDIA Framework) and were developed in close collaboration with domain experts, taking into account the telemetry data collected by the Central Telemetry Handler (CTH) and using state of the art ML techniques, as described in the deliverable D5.4 [92]. When more than one option is available to satisfy the same application requirement or constraint, a branch is created by the Deployment Scenarios Preparation mechanism that indicates that the same constraints can be satisfied in different ways.

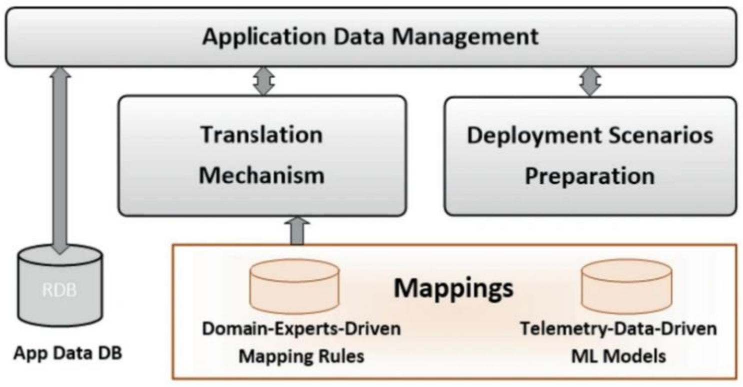

Application Data Management
Translation Mechanism
Deployment Scenarios Preparation
RDB
App Data DB
Mappings
Domain-Experts-Driven Mapping Rules
Telemetry-Data-Driven ML Models

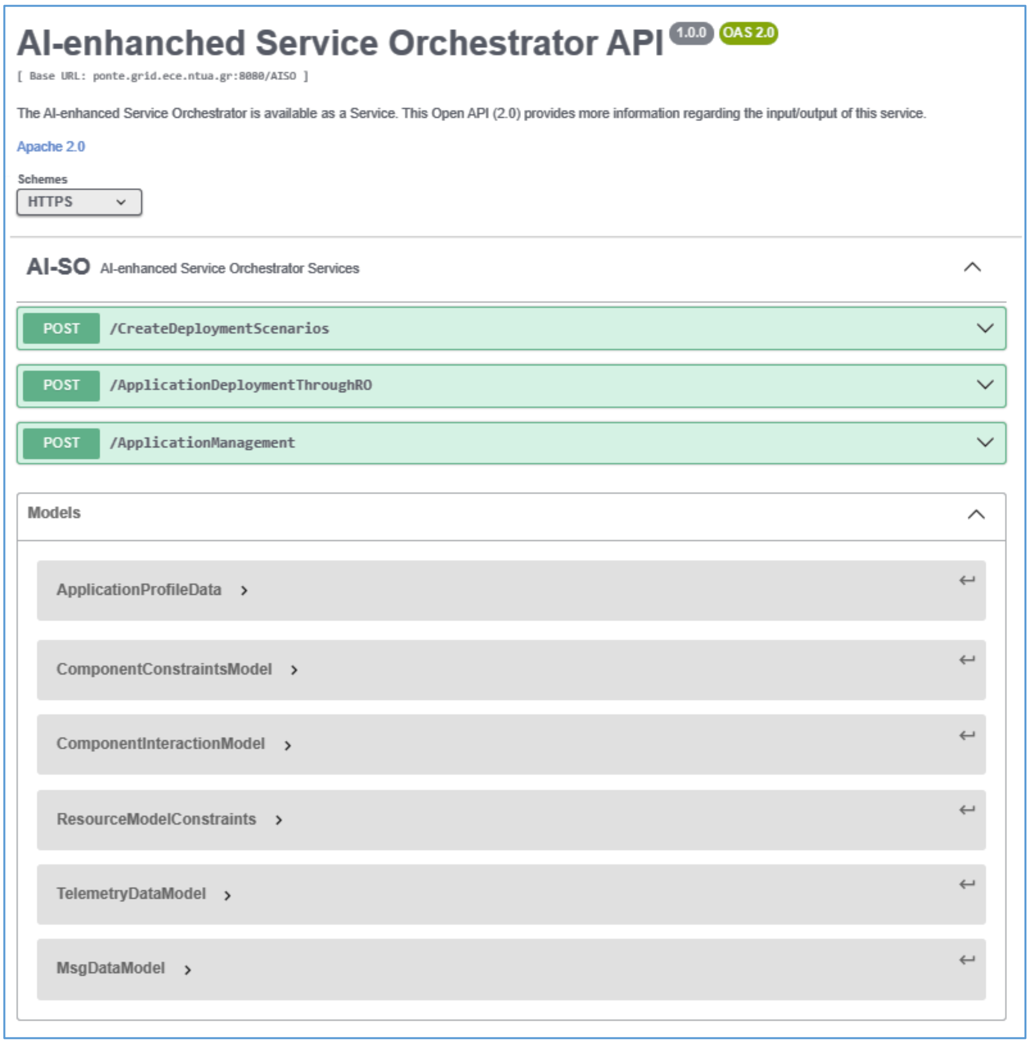
**Figure 2: AI-enhanced Service Orchestrator architecture and main components**

## 4.1.3 Integration details and REST APIs

### *4.1.3.1 AISO services*

The functionality provided by the AISO is available as REST services. Figure 3 presents the services available by the AISO. As can be noticed, there are three different services. The first service (*CreateDeploymentScenarios*) detects potential application or micro-services objectives based on the given application requirements and user goals or intents. The second service (*ApplicationDeploymentThroughRO*) uses the first service to detect objectives as above, and consequently sends the output to the RO (by invoking the relevant RO service) to deploy the application micro-services to the appropriate resources. It eventually service returns the deployment's unique ID. The third service (*ApplicationManagement*) gets the unique ID as input and can be used for service management purposes (e.g., un-deployment). Both the input and output of the AISO's first two services comply with the elements included in the Application and Resource Models, respectively.

The AISO was implemented using Java (v.1.8). The application was compiled and packaged to a WAR file using Apache Maven tool. Accordingly, the Docker image was prepared so that it could be finally deployed to UVT's K8s cluster following the CI/CD process through the specification of the appropriate Jenkins pipeline and Helm charts. Figure 4 presents the output of this process (through the Jenkins GUI).



AI-enhanched Service Orchestrator API 1.0.0 OAS 2.0

[ Base URL: ponte.grid.ece.ntua.gr:8080/AISO ]

The AI-enhanced Service Orchestrator is available as a Service. This Open API (2.0) provides more information regarding the input/output of this service.

Apache 2.0

Schemes: HTTPS

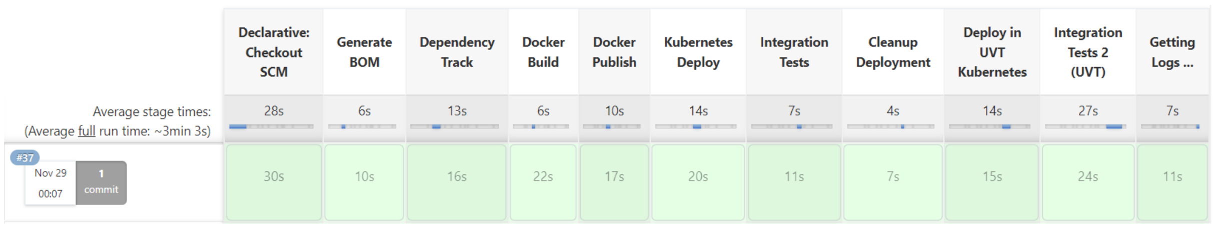
AI-SO AI-enhanced Service Orchestrator Services

- POST /CreateDeploymentScenarios
- POST /ApplicationDeploymentThroughRO
- POST /ApplicationManagement

Models

- ApplicationProfileData
- ComponentConstraintsModel
- ComponentInteractionModel
- ResourceModelConstraints
- TelemetryDataModel
- MsgDataModel

Figure 3: AI-enhanced Service Orchestrator REST Open API



| | Declarative: Checkout SCM | Generate BOM | Dependency Track | Docker Build | Docker Publish | Kubernetes Deploy | Integration Tests | Cleanup Deployment | Deploy in UVT Kubernetes | Integration Tests 2 (UVT) | Getting Logs ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Average stage times: (Average full run time: ~3min 3s) | 28s | 6s | 13s | 6s | 10s | 14s | 7s | 4s | 14s | 27s | 7s |
| #37 Nov 29 00:07 1 commit | 30s | 10s | 16s | 22s | 17s | 20s | 11s | 7s | 15s | 24s | 11s |

Figure 4: AI-enhanced Service Orchestrator deployment to UVT's K8s cluster using CI/CD pipeline

More information about the AISO services is available in Table 2. As already mentioned, there are three services that serve different purposes. The definition of input/output of each one of them is available at the OpenAPI YAML file. Figure 4 presents the tests that took place as part of the deployment process (i.e., Jenkins pipeline).

**Table 2 Integration details of AI-enhanced Service Orchestrator**

| | |
|---|---|
| **IP(s)/Port(s)** | AI-enhanced Service Orchestrator (AISO):<br>• https://ai-enhanced-service-orchestrator.services.cloud.ict-serrano.eu/AISO<br>• https://ai-enhanced-service-orchestrator.services.cloud.ict-serrano.eu/AISO/CreateDeploymentScenarios<br>• https://ai-enhanced-service-orchestrator.services.cloud.ict-serrano.eu/AISO/ApplicationDeploymentThroughRO<br>• https://ai-enhanced-service-orchestrator.services.cloud.ict-serrano.eu/AISO/ApplicationManagement<br>Resource Orchestrator (RO):<br>• https://resource-orchestrator.services.cloud.ict-serrano.eu<br>Central Telemetry Handler (CTH):<br>• https:// central-telemetry.services.cloud.ict-serrano.eu |
| **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** | *https://ai-enhanced-service-orchestrator.services.cloud.ict-serrano.eu/* |
| **API documentation** | *https://ai-enhanced-service-orchestrator.services.cloud.ict-serrano.eu/AISO/openapi/aiso_rest.yaml* |
| **Location of integration tests** | *https://ai-enhanced-service-orchestrator.services.cloud.ict-serrano.eu/AISO/tests/Jenkinsfile* |

#### *4.1.3.2 Interaction with Alien4Cloud (A4C)*

The functionality provided by the AISO has been integrated with the Alien4Cloud (A4C) platform (as presented in the deliverable D5.4), which enables users to graphically formulate the deployment descriptor of their application and specify their goals/intents. For this purpose, an Alien4Cloud Orchestrator Plugin (A4COP) was developed to deploy applications on the SERRANO ecosystem. The plugin generates a JSON file with a predefined format (based on the elements included in the Application Model) that encompasses both the application deployment descriptor and the user's goals/intents. This file, and in particular the given application requirements and goals/intents, can be accordingly processed by the AISO to detect the appropriate deployment options, which are then passed to the RO.

To ensure the successful interaction of the AISO with A4COP (e.g., correct format, missing fields, etc.) several tests were made to verify that the data provided through A4COP are in line with the required specifications and can be further processed by the AISO as well as the rest components of the SERRANO platform (mainly the RO).

The integration tests were performed using the Anomaly Manufacturing use-case, which was defined using the SERRANO TOSCA extension. Figure 5 shows the application modelled in the A4C Topology Editor interface. The application consists of three components, all of which depend on two SERRANO core services. These dependencies are managed by the A4COP, updating the Config Maps of the containers with the corresponding endpoints and credentials of the SERRANO services registered with A4C.

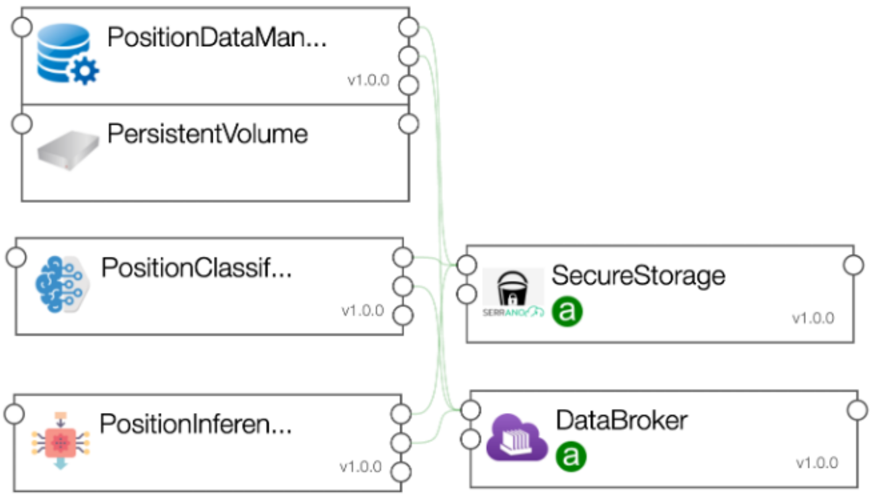


**Figure 5: Anomaly Detection use case modelled using TOSCA and presented in Alien4Cloud**

The TOSCA topology also contains information about the intent. An example is presented below:

```
topology_template:
  node_templates:
    PositionClassifierTrainer:
      type: serrano.nodes.PositionClassifierTrainer
      properties:
        intent:
          Application_Performance:
            Total_Execution_Time: "</= 200 ms"
      ...
```

The intent is translated by the A4COP using the specification of the AISO, and the relation between components is translated to Kubernetes descriptors (YAML). If there are any dependencies between the services, these will be reflected in the intent JSON, under the application workflow field. The output of the aforementioned process (including JSON and YAML) is then sent to the AISO for detecting potential deployment scenarios/objectives (e.g., target infrastructure, hardware accelerators, etc.) that facilitate the deployment process and consequently the RO for deploying the application microservices to the appropriate resources by utilizing the AISO services described in Section 4.1.3.1.

Several tests took place to check the integration between the A4COP, AI-enhanced Service Orchestrator, Resource Orchestrator, and Telemetry Services. All tests have been performed on the Anomaly Manufacturing use case and passed in the production environment at UVT.

The first two tests validate the functionality of the A4COP when connecting with the SERRANO core components. When creating an Orchestrator instance in A4C, the plugin executes an HTTP GET request to the root path of the AISO, Resource Orchestrator and Telemetry services using the configured credentials. A successful response is returned (i.e., 200 OK) and the Orchestrator instance is initiated successfully. If the plugin cannot establish a successful connection when contacting the core SERRANO components, (i.e., the Orchestrator instance is initiated, but the SERRANO Location is not available for deployment) the Orchestrator must be disabled, and the credentials must be verified before re-enabling the instance.

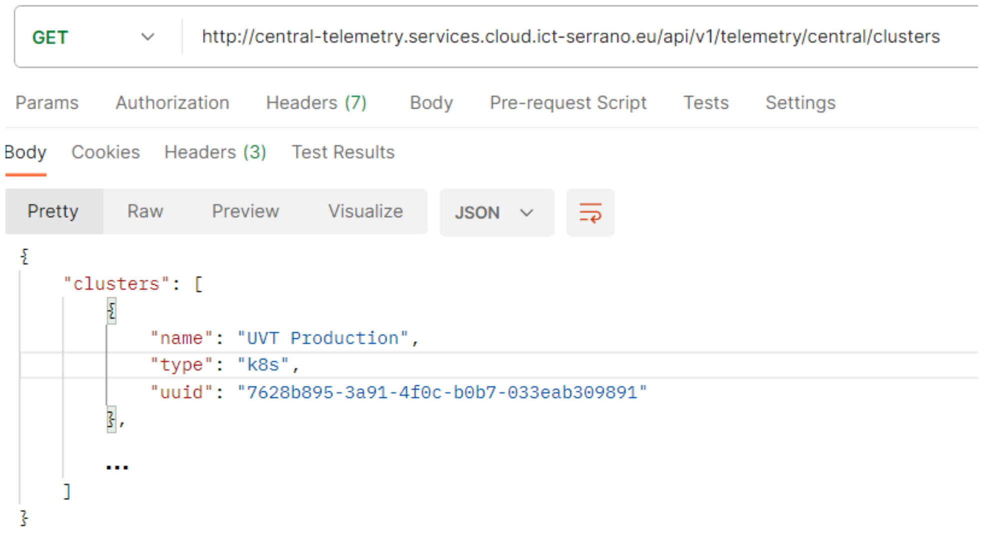
The next four tests are related to deploying an application defined using the SERRANO TOSCA extension after generating the correct documents for the AISO and RO components. The tests validate that the application can be deployed, and the status of the components can be investigated. In the third test, the A4C user selects a SERRANO Location and clicks the deploy button. The A4COP generates the Kubernetes YAML descriptor and the intent request for the AISO. The two generated documents respect the corresponding schemas and can be successfully processed when sent on a cold line as request to the AISO, more specifically, the "/CreateDeploymentScenarios" endpoint. In the fourth test, the two generated documents are sent to the AISO via HTTP REST to the "/ApplicationDeploymentThroughRO" endpoint and a successful status code is received. The body of the response contains the deployment ID. The application is presented as deployed in the A4C interface and status can be investigated at component level. The fifth test is about an unsuccessful response from the AISO. In this case, the A4C user is presented with the error and the application remains in the undeployed state. In the sixth test, the deployment ID obtained during the fourth test can be used to retrieve information from the Resource Orchestrator and the Telemetry services.

The final two tests validate the interaction for stopping an application. In the seventh test, the A4C user hits the un-deploy button. The A4COP contacts the AISO (the "/ApplicationManagement" endpoint) to stop the deployment execution with the ID received during the fourth test. The AISO service responds with a success code. The application is presented as undeployed in the A4C interface. Finally, in the eighth test, regards the case that the AISO times out or responds with an unsuccessful status code, e.g., when the service may not be able to un-deploy the application at that moment because some components may not be available due to network interruptions. In this case, the application remains deployed from the A4C perspective and the un-deploy button must be clicked again at a later point.

#### *4.1.3.3 Interaction with the Resource Orchestrator (RO) and Central Telemetry Handler (CTH)*

The output of the AISO is also a JSON description with a predefined format (based on the elements included in the Resource Model) that includes the given deployment descriptor (YAML file) along with the particular deployment scenarios/objectives detected. This description is provided to the RO, which accordingly uses the given data to allocate the appropriate resources for the deployment and execution of the application. On condition that the application has been successfully deployed, the RO returns to the AISO the deployment unique identifier that can be used for application management purposes (e.g., un-deploy).

In the above process, the Central Telemetry Handler is used in the background for collecting information about the infrastructure and its status. More precisely, the CTH is used to get information about the available platforms, their capabilities and the status of the associated resources. An example is presented in Figure 6. In this example, a GET request is sent to the CTH (*/api/v1/telemetry/central/clusters*) that returns a JSON response with the Unique IDs of available clusters (UUID), which can be accordingly provided to the CTH service (*/api/v1/telemetry/central/clusters/{UUID}*) for retrieving detailed information.


GET http://central-telemetry.services.cloud.ict-serrano.eu/api/v1/telemetry/central/clusters

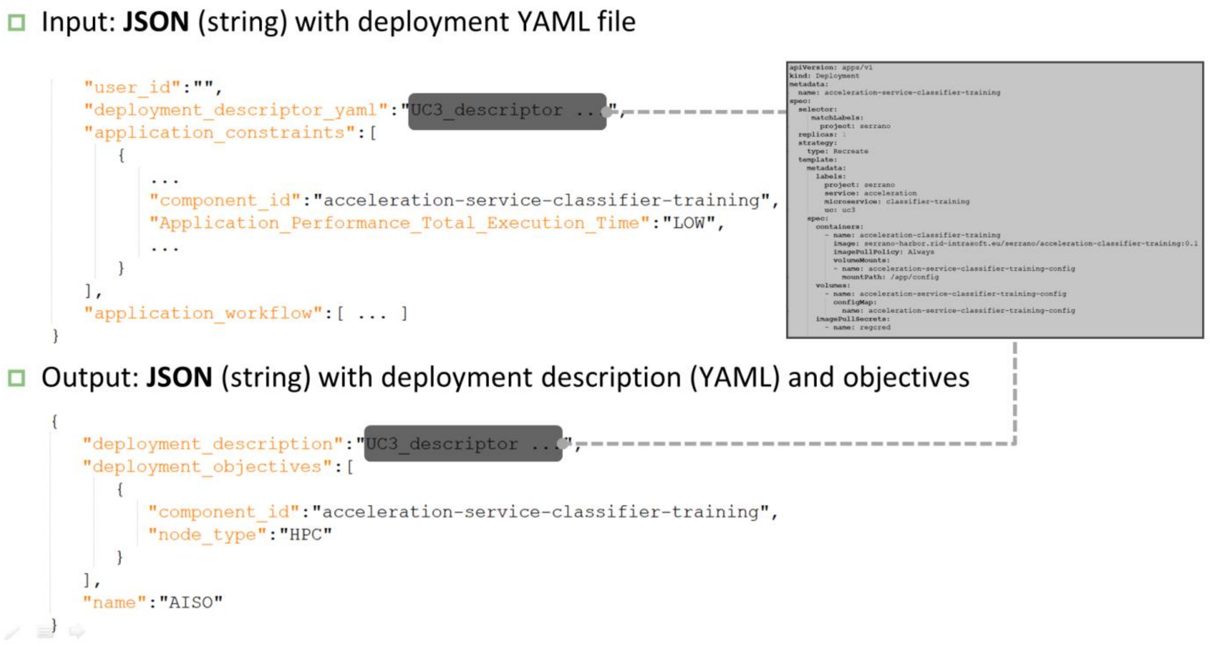
```
{
    "clusters": [
        {
            "name": "UVT Production",
            "type": "k8s",
            "uuid": "7628b895-3a91-4f0c-b0b7-033eab309891"
        },
        ...
    ]
}
```

**Figure 6: Interaction with the CTH**

Several tests were made to ensure the successful interaction of the AISO with the RO (e.g., correct format of deployment description and objectives). Initially, focus was given to successfully deploying the application micro-services to the appropriate resources through the provided deployment descriptor. Then, additional tests were made to ensure that the RO could properly use the produced deployment scenarios/objectives.

The examples presented in the following figures showcase the usage of the AISO and are based on the deployment and execution of the applications of a SERRANO UC, taking into account specific user requirements regarding the total execution time or energy consumption of the respective components/microservices. In order to present both the input and output of the AISO, the "CreateDeploymentScenarios" service was used (Figure 3), and hence the output of the AISO had to be manually provided to the RO. Alternatively, the "ApplicationDeploymentThroughRO" could be used, which directly provides the generated output to the RO.

In the first example (Figure 7), the application to be deployed is about training the ML model being used for anomalies detection in the third SERRANO UC, and the user requirement is minimum application execution time. The AISO suggests the usage of HPC (instead of edge resources) so that the total execution time of this process is minimized. The AISO provides this particular response since it has been informed (via the CTH services) that there are only two options available, i.e., usage of edge resources or HPC, and based on the data collected for these platforms the second one is the best option.



- Input: **JSON** (string) with deployment YAML file

```
  "user_id":"",
  "deployment_descriptor_yaml":"UC3_descriptor ...",
  "application_constraints":[
    {
      ...
      "component_id":"acceleration-service-classifier-training",
      "Application_Performance_Total_Execution_Time":"LOW",
      ...
    }
  ],
  "application_workflow":[ ... ]
}
```

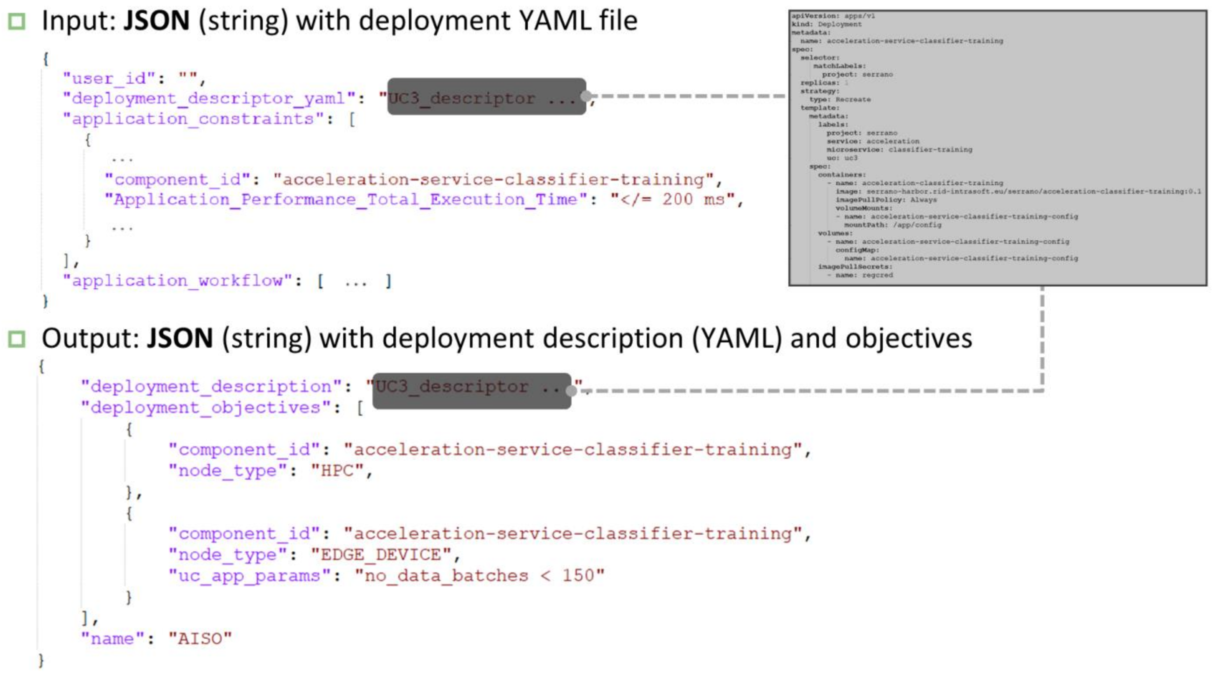
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: acceleration-service-classifier-training
spec:
  selector:
     matchLabels:
       project: serrano
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        project: serrano
        service: acceleration
        microservice: classifier-training
        uc: uc3
    spec:
      containers:
        - name: acceleration-classifier-training
          image: serrano-harbor.rid-intrasoft.eu/serrano/acceleration-classifier-training:0.1
          imagePullPolicy: Always
          volumeMounts:
          - name: acceleration-service-classifier-training-config
            mountPath: /app/config
      volumes:
        - name: acceleration-service-classifier-training-config
          configMap:
            name: acceleration-service-classifier-training-config
      imagePullSecrets:
        - name: regcred
```

- Output: **JSON** (string) with deployment description (YAML) and objectives

```
{
  "deployment_description":"UC3_descriptor ...",
  "deployment_objectives":[
    {
      "component_id":"acceleration-service-classifier-training",
      "node_type":"HPC"
    }
  ],
  "name":"AISO"
}
```

**Figure 7: AI-enhanced Service Orchestrator Example no. 1**

In the second example (Figure 8), the AISO detects that two options are available (i.e., either usage of HPC or an edge resource with the appropriate configuration), so that the total execution time of the above component/micro-service is below a predefined threshold.



- Input: **JSON** (string) with deployment YAML file

```
{
  "user_id": "",
  "deployment_descriptor_yaml": "UC3_descriptor ...",
  "application_constraints": [
    {
      ...
      "component_id": "acceleration-service-classifier-training",
      "Application_Performance_Total_Execution_Time": "</= 200 ms",
      ...
    }
  ],
  "application_workflow": [ ... ]
}
```

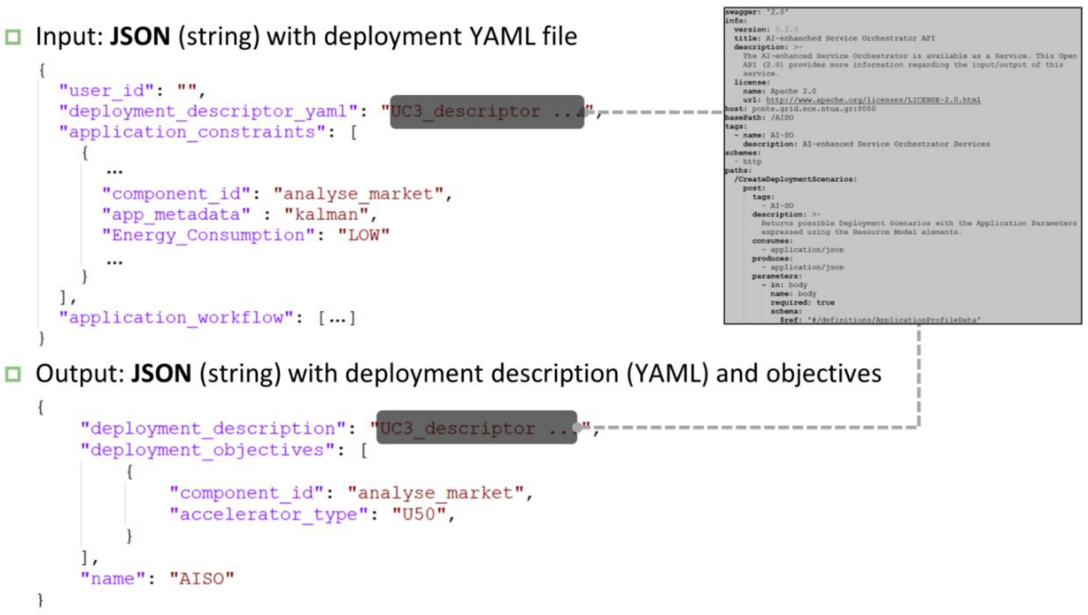
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: acceleration-service-classifier-training
spec:
  selector:
     matchLabels:
       project: serrano
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        project: serrano
        service: acceleration
        microservice: classifier-training
        uc: uc3
    spec:
      containers:
        - name: acceleration-classifier-training
          image: serrano-harbor.rid-intrasoft.eu/serrano/acceleration-classifier-training:0.1
          imagePullPolicy: Always
          volumeMounts:
          - name: acceleration-service-classifier-training-config
            mountPath: /app/config
      volumes:
        - name: acceleration-service-classifier-training-config
          configMap:
            name: acceleration-service-classifier-training-config
      imagePullSecrets:
        - name: regcred
```

- Output: **JSON** (string) with deployment description (YAML) and objectives

```
{
    "deployment_description": "UC3_descriptor ...",
    "deployment_objectives": [
        {
            "component_id": "acceleration-service-classifier-training",
            "node_type": "HPC",
        },
        {
            "component_id": "acceleration-service-classifier-training",
            "node_type": "EDGE_DEVICE",
            "uc_app_params": "no_data_batches < 150"
        }
    ],
    "name": "AISO"
}
```

**Figure 8: AI-enhanced Service Orchestrator Example no. 2**

In the third example (Figure 9), which concerns the market analysis microservice used by the second SERRANO use case, the AISO suggests that the best option for the lowest possible energy consumption would be the usage of a U50 accelerator.



- Input: **JSON** (string) with deployment YAML file

```
{
  "user_id": "",
  "deployment_descriptor_yaml": "UC3_descriptor ...",
  "application_constraints": [
    {
      ...
      "component_id": "analyse_market",
      "app_metadata" : "kalman",
      "Energy_Consumption": "LOW"
      ...
    }
  ],
  "application_workflow": [...]
}
```

```
swagger: '2.0'
info:
  version: 0.2.3
  title: AI-enhanched Service Orchestrator API
  description: >-
    The AI-enhanced Service Orchestrator is available as a Service. This Open
    API (2.0) provides more information regarding the input/output of this
    service.
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
host: ponte.grid.ece.ntua.gr:8080
basePath: /AISO
tags:
  - name: AI-SO
    description: AI-enhanced Service Orchestrator Services
schemes:
  - http
paths:
  /CreateDeploymentScenarios:
    post:
      tags:
        - AI-SO
      description: >-
        Returns possible Deployment Scenarios with the Application Parameters
        expressed using the Resource Model elements.
      consumes:
        - application/json
      produces:
        - application/json
      parameters:
        - in: body
          name: body
          required: true
          schema:
            $ref: '#/definitions/ApplicationProfileData'
```

- Output: **JSON** (string) with deployment description (YAML) and objectives

```
{
    "deployment_description": "UC3_descriptor ...",
    "deployment_objectives": [
        {
            "component_id": "analyse_market",
            "accelerator_type": "U50",
        }
    ],
    "name": "AISO"
}
```

**Figure 9: AI-enhanced Service Orchestrator Example no. 3**

In all the cases above, the output of the AISO (i.e., the JSON description with the particular scenarios/objectives along with the deployment descriptor YAML file) was accordingly sent to the RO, which returned back the deployment UID through which the respective application microservices status can be observed and un-deployed, when being necessary.

More precisely, a POST request is initially sent to the RO REST service (*/api/v1/orchestrator/deployments*) with the JSON description (i.e., the output of the aforementioned process) that encompasses both the deployment YAML description and the particular objectives detected, which is accordingly used for making the actual deployment of the application micro-services. The RO returns back the deployment unique ID (DUID). Then, by sending a GET request to the RO REST service *(/api/v1/orchestrator/deployments/{DUID}*) more information about the deployment process that took place can be retrieved through the JSON file that this service returns. More information about the last two services of the RO can be also found in Section 4.2.3.2.3.

## 4.2 Resource Orchestrator

### 4.2.1 Description

SERRANO adopts a hierarchical architecture to enable end-to-end cognitive resource orchestration and transparent application deployment over heterogeneous resources. The SERRANO Resource Orchestrator acts as the high-level orchestrator that interacts with multiple Local Orchestrators, each handling individual parts of the overall unified infrastructure. The Orchestration Drivers complete the implementation of the hierarchical resource orchestration.

An Orchestration Driver provides an abstraction layer for interacting with the specific edge, cloud, and HPC orchestration mechanisms, dealing with the low-level details of the heterogeneous Local Orchestrators at the individual platforms. The adopted design enables the SERRANO Resource Orchestrator to manage the underlying heterogeneous infrastructure more abstractly and disaggregated than the Local Orchestrators. Figure 10 depicts the architecture and the main components of the Resource Orchestrator and Orchestration Drivers.

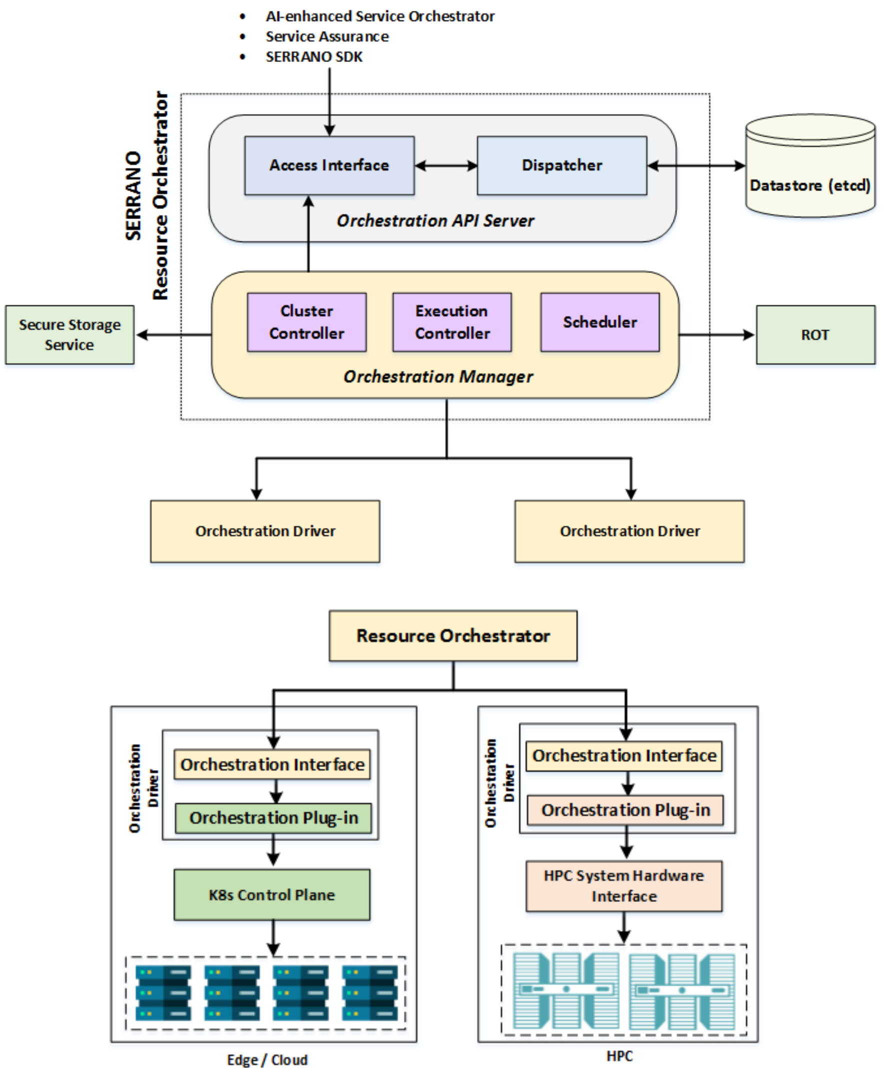


**Figure 10: Resource Orchestrator and Orchestration Drivers architecture and main components**

## 4.2.2 Inner components

The Resource Orchestrator consists of two primary services: *Orchestration API Server* and *Orchestration Manager*, while the Datastore component completes the architecture. The Datastore is based on etcd [1], an open-source distributed key-value store, and stores the SERRANO API objects that include configuration and state data for the available platforms, deployed applications, and SERRANO hardware and software accelerated kernels. It also facilitates the distributed communication among the Orchestration API Server, Orchestration Manager, and Orchestration Drivers.

Regarding the Orchestration API Service, the *Access Interface* exposes the appropriate interfaces to enable bidirectional communication for exchanging commands, information, and notifications. The *Dispatcher* handles the interaction with the Datastore by managing the SERRANO Orchestration API objects. These objects serve as the primary means of communication between the different components of the system. Additional information for the SERRANO Orchestration API objects is available in Section 9.1 at deliverable D5.4 (M31).

The Orchestration Manager implements the main part of the application logic and coordinates the resource allocation and application deployment, kernel execution, and secure storage policy management operations. It performs operations based on the SERRANO Orchestration API objects that are created through the API Server. The *Scheduler Controller* interacts with the ROT to retrieve the instructions for the cognitive application deployment and definition of secure storage policies. The *Cluster Controller* attaches Kubernetes clusters and HPC platforms to the Resource Orchestrator and oversees their operational state. The *Execution Controller* prepares the required application deployment instructions and triggers the actual deployment by interacting with the Orchestration Drivers at the selected edge/cloud and HPC platforms.

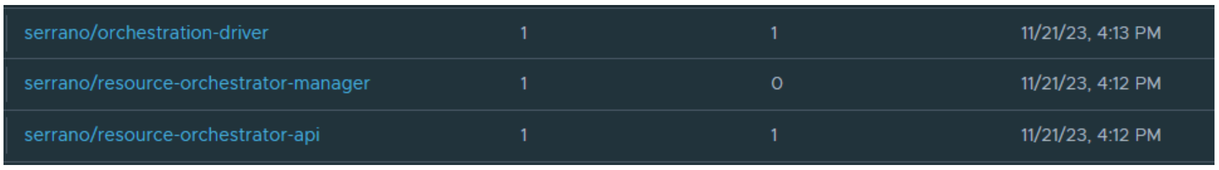
Since SERRANO unifies edge and cloud platforms that use Kubernetes (K8s) as the orchestration platform and HPC platforms with HPC resource managers [2] and batch jobs schedulers [3], two types of Orchestration Drivers are available. The *Orchestration Interface* component provides an infrastructure-agnostic interface between the Resource Orchestrator (i.e., Orchestration Manager) and the Local Orchestrators. It facilitates the generic description of the deployment preferences and constraints. The Orchestration Plug-in differs for each Orchestration Driver type and translates the Resource Orchestrator requests to specific actions for the Local Orchestrator at each platform.

Deliverables D5.3 (M15) and D5.4 (M31) provide more details for the overall design and implementation of the Resource Orchestrator and Orchestration Drivers.

### 4.2.3 Integration details and REST APIs

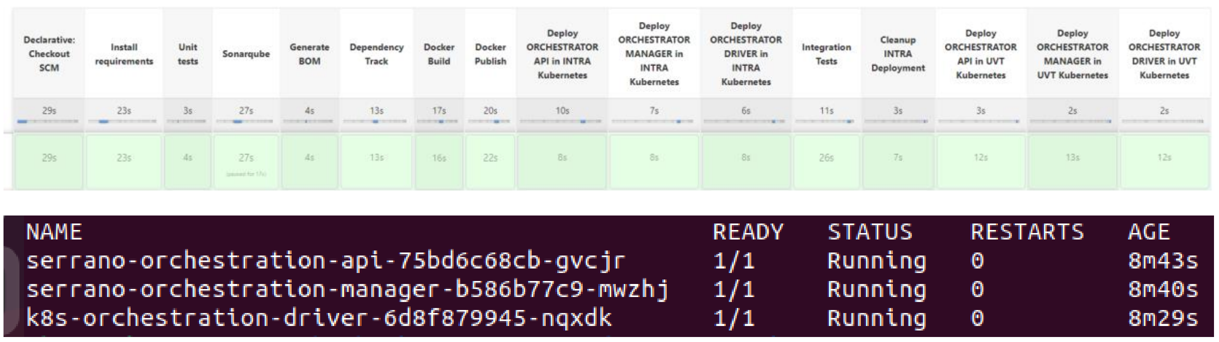
#### *4.2.3.1 Integration details*

The Resource Orchestrator services and Orchestration Drivers have been implemented in Python, leveraging popular open-source frameworks such as FastAPI [5], Pika [6], and PyQt [7]. These components, along with their configuration files are packaged as Python applications and seamlessly integrated into the SERRANO CI/CD pipeline. Within the SERRANO Harbor image repository [57], the Orchestration API and Orchestration Manager services are packaged within its dedicated container image, whereas both Orchestration Drivers share a common image (Figure 11). In addition, all the required Kubernetes YAML description files (i.e., ConfigMap, Deployment, Services, Ingress) were created to facilitate to facilitate their deployment on Kubernetes platforms. The developed components are also integrated into SERRANO's Jenkins pipeline, leading to their automatic deployment in the SERRANO testbed infrastructure (Figure 12). Figure 13 provides a visual representation of the setup employed for the final integration tests of the SERRANO Resource Orchestrator, demonstrating the comprehensive and well-integrated nature of our solution.



| | | | |
|---|---|---|---|
| serrano/orchestration-driver | 1 | 1 | 11/21/23, 4:13 PM |
| serrano/resource-orchestrator-manager | 1 | 0 | 11/21/23, 4:12 PM |
| serrano/resource-orchestrator-api | 1 | 1 | 11/21/23, 4:12 PM |

**Figure 11: Resource Orchestrator and Orchestration Drivers container images in SERRANO Harbor image repository**



| Declarative: Checkout SCM | Install requirements | Unit tests | Sonarqube | Generate BOM | Dependency Track | Docker Build | Docker Publish | Deploy ORCHESTRATOR API in INTRA Kubernetes | Deploy ORCHESTRATOR MANAGER in INTRA Kubernetes | Deploy ORCHESTRATOR DRIVER in INTRA Kubernetes | Integration Tests | Cleanup INTRA Deployment | Deploy ORCHESTRATOR API in UVT Kubernetes | Deploy ORCHESTRATOR MANAGER in UVT Kubernetes | Deploy ORCHESTRATOR DRIVER in UVT Kubernetes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 29s | 23s | 3s | 27s | 4s | 13s | 17s | 20s | 10s | 7s | 6s | 11s | 3s | 3s | 2s | 2s |
| 29s | 23s | 4s | 27s (paused for 17s) | 4s | 13s | 16s | 22s | 8s | 8s | 8s | 26s | 7s | 12s | 13s | 12s |

```
NAME                                        READY   STATUS    RESTARTS   AGE
serrano-orchestration-api-75bd6c68cb-gvcjr  1/1     Running   0          8m43s
serrano-orchestration-manager-b586b77c9-mwzhj 1/1   Running   0          8m40s
k8s-orchestration-driver-6d8f879945-nqxdk   1/1     Running   0          8m29s
```

**Figure 12: Resource Orchestrator and Orchestration Driver into Jenkins CI/CD pipeline and their deployment in UVT's K8s cluster**

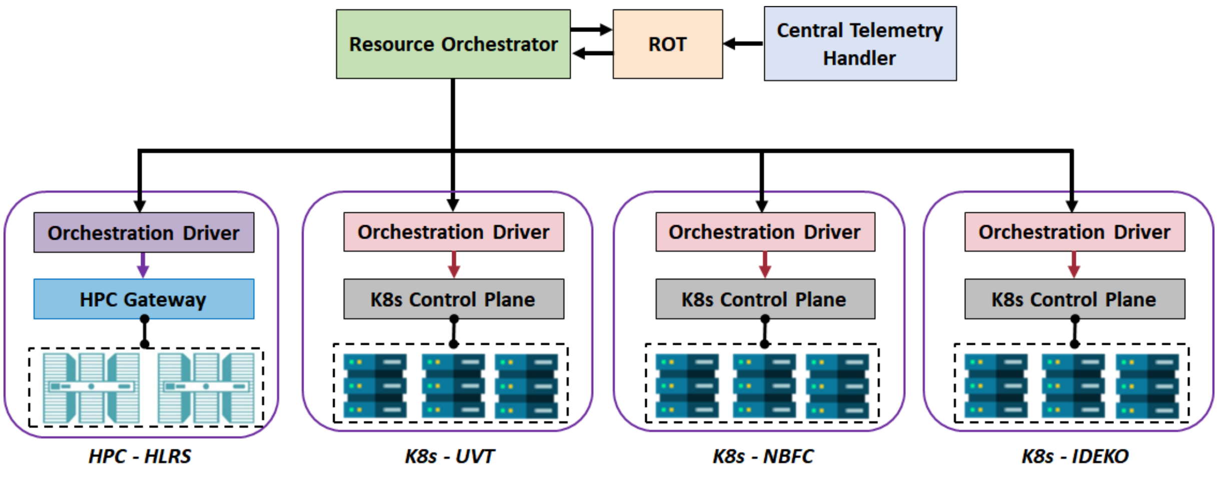
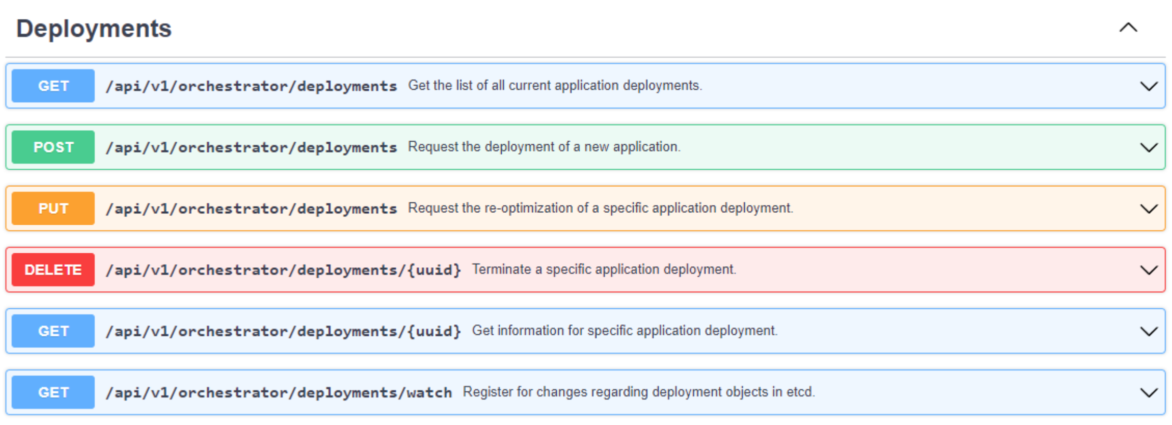


**Figure 13: Setup for the integration tests of SERRANO Resource Orchestrator**

**Table 3: Integration details of Resource Orchestrator**

| | |
|---|---|
| **IP(s)/Port(s)** | Resource Orchestrator:<br>• https://resource-orchestrator.services.cloud.ict-serrano.eu<br>ROT Controller:<br>• https://rot.services.cloud.ict-serrano.eu<br>Central Telemetry Handler:<br>• https:// central-telemetry.services.cloud.ict-serrano.eu<br>Orchestration Drivers:<br>• https://uvt-driver.services.cloud.ict-serrano.eu (K8s - UVT)<br>• https://nbfc-driver.services.cloud.ict-serrano.eu (K8s - NBFC)<br>• https://ideko-driver.services.cloud.ict-serrano.eu (K8s - IDEKO)<br>• https://hpc-interface.services.cloud.ict-serrano.eu (HPC) |
| **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** | https://resource-orchestrator.services.cloud.ict-serrano.eu |
| **API documentation** | *https://raw.githubusercontent.com/ict-serrano/Resource-Orchestrator/main/orchestrator_rest.yaml* |
| **Location of integration tests** | *https://raw.githubusercontent.com/ict-serrano/Resource-Orchestrator/main/Jenkinsfile* |

The final version of the exposed REST API includes several methods organized into two main categories. The first set of methods (Figure 14) enables the deployment and management of cloud-native applications, execution of SERRANO accelerated kernels, and the cognitive creation of secure storage policies. The second set (Figure 15) abstracts the interaction of the Orchestration Manager and Orchestration Driver services.


Deployments

GET /api/v1/orchestrator/deployments Get the list of all current application deployments.

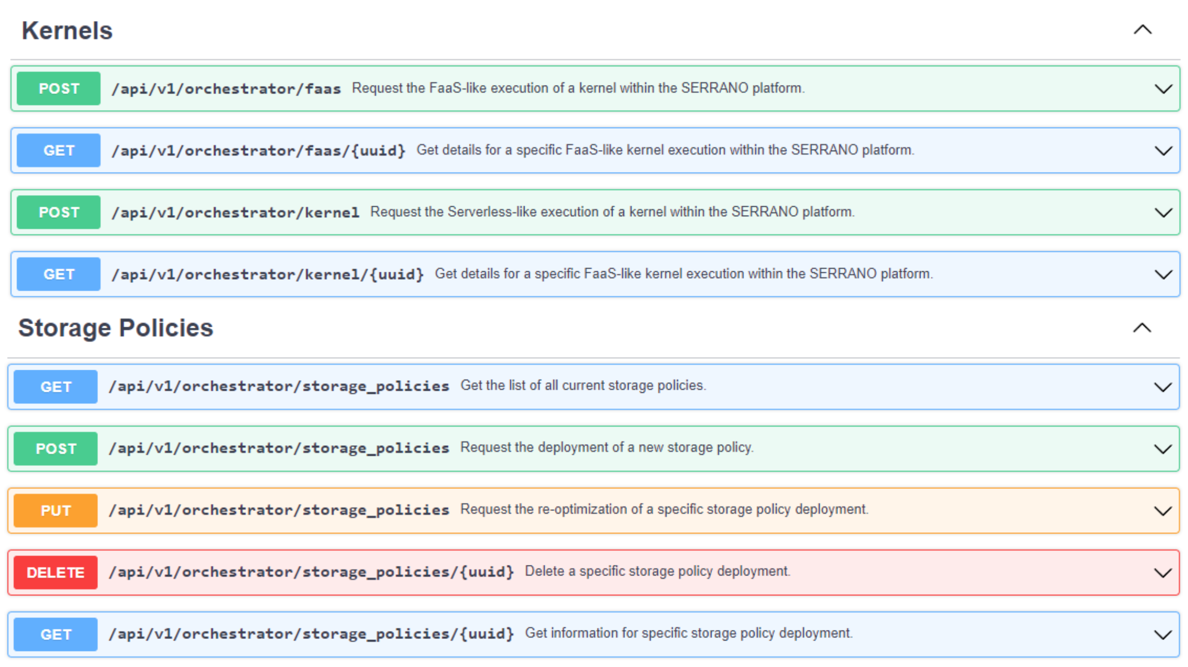
POST /api/v1/orchestrator/deployments Request the deployment of a new application.

PUT /api/v1/orchestrator/deployments Request the re-optimization of a specific application deployment.

DELETE /api/v1/orchestrator/deployments/{uuid} Terminate a specific application deployment.

GET /api/v1/orchestrator/deployments/{uuid} Get information for specific application deployment.

GET /api/v1/orchestrator/deployments/watch Register for changes regarding deployment objects in etcd.



Kernels

POST /api/v1/orchestrator/faas Request the FaaS-like execution of a kernel within the SERRANO platform.

GET /api/v1/orchestrator/faas/{uuid} Get details for a specific FaaS-like kernel execution within the SERRANO platform.

POST /api/v1/orchestrator/kernel Request the Serverless-like execution of a kernel within the SERRANO platform.

GET /api/v1/orchestrator/kernel/{uuid} Get details for a specific FaaS-like kernel execution within the SERRANO platform.

Storage Policies

GET /api/v1/orchestrator/storage_policies Get the list of all current storage policies.

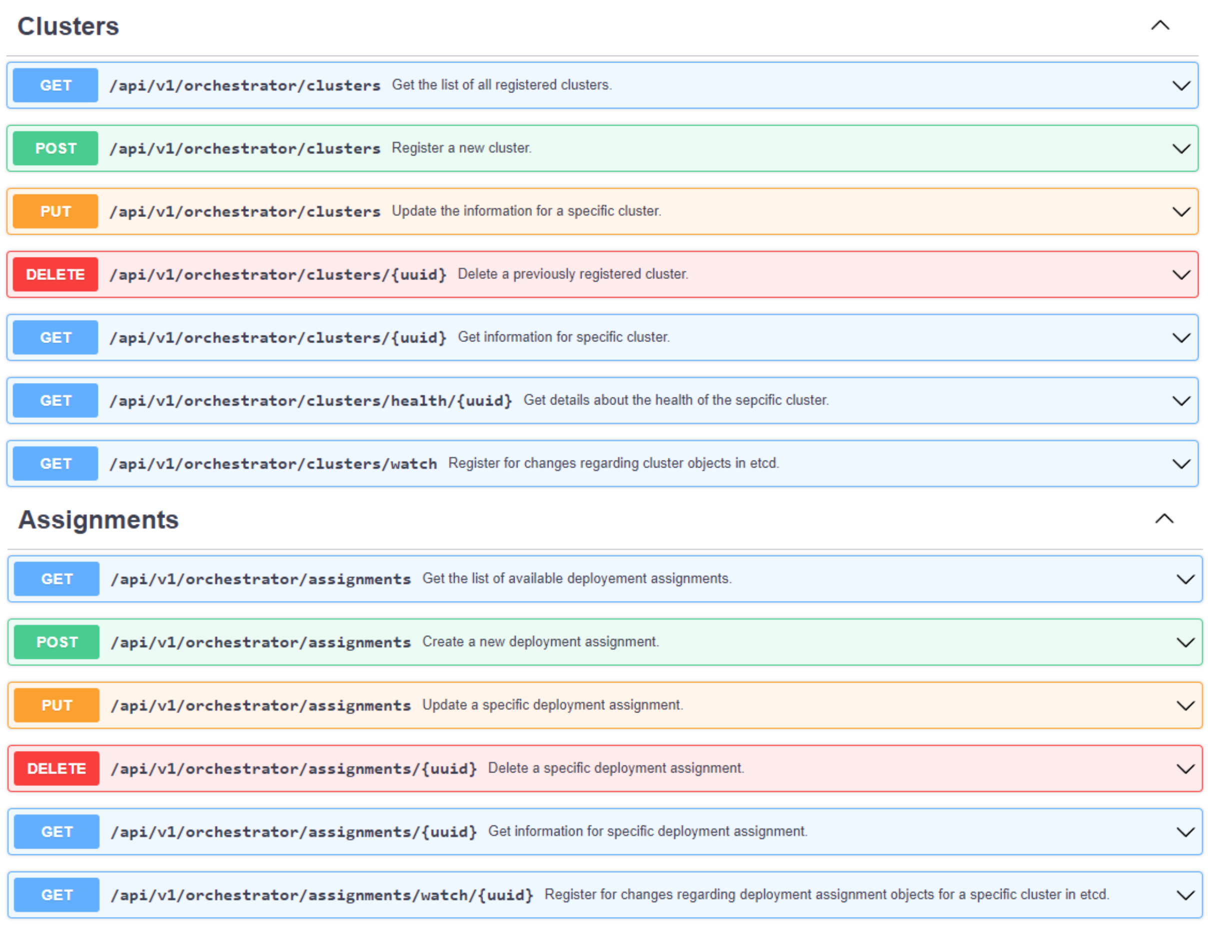
POST /api/v1/orchestrator/storage_policies Request the deployment of a new storage policy.

PUT /api/v1/orchestrator/storage_policies Request the re-optimization of a specific storage policy deployment.

DELETE /api/v1/orchestrator/storage_policies/{uuid} Delete a specific storage policy deployment.

GET /api/v1/orchestrator/storage_policies/{uuid} Get information for specific storage policy deployment.

**Figure 14: Resource Orchestrator REST API**



Clusters

GET /api/v1/orchestrator/clusters Get the list of all registered clusters.

POST /api/v1/orchestrator/clusters Register a new cluster.

PUT /api/v1/orchestrator/clusters Update the information for a specific cluster.

DELETE /api/v1/orchestrator/clusters/{uuid} Delete a previously registered cluster.

GET /api/v1/orchestrator/clusters/{uuid} Get information for specific cluster.

GET /api/v1/orchestrator/clusters/health/{uuid} Get details about the health of the sepcific cluster.

GET /api/v1/orchestrator/clusters/watch Register for changes regarding cluster objects in etcd.

Assignments

GET /api/v1/orchestrator/assignments Get the list of available deployement assignments.

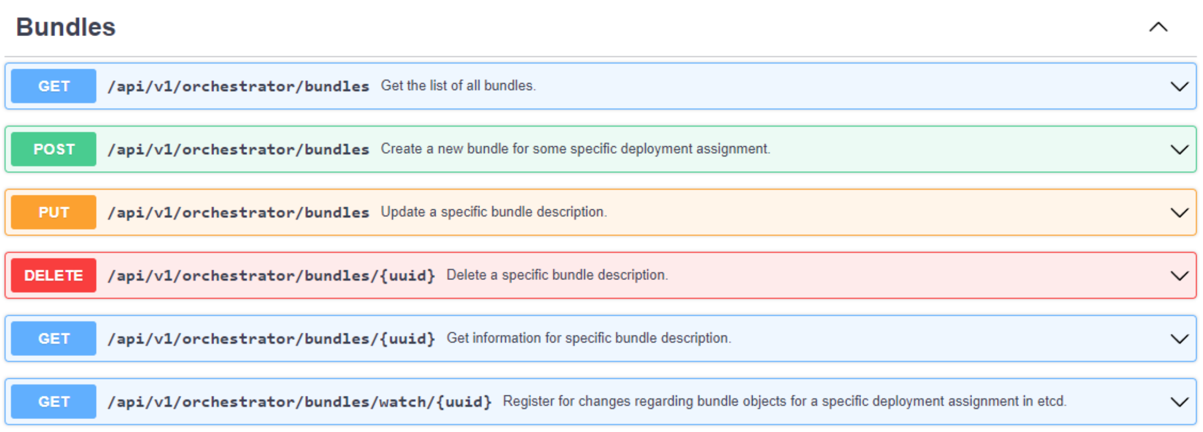
POST /api/v1/orchestrator/assignments Create a new deployment assignment.

PUT /api/v1/orchestrator/assignments Update a specific deployment assignment.

DELETE /api/v1/orchestrator/assignments/{uuid} Delete a specific deployment assignment.

GET /api/v1/orchestrator/assignments/{uuid} Get information for specific deployment assignment.

GET /api/v1/orchestrator/assignments/watch/{uuid} Register for changes regarding deployment assignment objects for a specific cluster in etcd.


Bundles

GET /api/v1/orchestrator/bundles Get the list of all bundles.

POST /api/v1/orchestrator/bundles Create a new bundle for some specific deployment assignment.

PUT /api/v1/orchestrator/bundles Update a specific bundle description.

DELETE /api/v1/orchestrator/bundles/{uuid} Delete a specific bundle description.

GET /api/v1/orchestrator/bundles/{uuid} Get information for specific bundle description.

GET /api/v1/orchestrator/bundles/watch/{uuid} Register for changes regarding bundle objects for a specific deployment assignment in etcd.

**Figure 15: Resource Orchestrator REST API – Methods related to inter-component communication**

### *4.2.3.2 Integration with SERRANO Services*

#### *4.2.3.2.1 Resource Orchestrator and Orchestration Drivers*

During their initialization, the Orchestration Drivers are registered to the Resource Orchestrator through the exposed REST endpoint by the Orchestration API Server. They also send a summary of the available resources in the platforms they manage. The Orchestration API Server uses this information to update the respective contents in Datastore. Bellow, there is an example of the available K8s and HPC clusters under the management of the Resource Orchestrator for the testbed setup in Figure 13. This information is available through the following GET request.

```
GET /api/v1/orchestrator/clusters
```

```
{"clusters":[{"cluster_uuid":"e65c33ac-3109-4a15-9cc2-9f4e90f82c2d","type":"k8s","
last_seen":"1700469459"}, {"cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","
type":"k8s","last_seen":"1700478141"}, {"cluster_uuid":"5a075716-7d7d-4b40-9566-bc
1a33ee70c2","type":"k8s","last_seen":"1700478258"}, {"cluster_uuid":"b7143497-a168
-4c8d-a899-8c56dccda8ad","type":"hpc","last_seen":"1700498273"}]}
```

#### *4.2.3.2.2 Secure storage policies cognitive creation*

The SERRANO platform supports creating automated secure storage policies based on significantly varying storage task requirements. This operation integrates the functionality of many platform components, such as the Secure Storage service (Section 4.8.1), the SERRANO Telemetry Framework (Section 4.4), the Resource Optimization Toolkit (Section 4.3), and the SERRANO Resource Orchestrator services.

**Storage request orchestration**

The SERRANO Resource Orchestrator provides a set of REST methods (*/api/v1/orchestrator/storage_policies*) (Figure 14) that enable external services to request the creation of a secure storage policy within the SERRANO platform. These requests are initially handled by the Orchestration API server. The Orchestration API Server validates the request parameters and creates the corresponding Storage Policy object in the Datastore.

```
{"name": "hybrid-policy", "description": "", "policy_parameters": {"data_size": 0,
"cost": 0, "volatility": 0, "availability": 1, "latency": 1, "lat": 45.7472357, "l
ng": 21.2316107}, "policy_uuid": "c13a8c59-a218-4c29-82d1-482e688b3d47", "kind": "
StoragePolicy", "decision": {}, "cc_policy_id": 0, "status": 2, "logs": [{"timesta
mp": 1700568600, "event": "Storage Policy description received."}, {"timestamp": 1
700568600, "event": "Request ROT decision"}], "updated_by": "Orchestration.Manager
", "created_at": 1700568600, "updated_at": 1700568600}
```

Next, the Orchestration Manager is notified of the new request and, through its Scheduler controller, triggers the orchestration of the secure storage policy request. To this end, it requests the orchestration decision from the ROT. Section 4.3.3.2 provides more technical details for this step. When the Orchestration Manager receives the ROT's decision, it updates the *decision* field in the corresponding Storage Policy object through the Orchestration API Server. The following example corresponds to a secure storage policy that utilizes cloud and edge storage locations.

```
{"name": "hybrid-policy", "description": "", "policy_parameters": {"data_size": 0,
"cost": 0, "volatility": 0, "availability": 1, "latency": 1, "lat": 45.7472357, "l
ng": 21.2316107}, "policy_uuid": "c13a8c59-a218-4c29-82d1-482e688b3d47", "kind": "
StoragePolicy", "decision": {"backends": [78,79], "edge_devices": [1, 2], "redunda
nt_packets": 1}, "cc_policy_id": 0, "status": 3, "logs": [{"timestamp": 1700568600
, "event": "Storage Policy description received."}, {"timestamp": 1700568600, "eve
nt": "Request ROT decision"}, {"timestamp": 1700568607, "event": "Get decision res
ponse from ROT"}], "updated_by": "Orchestration.Manager", "created_at": 1700568600
, "updated_at": 1700568607}
```

**Secure storage policy creation**

In the subsequent phase, the Orchestration Manager, through the Execution Controller, initiates the policy creation process by requesting it from the Secure Storage Service. It formats the appropriate request based on the provided orchestration decision and triggers the creation process by executing the exposed REST method (*POST /storage_policy*) provided by the Secure Storage Service (Section 4.8.1).

```
POST /storage_policy
Parameters:
{
   "name": "hybrid-policy", "description":"", "backends": [78, 79],
   "edge_devices": [1,2],
   "redundancy": {"redundant_packets": 1, "scheme": "RLNC"}
}
```

Finally, the Orchestration Manager updates the *cc_policy_id*, *status*, and *logs* fields according to the response status of the previous request.

```
{"name": "hybrid-policy", "description": "", "policy_parameters": {"data_size": 0,
"cost": 0, "volatility": 0, "availability": 1, "latency": 1, "lat": 45.7472357, "l
ng": 21.2316107}, "policy_uuid": "c13a8c59-a218-4c29-82d1-482e688b3d47", "kind": "
StoragePolicy", "decision": {"backends": [78,79], "edge_devices": [1, 2], "redunda
nt_packets": 1}, "cc_policy_id": 6318801792532480, "status": 5, "logs": [{"timesta
mp": 1700568600, "event": "Storage Policy description received."}, {"timestamp": 1
700568600, "event": "Request ROT decision"}, {"timestamp": 1700568607, "event": "G
et decision response from ROT"},{"timestamp": 1700568607, "event": "Request Storag
e Policy to Secure Storage Gateway"},{"timestamp": 1700568612, "event": "Storage P
olicy created successfully"}], "updated_by": "Orchestration.Manager", "created_at"
: 1700568600, "updated_at": 1700568612}
```

### *4.2.3.2.3 Cloud-native applications deployment*

Sections 9.1 and 9.4.2 in the deliverable D5.4 (M31) provide a detailed technical presentation regarding the transparent application deployment within the SERRANO platform. Next, we focus on the integration among the Resource Orchestrator services for deploying the Acceleration Service from the Anomaly Detection in Manufacturing Settings use case (Section 4.12), which includes three microservices.

**Orchestration API server and Orchestration Manager**

The Orchestration API server receives the requests for the application deployments through its exposed REST methods (*/api/v1/orchestrator/deployments*) (Figure 14) and creates the appropriate Deployment object[1] in the Datastore. Next, we list the Deployment object for the Acceleration Service deployment.

```
{
  "kind": "Deployment", "name": "UC3-Acceleration-Service",
  "deployment_uuid": "fdf45855-1299-47f1-8ea6-98be8d89030b",
  "deployment_description": "YAML DESCRIPTION",
  "assignments": [], "assignments_status": [],
  "logs":[{"timestamp":1700589809, "event":"Deployment description received."}],
  "status":1, "updated_by": "Orchestration.API","created_at": 1700589809,
  "updated_at": 1700589809
}
```

The Orchestration Manager is notified of the new request and, through its Scheduler controller, triggers the ROT to provide the high-level orchestration decision for the application deployment.  Then, the Orchestration Manager, through its Execution Controller, creates and stores in the Datastore the appropriate number of Assignment and Bundle objects according to the assignment of the application microservices into the individual edge, cloud, and HPC platforms as described in the ROT response.

[1] For clarity, we omitted the contents of the *deployment_description* field that include the YAML description of application's microservices.

```
{"kind": "Deployment", "assignments": [{"cluster_uuid": "e65c33ac-3109-4a15-9cc2-9
f4e90f82c2d", "deployments": ["acceleration-service-classifier-training", "acceler
ation-service-data-manager"]}, {"cluster_uuid": "5a075716-7d7d-4b40-9566-bc1a33ee7
0c2", "deployments": ["acceleration-service-model-inference"]}]}
```

The ROT allocated the three application microservices across two distinct K8s clusters. Specifically, two of these microservices have been assigned to the IDEKO cluster, identifiable by the unique identifier *"e65c33ac-3109-4a15-9cc2-9f4e90f82c2"*, while the third resides in the NBFC cluster with the unique identifier *"5a075716-7d7d-4b40-9566-bc1a33ee70c2"*. Consequently, the Execution Controller within the Orchestration Manager creates two Assignment objects (UUIDs *"4aae522e-3d56-42a2-b5f7-56d1cc4bef2b"* and *"4ea8f80a-8354-433f-8100-5ed8b469f54c"*), each associated with a selected platform. These objects are also linked with the initial Deployment object (*"fdf45855-1299-47f1-8ea6-98be8d89030b"*).

```
/serrano/orchestrator/assignments/5a075716-7d7d-4b40-9566-bc1a33ee70c2/assignment/
4aae522e-3d56-42a2-b5f7-56d1cc4bef2b
{"uuid": "4aae522e-3d56-42a2-b5f7-56d1cc4bef2b", "kind": "Deployment", "cluster_uu
id": "5a075716-7d7d-4b40-9566-bc1a33ee70c2", "deployment_uuid": "fdf45855-1299-47f
1-8ea6-98be8d89030b", "bundles": ["6f2e474f-06b8-4b96-afa3-e3f90a088a9a"], "status
": 1, "updated_by": "Orchestration.Manager", "logs": [{"timestamp": 1700592156, "e
vent": "Assignment created."}], "created_at": 1700592156, "updated_at": 1700592156
}
/serrano/orchestrator/assignments/e65c33ac-3109-4a15-9cc2-9f4e90f82c2d/assignment/
4ea8f80a-8354-433f-8100-5ed8b469f54c
{"uuid": "4ea8f80a-8354-433f-8100-5ed8b469f54c", "kind": "Deployment", "cluster_uu
id": "e65c33ac-3109-4a15-9cc2-9f4e90f82c2d", "deployment_uuid": "fdf45855-1299-47f
1-8ea6-98be8d89030b", "bundles": ["1499c29a-acbb-46c7-84fd-38eb6c4559f2", "8241f40
b-1d02-446c-b4e6-94a5944f84c7"], "status": 1, "updated_by": "Orchestration.Manager
", "logs": [{"timestamp": 1700592156, "event": "Assignment created."}], "created_a
t": 1700592156, "updated_at": 1700592156}
```

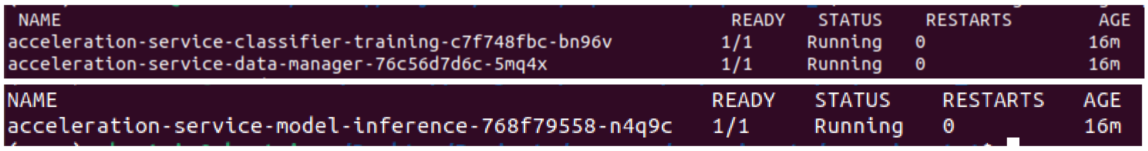
Furthermore, three Bundle objects (identified by UUIDs *"6f2e474f-06b8-4b96-afa3-e3f90a088a9a"*, *"1499c29a-acbb-46c7-84fd-38eb6c4559f2"*, and *"8241f40b-1d02-446c-b4e6-94a5944f84c7"*) has been generated, each corresponding to a specific application microservice. These Bundles were mapped to the corresponding Assignment objects and provide the required deployment descriptions for the platform-level orchestration mechanisms. It's important to note that, due to space constraints, the detailed descriptions of these Bundles are omitted from this text.

**<u>Orchestration Manager and Orchestration Drivers</u>**

Creating the Bundles and Assignment objects by the Orchestration Manager activates the Orchestration Drivers at the at the two selected platforms. Each Orchestration Driver receives the Assignment object that includes the list of all Bundles' unique identifiers related to the specific assignment. The Orchestration Driver retrieves the description for each Bundle object and uses the K8s API to apply the required deployment actions. Next, we present the log messages from the Orchestration Driver in NBFC K8s clusters that successfully executed the Bundle that was related to its assignment.

```
INFO:SERRANO.Orchestrator.OrchestrationDriver:Assignment event for key '/serrano/orchestrator/assignments/5a075716-7d7d-4b40-9566-bc1a33ee70c2/assignment/43a82209-c3d3-4cc9-a584-fa9155056c13'
INFO:SERRANO.Orchestrator.DriverKubernetes:Handler deployment request ...
DEBUG:SERRANO.Orchestrator.DriverKubernetes:{"uuid": "43a82209-c3d3-4cc9-a584-fa9155056c13", "kind": "Deployment", "cluster_uuid": "5a075716-7d7d-4b40-9566-bc1a33ee70c2", "deployment_uuid": "fd9a6e44-428f-4c18-b13d-3ddd93509222", "bundles": ["5d4bb088-01b5-46cf-aa79-fd4536718968"], "status": 1, "updated_by": "Orchestration.Manager", "logs": [{"timestamp": 1700595973, "event": "Assignment created."}], "created_at": 1700595973, "updated_at": 1700595973}
DEBUG:SERRANO.Orchestrator.DriverKubernetes:Create ConfigMap 'acceleration-service-model-inference-config'
DEBUG:SERRANO.Orchestrator.DriverKubernetes:Successful ConfigMap description for bundle '5d4bb088-01b5-46cf-aa79-fd4536718968'
DEBUG:SERRANO.Orchestrator.DriverKubernetes:Create Deployment 'acceleration-service-model-inference'
DEBUG:SERRANO.Orchestrator.DriverKubernetes:Successful Deployment description for bundle '5d4bb088-01b5-46cf-aa79-fd4536718968'
INFO:SERRANO.Orchestrator.DriverKubernetes:Deployment for assignment '43a82209-c3d3-4cc9-a584-fa9155056c13' successfully executed
```

Finally, the Orchestration Manager informs, through the Central Telemetry Handler, the SERRANO telemetry framework to start the automatic monitoring of the deployed application (Section 4.4.3). Figure 16 shows the three deployed application microservices in the two selected K8s clusters.



```
NAME                                                    READY   STATUS    RESTARTS   AGE
acceleration-service-classifier-training-c7f748fbc-bn96v   1/1     Running   0          16m
acceleration-service-data-manager-76c56d7d6c-5mq4x         1/1     Running   0          16m
```

```
NAME                                                READY   STATUS    RESTARTS   AGE
acceleration-service-model-inference-768f79558-n4q9c   1/1     Running   0          16m
```

**Figure 16: Acceleration Service microservices deployed across two K8s clusters within the SERRANO platform**

#### *4.2.3.2.4 On-demand SERRANO HW/SW accelerated kernels execution*

The SERRANO platform supports the on-demand execution of the SERRANO-accelerated kernels. The on-demand execution is based on the Functional as a Service (FaaS) execution model. The SERRANO SDK supports the on-demand execution of the kernels, whereas the orchestration and deployment mechanisms handle all the required operations and return the results to the application service. The overall process involves the following steps: (1) move input data to SERRANO storage services, which provides the data description from the next step; (2) request the execution of kernel; and (3) retrieve the results. Figure 17 summarizes the overall workflow from an end-user perspective, while additional information is available in deliverable D5.4 (M31).

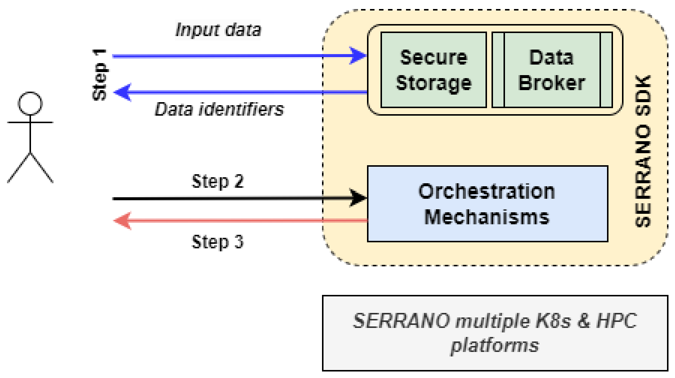


**Figure 17: Kernel execution and data handling from the end user's perspective, common approach for all supported modes and platforms**

## <u>Kernel execution in edge and cloud platforms</u>

We consider the on-demand execution of the KMEANS kernel from the Position Process service of the third use case. Initially, the service uses the appropriate method from the SERRANO SDK to push the input data for the kernel in the SERRANO platform.

```
{'queue_id': '84e0b309-e3f8-488e-a441-073720b55b4e', 'bucket_id': '84e0b309-e3f8-4
88e-a441-073720b55b4e', 'arguments': ['position', 'labels'], 'storage': 'broker',
'total_size_MB': 7.35}
```

Next, it requests the kernel execution from the SERRANO orchestration and deployment mechanisms. The Orchestration API server creates the corresponding Kernel object in the Datastore.

```
{"kind": "FaaS", "request_uuid": "1465c479-03d7-4123-8329-fac9fa580256", "kernel_n
ame": "kmeans", "deployment_objectives": {}, "data_description": {"queue_id": "3a0
47bd2-5558-410b-ae7c-8e7fd3953ae2", "bucket_id": "3a047bd2-5558-410b-ae7c-8e7fd395
3ae2", "arguments": ["position", "labels"], "storage": "broker", "total_size_MB":
7.35, "w": 200, "iterations": 2, "uuid": "1465c479-03d7-4123-8329-fac9fa580256"},
"assignment_uuid": "", "logs": [{"timestamp": 1700667284, "event": "Kernel descrip
tion received."}], "status": 2, "updated_by": "Orchestration.API", "created_at": 1
700667284, "updated_at": 1700667284}
```

The Orchestration Manager service contacts the ROT, which selects the appropriate type of accelerated resources and the specific platform that will assign the kernel execution. Next, the Orchestration Manager creates the necessary Assignment and Bundle objects in the Datastore. In the following example, the selected platform is NBFC K8s cluster (UUID "*5a075716-7d7d-4b40-9566-bc1a33ee70c2*") and the selected acceleration platform is GPU, as listed in the Bundle object.

```
/serrano/orchestrator/assignments/5a075716-7d7d-4b40-9566-bc1a33ee70c2/assignment/
b8f74a2e-a6fc-456a-94d2-34583b2ab7cb

{"uuid": "b8f74a2e-a6fc-456a-94d2-34583b2ab7cb", "kind": "FaaS", "cluster_uuid": "
5a075716-7d7d-4b40-9566-bc1a33ee70c2", "deployment_uuid": "1465c479-03d7-4123-8329
-fac9fa580256", "bundles": ["8d7c2128-1a97-4b13-bd42-69abf817ab7e"], "status": 1,
"updated_by": "Orchestration.Manager", "logs": [{"timestamp": 1700667284, "event":
"Assignment created."}], "created_at": 1700667284, "updated_at": 1700667284}

/serrano/orchestrator/bundles/bundle/8d7c2128-1a97-4b13-bd42-69abf817ab7e

{"uuid": "8d7c2128-1a97-4b13-bd42-69abf817ab7e", "description": {"kind": "FaaS", "
request_uuid": "1465c479-03d7-4123-8329-fac9fa580256", "kernel_name": "kmeans", "d
ata_description": {"queue_id": "3a047bd2-5558-410b-ae7c-8e7fd3953ae2", "bucket_id"
: "3a047bd2-5558-410b-ae7c-8e7fd3953ae2", "arguments": ["position", "labels"], "st
orage": "broker", "total_size_MB": 7.35, "mode": "gpu", "w": 200, "iterations": 2,
"uuid": "1465c479-03d7-4123-8329-fac9fa580256"}, "status": 1, "updated_by": "Orche
stration.Manager", "logs": [{"timestamp": 1700667284, "event": "Bundle created."}]
, "created_at": 1700667284, "updated_at": 1700667284}
```

The Orchestration Driver at the selected platform is notified for the kernel execution assignment and reads the instructions from the Bundle object. Then, it requests the actual execution from the OpenFaaS service, oversees the progress, and collects detailed monitoring information. The vAccel framework (Section 4.7.1.1) abstracts all the resource-specific details to enable the seamless deployment of the accelerated kernels across heterogeneous acceleration resources, such as GPUs and FPGAs.

```
INFO:SERRANO.Orchestrator.OrchestrationDriver:Assignment event(s) ...
INFO:SERRANO.Orchestrator.OrchestrationDriver:Assignment event for key '/serrano/o
rchestrator/assignments/5a075716-7d7d-4b40-9566-bc1a33ee70c2/assignment/b8f74a2e-a
6fc-456a-94d2-34583b2ab7cb'
DEBUG:SERRANO.Orchestrator.DriverKubernetes:{"uuid": "b8f74a2e-a6fc-456a-94d2-3458
3b2ab7cb", "kind": "FaaS", "cluster_uuid": "5a075716-7d7d-4b40-9566-bc1a33ee70c2",
"deployment_uuid": "1465c479-03d7-4123-8329-fac9fa580256", "bundles": ["8d7c2128-1
a97-4b13-bd42-69abf817ab7e"], "status": 1, "updated_by": "Orchestration.Manager",
"logs": [{"timestamp": 1700667284, "event": "Assignment created."}], "created_at":
1700667284, "updated_at": 1700667284}
INFO:SERRANO.Orchestrator.DriverKubernetes:Bundle for Faas kernel assignment 'b8f7
4a2e-a6fc-456a-94d2-34583b2ab7cb' is activated
INFO:SERRANO.Orchestrator.OrchestrationDriver.ExecutionWrapper:Submit execution re
quest to OpenFaaS service for request_uuid '1465c479-03d7-4123-8329-fac9fa580256'
```

When the kernel execution is completed, the Orchestration Driver updates the status of the Kernel and Assignment objects through the exposed REST methods from the Orchestration API server. At the same time, the OpenFaaS service forwards automatically to the end users through the Message Broker (Section 4.5.1.2)

**Kernel execution in HPC platform**

The workflow for the execution of SERRANO accelerated kernels into HPC platform is similar to the previous case. The difference lays in the final steps where the Orchestration Drivers for the HPC platforms interacts with the SERRANO HPC Gateway (Section 4.6) instead of the

OpenFaaS service. The HPC Gateway abstracts the interaction with the HPC infrastructure and the specific batch job scheduler mechanisms that used in this type environments.

When the HPC Orchestration Driver (cluster UUID *"b7143497-a168-4c8d-a899-8c56dccda8ad"*) is notified for the kernel execution assignment (UUID *"3543bf88-7ca6-45dc-a259-72001d17484a"*), it reads the instructions from the Bundle object. In this case, it is also responsible for moving the input data from the SERRANO Secure Storage service into the HPC. This operation utilizes the exposed data endpoints of the SERRANO HPC Gateway (Figure 37). Then, it requests the execution from the HPC Gateway service, oversees the progress, and collects detailed monitoring information. Moreover, the HPC Orchestration Driver has to transfer the results from the HPC platform through the HPC gateway and push them to the Message Broker to be forwarded automatically to the end users. Next, we present the log messages from the HPC Orchestration Driver that correspond to the execution of the KMEANS kernel in the SERRANO HPC infrastructure. Figure 18 summarizes the workflow for executing a SERRANO accelerated kernel in HPC platforms, highlighting the role of the Orchestration Driver and its interaction with the SERRANO HPC Gateway.

```
INFO:SERRANO.Orchestrator.OrchestrationDriver:Assignment event(s) ...
INFO:SERRANO.Orchestrator.OrchestrationDriver:Assignment event for key '/serrano/o
rchestrator/assignments/b7143497-a168-4c8d-a899-8c56dccda8ad/assignment/3543bf88-7
ca6-45dc-a259-72001d17484a'
INFO:SERRANO.Orchestrator.DriverHPC:Handle deployment request ...
INFO:SERRANO.Orchestrator.DriverHPC:Retrieve Bundle 'd0436f18-033d-4db1-befb-c7d82
d6d4d5c'
INFO:SERRANO.Orchestrator.DriverHPC:Bundle for assignment '3543bf88-7ca6-45dc-a259
-72001d17484a' is activated
INFO:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper:Move data from bucket 'e90a23
82-2008-468e-a0fd-4964c19a7331' to HPC Gateway Service
DEBUG:SERRANO.Orchestrator.OrchestrationDriver.ExecutionWrapper:Move object 'posit
ion_input_data' from bucket 'e90a2382-2008-468e-a0fd-4964c19a7331' to HPC Gateway
Service
DEBUG:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper:Format execution request for
HPC Gateway Service
DEBUG:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper: {'services': ['kmean'], 'in
frastructure': 'excess_slurm', 'params': {'read_input_data': '/Init_Data/raw_data_
position/from_s3_position_input_data'}}
INFO:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper:Submit execution request to H
PC Gateway Service for request_uuid '7b7435dd-c9bd-4afa-b399-1b9e3047184a'
INFO:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper: HPC Gateway Service return j
obid  '0046313f-2df5-4fbc-81ec-b82880b48ff1' for request_uuid '7b7435dd-c9bd-4afa-
b399-1b9e3047184a'
INFO:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper:Checking the execution status
for request_uuid '7b7435dd-c9bd-4afa-b399-1b9e3047184a'
INFO:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper: request_uuid '7b7435dd-c9bd-
4afa-b399-1b9e3047184a' executed successfully
INFO:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper:Move results from HPC Gateway
Service to bucket 'e90a2382-2008-468e-a0fd-4964c19a7331'
DEBUG:SERRANO.Orchestrator.OrchestrationDriver.ExecutionWrapper:Move results 'serr
ano/data/Output_Data/KMean/KMean_cluster.csv' from HPC Gateway Service to object '
results_7b7435dd-c9bd-4afa-b399-1b9e3047184a' in bucket 'e90a2382-2008-468e-a0fd-4
964c19a7331'
```

```
INFO:SERRANO.Orchestrator.DriverHPC.ExecutionWrapper: Forward results from secure
storage service to message broker
```

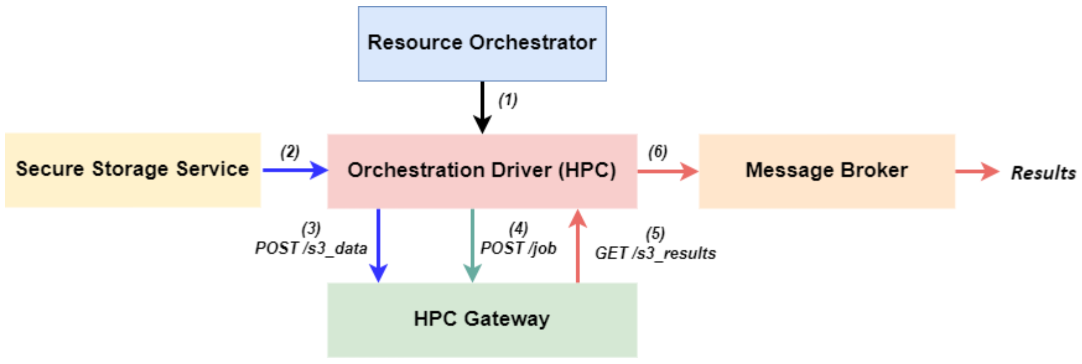


**Figure 18: SERRANO accelerated kernel execution in HPC platform**

## 4.3 Resource Optimization Toolkit

### 4.3.1 Description

The Resource Optimization Toolkit (ROT) integrates the designed resource allocation algorithms in the SERRANO platform, implementing the deciding part at the envisioned closed-loop control. It supports the SERRANO Resource Orchestrator, providing the logic to allocate the edge, cloud, and HPC resources to satisfy the applications' requirements and support the efficient movement of required data across the selected resources. Figure 19 presents the architecture of the ROT, its main components, and the interactions with other components within the SERRANO architecture.

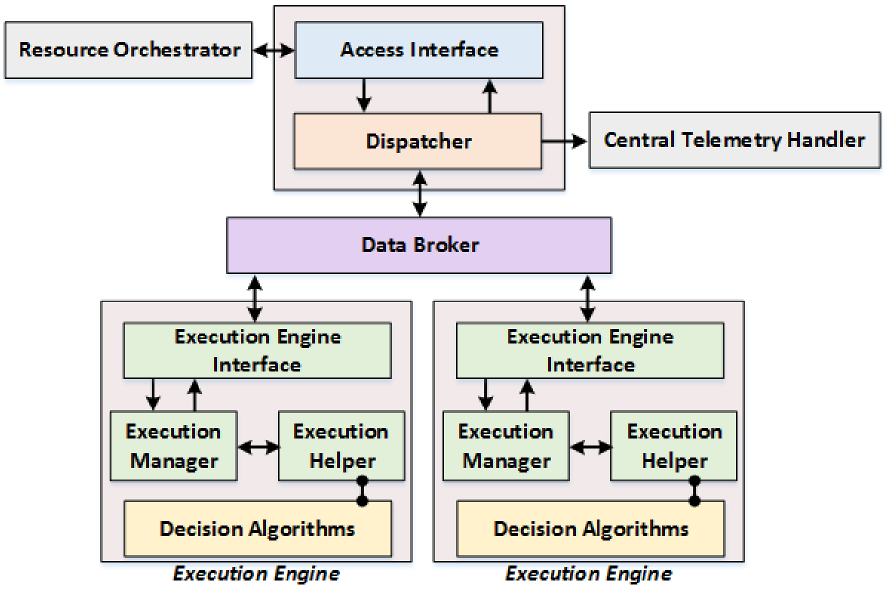


**Figure 19: Resource Optimization Toolkit architecture and main components**

## 4.3.2 Inner components

The design includes one ROT Controller but multiple Execution Engines, the actual workers. The overall design and the implementation provide a robust and adaptable cloud-native application.

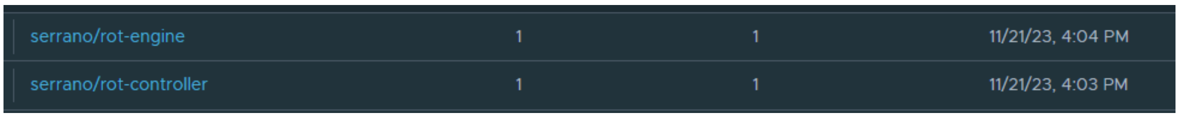
The controller includes the *Access Interface* and *Dispatcher* components. The former exposes the interfaces that allow bidirectional communication for exchanging commands, information, and notifications. The latter manages the execution of the requests and handles the interaction with the multiple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

The Execution Engine, through the *Execution Engine Interface, Execution Manager*, and *Execution Helper,* receives requests from the ROT Controller for starting or terminating algorithm executions and performs all the required actions. In addition, the *Decision Algorithms* is the library of developed orchestration algorithms that include: (i) a simple first-fit allocation algorithm, (ii) the best-fit heuristic for the security-aware deployment, (iii) the greedy resource allocation algorithm (GRAA), and (iv) the heuristic for the distributed storage allocation. Deliverables D5.2 (M15) and D5.4 (M31) provide more details for the overall design and implementation of the ROT framework along with a thorough analysis of the developed algorithms in the context of SERRANO.

## 4.3.3 Integration details and REST APIs

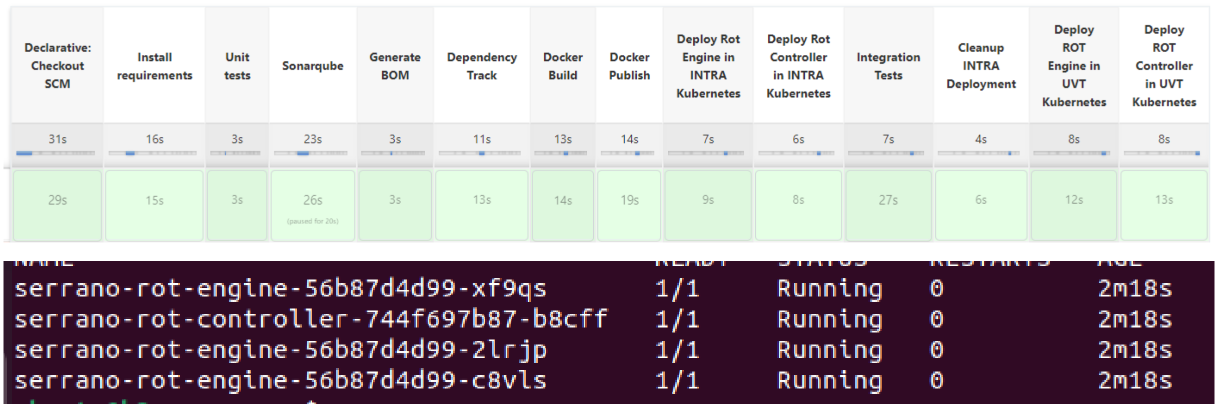
### *4.3.3.1 Integration details*

The ROT is implemented in Python language, using additional frameworks such as Flask 2.0 [4], Pika [6], and PyQt [7], and packaged in container images using the SERRANO CI/CD services, ensuring a smooth and efficient development workflow. There are separate container images for the ROT Controller and ROT Execution Engine (Figure 20), accessible through the official SERRANO Harbor image repository [57]. Moreover, the corresponding Kubernetes YAML description files were created to facilitate effortless deployment on Kubernetes platforms. These files enable the automatic deployment and scaling of the ROT Controller and ROT Execution Engines within Kubernetes. They are also used in SERRANO's Jenkins pipeline, leading to the automatic deployment of the ROT services in the SERRANO testbed infrastructure (Figure 21).



| | | | |
|---|---|---|---|
| serrano/rot-engine | 1 | 1 | 11/21/23, 4:04 PM |
| serrano/rot-controller | 1 | 1 | 11/21/23, 4:03 PM |

**Figure 20: ROT container images into SERRANO Harbor image repository**



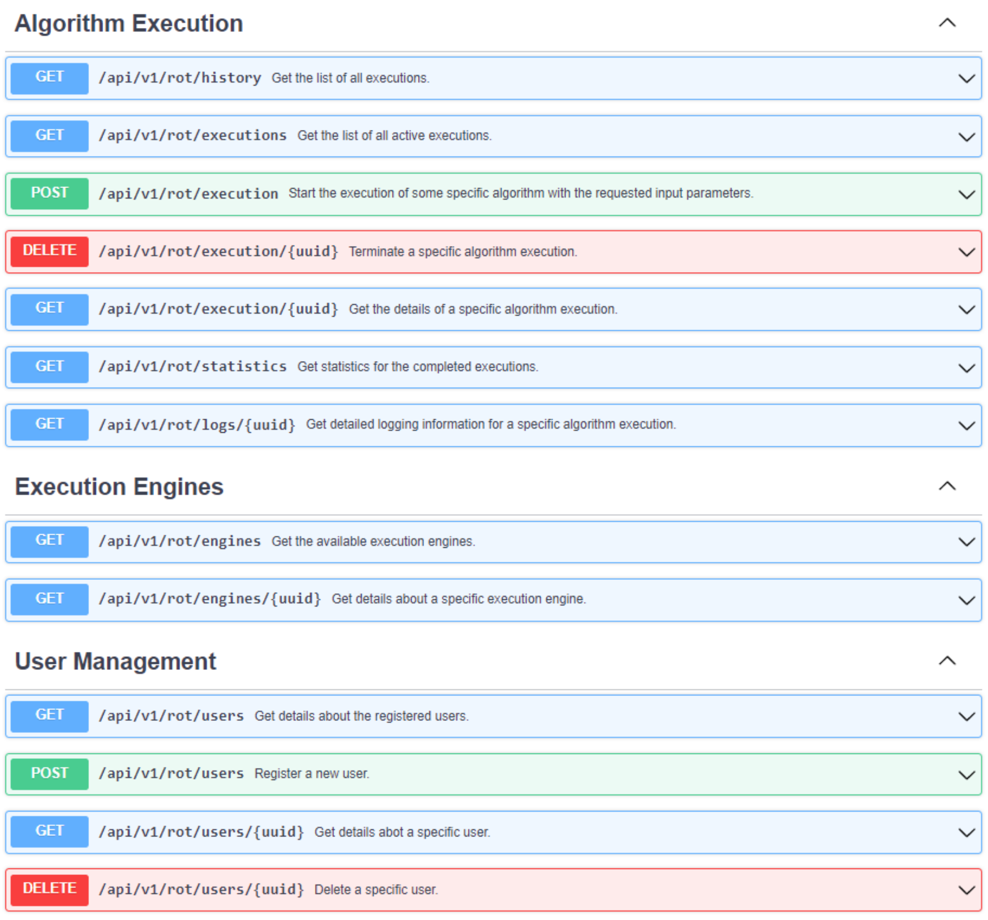
| Declarative: Checkout SCM | Install requirements | Unit tests | Sonarqube | Generate BOM | Dependency Track | Docker Build | Docker Publish | Deploy Rot Engine in INTRA Kubernetes | Deploy Rot Controller in INTRA Kubernetes | Integration Tests | Cleanup INTRA Deployment | Deploy ROT Engine in UVT Kubernetes | Deploy ROT Controller in UVT Kubernetes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31s | 16s | 3s | 23s | 3s | 11s | 13s | 14s | 7s | 6s | 7s | 4s | 8s | 8s |
| 29s | 15s | 3s | 26s (paused for 20s) | 3s | 13s | 14s | 19s | 9s | 8s | 27s | 6s | 12s | 13s |

```
serrano-rot-engine-56b87d4d99-xf9qs       1/1     Running    0          2m18s
serrano-rot-controller-744f697b87-b8cff   1/1     Running    0          2m18s
serrano-rot-engine-56b87d4d99-2lrjp       1/1     Running    0          2m18s
serrano-rot-engine-56b87d4d99-c8vls       1/1     Running    0          2m18s
```

**Figure 21: ROT into Jenkins CI/CD pipeline and deployment in UVT's K8s cluster**

**Table 4: Integration details of Resource Optimization Toolkit**

| | |
|---|---|
| **IP(s)/Port(s)** | ROT Controller:<br>• https://rot.services.cloud.ict-serrano.eu<br>Resource Orchestrator:<br>• https://resource-orchestrator.services.cloud.ict-serrano.eu<br>Central Telemetry Handler:<br>• https://central-telemetry.services.cloud.ict-serrano.eu |
| **Publicly accessible (y/n and other details)** | The service is publicly accessible, but the access has been restricted though token authentication. |
| **Type of API** | REST and asynchronous (AMQP) |
| **Associated host names** | https://rot.services.cloud.ict-serrano.eu |
| **API documentation** | *https://raw.githubusercontent.com/ict-serrano/Resource-Optimization-Toolkit/main/ROT_swagger_api_v1.1.yaml* |
| **Location of integration tests** | *https://raw.githubusercontent.com/ict-serrano/Resource-Optimization-Toolkit/main/Jenkinsfile* |


Algorithm Execution

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

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

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

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

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

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

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

Execution Engines

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

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

User Management

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

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

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

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

**Figure 22: Resource Optimization Toolkit REST API**

The ROT exposes two primary sets of interfaces through which external services or applications can interact to consume the provided capabilities. The first is based on REST APIs, and the second is an asynchronous messaging interface based on the Advanced Message Queuing Protocol (AMQP). The former exposes control operations to manipulate and inspect the execution of deployment algorithms, get information for the available Execution Engines, and manage end users. The latter offers asynchronous communication between the ROT Controller and end users for exchanging notification messages and results. A detailed presentation of these interfaces is available in deliverable D5.4 (M31). Figure 22 summarizes the final version of the exposed REST API.

A Python API is also available that abstracts the integration with the ROT controller and the exposed northbound interfaces. This API is part of the SERRANO SDK and is also used by the Orchestration Manager (Section 4.2.2) to interact with the ROT. The API includes a set of methods to abstract the interaction with the REST interface and various events to handle the low-level operations for interacting with the asynchronous communication over the Message Broker (Section 4.5.1).

### 4.3.3.2 Service functionality

Next, we present the interaction among the ROT components, the Resource Orchestrator, and the Central Telemetry Handler for executing a resource allocation algorithm in the SERRANO platform. More specifically, we consider the creation of a SERRANO secure storage policy.

**Step 1: Resource Orchestrator and ROT Controller**

When the Resource Orchestrator receives the end user request to create a storage policy, it triggers the ROT decision-making process. To this end, the Orchestration Manager service uses the provided Python API to request the execution of the appropriate algorithm with the provided input parameters.

SERRANO Python API:

```
res = self.client.post_execution("StoragePolicy", request_params)
```

```
Parameters:
{"name": "edge-policy","description": "Secure storage policy with only SERRANO edg
e storage locations", "policy_parameters": {"data_size":0, "cost":0, "latency": 1,
"volatility":0, "availability": 1, "lat": 45.7472357, "lng": 21.2316107}}
```

Then, the ROT Controller, through its Access Interface, assigns a unique identifier (*"bbbc8b87-d896-4c5b-bdea-570a191a4f10"*) in the request, validates it, and forwards the request to the Dispatcher.

```
{"execution_id": "bbbc8b87-d896-4c5b-bdea-570a191a4f10", "status": "Accepted"}
```

**Step 2: ROT Controller and Central Telemetry Handler**

The Dispatcher, among other actions, interacts with the Central Telemetry Handler to retrieve the required operational and monitoring data. This operation uses the exposed REST methods by the SERRANO telemetry framework (Section 4.4.3). This example uses the endpoint that provides detailed information about the available cloud and edge storage locations within the SERRANO platform, along with their basic characteristics. Other endpoints, such as *"/api/v1/telemetry/central/infrastructure"* are used to execute the algorithms that support the application and the on-demand kernel deployments.

```
GET /api/v1/telemetry/central/storage_locations
```

```
{"cloud_storage":[{"cloud_provider_jurisdiction":"United States","cloud_provider_n
ame":"Microsoft Azure","cloud_provider_url":"https://azure.microsoft.com/","countr
y":"United States","countrycode":"US","download_errors_in_last_12_months":2,"downl
oad_price":0.0,"id":25,"is_gdpr":false,"lat":37.4315734,"lng":-78.6568942,"locatio
n":"Virginia","rtt_download_1B_ms":[116,116,116,115,115,116,116,122,115,115,116,11
6,115,116,116,116,116,116,115,116,122,116,116,116,115,115,116,115,118,120],"rtt_do
wnload_1MB_ms":[1128,1134,1016,1135,1128,1128,1016,1029,1050,1175,1144,1151,1126,1
058,905,1137,1019,1016,1664,1013,1022,1021,1129,1127,1015,1033,1016,1127,1154,1046
],"rtt_upload_1B_ms":[119,119,118,118,119,121,118,119,117,118,118,118,117,119,118,
120,299,118,119,117,127,119,122,124,119,117,119,117,120,129],"rtt_upload_1MB_ms":[
1282,1488,1424,1283,1333,1300,1415,1335,1281,1294,1304,1291,1286,1333,1277,1301,16
17,1296,1279,1274,1430,1391,1476,1316,1440,1426,1403,1554,1303,1315],"storage_pric
e":21.0,"upload_errors_in_last_12_months":2,"upload_price":0}],
"edge_storage":[{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","id":1,"lat
":45.7472357,"lng":21.2316107,"metrics":[{"minio_node_disk_free_bytes":19901820928
.0,"minio_node_disk_total_bytes":31509614592.0,"minio_node_disk_used_bytes":116077
93664.0}],"minio_node_disk_total_bytes":8333520896.0,"name":"edge-storage-devices-
0","node":"serrano-k8s-worker-02"}]}
```

The response above is indicative since there are many cloud and edge storage locations in practice, and the actual response will be unreadable.

### Step 3: ROT Controller and Execution Engine

After, the controller assigns the execution request to the Execution Engine with the least workload.

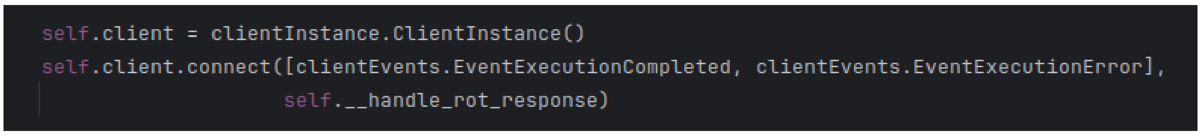
```
DEBUG:SERRANO.ROT.Dispatcher:Execution "bbbc8b87-d896-4c5b-bdea-570a191a4f10" is a
ssigned to engine: "a6db7225-f759-4805-9b04-13b63fd93ac1"
```

The selected Execution Engine rexecutes the selected algorithm with the provided input data. Then, it sends the resource allocation decisions to the controller through the Message Broker.

```
DEBUG:SERRANO.Orchestrator.ROTInterface: ROT response for execution uuid "bbbc8b87
-d896-4c5b-bdea-570a191a4f10"
DEBUG:SERRANO.Orchestrator.ROTInterface: {"kind": "StoragePolicy", "backends": [],
"edge_devices": [1, 2], "redundant_packets": 1}}
```

### Step 4: ROT Controller and Resource Orchestrator

The ROT API provides a simple and effective event-driven mechanism through which applications asynchronously receive the ROT controller responses to their requests. The API provides the appropriate methods (*connect()* method in the following code snippet) that enable applications to register for specific events, providing also the corresponding custom methods that will be executed when each event occurs. The Resource Orchestrator uses this event-handling mechanism while interacting with the ROT Controller. Under the hood, the event handling mechanism leverages the functionalities provided by the SERRANO Message Broker component (Section 4.5.1).



```
self.client = clientInstance.ClientInstance()
self.client.connect([clientEvents.EventExecutionCompleted, clientEvents.EventExecutionError],
                    self.__handle_rot_response)
```

When the ROT Controller receives the execution response, it forwards the results to the appropriate client. Finally, the Resource Orchestrator receives the response from the ROT regarding the selected parameters for creating the requested secure storage policy.

```
INFO:SERRANO.Orchestrator.ROTInterface:Receive ROT response for execution uuid 'bb
bc8b87-d896-4c5b-bdea-570a191a4f10'
DEBUG:SERRANO.Orchestrator.ROTInterface:{"kind": "StoragePolicy", "backends": [],
"edge_devices": [1, 2], "redundant_packets": 1}
```

## 4.4 SERRANO Telemetry Framework

### 4.4.1 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 the redeployment of applications. The SERRANO telemetry stack consists of four key building blocks: (a) the Central Telemetry Handler, (b) Enhanced Telemetry Agents, (c) Monitoring Probes, and (d) Persistent Monitoring Data Storage service.

The Central Telemetry Handler is the root element of the SERRANO hierarchical telemetry infrastructure. The various Enhanced Telemetry Agents are responsible for a specific set of Monitoring Probes. The collection and exchange of monitored information is performed periodically, while the granularity can be adapted, and other telemetry operations can be activated based on detected events or explicitly by entities at upper layers. The Persistent Monitoring Data Storage (PMDS) service provides long-term storage for the collected telemetry data. It operates as a central repository that provides historical data to the SERRANO orchestration (Sections 4.1 and 4.3) and service assurance mechanisms (Section 4.9). The telemetry functionalities are spread into several layers to meet the scalability requirement while enabling immediate reaction to events that affect the performance of the deployed applications at individual parts within the SERRANO platform.

### 4.4.2 Telemetry framework components

The Central Telemetry Handler and Enhanced Telemetry Agents provide the same core functions at different scales and views of the infrastructure resources and deployed applications. They expose RESTful methods that enable the on-demand change of their current operational configuration. The Central Telemetry Handler manages multiple instances of Enhance Telemetry Agents, while each agent controls a specific set of Monitoring Probes. SERRANO's hierarchical monitoring infrastructure is depicted in Figure 23.

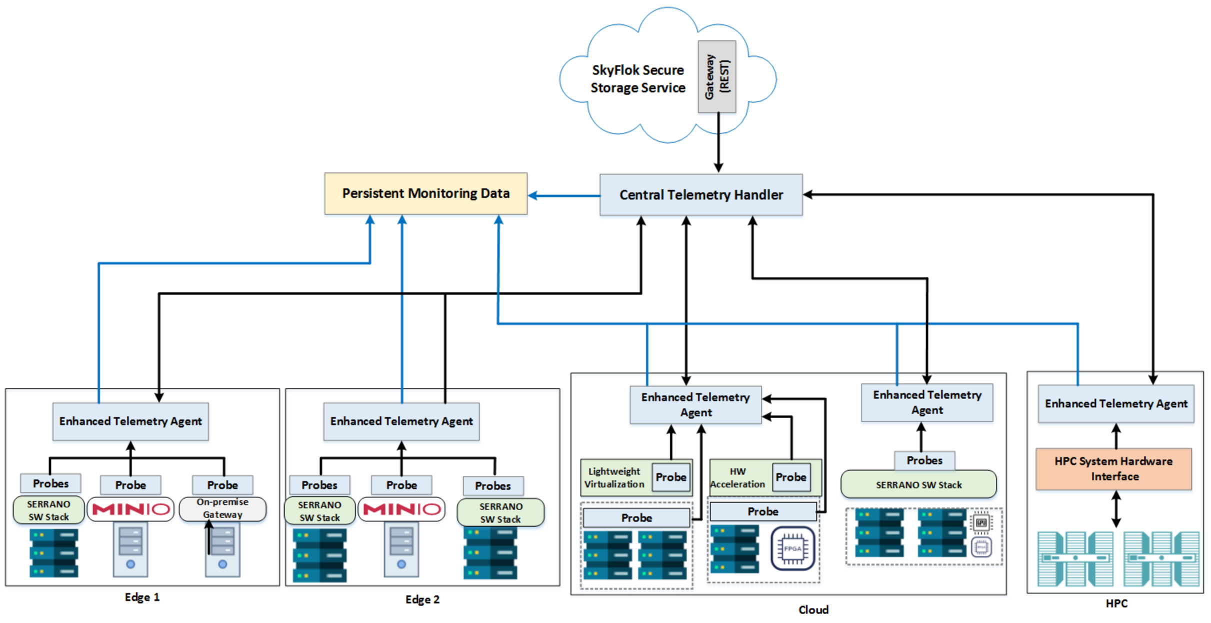


**Figure 23: SERRANO hierarchical telemetry architecture**

Monitoring probes collect valuable information about the infrastructure resources, services, and deployed applications within the SERRANO platform. SERRANO provides a collection of specialized probes, each dedicated to monitoring a specific infrastructure type. More specifically, three different monitoring probes are available: Kubernetes Monitoring Probe, HPC Monitoring Probe, and SERRANO Edge Devices Monitoring Probe. Finally, the SERRANO telemetry framework includes the PMDS service for storing the collected timestamped telemetry data from the various monitoring probes. It is based on InfluxDB [62], an open-source time-series database. The PMDS exposes methods (Figure 27) that allow end users and external services to retrieve historical telemetry data.

The SERRANO telemetry framework also includes several operational databases that store information related to the deployed components of the framework and their configurations, along with the most up-to-date information for the infrastructure resources, deployed applications, and executed SERRANO-accelerated kernels. The databases are based on MongoDB [61]. Deliverables D5.2 (M15) and D5.4 (M31) provide more details for the overall design and implementation of the Resource Orchestrator and Orchestration Drivers.

## 4.4.3 Integration details and REST APIs

### *4.4.3.1 Integration details*

The SERRANO telemetry framework has been implemented in Python. The framework's services and their corresponding configuration files have been packaged as Python applications and integrated using the SERRANO CI/CD pipeline. Different container images are available for the key components such as the Central Telemetry Handler, Enhanced Telemetry Agent, PMDS, and Monitoring Probes. These components have been seamlessly integrated into SERRANO's Jenkins pipeline, facilitating their automatic deployment within the SERRANO testbed infrastructure. The integration tests of the telemetry framework and the evaluation of the final SERRANO platform release were performed using the setup that is depicted in Figure 24.

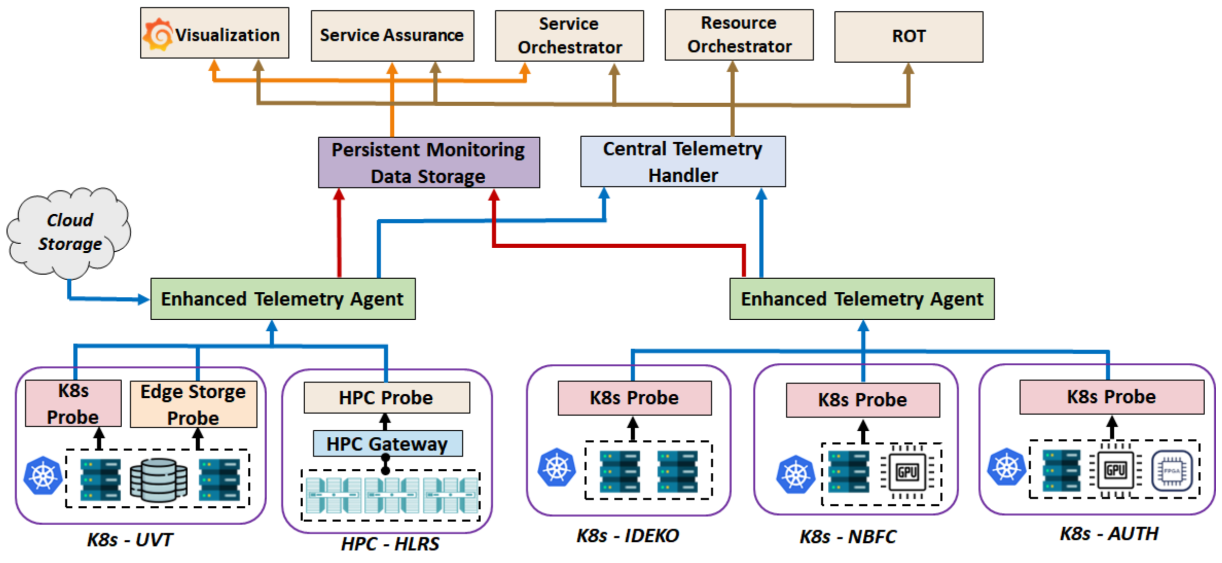
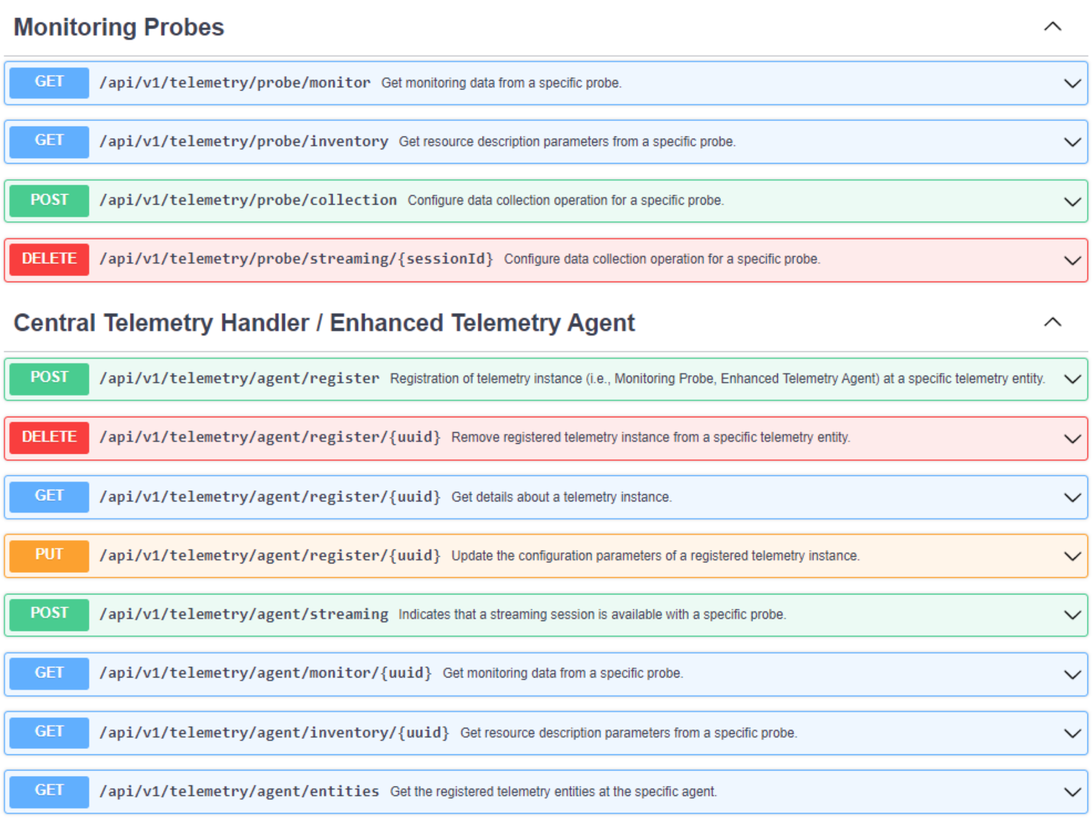


Figure 24: Setup for the final integration tests of SERRANO telemetry framework

Table 5: Integration details of Resource Optimization Toolkit

| | |
|---|---|
| **IP(s)/Port(s)** | Central Telemetry Handler:<br>• https://central-telemetry.services.cloud.ict-serrano.eu<br>Enhanced Telemetry Agents:<br>• https://telemetry-agent.services.cloud.ict-serrano.eu<br>• http://85.120.206.26:30090<br>PMDS:<br>• https://pmds.services.cloud.ict-serrano.eu<br>Grafana:<br>• http://85.120.206.26:32000<br>Monitoring Probes:<br>• Cloud storage: https://on-premise-storage-gateway.services.cloud.ict-serrano.eu<br>• Edge storage: https://edge-storage-probe.services.cloud.ict-serrano.eu<br>• UVT K8s: https://k8s-probe.wp5.services.cloud.ict-serrano.eu<br>• NBFC K8s: https://serrano.nbfc.io:9080<br>• IDEKO K8s: https://extranet.danobatgroup.com/serranok8sprobe<br>• AUTH K8s: https://155.207.169.212:9080<br>• HPC: https://hpc-interface.services.cloud.ict-serrano.eu |
| **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** | https://central-telemetry.services.cloud.ict-serrano.eu |
| **API documentation** | *https://raw.githubusercontent.com/ict-serrano/Telemetry-Framework/master/telemetry_rest.yaml* |
| **Location of integration tests** | *https://raw.githubusercontent.com/ict-serrano/Telemetry-Framework/master/Jenkinsfile* |

The following figures summarize the final version of the REST API exposed by the SERRANO telemetry framework services. The API related to CTH, ETA, and Monitoring Probes functionality includes methods organized into two main categories. The first category allows other SERRANO services (such as the AI-Enhanced Service Orchestrator, Event Detection Engine, Resource Optimization Toolkit, and Resource Orchestrator) to interact with the telemetry framework. The second category includes methods that support the operation of telemetry framework services and data exchange among its components.



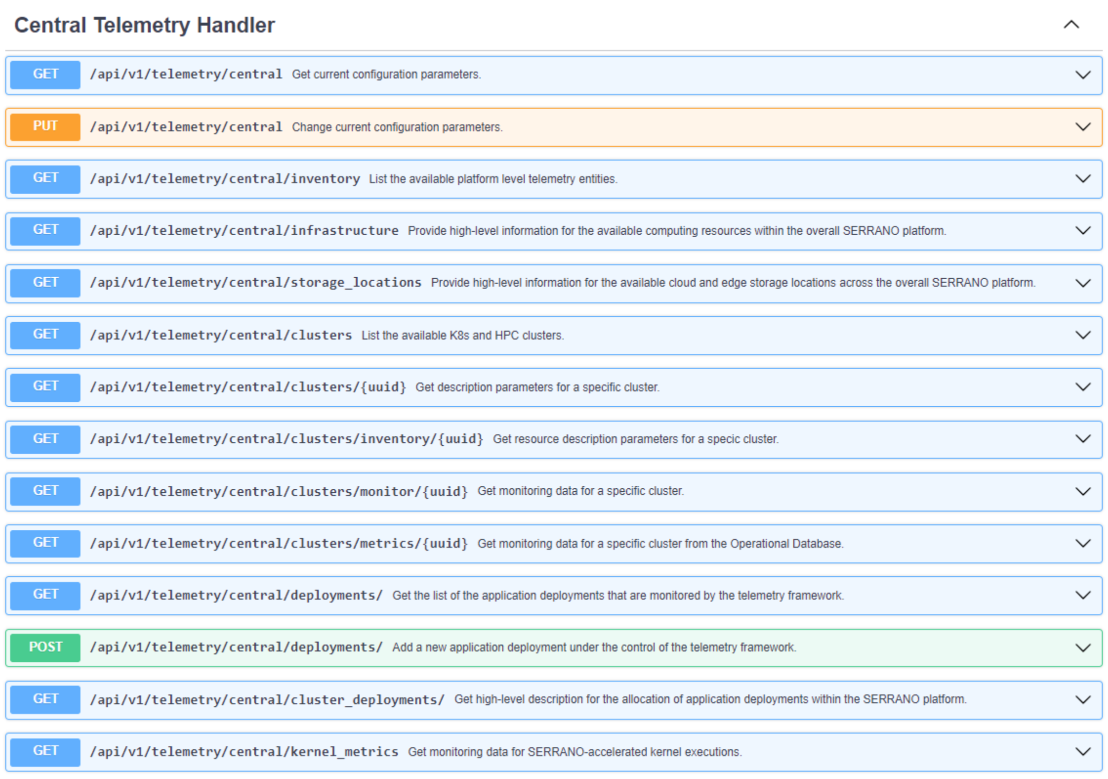
Monitoring Probes

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

Central Telemetry Handler / Enhanced Telemetry Agent

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

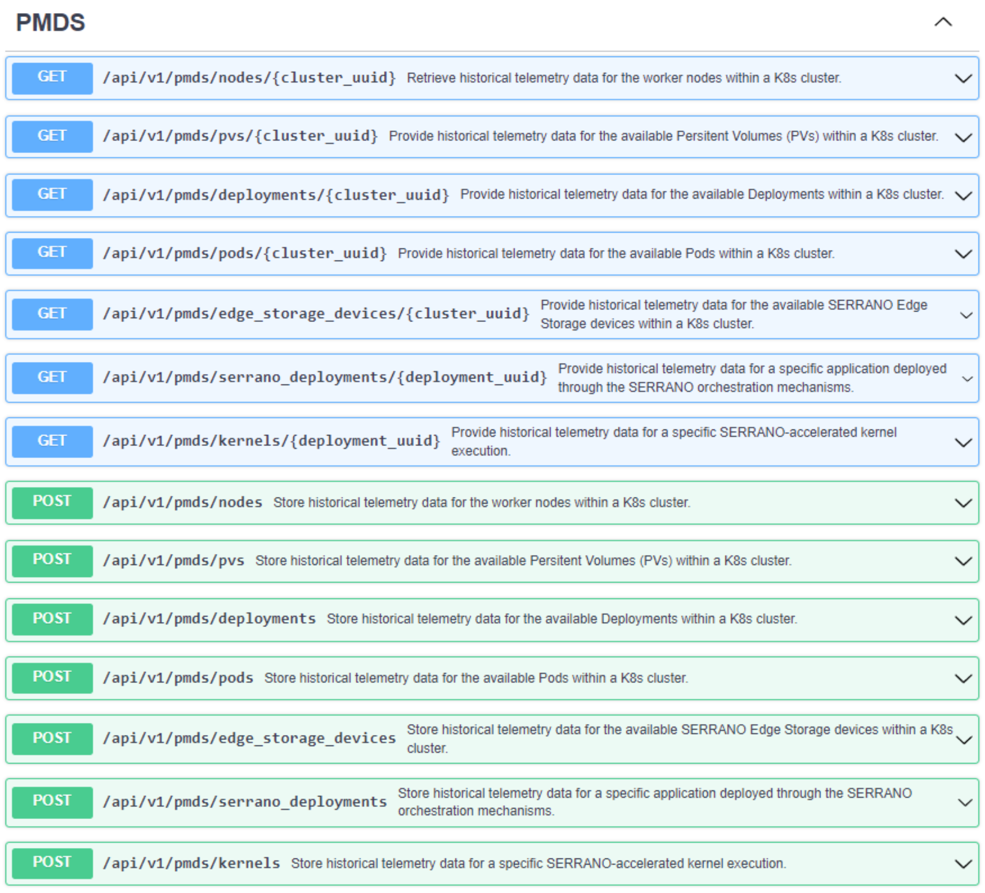
**Figure 25: Telemetry framework REST interfaces – Control and management methods**



Central Telemetry Handler

GET /api/v1/telemetry/central Get current configuration parameters.
PUT /api/v1/telemetry/central Change current configuration parameters.
GET /api/v1/telemetry/central/inventory List the available platform level telemetry entities.
GET /api/v1/telemetry/central/infrastructure Provide high-level information for the available computing resources within the overall SERRANO platform.
GET /api/v1/telemetry/central/storage_locations Provide high-level information for the available cloud and edge storage locations across the overall SERRANO platform.
GET /api/v1/telemetry/central/clusters List the available K8s and HPC clusters.
GET /api/v1/telemetry/central/clusters/{uuid} Get description parameters for a specific cluster.
GET /api/v1/telemetry/central/clusters/inventory/{uuid} Get resource description parameters for a specic cluster.
GET /api/v1/telemetry/central/clusters/monitor/{uuid} Get monitoring data for a specific cluster.
GET /api/v1/telemetry/central/clusters/metrics/{uuid} Get monitoring data for a specific cluster from the Operational Database.
GET /api/v1/telemetry/central/deployments/ Get the list of the application deployments that are monitored by the telemetry framework.
POST /api/v1/telemetry/central/deployments/ Add a new application deployment under the control of the telemetry framework.
GET /api/v1/telemetry/central/cluster_deployments/ Get high-level description for the allocation of application deployments within the SERRANO platform.
GET /api/v1/telemetry/central/kernel_metrics Get monitoring data for SERRANO-accelerated kernel executions.

**Figure 26: Telemetry framework REST interfaces – High-level CTH methods**



PMDS

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

**Figure 27: Persistent Monitoring Data Storage (PMDS) RESTful interface**

### 4.4.3.2 Integration with SERRANO services

**Enhanced Telemetry Agent and Monitoring Probes**

Deliverables D6.3 (M18) and D5.4 (M31) provide a detailed presentation for the integration of the three developed SERRANO Monitoring Probes, along with their registration and overall management from the Enhanced Telemetry Agent. It follows an example for the monitoring data that the Enhanced Telemetry Agent collects from the final version of the probe that monitors the SERRANO edge storage devices that are deployed in the UVT Kubernetes cluster.

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

```
{"edge_storage_devices":[{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","m
inio_bucket_usage_object_total":199.0,"minio_bucket_usage_total_bytes":3186080021.
0,"minio_node_disk_free_bytes":18935615488.0,"minio_node_disk_total_bytes":3150961
4592.0,"minio_node_disk_used_bytes":12573999104.0,"minio_s3_requests_total":0,"min
io_s3_requests_waiting_total":0.0,"minio_s3_traffic_received_bytes":3436003534.0,"
minio_s3_traffic_sent_bytes":16726678916.0,"name":"edge-storage-devices-0","node":
"serrano-k8s-worker-02"},{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","m
inio_bucket_usage_object_total":183.0,"minio_bucket_usage_total_bytes":3092835141.
0,"minio_node_disk_free_bytes":18935615488.0,"minio_node_disk_total_bytes":3150961
4592.0,"minio_node_disk_used_bytes":12573999104.0,"minio_s3_requests_total":0,"min
io_s3_requests_waiting_total":0.0,"minio_s3_traffic_received_bytes":3092835377.0,"
minio_s3_traffic_sent_bytes":932.0,"name":"edge-storage-devices-1","node":"serrano
-k8s-worker-02"},{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","minio_buc
ket_usage_object_total":199.0,"minio_bucket_usage_total_bytes":3186080021.0,"minio
_node_disk_free_bytes":18935615488.0,"minio_node_disk_total_bytes":31509614592.0,"
minio_node_disk_used_bytes":12573999104.0,"minio_node_process_cpu_total_seconds":6
017.01,"minio_node_process_resident_memory_bytes":126976000.0,"minio_node_process_
uptime_seconds":3113825.842905024,"minio_s3_requests_errors_total":0,"minio_s3_req
uests_rejected_invalid_total":0,"minio_s3_requests_total":0,"minio_s3_requests_wai
ting_total":0.0,"minio_s3_traffic_received_bytes":3436003467.0,"minio_s3_traffic_s
ent_bytes":16726678782.0,"name":"edge-storage-devices-2","node":"serrano-k8s-worke
r-02"},{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","minio_bucket_usage_
object_total":183.0,"minio_bucket_usage_total_bytes":3092835141.0,"minio_node_disk
_free_bytes":18935615488.0,"minio_node_disk_total_bytes":31509614592.0,"minio_node
_disk_used_bytes":12573999104.0,"minio_s3_requests_total":0,"minio_s3_requests_wai
ting_total":0.0,"minio_s3_traffic_received_bytes":3092835377.0,"minio_s3_traffic_s
ent_bytes":16726678916.0,"name":"edge-storage-devices-3","node":"serrano-k8s-worke
r-02"}],"type":"Probe.EdgeStorage","uuid":"4c1f3be3-9d47-4e5a-9199-cd4a01d6775a"}
```

**Configuration of Enhanced Telemetry Agent and monitoring probes**

The framework supports the dynamic configuration of the monitoring probes and Enhanced Telemetry Agents through their exposed REST methods. An Enhanced Telemetry Agent is able to configure the operation of its monitoring probes, while it can also be configured from the Central Telemetry Handler. In the following example, the Central Telemetry Handler changes the configuration of one of the two Enhanced Telemetry Agents and sets the data retaining period in its operational database to three minutes and disables the emit of notifications related to its operation through the SERRANO Stream Handler.

```
POST /api/v1/telemetry/agent
```

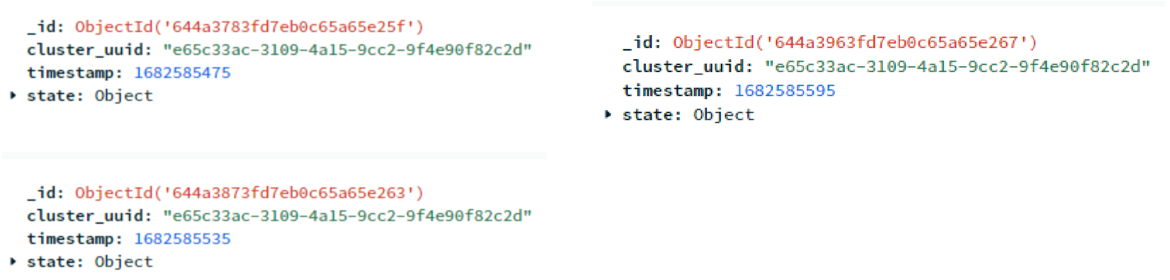
```
{"data_retain_period": 180, "active_notifications": false}
```

Similarly, the Enhanced Telemetry Agent can dynamically change the rate at which a monitoring probe forwards its collected telemetry data. The following example sets the reporting period to 60 seconds in the probe that collects information from the IDEKO K8s cluster.

```
POST /api/v1/telemetry/probe
```

```
{"query_interval": 60}
```

The figure bellow shows the contents of the operational database after the previous two configurations. As expected, it contains only the last three monitoring samples for the K8s cluster (UUID "*e65c33ac-3109-4a15-9cc2-9f4e90f82c2d*") that monitors the specific probe.



```
_id: ObjectId('644a3783fd7eb0c65a65e25f')
cluster_uuid: "e65c33ac-3109-4a15-9cc2-9f4e90f82c2d"
timestamp: 1682585475
▸ state: Object
```

```
_id: ObjectId('644a3963fd7eb0c65a65e267')
cluster_uuid: "e65c33ac-3109-4a15-9cc2-9f4e90f82c2d"
timestamp: 1682585595
▸ state: Object
```

```
_id: ObjectId('644a3873fd7eb0c65a65e263')
cluster_uuid: "e65c33ac-3109-4a15-9cc2-9f4e90f82c2d"
timestamp: 1682585535
▸ state: Object
```

## <u>Central Telemetry Handler and SERRANO platform services</u>

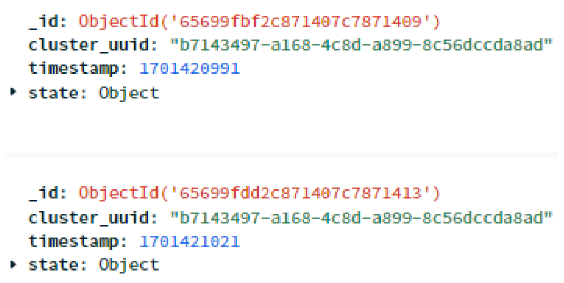
The Central Telemetry Handler exposes methods that facilitate the interaction of other core SERRANO services with the telemetry services. The following GET method lists the cloud-native applications that are deployed within the SERRANO platform.

```
GET /api/v1/telemetry/central/deployments
```

```
{"deployments":[{"clusters":["7628b895-3a91-4f0c-b0b7-033eab309891"],"deployment_u
uid":"649decae-63ec-40cb-9c5f-eb16f5b93590","timestamp":1695371235},{"clusters":["
7628b895-3a91-4f0c-b0b7-033eab309891"],"deployment_uuid":"d057ad52-8a6d-49b8-9dbc-
b2570e93c079","timestamp":1700827414},{"clusters":["7628b895-3a91-4f0c-b0b7-033eab
309891"],"deployment_uuid":"395fa908-08d5-4317-8805-feecbc93d7c7","timestamp":1700
829240},{"clusters":["7628b895-3a91-4f0c-b0b7-033eab309891","5a075716-7d7d-4b40-95
66-bc1a33ee70c2"],"deployment_uuid":"78ba07f8-713d-42ba-bab2-1725a9402f1d","timest
amp":1700838788}]}
```

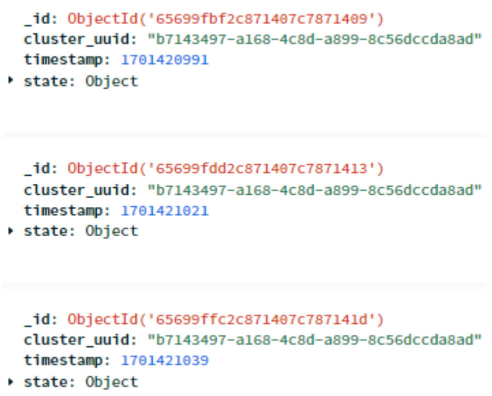
Section 4.2.3.2 includes additional CTH methods used by other services, such as the ROT and Resource Orchestrator. There is also a method for retrieving monitoring data for a specific edge, cloud, or HPC infrastructure. It provides the most up-to-date data by querying the appropriate probe. It also automatically updates the operational database and PMDS service. Below is an example of getting the monitoring data from the HPC cluster, while the figure shows the respective contents of the operational database before and after the request. The previous entries were every 60 seconds but the last one is from the request.

```
GET /api/v1/telemetry/central/cluster/monitor/b7143497-a168-4c8d-a899-8c56dccda8ad
```



```
_id: ObjectId('65699fbf2c871407c7871409')
cluster_uuid: "b7143497-a168-4c8d-a899-8c56dccda8ad"
timestamp: 1701420991
‣ state: Object

_id: ObjectId('65699fdd2c871407c7871413')
cluster_uuid: "b7143497-a168-4c8d-a899-8c56dccda8ad"
timestamp: 1701421021
‣ state: Object
```



```
_id: ObjectId('65699fbf2c871407c7871409')
cluster_uuid: "b7143497-a168-4c8d-a899-8c56dccda8ad"
timestamp: 1701420991
‣ state: Object

_id: ObjectId('65699fdd2c871407c7871413')
cluster_uuid: "b7143497-a168-4c8d-a899-8c56dccda8ad"
timestamp: 1701421021
‣ state: Object

_id: ObjectId('65699ffc2c871407c787141d')
cluster_uuid: "b7143497-a168-4c8d-a899-8c56dccda8ad"
timestamp: 1701421039
‣ state: Object
```

**Automatic monitoring of deployed applications**

The telemetry framework provides the autonomous collection of monitoring metrics for the performance of the deployed applications across the SERRANO heterogeneous and distributed resources, regardless of the individual platforms that host them. Next, we present this functionality using the application deployment from Section 4.2.3.2.3 as an example.

During the final steps of the deployment phase, each Orchestration Driver updates the Orchestration API server for the status of its Assignment object. The Orchestration API server uses this information to determine if a deployment request has been served successfully. Then, the Orchestration API server informs the SERRANO telemetry framework to monitor the deployed application automatically and register the deployed application to the Service Assurance mechanisms (Section 4.9).

```
DEBUG:SERRANO.Orchestrator.Dispatcher:Update deployment "fdf45855-1299-47f1-8ea6-9
8be8d89030b" status after assignment "4aae522e-3d56-42a2-b5f7-56d1cc4bef2b" progre
ss update ..
DEBUG:SERRANO.Orchestrator.Dispatcher:Update deployment "fdf45855-1299-47f1-8ea6-9
8be8d89030b" status after assignment "4ea8f80a-8354-433f-8100-5ed8b469f54c " progr
ess update ..
INFO:SERRANO.Orchestrator.Dispatcher:All assignments were successful
DEBUG:SERRANO.Orchestrator.Dispatcher:Now enable monitoring and service assurance
for Deployment "fdf45855-1299-47f1-8ea6-98be8d89030b"
DEBUG:urllib3.connectionpool:http://85.120.206.26:30070 "POST /api/v1/telemetry/ce
ntral/deployments HTTP/1.1" 201 3
```

The Orchestration API server notifies, through the CTH, the telemetry framework to monitor the deployed application. For the previous example, it makes the following POST request.

```
POST /api/v1/telemetry/central/deployments
```

```
{"deployment_uuid": "fdf45855-1299-47f1-8ea6-98be8d89030b","clusters":["e65c33ac-3
109-4a15-9cc2-9f4e90f82c2d","5a075716-7d7d-4b40-9566-bc1a33ee70c2"],"7628b895-3a91
-4f0c-b0b7-033eab309891":[{"k8s_deployment_name":"acceleration-service-data-manage
r", "k8s_deployment_namespace":"integration","k8s_deployment_uuid":"06802ae3-9e00-
41ae-a7a0-15ffe2505db0"},{"k8s_deployment_name":"acceleration-service-classifier-t
raining", "k8s_deployment_namespace":"integration","k8s_deployment_uuid":"97271b19
-36c0-4458-b534-4736bae470b2"}],"5a075716-7d7d-4b40-9566-bc1a33ee70c2":[{"k8s_depl
oyment_name": "acceleration-service-model-inference", "k8s_deployment_namespace":
"integration", "k8s_deployment_uuid": "e10e05a4-dc11-4cef-bb09-21a9d90c6a5b"}]}
```

Moreover, the Orchestration API server notifies the Event Detection Engine (EDE) component of the Service Assurance service to monitor the performance of the newly deployed application. To this end, it makes the following PUT request.

```
PUT /api/v1/config/connector
```

```
{
  "Deployment_id": "fdf45855-1299-47f1-8ea6-98be8d89030b",
  "Groups": ["general", "cpu", "memory", "network", "storage"]
}
```

After the successful execution of the previous configurations the collected telemetry data are automatically stored in the corresponding operational database and PMDS service. Finally, the information is available from the CTH through the following request.

```
GET /api/v1/telemetry/central/deployments/fdf45855-1299-47f1-8ea6-98be8d89030b
```

```
{"metrics":[{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","creation_times
tamp":1700838818,"deployment_uuid":"fdf45855-1299-47f1-8ea6-98be8d89030b","group_i
d":"s1","host_ip":"192.168.12.110","name":"acceleration-service-data-manager-475b4
c6b85-k89b5","namespace":"integration","node":"serrano3","phase":"Running","pod_ip
":"192.168.213.61","restarts":0,"start_time":1700838818,"timestamp":1700838850,"us
age":{"cpu":"1002310835n","memory":"150672Ki"}},{"cluster_uuid":"5a075716-7d7d-4b4
0-9566-bc1a33ee70c2","creation_timestamp":1700838818,"deployment_uuid":"fdf45855-1
299-47f1-8ea6-98be8d89030b","group_id":"s2","host_ip":"192.168.8.117","name":"acce
leration-service-model-inference-5b4c6b4785-g7jb5","namespace":"integration","node
":"bf","phase":"Running","pod_ip":"192.168.231.40","restarts":0,"start_time":17008
38818,"timestamp":1700838850,"usage":{"cpu":"1006083037n","memory":"175124Ki"}},{"
cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","creation_timestamp":17008388
18,"deployment_uuid":"fdf45855-1299-47f1-8ea6-98be8d89030b","group_id":"s3","host_
ip":"192.168.12.110","name":"acceleration-service-classifier-training-21a9d90c6a-5
b4c6","namespace":"integration","node":"serrano3","phase":"Running","pod_ip":"192.
168.231.118","restarts":0,"start_time":1700838818,"timestamp":1700838850,"usage":{
"cpu":"1004769336n","memory":"204008Ki"}}]}
```

**Automatic monitoring of SERRANO accelerated kernels**

The telemetry mechanisms also dynamically collect performance metrics for the on-demand execution of the SERRANO-accelerated kernels. The telemetry data are collected from the Orchestration Drivers based on the provided detailed performance metrics from the vAccel (Section 4.7) and SERRANO HPC Gateway (Section 4.6). An Orchestration Driver forwards the collected data to the Enhanced Telemetry Agent that manages the corresponding part of the infrastructure. Next is an example of collected metrics after executing the SERRANO hardware-accelerated version of the KNN kernel.

```
INFO:SERRANO.Orchestrator.DriverKubernetes:Handler deployment request ...
INFO:SERRANO.Orchestrator.DriverKubernetes:Bundle for Faas kernel assignment '765c
0781-e78e-448c-b1f4-707e1b224135' is activated
INFO:SERRANO.Orchestrator.OrchestrationDriver.ExecutionWrapper:Submit execution re
quest to OpenFaaS service for request_uuid '35923a24-8ee1-4fdd-ad12-82ce97b9fcbf'
DEBUG:SERRANO.Orchestrator.DriverKubernetes:__post_metric_log_data
DEBUG:SERRANO.Orchestrator.DriverKubernetes:[{"uuid":"35923a24-8ee1-4fdd-ad12-82ce
97b9fcbf","kind":"KernelMetrics","deployment_mode":"FaaS","cluster_uuid":"5a075716
-7d7d-4b40-9566-bc1a33ee70c2","kernel_name":"knn","input_total_size_MB":66.15,"dep
loyed_at":1701423250,"completed_at":1701423285,"kernel_mode":"gpu","metrics":{"loa
d_vaccel_libs_ms":1074,"load_model_libs_ms":21999,"read_input_from_backend_ms":322
9,"parse_model_ms": 4134,"parse_input_ms":37,"setup_vaccel_args_ms":1,"run_kernel_
ms":1237,"push_output_to_backend_ms":323,"total_ms":33387}, "status": 1}]
```

We can get the monitoring data for the on-demand executions of a SERRANO-accelerated kernel using the following request:

```
GET /api/v1/telemetry/central/kernels_metrics?kernel_name=knn
```

```
{"metrics":[{"cluster_uuid":"3984f92a-21a0-4ce5-85a4-7febd261b794","completed_at":
1700469441,"deployed_at":1700469431,"deployment_mode":"FaaS","input_total_size_MB"
:66.15,"kernel_mode":"gpu","kernel_name":"knn","kind":"KernelMetrics","metrics":{"
load_model_libs_ms":1330,"load_vaccel_libs_ms":53,"output_ms":0,"parse_input_ms":6
,"parse_model_ms":1457,"push_output_to_backend_ms":117,"read_input_from_backend_ms
":6390,"run_kernel_ms":226,"setup_vaccel_args_ms":0,"total_ms":9678},"uuid":"5133c
bbb-f4ab-4e7f-8947-a180e0bbc373"},{"cluster_uuid":"5a075716-7d7d-4b40-9566-bc1a33e
e70c2","completed_at":1701177150,"deployed_at":1701177137,"deployment_mode":"FaaS"
,"input_total_size_MB":66.15,"kernel_mode":"gpu","kernel_name":"knn","kind":"Kerne
lMetrics","metrics":{"load_model_libs_ms":3960,"load_vaccel_libs_ms":221,"output_m
s":0,"parse_input_ms":22,"parse_model_ms":3734,"push_output_to_backend_ms":111,"re
ad_input_from_backend_ms":3357,"run_kernel_ms":509,"setup_vaccel_args_ms":1,"total
_ms":12226},"status":1,"uuid":"f581bf77-53e2-4012-98d0-ad56be3a2be1"},{"cluster_uu
id":"5a075716-7d7d-4b40-9566-bc1a33ee70c2","completed_at":1701423285,"deployed_at"
:1701423250,"deployment_mode":"FaaS","input_total_size_MB":66.15,"kernel_mode":"gp
u","kernel_name":"knn","kind":"KernelMetrics","metrics":{"load_model_libs_ms":2199
9,"load_vaccel_libs_ms":1074,"output_ms":0,"parse_input_ms":37,"parse_model_ms":41
34,"push_output_to_backend_ms":323,"read_input_from_backend_ms":3229,"run_kernel_m
s":1237,"setup_vaccel_args_ms":1,"total_ms":33387},"status":1,"uuid":"35923a24-8ee
1-4fdd-ad12-82ce97b9fcbf"}]}
```

**Interaction with the Persistent Monitoring Data Storage service**

Apart from the exposed REST methods from the PMDS, there is also available a Python API that facilitates the interaction with the services by offering abstractions and a range of filtering parameters. The following example queries the service to retrieve historical data about the memory usage the last 2 minutes for a specific worker node (*"nuc5"*) within the SERRANO NBFC K8s cluster.

```
NBFC = “5a075716-7d7d-4b40-9566-bc1a33ee70c2”
pmds_service_query_nodes(NBFC, group="memory", node_name="nuc5", start="-2m")
```

```
{'nuc5': [{'node_memory_MemAvailable_bytes': 59249152000.0, 'node_memory_MemFree_b
ytes': 18508849152.0, 'node_memory_MemTotal_bytes': 67436380160.0, 'node_memory_Me
mUsed_bytes': 48927531008.0, 'node_memory_usage_percentage': 72.55, 'time': '2023-
11-30T14:50:04.520099+00:00'}, {'node_memory_MemAvailable_bytes': 59316523008.0, '
node_memory_MemFree_bytes': 18575515648.0, 'node_memory_MemTotal_bytes': 674363801
60.0, 'node_memory_MemUsed_bytes': 48860864512.0, 'time': '2023-11-30T14:50:34.567
482+00:00'}, {'node_memory_MemAvailable_bytes': 59296083968.0, 'node_memory_MemFre
e_bytes': 18554392576.0, 'node_memory_MemTotal_bytes': 67436380160.0, 'node_memory
_MemUsed_bytes': 48881987584.0, 'node_memory_usage_percentage': 72.49, 'time': '20
23-11-30T14:51:04.348757+00:00'}, {'node_memory_MemAvailable_bytes': 59292823552.0
, 'node_memory_MemFree_bytes': 18550431744.0, 'node_memory_MemTotal_bytes': 674363
80160.0, 'node_memory_MemUsed_bytes': 48885948416.0, 'node_memory_usage_percentage
': 72.49, 'time': '2023-11-30T14:51:34.669745+00:00'}]}
```

Moreover, the next example retrieves historical telemetry data for the application deployment with UUID “*649decae-63ec-40cb-9c5f-eb16f5b93590*” for the last 48 hours. Note that we only present part of the provided data by the PMDS service.

```
pmds_service_query_serrano_deployments("649decae-63ec-40cb-9c5f-eb16f5b93590", for
mat="compact", start="-2d")
```

```
{'position-service-classifier-training-699448bc9d-6gv5d': [{'cpu_usage': '188567n'
, 'group_id': 's3', 'memory_usage': '257776Ki', 'phase': 'Running', 'restarts': 1,
'time': '2023-11-30T14:42:34.149451+00:00'}, {'cpu_usage': '163545n', 'group_id':
's3', 'memory_usage': '257776Ki', 'phase': 'Running', 'restarts': 1, 'time': '2023
-11-30T14:43:04.101261+00:00'}, {'cpu_usage': '206514n', 'group_id': 's3', 'memory
_usage': '257776Ki', 'phase': 'Running', 'restarts': 1, 'time': '2023-11-30T14:43:
34.078419+00:00'}], 'position-service-data-manager-b87ccb677-t89bp': [{'cpu_usage'
: '1006358704n', 'group_id': 's1', 'memory_usage': '207200Ki', 'phase': 'Running',
'restarts': 0, 'time': '2023-11-30T14:42:34.154989+00:00'}, {'cpu_usage': '1010958
535n', 'group_id': 's1', 'memory_usage': '218944Ki', 'phase': 'Running', 'restarts
': 0, 'time': '2023-11-30T14:43:04.106991+00:00'}, {'cpu_usage': '1004959533n', 'g
roup_id': 's1', 'memory_usage': '207196Ki', 'phase': 'Running', 'restarts': 0, 'ti
me': '2023-11-30T14:43:34.084241+00:00'}], 'position-service-model-inference-84ddd
49947-66ztp': [{'cpu_usage': '1030650398n', 'group_id': 's2', 'memory_usage': '498
816Ki', 'phase': 'Running', 'restarts': 0, 'time': '2023-11-30T14:42:34.160566+00:
00'}, {'cpu_usage': '1880507280n', 'group_id': 's2', 'memory_usage': '621636Ki', '
phase': 'Running', 'restarts': 0, 'time': '2023-11-30T14:43:34.089212+00:00'}]}
```

**Streaming telemetry**

The SERRANO telemetry framework supports the collection of measurements based on the streaming telemetry approach, where continuous measurements are sent at a rate much shorter than the typical monitoring approach. The operation is activated from the Central Telemetry Handler (CTH) or Enhanced Telemetry Agents (ETA) and is implemented based on gRPC [64], which also supports streaming RPCs.

In the following example, the framework has been notified by the Event Detection Engine (EDE) component of the Service Assurance service for anomalous performance in a worker node within the UVT K8s cluster. Hence, the CTH instructed the corresponding ETA to start the streaming telemetry operation with the appropriate monitoring probe. The ETA initiates a streaming session by requesting detailed telemetry data every 5 seconds regarding the CPU and memory performance at the worker node of interest.

```
POST /api/v1/telemetry/agent/streaming
```

```
{"cluster_uuid": "7628b895-3a91-4f0c-b0b7-033eab309891", "action": "start", "repor
ting_rate": 5, "nodes": ["serrano-k8s-worker-02"], "metrics": ["cpu","memory"]}
```

The ETA and monitoring probe use the client-streaming RPC approach where the monitoring probe sends a stream of messages to the ETA. Next are some indicative log messages from the operation of the Kubernetes monitoring probe in the UVT cluster and the ETA.

```
INFO:SERRANO.TelemetryProbe.StreamingTelemetry:Streaming telemetry activated at ti
mestamp '1701548446'
DEBUG:SERRANO.TelemetryProbe.StreamingTelemetry:Streaming telemetry parameters: {"
cluster_uuid": "7628b895-3a91-4f0c-b0b7-033eab309891", "action": "start", "request
": "streaming", "reporting_rate": 5, "nodes": ["serrano-k8s-worker-02"], "metrics"
: ["cpu", "memory"]}
DEBUG:SERRANO.TelemetryProbe.StreamingTelemetry:Forward streaming telemetry metric
s at Enhanced Telemetry Agent - timestamp: 1701548447
DEBUG:SERRANO.TelemetryProbe.StreamingTelemetry:Forward streaming telemetry metric
s at Enhanced Telemetry Agent - timestamp: 1701548452
DEBUG:SERRANO.TelemetryProbe.StreamingTelemetry:Forward streaming telemetry metric
s at Enhanced Telemetry Agent - timestamp: 1701548457
DEBUG:SERRANO.TelemetryProbe.StreamingTelemetry:Forward streaming telemetry metric
s at Enhanced Telemetry Agent - timestamp: 1701548462
```

```
INFO:SERRANO.EnhancedTelemetryAgent.StreamTelemetry:Get streaming telemetry data
DEBUG:SERRANO.EnhancedTelemetryAgent.StreamTelemetry:{'serrano-k8s-worker-02': {'t
imestamp':1701548447,'node_cpus':[{'idle':508444.15,'used':24175.47,'label':'0'},{
'idle':508469.32,'used':23973.739999999998,'label':'1'},{'idle':508393.15,'used':2
3790.52,'label':'2'},{'idle':508454.94,'used':23825.629999999997,'label':'3'}],'me
mory':{'node_memory_MemAvailable_bytes':6750740480,'node_memory_MemFree_bytes':335
2383488,'node_memory_MemTotal_bytes':8331374592,'node_memory_MemUsed_bytes':497899
1104,'node_memory_usage_percentage':59.76}}}
INFO:SERRANO.EnhancedTelemetryAgent.PMDSInterface:Store streaming telemetry for cl
uster '7628b895-3a91-4f0c-b0b7-033eab309891'
```

```
INFO:SERRANO.EnhancedTelemetryAgent.StreamTelemetry:Get streaming telemetry data
INFO:SERRANO.EnhancedTelemetryAgent.StreamingTelemetry:{'serrano-k8s-worker-02':{'
timestamp':1701548452,'node_cpus':[{'idle':508449.01,'used':24175.8,'label':'0'},{
'idle':508474.37,'used':23973.88,'label':'1'},{'idle':508398.16,'used':23790.72,'l
abel':'2'},{'idle':508459.95,'used':23825.82,'label':'3'}],'memory':{'node_memory_
MemAvailable_bytes':6751502336,'node_memory_MemFree_bytes':3353116672.0,'node_memo
ry_MemTotal_bytes':8331374592,'node_memory_MemUsed_bytes':4978257920,'node_memory_
usage_percentage':59.75}}}
INFO:SERRANO.EnhancedTelemetryAgent.PMDSInterface:Store streaming telemetry for cl
uster '7628b895-3a91-4f0c-b0b7-033eab309891'
```

Figure 28 illustrates the different granularity of collected telemetry data regarding the memory performance of the worker node *'serrano-k8s-worker-02'* in the UVT K8s cluster. The left chart corresponds to data collected from the typical operation of the telemetry framework, and the right from the streaming telemetry.

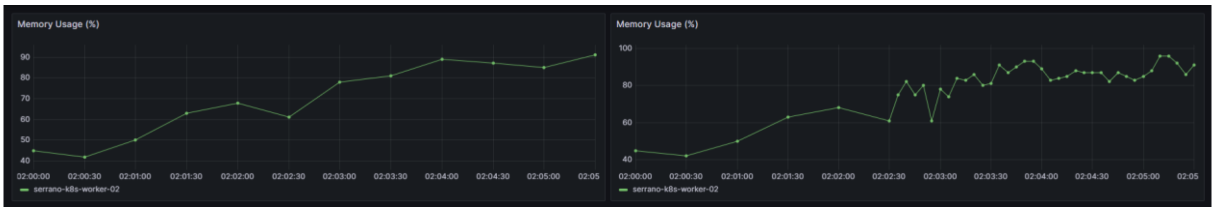


**Figure 28: Memory performance monitoring data: (a) typical operation of telemetry framework, (b) streaming telemetry**

**<u>Telemetry data visualization</u>**

Several custom Grafana [63] dashboards were created for visualizing the collected telemetry and to support the operation and evaluation of the final release of the SERRANO platform. Deliverable D6.8 (M38) provides additional information for the available Grafana dashboards.

## 4.5 Data Broker

### 4.5.1 Message Broker

#### *4.5.1.1 Description*

The SERRANO architecture includes the Data Broker service that provides the appropriate communication mechanisms to interconnect the individual components and enable the exchange of messages and events within the distributed SERRANO platform. The Data Broker includes two components, Message Broker and Stream Handler.

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 [58] that supports different transport and messaging protocols, such as the different versions of Advanced Message Queuing Protocol (AMQP) and MQ Telemetry Transport (MQTT) [59]. The SERRANO services and end users interact with the Message Broker using the well-known and opensource Pika [6] and Paho [60] Python libraries.

#### 4.5.1.2 Integration details

**Table 6: Integration details of Message Broker**

| | |
|---|---|
| **IP(s)/Port(s)** | Message Broker (AMQP): 85.120.206.30:5672<br>Message Broker (MQTT): 85.120.206.30:1883 |
| **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 |

**Resource Optimization Toolkit**

The ROT is one of the SERRANO components that uses the Message Broker functionalities to enable asynchronous communication between its components (Section 4.3.2). Additional details for using the Message Broker in the context of the ROT are available in deliverables D5.3 (M15) and D5.4 (M31). Moreover, D6.3 (M18) provided several integration examples related to the operation of the ROT.

The functionality of the Message Broker is also used internally by the developed Python API for implementing the asynchronous delivery of the ROT responses to the end users. For this operation, the ROT uses the “direct” exchange type with the routing key the *“client_uuid”*. An example of a response for successful execution of the Storage Policy algorithm follows below for a client with the identifier “42f534cf-1ec5-46a7-8f80-ed8d2f8dc7d2”.

```
{"execution_id": "9eecb659-299f-43e5-bb53-00890deea309", "status": 2,
 "client_uuid": "42f534cf1ec5-46a7-8f80-ed8d2f8dc7d2", "timestamp": 1700246732,
 "results": {"kind": "StoragePolicy", "backends": [], "edge_devices": [1, 2],
 "redundant_packets": 1}}
```

**Seamless execution of SERRANO accelerated kernels**

The SERRANO SDK abstracts the interaction with the various SERRANO platform services to enable the seamless execution of SERRANO HW/SW accelerated kernels, as described in Section 4.2.3.2.4. The Message Broker provides the data handling operations to enable the transparent input data movement from the end-user to the selected platform with accelerated resources and provide results back to users. Figure 29 illustrates the related functionalities, focusing on the first and last steps of the overall workflow for the kernel execution within the SERRANO platform.

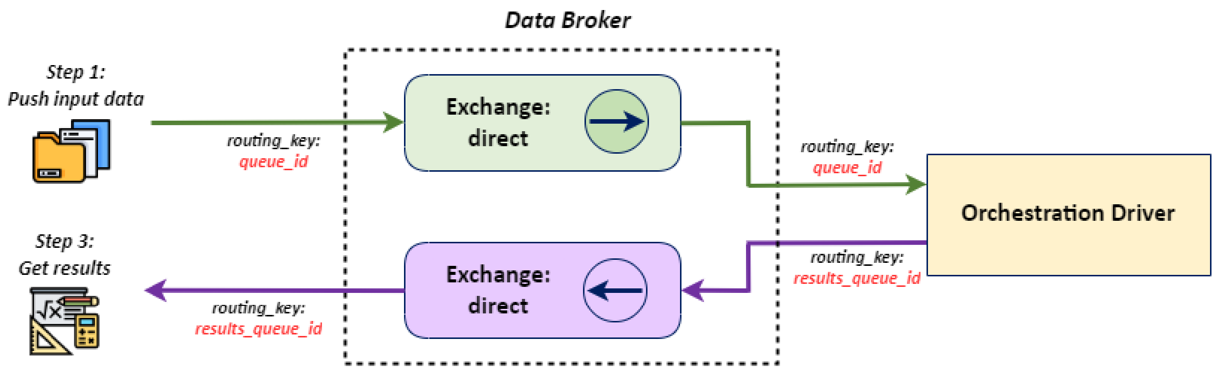


**Figure 29: Message Broker and execution of SERRANO accelerated kernels**

Next is an example of the information returned by the method that pushes the input data for the KNN kernel execution. This data description is then forwarded to the SERRANO orchestration and deployment mechanisms, which they use to retrieve all the required input data.

```
{"queue_id": "0ead8e8f-75af-45dc-8582-f4f3e955d85f", "arguments": ["position",
 "labels", "input_data"], "uuid": "61e75310-d826-4541-a638-a08d5b9bb44e",
 "total_size_MB": 66.1, "cluster_uuid": "3984f92a-21a0-4ce5-85a4-7febd261b794",
 "storage": "broker"}
```

## <u>Anomaly Detection in Manufacturing Settings</u>

As described in Section 4.12, this use case developed 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. The application includes several microservices that, among others, exchange information through a MQTT broker. The Message Broker component of the SERRANO platform provides the required functionality. Below, there is an example from the position-model-inference microservice that is registered to specific topics ("*data/+/+/position/cycle*") and received data from machine3 ("*/data/machine3/x_axis/position/cycle*") for further analysis.

```
2023-11-17 07:32:42,019 - subscriber - INFO - Sending SUBSCRIBE (d0, m545) [(b'dat
a/+/+/position/cycle', 0)]
2023-11-17 07:32:42,030 - subscriber - INFO - Received SUBACK
2023-11-17 07:32:42,040 - subscriber - INFO - Subscribed: 545 (0,)
2023-11-17 07:32:45,163 - subscriber - INFO - Received PUBLISH (d0, q0, r0, m0), '
data/machine3/x_axis/position/cycle', ...  (40814 bytes)
```

## 4.5.2 Stream Handler

*For completeness, some subsections that existed in D6.3 have been repeated in the contents of this section. The below information has been updated according to the current status.*

### *4.5.2.1 Description*

SERRANO's distributed streaming platform allows publishing and subscribing to streams of records. This part of the messaging infrastructure supports 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.

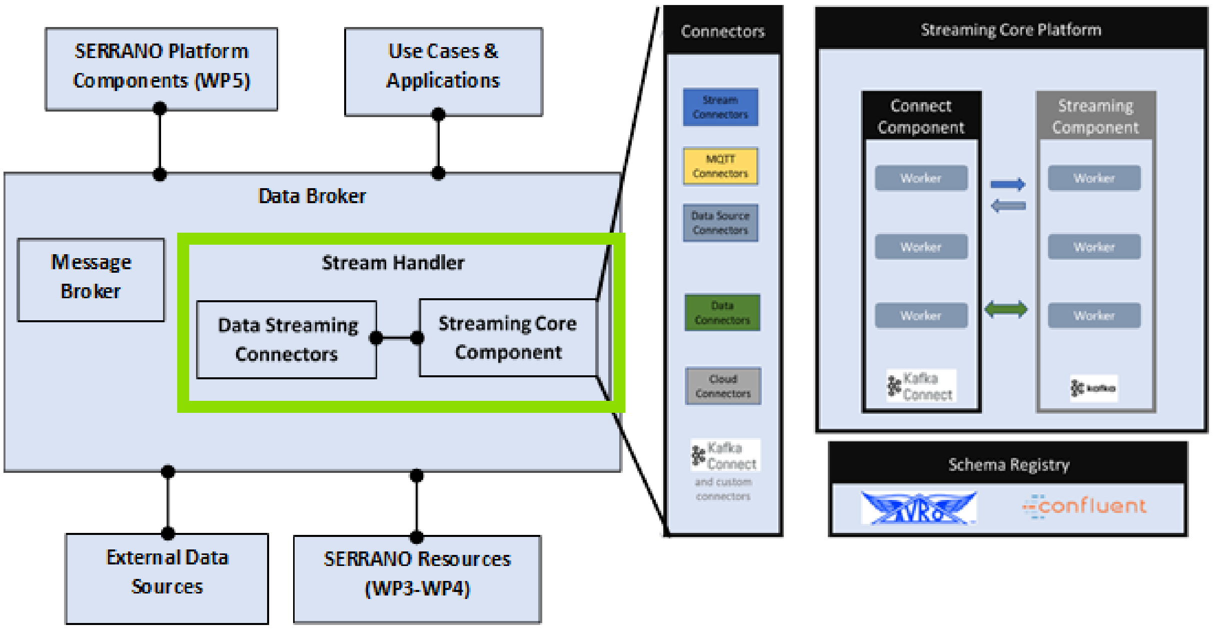


**Figure 30: Stream Handler and possible integrations with data sources and other infrastructure**

The implementation of Stream Handler is based on existing and well-known software platforms that support critical features like the loose coupling of components, increased scalability, and security. Figure 30 shows the key building blocks and their interactions with other SERRANO components.

### *4.5.2.2 Inner components*

#### *4.5.2.2.1 Streaming Component (Kafka Cluster)*

A Kafka [11] 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 [12] to manage and coordinate 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.

##### 4.5.2.2.2 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.

##### 4.5.2.2.3 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.

##### 4.5.2.2.4 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.

#### 4.5.2.3 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 31). The described data sources are 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 [13], Apache HBASE [14], etc. as well as other persistence approaches such as Mongo [15], MySQL [16], 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 [17], Hadoop [18], Kafka Streams [19], Spark Streaming [20] 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 [21], DeepLearning4J [22], or H2O.ai [23].

### *4.5.2.4 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.

#### *4.5.2.4.1 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 7 describes the notification messages exposed by the SERRANO telemetry framework.

**Table 7: Telemetry notification messages**

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

Below there is an example of a notification message concerning a monitoring probe status.

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

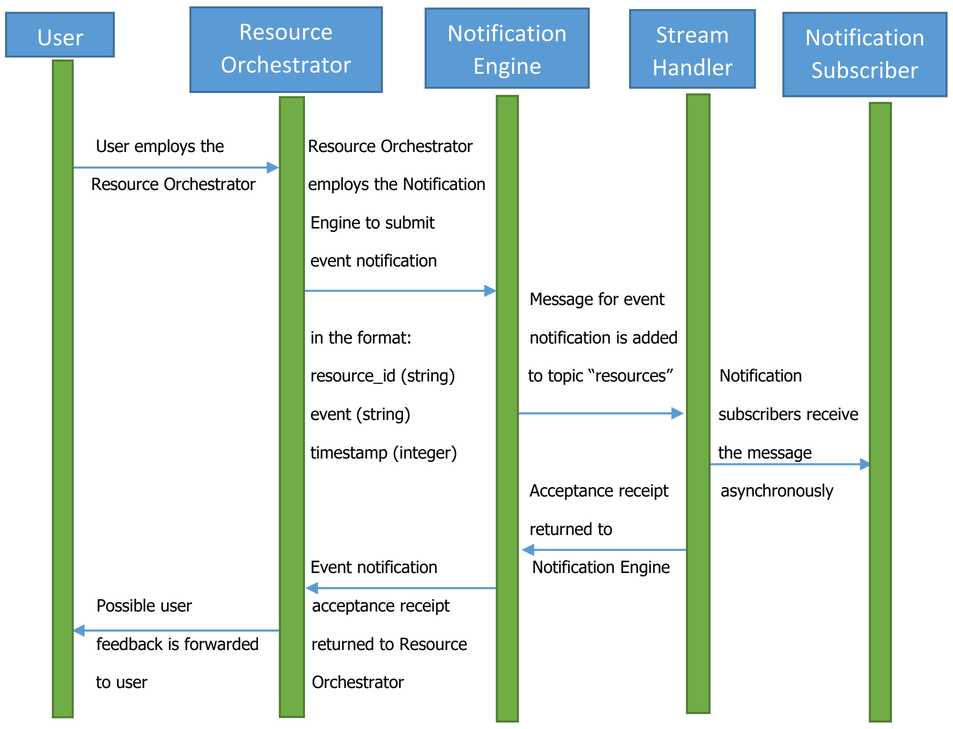

User
Resource Orchestrator
Notification Engine
Stream Handler
Notification Subscriber
User employs the Resource Orchestrator
Resource Orchestrator employs the Notification Engine to submit event notification
in the format: resource_id (string) event (string) timestamp (integer)
Message for event notification is added to topic "resources"
Notification subscribers receive the message asynchronously
Acceptance receipt returned to Notification Engine
Event notification acceptance receipt returned to Resource Orchestrator
Possible user feedback is forwarded to user

**Figure 31: Resource Orchestrator Interaction with Stream Handler**

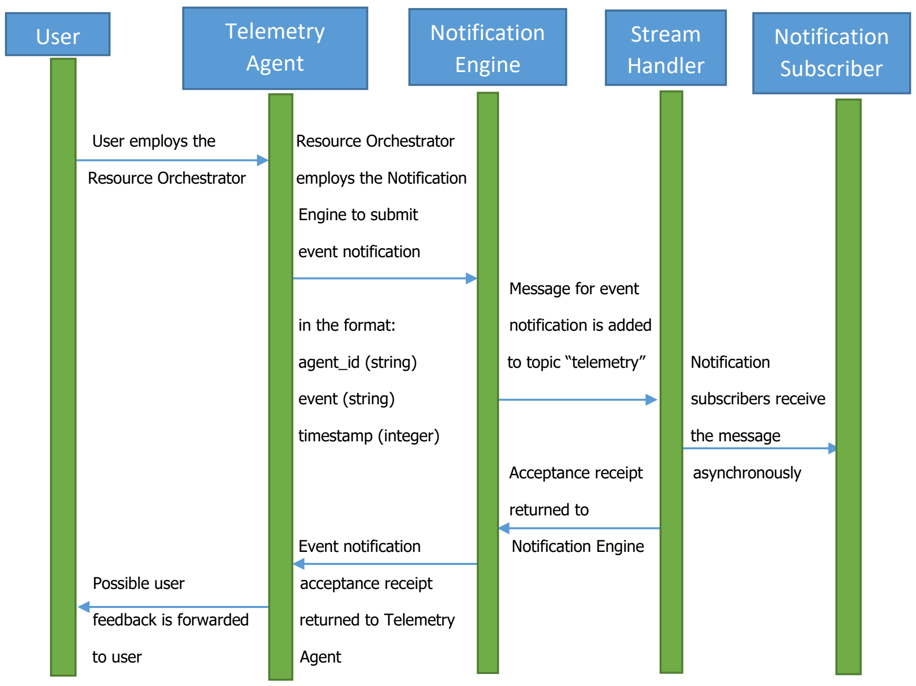

User
Telemetry Agent
Notification Engine
Stream Handler
Notification Subscriber
User employs the Resource Orchestrator
Resource Orchestrator employs the Notification Engine to submit event notification
in the format: agent_id (string) event (string) timestamp (integer)
Message for event notification is added to topic "telemetry"
Notification subscribers receive the message asynchronously
Acceptance receipt returned to Notification Engine
Event notification acceptance receipt returned to Telemetry Agent
Possible user feedback is forwarded to user

**Figure 32: Telemetry Agent Interaction with Stream Handler**

#### *4.5.2.4.2 Service Assurance*

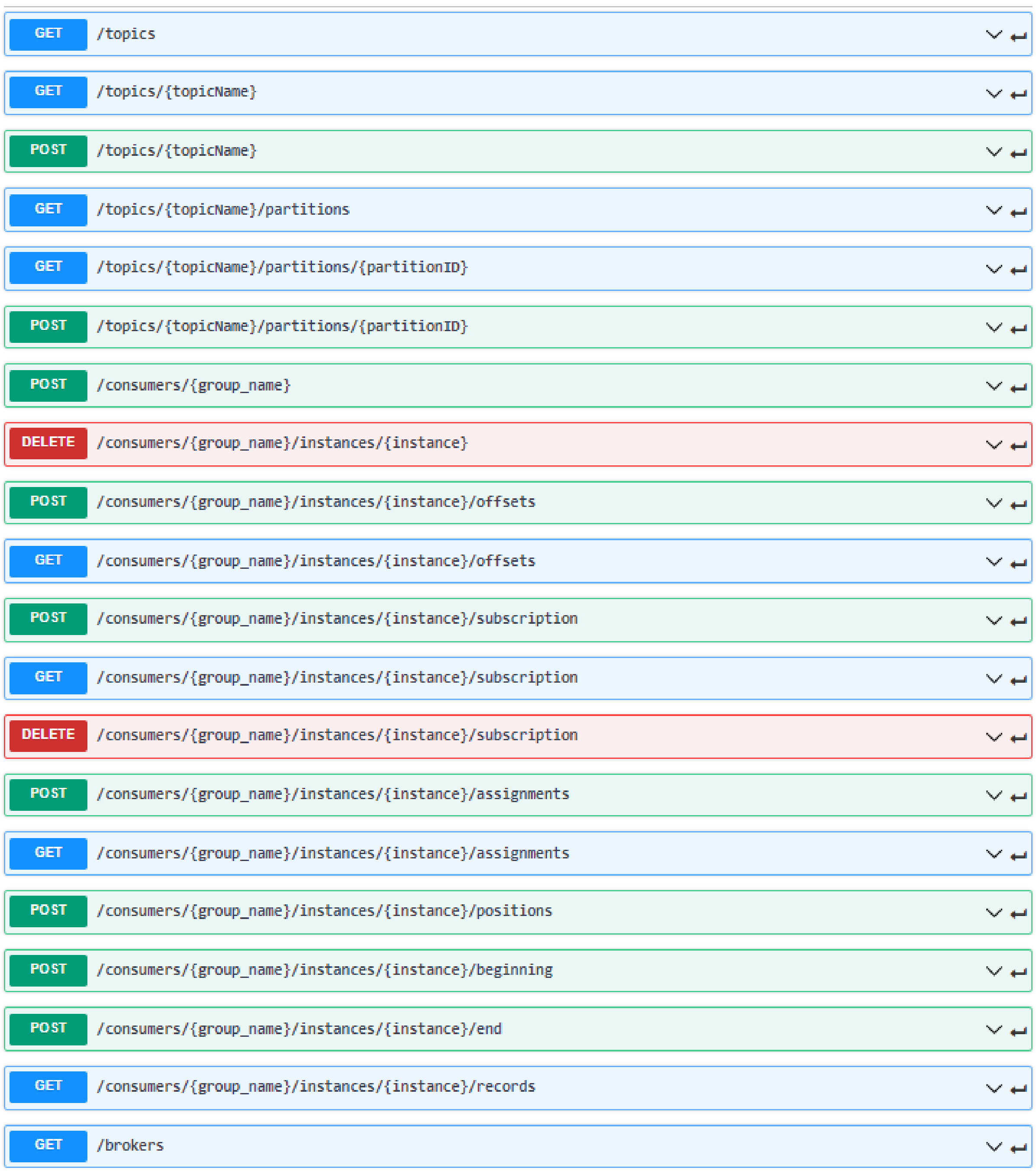
As described in more detail in Section 4.9.1, when an event anomaly is detected by the Event Detection Engine (EDE) component of service assurance (specifically by its prediction sub-component), it is necessary to notify the SERRANO components in charge of orchestration, scheduling, or remediation. Thus, 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 54.

#### *4.5.2.4.3 Integration details*

**Table 8: Integration details of Stream Handler**

| | |
|---|---|
| **IP(s)/Port(s)** | • Kafka protocol over TCP: 88.198.124.99:9092<br>• Rest-proxy: 88.198.124.99:8082<br>• 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://github.com/ict-serrano/streamhandler/blob/develop/rest_proxy.yaml<br>https://github.com/ict-serrano/streamhandler/blob/develop/schema_registry.yaml |
| **Location of integration tests** | https://github.com/ict-serrano/streamhandler/blob/develop/Jenkinsfile |

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



GET /topics

GET /topics/{topicName}

POST /topics/{topicName}

GET /topics/{topicName}/partitions

GET /topics/{topicName}/partitions/{partitionID}

POST /topics/{topicName}/partitions/{partitionID}

POST /consumers/{group_name}

DELETE /consumers/{group_name}/instances/{instance}

POST /consumers/{group_name}/instances/{instance}/offsets

GET /consumers/{group_name}/instances/{instance}/offsets

POST /consumers/{group_name}/instances/{instance}/subscription

GET /consumers/{group_name}/instances/{instance}/subscription

DELETE /consumers/{group_name}/instances/{instance}/subscription

POST /consumers/{group_name}/instances/{instance}/assignments

GET /consumers/{group_name}/instances/{instance}/assignments

POST /consumers/{group_name}/instances/{instance}/positions

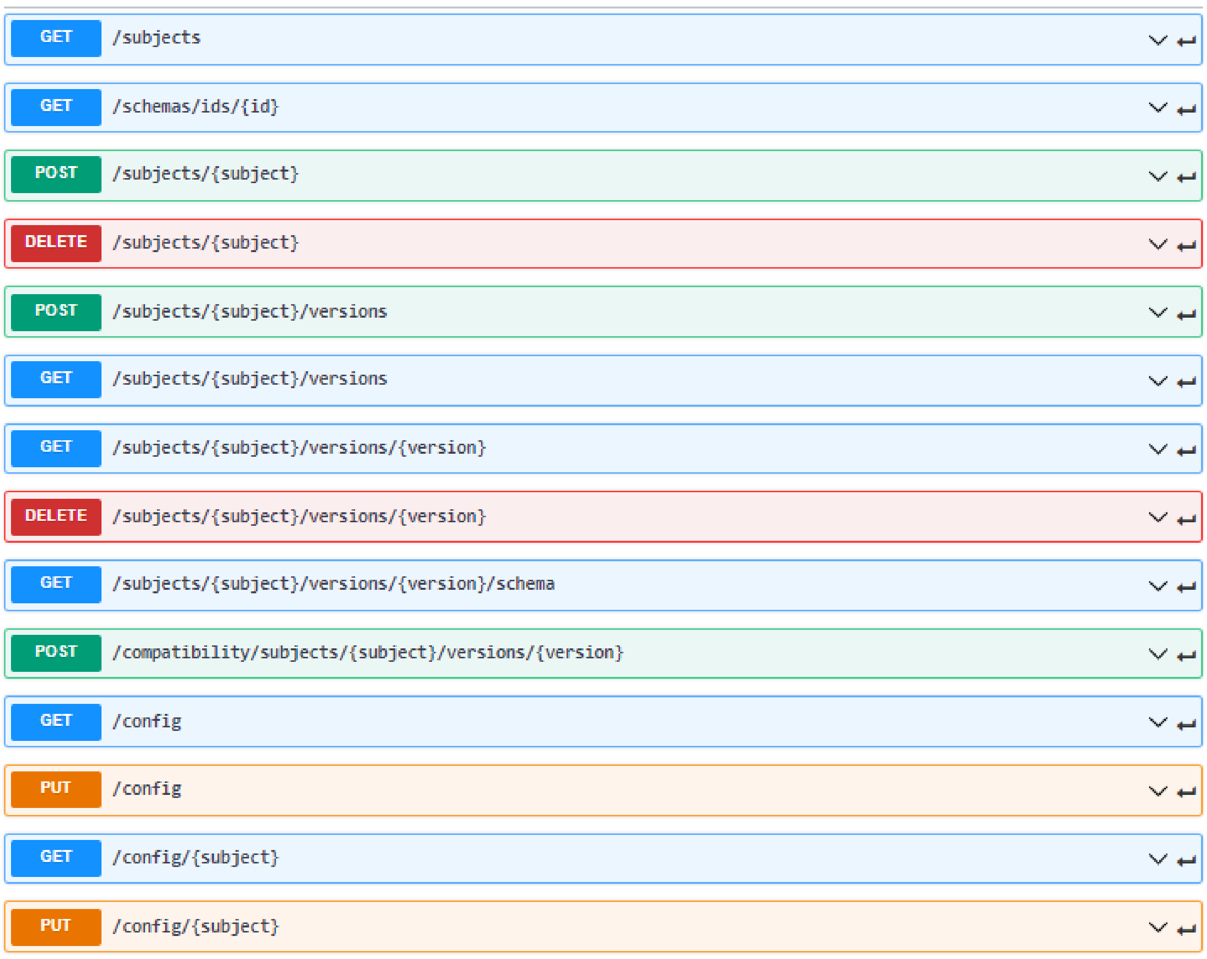
POST /consumers/{group_name}/instances/{instance}/beginning

POST /consumers/{group_name}/instances/{instance}/end

GET /consumers/{group_name}/instances/{instance}/records

GET /brokers

**Figure 33: REST Endpoints exposed by Streaming Core Platform (through the REST Proxy)**



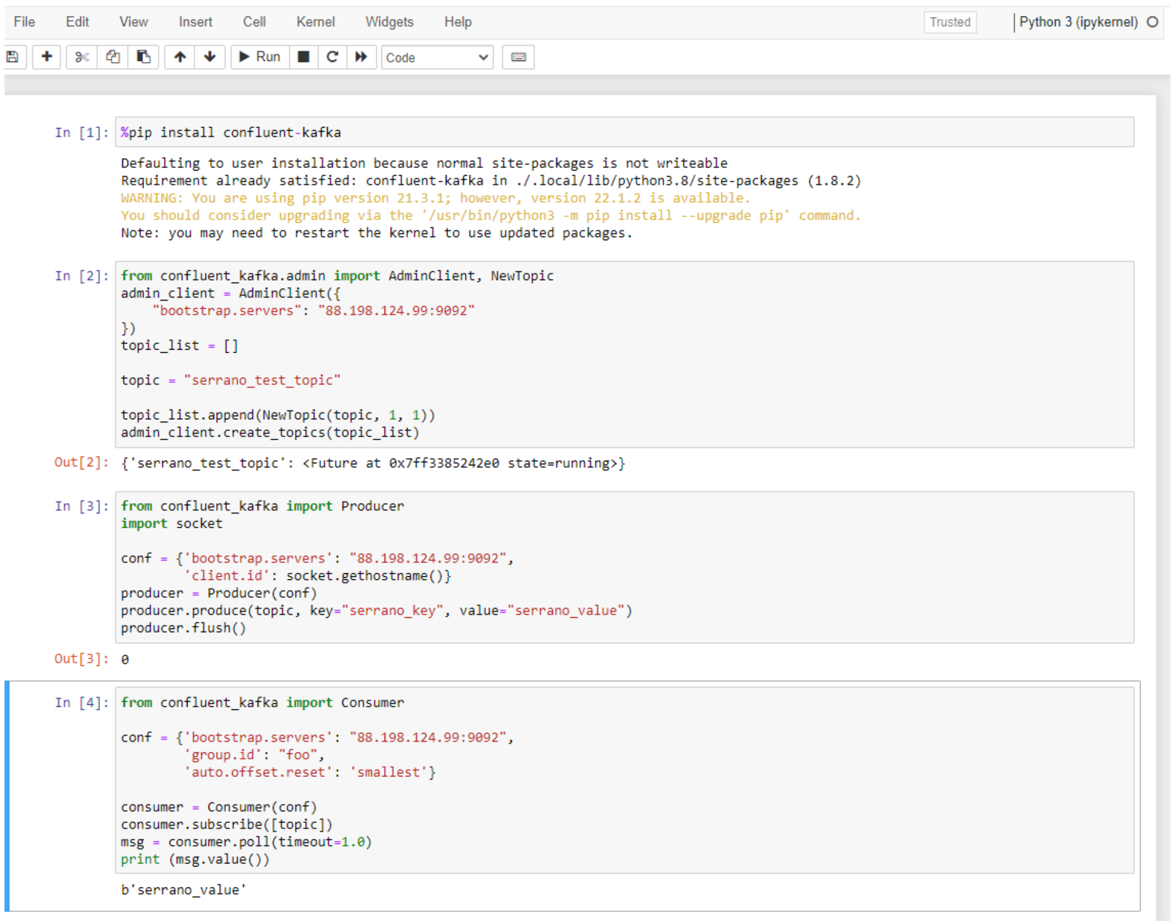
```
GET     /subjects
GET     /schemas/ids/{id}
POST    /subjects/{subject}
DELETE  /subjects/{subject}
POST    /subjects/{subject}/versions
GET     /subjects/{subject}/versions
GET     /subjects/{subject}/versions/{version}
DELETE  /subjects/{subject}/versions/{version}
GET     /subjects/{subject}/versions/{version}/schema
POST    /compatibility/subjects/{subject}/versions/{version}
GET     /config
PUT     /config
GET     /config/{subject}
PUT     /config/{subject}
```

**Figure 34: Schema Registry REST API**

### *4.5.2.4.4 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 35 does not involve communication encryption which is enabled when transferring data that are relevant to the SERRANO platform operations.



```
In [1]: %pip install confluent-kafka

        Defaulting to user installation because normal site-packages is not writeable
        Requirement already satisfied: confluent-kafka in ./.local/lib/python3.8/site-packages (1.8.2)
        WARNING: You are using pip version 21.3.1; however, version 22.1.2 is available.
        You should consider upgrading via the '/usr/bin/python3 -m pip install --upgrade pip' command.
        Note: you may need to restart the kernel to use updated packages.

In [2]: from confluent_kafka.admin import AdminClient, NewTopic
        admin_client = AdminClient({
            "bootstrap.servers": "88.198.124.99:9092"
        })
        topic_list = []

        topic = "serrano_test_topic"

        topic_list.append(NewTopic(topic, 1, 1))
        admin_client.create_topics(topic_list)

Out[2]: {'serrano_test_topic': <Future at 0x7ff3385242e0 state=running>}

In [3]: from confluent_kafka import Producer
        import socket

        conf = {'bootstrap.servers': "88.198.124.99:9092",
                'client.id': socket.gethostname()}
        producer = Producer(conf)
        producer.produce(topic, key="serrano_key", value="serrano_value")
        producer.flush()

Out[3]: 0

In [4]: from confluent_kafka import Consumer

        conf = {'bootstrap.servers': "88.198.124.99:9092",
                'group.id': "foo",
                'auto.offset.reset': 'smallest'}

        consumer = Consumer(conf)
        consumer.subscribe([topic])
        msg = consumer.poll(timeout=1.0)
        print (msg.value())

        b'serrano_value'
```

**Figure 35: 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 8 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
}
```

## 4.6 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 and PBS-based (e.g., TORQUE, OpenPBS). Further information can be found in deliverable D4.2 [87].

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), submit the job, and monitor the status of the job and the partitions, as shown in Figure 36. Moreover, SERRANO HPC Gateway provides endpoints for remote (HTTP, S3) file transfers into HPC infrastructure, as well as transferring results from HPC into S3.

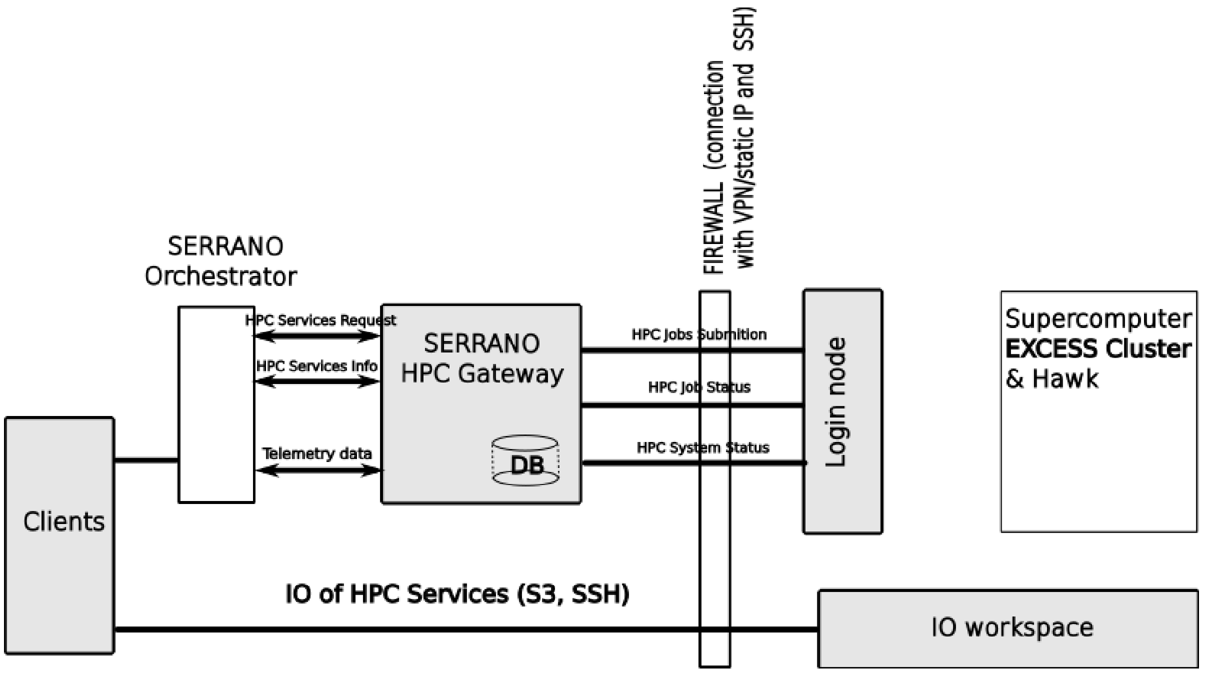


**Figure 36: Interaction between HPC Gateway and HPC infrastructure**

## 4.6.1 Integration details and REST APIs

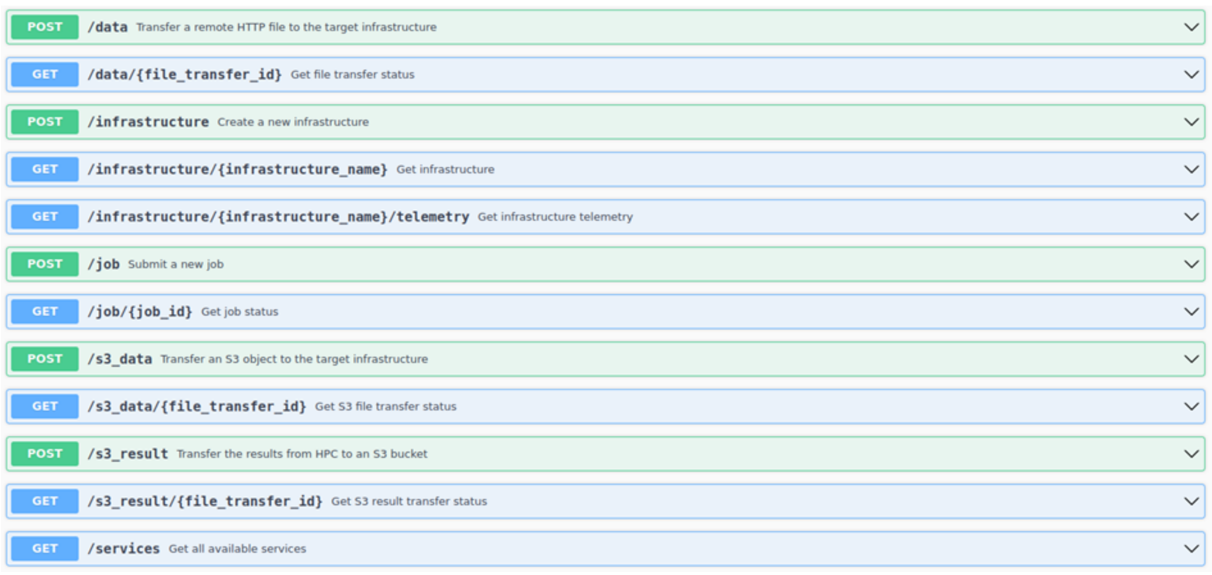
### *4.6.1.1 Integration details*

The HPC System Hardware Interface (HPC Gateway) is integrated with the SERRANO platform. It exposes REST API endpoints (Figure 37) needed for the Resource Orchestrator and Telemetry Framework for executing HPC services (or HPC kernels) and monitoring the state of the HPC infrastructure. Clients can utilise data endpoints of the HPC Gateway to transfer data from HTTP and S3 endpoints, such as the SERRANO Secure Storage service (WP3), into HPC and move the results data back to S3. The HPC Gateway is implemented as a service and interacts with the target HPC infrastructure using SSH protocol (Figure 36). The administrator maintains SSH keys that are used for authentication with the infrastructure.

Furthermore, the API calls that require a longer time to process, e.g., submission of jobs, file transfers, are non-blocking and return the ID of the operation immediately. A client then can monitor the respective state of the operation until the operation is finished, either successfully or with failures.

**Table 9: Integration details of SERRANO HPC System Hardware Interface**

| | |
|---|---|
| **Host name(s)/Port(s)** | https://hpc-interface.services.cloud.ict-serrano.eu |
| **Publicly accessible (y/n and other details)** | The IP is publicly accessible |
| **Type of API** | REST API |
| **API documentation** | https://github.com/ict-serrano/hpc-interface/blob/main/openapi-spec.yaml |
| **Location of integration tests** | https://github.com/ict-serrano/hpc-interface/blob/main/Jenkinsfile |



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

**Figure 37: 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 and kernels is returned. The list is being updated, once the service is deployed on the target HPC system. In this sample, Kalman, FFT, Min-Max, Savitzky-Golay filters and k-Means and k-NN kernels are returned.

```
GET /services
```

```
[
  { "name": "kalman" },
  { "name": "fft" },
  { "name": "min_max" },
  { "name": "savitzky_golay" },
  { "name": "kmean" },
  { "name": "knn" }
]
```

***HPC infrastructure management and telemetry***

An administrator can provide infrastructure details to the HPC Gateway, which include a 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"
}
```

```
Response:
{
  "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 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
```

```
Response:
{
  "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 the 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",
  "services": [ "kalman", "fft" ]
  "params": {
        "read_input_data": "/path/to/input/data"
  }
}
```

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

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

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

### ***File transfer into HPC infrastructure***

Two endpoints are available for file transfer from a source to a destination in an HPC infrastructure: from an HTTP source *(/data*) and an S3 *(/s3_data*) source. In each case, the file transfer status shall be monitored until the completion or failure states are reached.

```
POST /data
Body:
{
  "infrastructure": "cluster_name",
  "src": "https://example.com/some-file.csv",
  "dst": "/tmp/some-file.csv",
}
```

```
Response:
{
  "id": "6f67b9b4-1821-41df-991f-c7fbdfc7f959",
  "infrastructure": "cluster_name",
  "src": "https://example.com/some-file.csv",
  "dst": "/tmp/some-file.csv",
  "status": "transferring",
  "reason": ""
}
```

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

```
Response:
{
  "id": "6f67b9b4-1821-41df-991f-c7fbdfc7f959",
  "infrastructure": "cluster_name",
  "src": "https://example.com/some-file.csv",
  "dst": "/tmp/some-file.csv",
  "status": "completed" | "failed",
  "reason": "" | "error description"
}
```

```
POST /s3_data
Body:
{
  "infrastructure": "cluster_name",
  "endpoint": "https://on-premise-storage-gateway.services.cloud.ict-
serrano.eu/s3",
  "bucket": "bucket-name",
  "object": "object-name",
  "region": "local",
  "access_key": "access_key",
  "secret_key": "secret_key",
  "dst": "/tmp/some-file.csv",
}
```

```
Response:
{
  "id": "6f67b9b4-1821-41df-991f-c7fbdfc7f959",
  "infrastructure": "cluster_name",
  "endpoint": "...",
  "bucket": "bucket-name",
  "object": "object-name",
  "region": "local",
  "dst": "/tmp/some-file.csv",
  "status": "transferring",
  "reason": ""
}
```

```
GET /s3_data/6f67b9b4-1821-41df-991f-c7fbdfc7f959
```

```
Response:
{
  "id": "6f67b9b4-1821-41df-991f-c7fbdfc7f959",
  "infrastructure": "cluster_name",
  "endpoint": "...",
  "bucket": "bucket-name",
  "object": "object-name",
  "region": "local",
  "dst": "/tmp/some-file.csv",
  "status": "completed" | "failed",
  "reason": "" | "error description"
}
```

***<u>Results retrieval from HPC infrastructure</u>***

The HPC Gateway provides an endpoint for transferring the results available after an HPC job execution from an HPC infrastructure into an S3 storage. The results transfer status shall also be monitored until the completion or failure states are reached.

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

```
Response:
{
  "id": "6f67b9b4-1821-41df-991f-c7fbdfc7f959",
  "infrastructure": "cluster_name",
  "endpoint": "...",
  "bucket": "bucket-name",
  "object": "object-name",
  "region": "local",
  "src": "/path/to/results",
  "status": "transferring",
  "reason": ""
}
```

```
GET /s3_result/6f67b9b4-1821-41df-991f-c7fbdfc7f959
```

```
Response:
{
  "id": "6f67b9b4-1821-41df-991f-c7fbdfc7f959",
  "infrastructure": "cluster_name",
  "endpoint": "...",
  "bucket": "bucket-name",
  "object": "object-name",
  "region": "local",
  "src": "/path/to/results",
  "status": "completed" | "failed",
  "reason": "" | "error description"
}
```

## 4.7 HW Acceleration Abstractions and Trusted Execution

### 4.7.1 HW Acceleration Abstractions

In SERRANO, we enable flexible and interoperable hardware acceleration through vAccel, a hardware acceleration framework that decouples the function call from the hardware-specific implementation. In this section, we briefly go through the vAccel framework (extensively described in D2.3 [83], D4.3 [88], and D4.4 [89]), present the integration done with sandboxed container runtimes (D5.4 [92]) and describe an end-to-end FaaS execution example triggered by the SERRANO Resource Orchestrator.

#### *4.7.1.1 The vAccel framework*

In SERRANO, we introduce vAccel [65], a framework that enables virtualized workloads to access hardware accelerators securely and efficiently. vAccel is addressing this situation in two ways. Firstly, it enables the development of hardware independent applications by logically separating an application into two parts: (i) the user code which is part of the application logic itself and (ii) the hardware specific code which is the part of the application that runs on a hardware accelerator. Second, it enables hardware acceleration within virtualized guests by employing an efficient API remoting approach at the granularity of function calls to delegate accelerable code in a vAccel agent on the host system.

The vAccel software framework has been described in detail in D4.3 [87]. Additionally, OpenFaaS is also described in the deliverable above. In the following sections, we provide a brief overview of the software stack, along with the developments of porting the various SERRANO kernels to vAccel and OpenFaaS.

#### *4.7.1.1.1 vAccel*

vAccel enables workloads to enjoy hardware acceleration while running on environments that do not have direct (physical) access to acceleration devices.

The design goals of vAccel are:

1. **Portability**: vAccel applications can be deployed in machines with different hardware accelerators without re-writing or re-compilation.
2. **Security**: A vAccel application can be deployed, *as is*, in a VM to ensure isolation in multi-tenant environments. QEMU [66], AWS Firecracker [67], and Cloud Hypervisor [68] are currently supported.
3. **Compatibility**: vAccel supports the OCI container format through integration with the Kata containers [69] framework.
4. **Low-overhead**: vAccel uses a very efficient transport layer for offloading "accelerate-able" functions from inside the VM to the host, incurring minimum overhead.
5. **Scalability**: Integration with k8s allows the deployment of vAccel applications at scale.

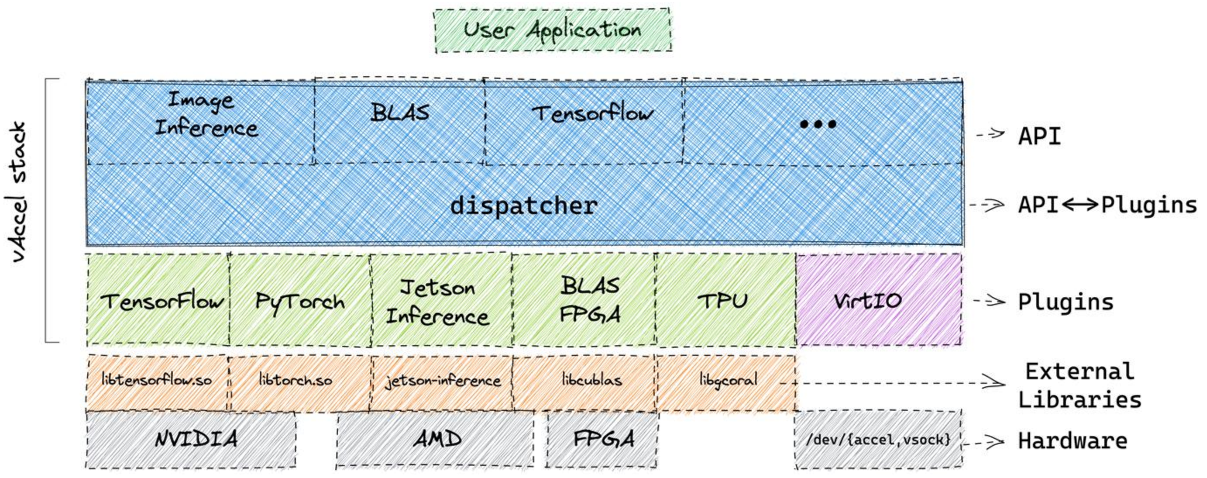


**Figure 38: vAccel software stack**

The core component of vAccel is the vAccel runtime library (vAccelRT). vAccelRT is designed in a modular way: the core runtime exposes the vAccel API to user applications and dispatches requests to one of many *backend plugins*, which implement the glue code between the vAccel API operations on a particular hardware accelerator.

The user application links against the core runtime library and the plugin modules are loaded at runtime. This workflow decouples the application from the hardware accelerator-specific parts of the stack, allowing for seamless migration of the same binary to different platforms with different accelerator capabilities, without the need to recompile user code.

##### 4.7.1.1.1.1 Virtualization Abstraction

Hardware acceleration for virtualized guests is, still, a real challenge. Typical solutions involve device pass-through or paravirtual drivers that expose hardware semantics inside the guest. vAccel differentiates itself from these approaches by exposing coarse-grain "accelerate-able" functions in the guest over a generic transport layer.

The semantics of the transport layer are hidden from the programmer. A vAccel application that runs on baremetal with an Nvidia GPU can run *as is* inside a VM using our appropriate *VirtIO* backend plugin.

We have implemented the necessary parts for our VirtIO driver in our forks of QEMU [70] and Firecracker [71] hypervisors.

Additionally, we have designed the above transport protocol over sockets, allowing vAccel applications to use any backend, if there is a socket interface installed between the two peers. Existing implementations include VSOCK and TCP sockets. Any hypervisor supporting virtio-vsock can support vAccel.

##### 4.7.1.1.1.2 Container Runtime Integration

To facilitate the deployment of vAccel-enabled applications, we integrate vAccel to a popular container runtime, kata-containers [69].

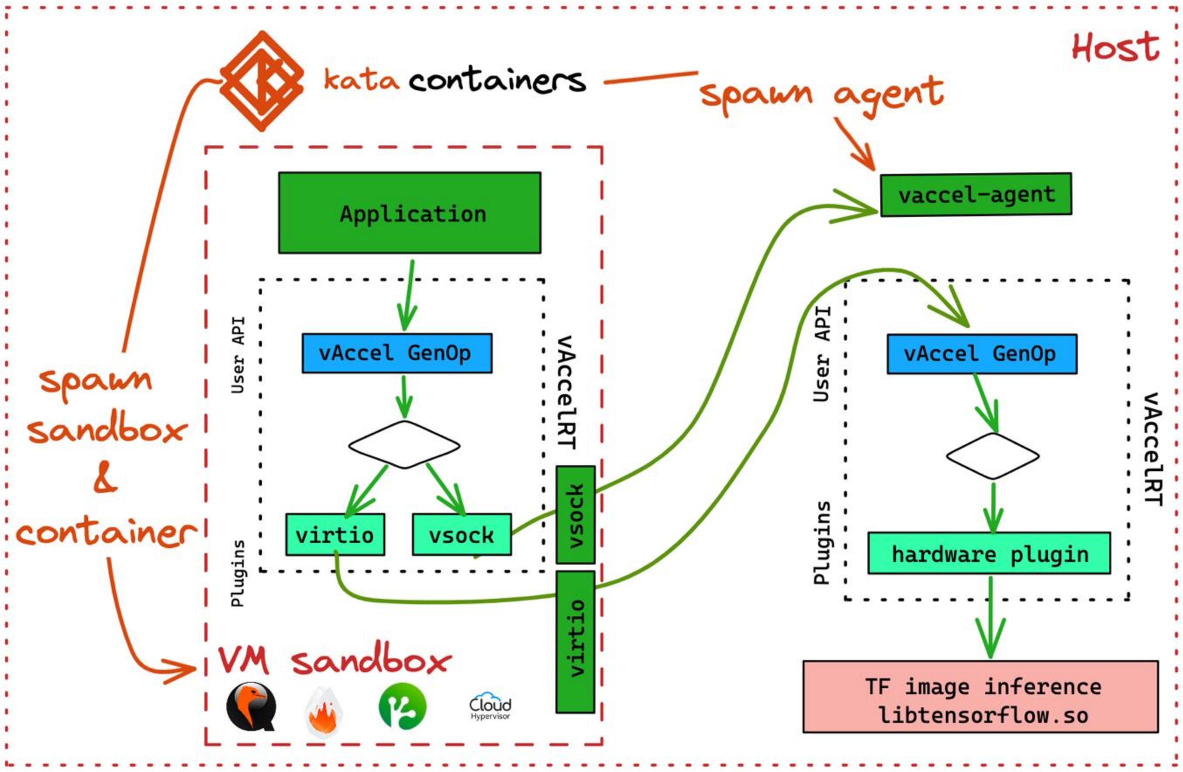


Figure 39: vAccel integration with container runtimes

Kata Containers enable containers to be seamlessly executed in sandbox Virtual Machines. Kata Containers are as light and fast as containers and integrate with the container management layers, while also delivering the security advantages of VMs. Kata Containers is the result of merging two existing open-source projects: Intel Clear Containers and Hyper runV.

vAccel integration to kata comes in both modes: virtio and vsock. An overview of the software stack is shown in Figure 39.

Our current, downstream implementation for Kata-containers v3 includes support for both the AWS Firecracker sandbox and their custom, tailor-made Dragonball backend, using the vsock mode of vAccel.

#### 4.7.1.1.1.3 Framework and Language Bindings

To facilitate the use of vAccel, we provide bindings for popular languages apart from C. Essentially, the vAccel C API can be called from any language that interacts with C libraries. Building on this, we provide support for Python [72] and Rust while working on extending support for various other high- or low-level languages. In SERRANO, the serverless function implementation for all kernels is implemented using the vAccel Python bindings.

Additionally, we have implemented a subset of Tensorflow [73] and PyTorch APIs in a way that the user can execute an application written for those frameworks over vAccel with minimal and/or no changes.

#### 4.7.1.1.1.4 SERRANO Kernels on vAccel

We focused on hardware interoperability and ease-of-deployment to port the SERRANO hardware accelerated kernels on vAccel.

***Interoperability***

One of the key merits of the vAccel framework is the fact that users write their code using the vAccel API and the underlying plugin executes this code in the respective accelerator device. This enables hardware interoperability as the user does not need to rewrite or even re-compile their code if they want to run on a different hardware accelerator. This greatly facilitates the scaling of hardware-accelerated applications throughout the cloud-edge continuum, as the user builds a container image with their vAccel API code, deploys it in the SERRANO platform and this code can use hardware accelerators in the Cloud (Generic, NVIDIA GPUs), at the Edge (Jetson GPUs, Orin/Xavier/Nano), or even CPUs when there is no hardware accelerator available (e.g., on a RPi4).

With this in mind, we ported KNN, K-MEANS, Black-Scholes, and SavGol to vAccel, developing plugin implementations for CPU, GPU, and FPGA hardware accelerators. In the following sections, we briefly elaborate on the porting methodology and the performance implications this integration imposes.

***Libification***

The primary way of allowing applications to run on the vAccel framework is by separating the part we want to abstract away from the core I/O part of the application. Since the actual application is essentially the kernel to be abstracted, nearly all the code from the kernel resides in the plugin part of the vAccel stack. Instead of developing separate API calls and plugins for all the available kernels and execution modes, we chose to abstract this functionality to a simple exec operation: we "libify" the hardware-accelerated part of the application and build it using the same methods as the generic kernel (e.g., for GPU code, we use *nvcc*, and the output binary is a shared library, e.g., *libknn_app_gpu.so*, exposing the symbol of the kernel we are porting, e.g., *knn_app*).

We followed the above method for all kernels. The summary of kernels and libraries available is shown in Table 10.

**Table 10: SERRANO kernels ported to vAccel**

| Kernel | Symbol | Library | Hardware |
|---|---|---|---|
| k-NN | knn_app | libknn_app_cpu.so | CPU |
| | | libknn_app_gpu.so | GPU |
| | | libknn_app_fpga.so | FPGA |
| k-MEANS | kmeans_app | libkmeans_app_cpu.so | CPU |
| | | libkmeans_app_gpu.so | GPU |
| | | libkmeans_app_fpga.so | FPGA |
| BS | bs_app | libbs_app_cpu.so | CPU |
| | | libbs_app_fpga.so | FPGA |
| SAVGOL | savgol_app | libsavgol_app_cpu.so | CPU |
| | | libsavgol_app_gpu.so | GPU |
| | | libsavgol_app_fpga.so | FPGA |

Essentially, to port the kernels to vAccel, we followed the steps below:

- Step 1: We used the host application as the "frontend" and replaced the call to the relevant function with a library call implemented by all execution modes for the specified kernel. We implemented "plugin" libraries for each of the core code versions (CPU, GPU, FPGA) and verified the execution is exactly the same as the original code.
- Step 2: We replaced this library call with a vAccel-specific call. This library, essentially, the "frontend library", enabled us to set up the necessary data structures to ensure input and output consistency. Afterwards, using the same plugin libraries as before, we were able to specify which plugin library we want to use for each execution example: as we used the vaccel-exec operation, all we needed to do is provide the frontend with the shared object to be executed on the host, and a symbol (summarised in Table 10).

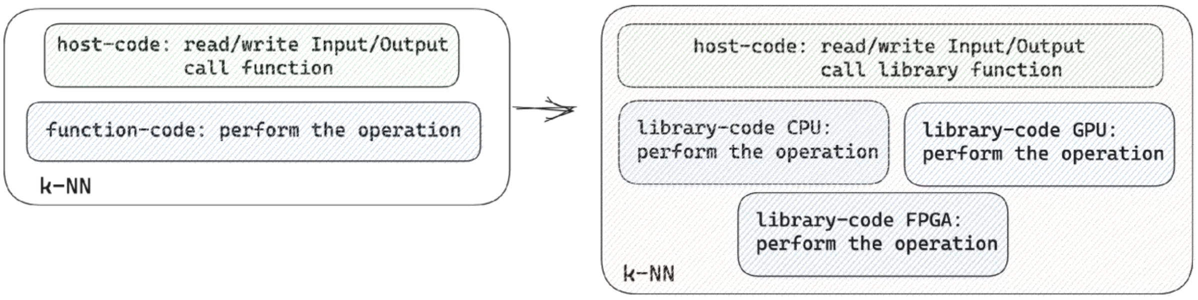


**Figure 40: Libification of original kernel**

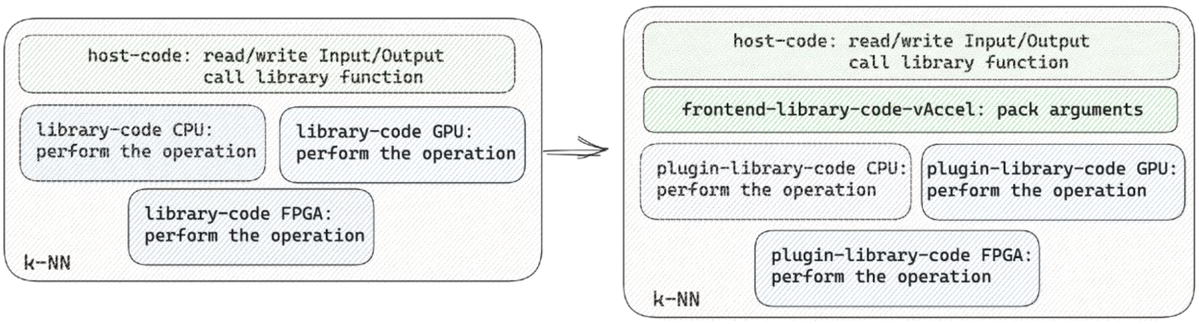


**Figure 41: vAccel port**

Figure 40 and Figure 41 illustrate the above process as steps 1 and 2.

To assess the overhead imposed by this process on the specific kernels, we performed an initial evaluation on a Jetson Xavier AGX system (CPU and GPU execution). We measured execution time with the identical input provided by the partners that developed the kernels.

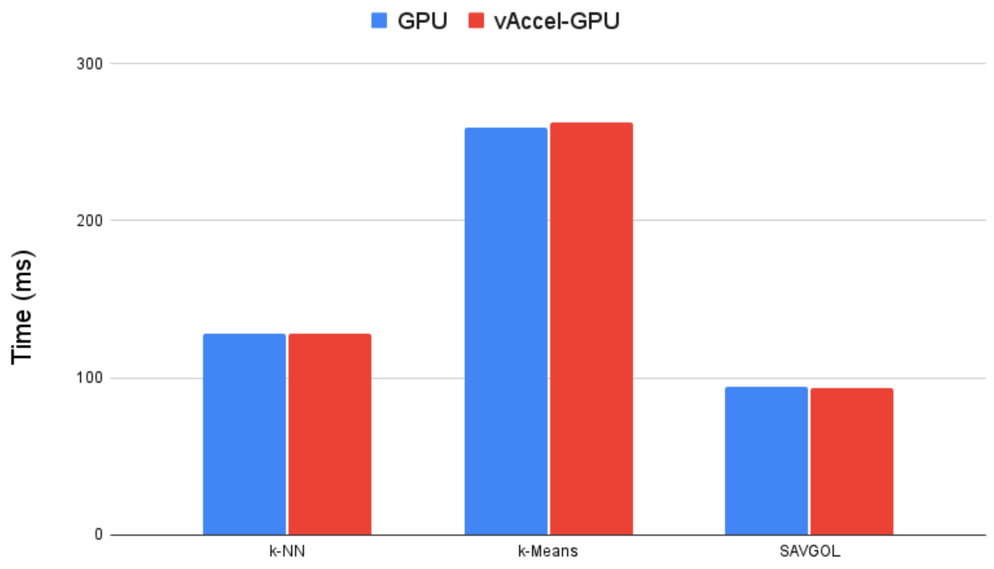


**Figure 42: Performance overhead of vAccel on local execution (library overhead)**

Figure 42 presents the absolute execution time (in ms) for the GPU version of each of the three SERRANO hardware-accelerated kernels studied: k-NN, k-Means, and SAVGOL. The blue bars present the execution time of the stock kernels provided by the partners vs the vAccel-ported ones. Figure 42 shows that running the kernels via vAccel on the same host imposes negligible overhead.

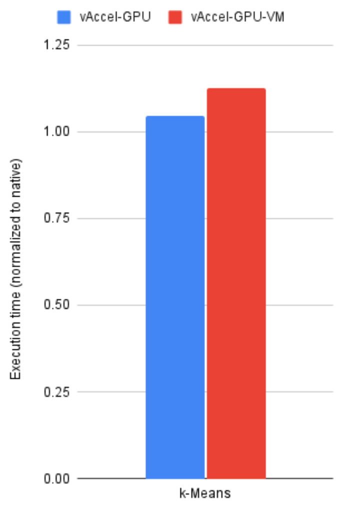


**Figure 43: Performance overhead of vAccel for VM execution**

Figure 43 shows the normalised execution time of the k-Means kernel to native execution when running on the host (vAccel-GPU, blue bars) and on a virtual machine (vAccel-GPU-VM, red bars). We are investigating the source of the overhead imposed on the VM execution. Part of this is accounted to the data transfer between the VM and the host.

#### *4.7.1.1.2 OpenFaaS*

In SERRANO, we build on OpenFaaS [75] to provide short-lived task execution functionality. OpenFaaS is a serverless framework that allows users to deploy functions written in any language to a Kubernetes cluster or standalone VM inside containers. It provides auto-scaling and metrics for the deployed functions. It abstracts the underlying infrastructure and allows users to deploy their services using a high-level CLI tool or Web UI.

***Porting the SERRANO Kernels to Serverless Functions***

To accommodate the diverse input/output modes of the kernels, as well as the various modes of execution, we used the vAccel python bindings to facilitate the process of porting the kernels to serverless functions.

Essentially, the logic of the execution remains the same; the only thing that changes is the way we get the input, and we provide the output.

Since the plugin libraries for executing different algorithms are the same as described in Subsection "Libification", we can use them over the vAccel API by executing the *exec_with_resource* function. We have developed tests to ensure the proper interaction and integration between the algorithm and the plugin library, through vAccel which enables them to interact efficiently.

##### 4.7.1.1.2.1 k-NN

For the k-NN, after loading the necessary libraries for the interaction with vAccel, we must convert the .csv files that will be processed into a format suitable for execution. We establish the appropriate casting for the input and output parameters and pack them appropriately. Since the arguments are in the required format, we execute the *exec_with_resource* function with the necessary input arguments:

- *object*: libknn_app library
- *symbol*: The symbol that implements the k-NN algorithm in the context of the plugin, eg: knn_app
- *arg_read*: The converted read arguments we have packed appropriately.
- *arg_write*: The converted write arguments we have packed appropriately.

##### 4.7.1.1.2.2 k-Means

For the k-Means, we are working again in a similar way. After loading the necessary libraries for the interaction with vAccel, we convert the .csv files that will be processed into the format we want. After that, we establish the appropriate casting for the input and output parameters and pack them appropriately. Since the arguments are in the required format, we execute the *exec_with_resource* function with the necessary input arguments:

- *object*: lib_kmeans_app library
- *symbol*: The symbol that implements the k-Means algorithm in the context of the plugin, eg: kmeans_app
- *arg_read*: The converted read arguments we have packed appropriately.
- *arg_write*: The converted write arguments we have packed appropriately.

##### 4.7.1.1.2.3 KALMAN

For the KALMAN, we are working again in a similar way. After loading the necessary libraries for the interaction with vAccel, we convert the .csv file that will be processed into the format we want. Next, we establish the appropriate casting for the input and output parameters and pack them appropriately. Since the arguments are in the required format, we execute the *exec_with_resource* function with the necessary input arguments:

- *object*: kalman_app library
- *symbol*: The identifier of Kalman library
- *arg_read*: The converted read arguments we have packed appropriately.
- *arg_write*: The converted write arguments we have packed appropriately.

An example Python program that calls the k-MEANS kernel using vAccel is shown in Table 11.

**Table 11: Example python snippet that implements the k-MEANS execution over Python vAccel**

```
def k-NN_vAccel(INPUT_PATH, LABELS_PATH, MODE, iterations):
    t0 = time.time_ns() // 1_000_000
    # Setup input
    start = time.time()
    timeseries= transformed_time_series(INPUT_PATH).astype(np.float32).flatten()
    print('Time for dataset read + transform: ', round(time.time() - start,3), 's')

    labels = load_labels(LABELS_PATH).astype(np.int32)
    golden_labels = labels.copy()
    nr_iter = iterations
    w = 200

    # Setup shared object (plugin) CPU/GPU/FPGA
    obj = "libkmeans_app_%s.so" % MODE

    t1 = time.time_ns() // 1_000_000
    c1 = timeseries[:N_FEATURES]
    c2 = timeseries[N_FEATURES+1:2*N_FEATURES]

    # Setup vAccel parameters
    pa = ffi.cast(f"float[{len(timeseries)}]", ffi.from_buffer(timeseries))
    pc1 = ffi.cast(f"float[{len(c1)}]", ffi.from_buffer(c1))
    pc2 = ffi.cast(f"float[{len(c2)}]", ffi.from_buffer(c2))
    pc = ffi.cast(f"int [{len(labels)}]", ffi.from_buffer(labels))

    # Pack arguments
    arg_read_local = [pa, nr_iter, w, pc1, pc2]
    arg_write = [pc]

    t2 = time.time_ns() // 1_000_000

    # Execute command
    res      =      Exec_with_resource.exec_with_resource(obj,      "kmeans_app",
arg_read=arg_read_local, arg_write=arg_write)
    t3 = time.time_ns() // 1_000_000

    labels_new = ffi.unpack(arg_write[0],len(arg_write[0]))
    total_elements = len(labels_new)
    matching_elements = sum(a == b for a, b in zip(golden_labels, labels_new))
    convergence_percentage = (matching_elements / total_elements) * 100

    t4 = time.time_ns() // 1_000_000
    print(convergence_percentage)
```

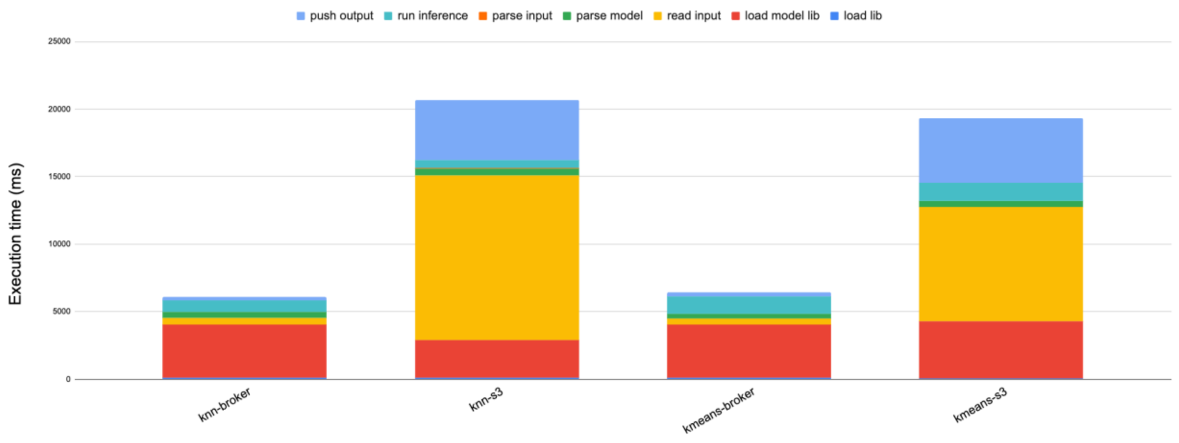


**Figure 44: Performance overhead of end-to-end operation with sandboxed OpenFaaS container and vAccel**

Figure 44 presents the end-to-end execution time (in ms) for k-NN and k-MEANS when called as serverless functions. To perform this test, we built a serverless function that receives a JSON object as input in the format of the Table 12.

**Table 12: Input format for the serverless function**

| Parameter | | Description |
|---|---|---|
| *queue_id* | | A random UUID, acting as the identifier for the storage backend |
| *arguments* | *position* | input data |
| | *labels* | input data |
| | *input file* | input data |
| *uuid* | | a unique id, acting as the identifier for the kernel execution |
| *mode* | | the accelerator to be used (CPU, GPU, or FPGA) |
| *storage* | | the storage backend to be used (data broker or s3) |
| *creds* | *ip* | IP address of the storage backend |
| | *user* | username for the storage backend |
| | *pass* | password for the storage backend |

The integration with the SERRANO Resource Orchestrator is through the SERRANO SDK. The JSON object is constructed using information from the SERRANO orchestration and deployment mechanisms, and upon request from a specific UC application, the function gets invoked and spawned/scaled accordingly. The results flow back through the storage backend (broker or S3) to the requested application to continue its execution.

## 4.7.2 Multi-tenant Isolation and Trusted Execution

In the context of the SERRANO project, multi-tenant isolation has been achieved using lightweight virtualization mechanisms. Trusted execution is realized by establishing the root of trust during the early boot stages of a node and by providing attestation mechanisms for the workloads running on the node. This deliverable contains the final update of the material published in D6.3 (M18). Most of the additions revolve around features that have been integrated and implemented during the second iteration of implementation (M19-M36) to satisfy the requirements of the three use cases. This includes integrating the policy controller, the signing mechanism for container images, and the finalization of the SERRANO security tiers. Isolation and trusted execution mechanisms have been described in greater detail in D3.4 [84]. Next, we briefly describe the mechanisms and present the integration results.

### *4.7.2.1 Multi-tenant isolation*

In SERRANO, we achieve multi-tenant isolation on shared resources via lightweight virtualization mechanisms. We have extensively described sandboxed container runtimes, namely kata-containers, in the context of D5.4 [92] and D3.4 [84]. The integration of these sandboxed container runtimes that are able to spawn containers in microVMs has been achieved through Kubernetes (K8s) Runtime Class functionality. Workloads in the context of SERRANO, depending on their security requirements, are spawned as either generic containers (no further isolation), as sandboxed containers (microVM / virtualization isolation), or as unikernels (reduced attack surface and virtualization isolation). To enable the integration of all these types of execution modes in K8s we have developed customized container runtimes (D5.4). The deployment of workloads in these execution modes is identical to any other type of workload, with the addition of an extra flag that specifies the runtime class (e.g., kata, kata-vaccel, urunc, etc.).

### *4.7.2.2 Trusted Execution*

Security has long been one of the key goals of systems design[2]. Cryptography has enabled the safe storage (at rest) and transmission (in flight) of important data. However, there is still a situation when data can be vulnerable. The applications decrypt the data in order to save them; therefore, the decrypted version of data is stored in RAM, CPU caches, and registers. In recent years, a high number of memory scraping and CPU side-channel attacks have been reported. Under these circumstances, the wide adoption of cloud and edge computing, where users cannot control the underlying infrastructure, raises significant concerns regarding the security of data in use. In that context, the user cannot trust any parts of the system stack that cannot control such as the host Operating System and the hypervisor.

The encryption and signing keys that are used from a Trusted Execution Environment (TEE) should be saved in a hardware module. That module can be the starting point, Root of Trust (RoT) and should be trustworthy. Except for encryption and signing keys, the RoT might

[2] L. Smith, "Architectures for secure computing systems," MITRE CORP BEDFORD MASS, 1975.

contain other root secrets and a set of functions needed for the encryption or validation of data. The code and data (keys) of a RoT are usually stored in a read-only memory (ROM), restricting any modifications. Trusted Platform Modules (TPMs) described in D3.4 [84] are examples of RoT that can generate cryptographic keys and protect critical information such as cryptographic and signing keys, and passwords.

Using the RoT platforms can secure the underlying firmware and extend the trust to higher levels of the software stack. A verified firmware can verify the OS boot loader, which can verify the Operating System and extend the trust to the hypervisor and/or container engine. The process of extending the trust from a RoT to higher levels of the software stack is called a Chain of Trust (CoT).

Apart from the isolation, a TEE should be able to verify the integrity of an application code. Even if the code inside a TEE is isolated and cannot be changed, there is still the danger of someone tweaking that code before it is launched inside a TEE. To be able to verify that the workload running on the hardware node is indeed the one intended by the system, we use attestation: through attestation, the workload tenant can verify that the workload is running on a genuine, authenticated platform and that the initial software stack is the expected one.

###### *4.7.2.2.1 Workload attestation*

To securely sign, verify, and provide attestable metadata to containers that will be deployed on a cluster, we are using the Sigstore[3] project. Sigstore is an open-source project that provides digital signing and verification of container images. Within the container image supply chain, it establishes confidence and maintains the image's integrity by utilizing cryptographic digital signatures and transparency log technologies.

Sigstore consists of a set of tools:

- Cosign (signing, verification, and storage for containers and other artifacts)
- Fulcio (root certificate authority)
- Rekor (transparency log)
- OpenID Connect (means of authentication)
- policy-controller (enforcing container orchestration policy)

<u>Cosign</u>: Tool for signing/verifying containers (and other artifacts) that ties the rest of Sigstore together, making signatures invisible infrastructure. It includes storage in an Open Container Initiative (OCI) registry.

<u>Fulcio</u>: A free root certification authority, issuing temporary certificates to an authorized identity and publishing them in the Rekor transparency log.

<u>Rekor</u>: A built-in transparency and timestamping service, Rekor records signed metadata to a ledger that can be searched but cannot be tampered with.

[3] Sigstore - https://www.sigstore.dev

OpenID Connect: An identity layer that checks if you're who you say you are. It lets clients request and receive information about authenticated sessions and users.

Policy Controller: An admission controller for Kubernetes for enforcing policy on containers allowed to run.

**How Sigstore works**

We are using cosign to sign and verify software artifacts, such as container images and blobs. Cosign operates in two different modes: key pair mode and keyless mode. We have chosen the keyless mode as our preferred option to simplify the process and avoid the burdensome task of securely managing and distributing keys. In keyless mode, the Sigstore associates identities, rather than keys, with an artifact signature. To do that, it utilizes Fulcio to issue short-lived certificates, binding an ephemeral key to an OpenID Connect (OIDC) identity. Fulcio uses OIDC tokens to authenticate requests. Subject-related claims from the OIDC token are extracted and included in issued certificates. Signing events are logged in Rekor, a signature transparency log, providing an auditable record of when a signature was created.

**Verifying identity and signing the artifact**

The process of verifying identity and signing the artifact is the following:

- An in-memory public/private key pair is created.
- The identity token is retrieved.
- Sigstore's certificate authority verifies the identity token of the user signing the artifact and issues a certificate attesting to their identity. The identity is bound to the public key. Decrypting with the public key will prove the identity of the private key holder.
- For security, the private key is destroyed shortly after, and the short-lived identity certificate expires.

Users wishing to verify the software will use the transparency log entry rather than relying on the signer to safely store and manage the private key.

**Recording signing event**

To create the transparency log entry, a Sigstore client creates an object containing information allowing signature verification without the (destroyed) private key. The object contains the hash of the artifact, the public key, and the signature. Crucially, this object is timestamped. The Rekor transparency log "witnesses" the signing event by entering a timestamped entry into the records that attests that the secure signing process has occurred. Clients upload signing events to the transparency log so that the events are publicly auditable. Artifact owners should monitor the log for their identity to verify each occurrence. The software creator publishes the timestamped object, including the hash of the artifact, public key, and signature.

**Verifying the signed artifact**

When a software consumer wants to verify the software's signature, Sigstore compares a tuple of signature, key/certificate, and artifact from the timestamped object against the timestamped Rekor entry. If they match, it confirms that the signature is valid because the user knows that the expected software creator, whose identity was certified when signing, published the software artifact in their possession. The entry in Rekor's immutable transparency log means that the signer will monitor the log for occurrences of their identity and will know if there is an unexpected signing event.

#### *4.7.2.2.2 Incorporating Sigstore in SERRANO*

**Signing**

A set of steps is required to enable image signing using the Sigstore (Figure 45).

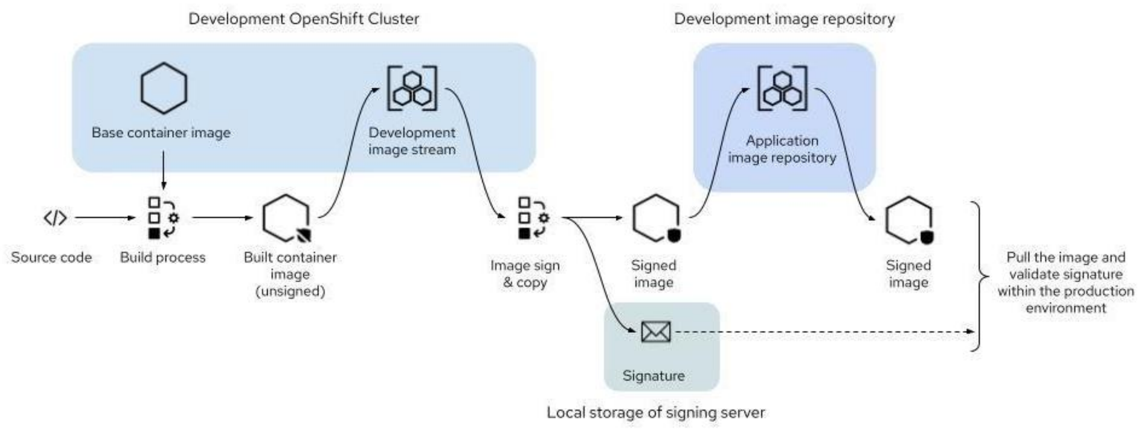

Development OpenShift Cluster
Development image repository
Base container image
Development image stream
Application image repository
Source code
Build process
Built container image (unsigned)
Image sign & copy
Signed image
Signed image
Pull the image and validate signature within the production environment
Signature
Local storage of signing server

**Figure 45: Image and signature creation**[4]

First, the image building process is automated using GitHub Action workflows. This approach grants us access to a GitHub Workflow identity token, which GitHub provides for each workflow run. This identity is specifically associated with the corresponding GitHub Action workflow. It includes additional metadata that helps identify the GitHub repository of the workflow, the workflow name, and more.

Using this OpenID Connect token available in the workflow's environment, we can sign the produced image using cosign. Acting as a Sigstore client, cosign will generate an in-memory public/private key pair and request a new short-lived certificate from Fulcio using the OIDC token and the key pair.

Fulcio then provides the certificate to sign the image. Fulcio will append the certificate to an immutable, append-only, cryptographically verifiable certificate transparency (CT) log, allowing for publicly auditable issuance.

[4] Source: RedHat

Given the certificate, cosign will sign the image using the provided certificate and push the signature to the OCI Image Registry, where the image is stored.

The signing event is recorded in a transparency log entry. To achieve this, cosign creates an object containing information allowing signature verification without the (destroyed) private key. The object contains the hash of the artifact, the public key, and the signature. Crucially, this object is timestamped.

The Rekor transparency log "witnesses" the signing event by entering a timestamped entry into the records that attests that the secure signing process has occurred.

**Verifying**

When a software consumer wants to verify the software’s signature, Sigstore compares a tuple of signature, key/certificate, and artifact from the timestamped object against the timestamped Rekor entry.

If they match, it confirms that the signature is valid because the user knows that the expected software creator, whose identity was certified when signing, published the software artifact in their possession.

The entry in Rekor’s immutable transparency log means that the signer will be monitoring the log for occurrences of their identity and will know if there is an unexpected signing event.

**Consuming verified containers**

To ensure that only legitimate container images are deployed in our Kubernetes (k8s) cluster, we can utilize Sigstore's policy controller admission controller. This controller is responsible for enforcing policies that validate the proper signing of images and the presence of verifiable supply-chain metadata. Additionally, the policy controller resolves the image tags to ensure that the image being executed is identical to the one initially admitted.

By verifying each image against the workflow that created it, the policy controller can validate that the image was signed by a workflow deployed by a specific entity and within a specific GitHub repository. This verification process guarantees that the image has not been tampered with since its creation, providing an added layer of security.

##### *4.7.2.2.3 Policy Controller*

The policy controller admission controller[5] can enforce policy on a Kubernetes cluster based on verifiable supply-chain metadata from cosign[6]. It also resolves the image tags to ensure the image being deployed is not different from when it was admitted.

By default, the policy controller admission controller will only validate resources in namespaces chosen to opt-in. This can be done by adding the label *policy.sigstore.dev/include: "true"* to the namespace resource.

[5] https://docs.sigstore.dev/policy-controller/overview/
[6] https://github.com/sigstore/cosign

An image is admitted after it has been validated against all *ClusterImagePolicy* that matched the image's digest and that there was at least one passing authority in each of the matched *ClusterImagePolicy*. Hence, each *ClusterImagePolicy* that matches is AND for admission, and within each *ClusterImagePolicy* authorities are OR.

In addition, the policy controller offers a configurable behaviour defining whether to allow, deny or warn whenever an image does not match a policy. This behaviour can be configured using the config-policy-controller ConfigMap created under the release namespace (by default cosign-system), and by adding an entry with the property no-match-policy and its value *warn|allow|deny*. By default, any image that does not match a policy is rejected whenever no-match-policy is not configured in the ConfigMap.

#### *4.7.2.2.4 ImagePolicyWebhook*

The ImagePolicyWebhook admission controller is an alternative method to statically attest images pulled to the specific node. It is a Kubernetes feature that allows us to enforce policies for image verification at runtime by calling an external webhook that can verify the digital signature of container images.

To use ImagePolicyWebhook, we follow these steps:

- Create a webhook service that can verify the digital signature of container images. The webhook service should be able to receive a request from the ImagePolicyWebhook admission controller and return a response indicating whether the image should be allowed or denied.
- Deploy the webhook service on the cluster.
- Configure the ImagePolicyWebhook admission controller to call the webhook service for image verification. This can be done by adding the ImagePolicyWebhook admission controller to the list of admission controllers in the Kubernetes API server configuration file and specifying the URL for the webhook service.

As mentioned earlier, we use the cosign tool to provide attestable metadata to containers, in order to be used by the ImagePolicy admission controller. Cosign, Image Signing, and ImagePolicy Verification are the three components that make up the Attestation Mechanism for SERRANO.

***ImagePolicy Admission Controller***

The ImagePolicy admission controller acts as a gatekeeper for deploying container images inside our k8s clusters. It does this by enforcing regulations that specify which container images are permitted to operate and are therefore considered genuine.

When a container image is presented for deployment, the ImagePolicy admission controller validates the image by carrying out the following procedures to ensure that it is authentic[7]:

[7] https://cloud.redhat.com/blog/signing-and-verifying-container-images

a. Image Retrieval: The admission controller is responsible for retrieving the container image from the registry or repository that has been defined.
b. Validation of the Signature: The Admission Controller makes use of Cosign in order to validate the embedded cryptographic signature that is contained within the image information. It compares the signature to the associated public key to confirm that the image has not been tampered with and came from a reliable source. This is done to ensure that the signature is valid.
c. Policy Evaluation: The admission controller examines the image in light of previously established policies. These policies may take into consideration aspects such as the image's provenance, the findings of vulnerability scanning, and the requirements for compliance. Our organization's security requirements and industry best practices served as the basis for establishing these rules.
d. Decision Making: After the findings of the signature validation and policy evaluation have been analyzed, the ImagePolicy admission controller will either decide to accept or refuse the deployment of the container image. If the image meets all of the requirements and is able to pass verification, it will be recognized as valid and will be granted permission to run inside of the cluster. In that case, it is rejected, which eliminates any potential threats to security.

We ensure that only trusted and certified container images are deployed in our cluster by using the Cosign software attestation mechanism and combining it with the ImagePolicy admission controller. This strategy improves the safety and integrity of our containerized programs and reduces the likelihood that those applications would run corrupt or modified image files.

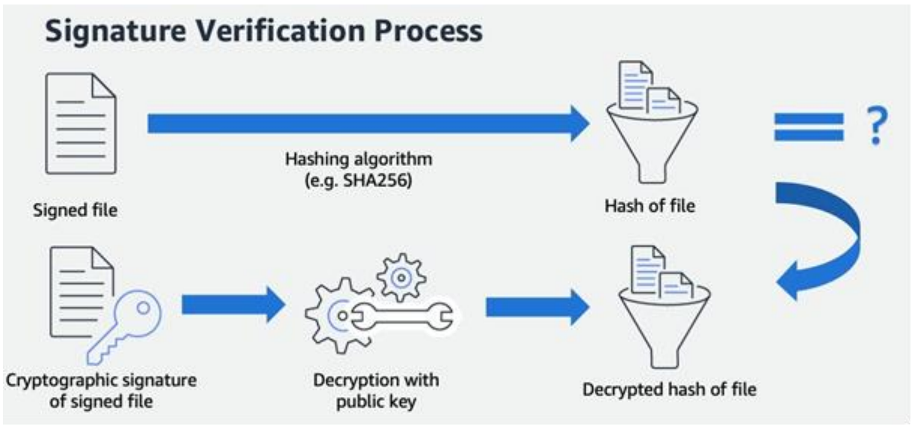

Signature Verification Process
Signed file
Hashing algorithm (e.g. SHA256)
Hash of file
= ?
Cryptographic signature of signed file
Decryption with public key
Decrypted hash of file

**Figure 46: Signature Verification Process8**

We use GitHub's OpenID to provide a third-party authentication service for cosign's keyless mode. Below, we elaborate on how a serverless function is built, packaged as a container, signed, deployed, and verified in the SERRANO platform on a securely booted device.

[8] Image Source: AWS

The example is based on the k-NN function.

**Build container image**

To build the k-NN function we use OpenFaaS template and include vAccel Python bindings and the actual code that calls the k-NN kernel.

```
FROM ghcr.io/openfaas/classic-watchdog:latest as watchdog
FROM harbor.nbfc.io/nubificus/serrano/knn-fpga:x86_64
COPY --from=watchdog /fwatchdog /usr/bin/fwatchdog
RUN chmod +x /usr/bin/fwatchdog
EXPOSE 8080
HEALTHCHECK --interval=3s CMD [ -e /tmp/.lock ] || exit 1

# function
COPY fetch.py /app/fetch.py
# workaround to reduce runtime latency
COPY test.py /app/test.py
RUN python3 test.py

ENV write_debug="true"
ENV debug_headers="true"
ENV marshal_request="true"

ENTRYPOINT ["fwatchdog"]
```

The *"harbor.nbfc.io/nubificus/serrano/knn-fpga:x86_64"* base image is, essentially, a container image with the vAccel frontend library installed, vAccel's Python bindings and the shared objects, including the FPGA, GPU, and CPU plugin code for the k-NN kernel. The function (fetch.py) contains helper functions to parse input, fetch data from the storage backends, and trigger the execution of the kernel. Snippets from this file are shown below:

```
def knn_vaccel(position, labels, input_file, mode):

    timeseries = transformed_time_series(position).astype(np.float32).flatten()
    labels = load_labels(labels).astype(np.int32)
    k = 3
    w = 200
    obj = "libknn_app_%s.so" % mode

    b = np.genfromtxt(io.BytesIO(input_file),delimiter=';')[1:].astype(np.float32)

    # Setup vAccel exec vars
    pa = ffi.cast(f"float[{len(timeseries)}]", ffi.from_buffer(timeseries))
    pb = ffi.cast(f"float[{len(b)}]", ffi.from_buffer(b))
    pc = ffi.cast(f"int[{len(labels)}]", ffi.from_buffer(labels))
    result = bytes(4)

    # Pack arguments
    arg_read_local = [pa, pb, k, w]
    arg_write = [pc, result]

    # execute command
    res        =        Exec_with_resource.exec_with_resource(obj,        "knn_app",
arg_read=arg_read_local, arg_write=arg_write)
    data = struct.unpack('<I', result[:4])[0]
    return data

if __name__ == "__main__":
    for line in sys.stdin:
        if 'Exit' == line.rstrip():
            break

    try:
        request = json.loads(line)
        raw_data = request['body']['raw']
        string = base64.b64decode(raw_data)
        description = json.loads(string)
        desc = json.loads(description)
        storage = desc["storage"]
        creds = desc["creds"]
        if "broker" in storage:
            position,  labels,  input_file  =  fetch_data(creds,  desc["queue_id"],
desc["arguments"])
        elif "s3" in storage:
            position, labels, input_file = fetch_data_s3(creds, desc["queue_id"],
desc["arguments"])
        else:
            print("No storage backend specified")
            sys.exit(1)
        uuid = desc["uuid"]
        mode = desc["mode"]
    except Exception as err:
        print("an error occured")
        print(str(err))
        sys.exit(1)
    data = knn_vaccel(position, labels, input_file, mode)
    data = "timeseries,label\n0,%d\n" % data

    if "broker" in storage:
        push_results_to_databroker(creds, uuid, data)
    elif "s3" in storage:
        push_results_to_s3(creds, desc["queue_id"], uuid, data)
    else:
        print("No storage backend specified")
        sys.exit(1)
```

To build the container image we use a generic command:

```
docker build -f Dockerfile -t harbor.nbfc.io/nubificus/knn-function:generic .
```

To sign the container image we use the following:

```
cosign       sign       --yes       harbor.nbfc.io/nubificus/knn-function@sha256:
72bf63ef9079d7da0dd2e3d4530393bc2316c7370512360108519a3430aae90a\
        -a "ref=60108519a3430aae90a” \
        -a "author=Nubificus LTD"
```

To facilitate the building, signing, and verification of container images, we use GitHub Actions to build and sign while taking advantage of GitHub's OpenID functionality. As a result, the container image is produced using a GitHub Action’s workflow, signed using GitHub’s OpenID, and verified against GitHub when deployed.

The action snippet is show below:

```
    - name: Build and push
      id: build-and-push
      uses: docker/build-push-action@master
      with:
        context: .
        file: ./Dockerfile
        push: true
        tags: ${{ steps.docker_meta.outputs.tags }}
    - name: Sign the published Docker image
      env:
        COSIGN_EXPERIMENTAL: "true"
        DIGEST: ${{ steps.build-and-push.outputs.digest }}
        TAGS: ${{ steps.docker_meta.outputs.tags }}
      # run: echo "${{ steps.meta.outputs.tags }}" | xargs -I {} cosign sign
{}@${{ steps.build-and-push.outputs.digest }}
      run: |
        cosign sign --yes harbor.nbfc.io/nubificus/knn-function@${{steps.build-
and-push.outputs.digest}} \
        -a "repo=${{github.repository}}" \
        -a "workflow=${{github.workflow}}" \
        -a "ref=${{github.sha}}" \
        -a "author=Nubificus LTD"
```

To validate the deployed container image, we enable the policy controller in a specific namespace (serrano-deployments) of our K8s installation:

```
apiVersion: v1
kind: Namespace
metadata:
  labels:
    policy.sigstore.dev/include: "true"
    kubernetes.io/metadata.name: serrano-deployments
  name: serrano-deployments
spec:
  finalizers:
  - kubernetes
```

And we add the policy that verifies the produced (signed) container images:

```
apiVersion: policy.sigstore.dev/v1beta1
kind: ClusterImagePolicy
metadata:
  name: serrano-policy
spec:
  authorities:
  - keyless:
      identities:
      - issuer: https://token.actions.githubusercontent.com
        subjectRegExp: https://github.com/nubificus/.*/.github/workflows/*@*
      url: https://fulcio.sigstore.dev
    name: authority-0
  images:
  - glob: '**'
  mode: enforce
```

## 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.

This deliverable contains the final update of the material previously published in Deliverable 6.3. Most of the additions revolve around features that have been implemented since the publishing of D6.3 to satisfy the requirements of the three use cases. This includes caching, support for HTTP Range headers, multipart uploads and authentication based on AWS Signature V4. The Secure Storage Service has been described in greater detail in Deliverable 2.4 [84] and Deliverable 3.4 [86]. Here, we only include a short excerpt.

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.

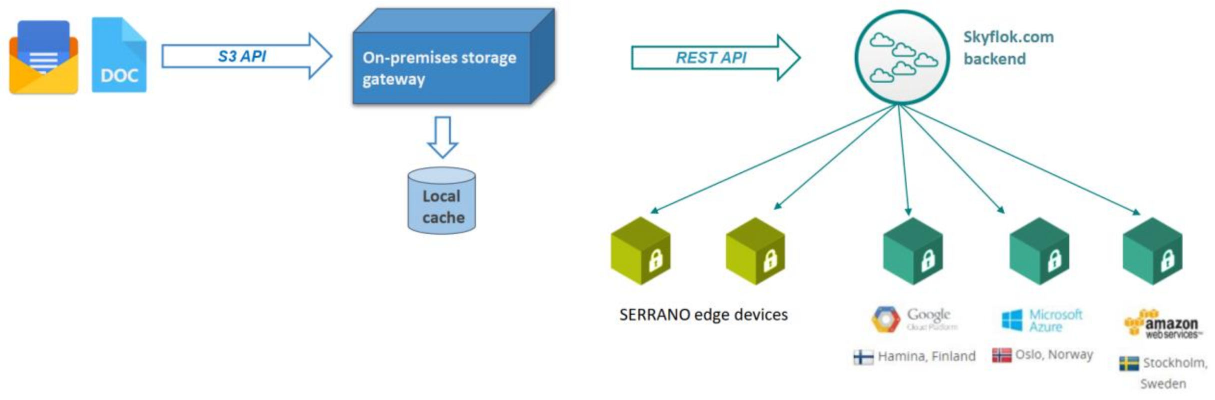


**Figure 47: The components of the Secure Storage Service**

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 [24] seamlessly. This also includes on-premises object stores hosted using Ceph [25], Openstack Swift [26] or MinIO [8]. By supporting the S3 multipart upload and HTTP range query feature, large files can be more quickly uploaded and downloaded. Thirdly, the introduction of an edge cache further increases performance.

#### *4.8.1.1 Inner components*

An overview of how the core components are connected can be seen on Figure 47.

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 3.11 using FastAPI [5], a modern ASGI framework. The Gateway runs as a containerized application and can be deployed using the SERRANO orchestration and deployment 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 into a SERRANO-managed Kubernetes (K8s) [27] cluster. Each is a separate instance of MinIO [7], 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 K8s, MinIO can make use of a wide range of available storage resources through K8s Persistent Volumes [28]. 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 of 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 Data Locations
54 Data Locations • 15 Countries

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 (24 out of 59 cloud locations are GDPR-compliant). 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.

The monitoring of cloud locations requires the ability to reliably schedule measurements that ascertain both the availability of each location and its performance characteristics with regards to uploading and downloading files. The Skyflok.com backend provides the scheduling, and the **Cloud Benchmarker Service** performs the measurements and stores the results. These are exposed through the Gateway's Cloud Telemetry API. An overview of these separate responsibilities is shown below on Figure 48.

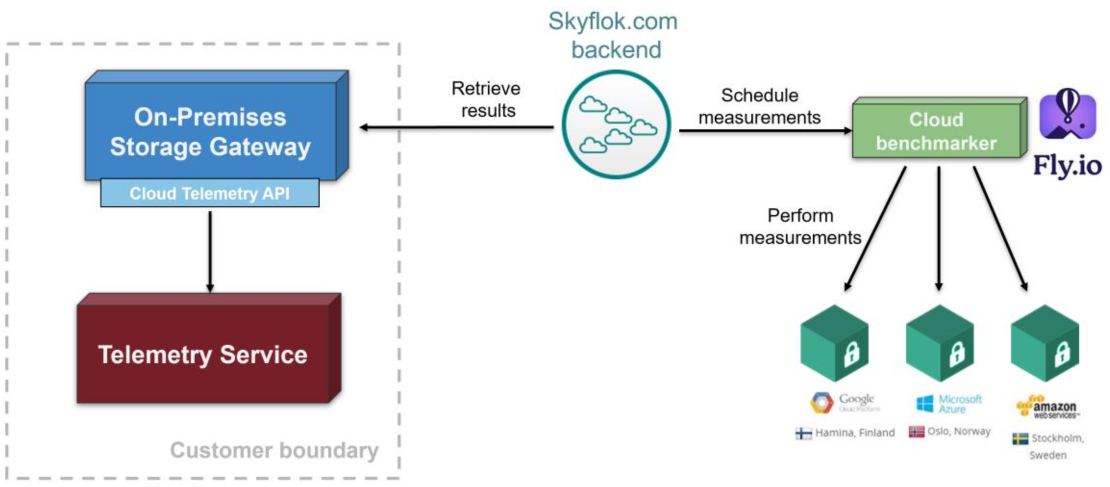
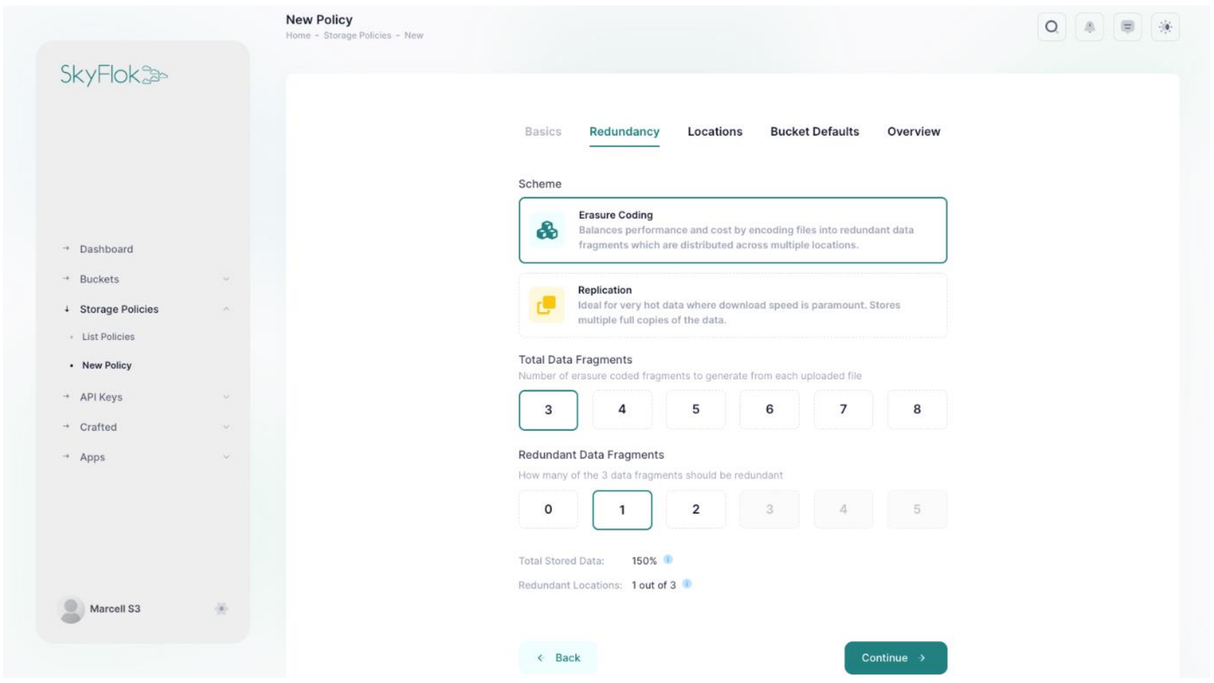


Figure 48: The components of the cloud monitoring features of the SERRANO-enhanced Storage Service.

### 4.8.1.2 Developer web portal

To enhance the usability of the SERRANO-enhanced Storage Service, we created a web portal to cater to the needs of the developers who will use the service. The features have been selected by studying the online interfaces of object storage providers and based on the project's requirements with particular emphasis on the use cases. The developer portal will also play an important role in the exploitation of the project's outcomes.



Scheme

**Erasure Coding**
Balances performance and cost by encoding files into redundant data fragments which are distributed across multiple locations.

**Replication**
Ideal for very hot data where download speed is paramount. Stores multiple full copies of the data.

Total Data Fragments
Number of erasure coded fragments to generate from each uploaded file

[✓ 3] 4 5 6 7 8

Redundant Data Fragments
How many of the 3 data fragments should be redundant

0 [✓ 1] 2 3 4 5

Total Stored Data: 150%
Redundant Locations: 1 out of 3

Figure 49: Developer portal – second step of the new storage policy creation wizard.

Its features revolve around letting developers manage three core entity types: S3 buckets, storage policies, and API keys. Compared to the REST APIs, it presents a friendly, graphical environment suitable for non-technical users as well. Figure 49 shows one frame of the storage policy creation wizard.

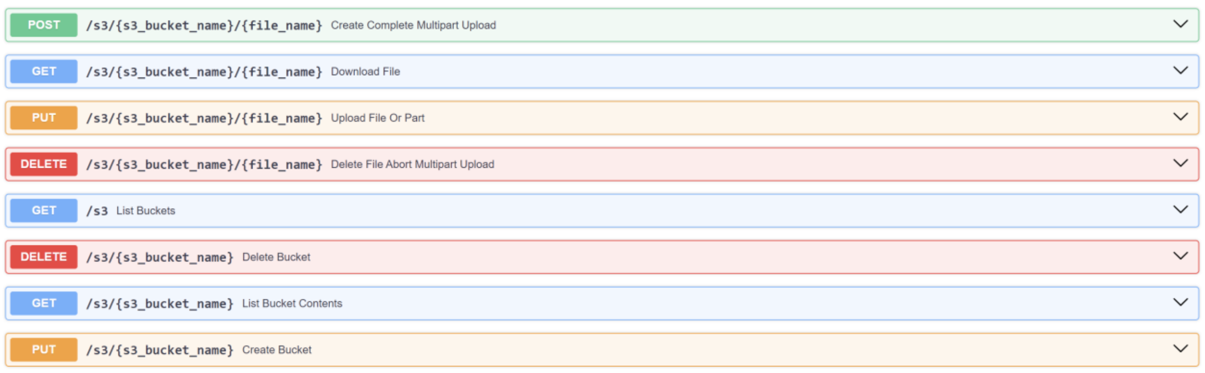
### 4.8.1.3 Integration details and REST APIs

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. In general, object storage semantics are somewhat less permissive than those of file systems. Despite this, cloud storage solutions like Dropbox and Google Drive have shown that it is possible to build a file storage service on top of an object storage system. Indeed, SkyFlok also follows this schema to offer its users a file system that is built on top of object storage.

| | |
|---|---|
| **Host names (s)/Port(s)** | https://on-premise-storage-gateway.services.cloud.ict-serrano.eu/s3 |
| **Publicly accessible (y/n and other details)** | The IP is publicly accessible, and authentication is performed using AWS Signature V4. Chocolate Cloud has made credentials available to all partners who wished to access the service. |
| **Type of API** | REST, XML responses, as defined by AWS S3 |
| **API documentation** | https://github.com/ict-serrano/On-Premise-Storage-Gateway/blob/master/openapi.json, https://on-premise-storage-gateway.services.cloud.ict-serrano.eu/docs |
| **Location of integration tests** | https://github.com/ict-serrano/On-Premise-Storage-Gateway/tree/master/tests |

The Secure Storage API supports all major Create, Read, Update, Delete (CRUD) features of both objects and buckets, with support for both compulsory and most optional parameters. It also supports all features related to multipart uploads. An API reference can be found on Amazon's website [29]. Figure 50 shows the list of supported endpoints, with many URLs serving more than one S3 feature.



```
POST    /s3/{s3_bucket_name}/{file_name}  Create Complete Multipart Upload
GET     /s3/{s3_bucket_name}/{file_name}  Download File
PUT     /s3/{s3_bucket_name}/{file_name}  Upload File Or Part
DELETE  /s3/{s3_bucket_name}/{file_name}  Delete File Abort Multipart Upload
GET     /s3  List Buckets
DELETE  /s3/{s3_bucket_name}  Delete Bucket
GET     /s3/{s3_bucket_name}  List Bucket Contents
PUT     /s3/{s3_bucket_name}  Create Bucket
```

**Figure 50: 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 after the end of the SERRANO project 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.

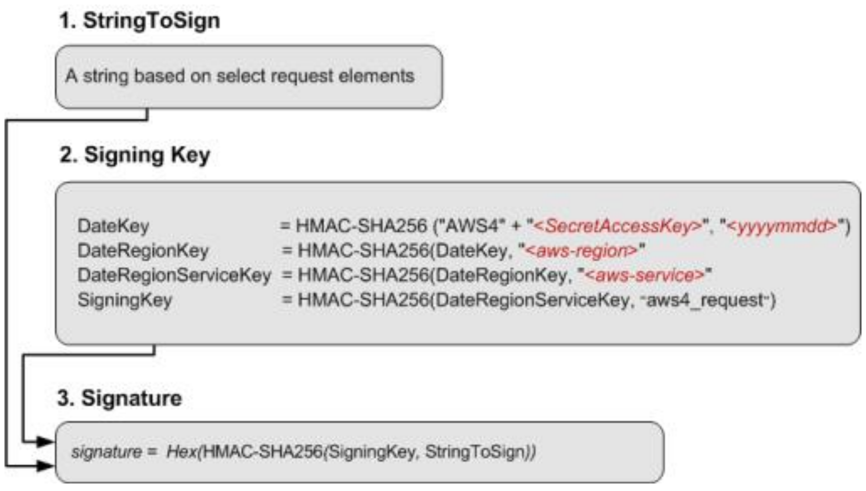

1. StringToSign
A string based on select request elements
2. Signing Key
DateKey = HMAC-SHA256 ("AWS4" + "<SecretAccessKey>", "<yyyymmdd>")
DateRegionKey = HMAC-SHA256(DateKey, "<aws-region>"
DateRegionServiceKey = HMAC-SHA256(DateRegionKey, "<aws-service>"
SigningKey = HMAC-SHA256(DateRegionServiceKey, "aws4_request")
3. Signature
signature = Hex(HMAC-SHA256(SigningKey, StringToSign))

**Figure 51: Overview of Amazon AWS Signature Version 4 process**Error! Bookmark not defined.

Authentication is handled through AWS Signature Version 4, though support for Version 3 may be added in the future if required by our customers. An overview of the process is shown in Figure 51[9]. The headers and body of incoming HTTPS requests are signed to protect their integrity using the client's secret access key. This is also the authentication mechanism of the S3 service.

The Gateway also exposes further REST APIs to offer services to SERRANO platform components. 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.

#### *4.8.1.4 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>PhotoBucket</Name>
    <Prefix/>
    <KeyCount>5</KeyCount>
    <Contents>
        <Key>flowers.bmp</Key>
        <LastModified>2023-12-12T12:22:28.000Z</LastModified>
        <Size>2475410</Size>
    </Contents>
    <Contents>
       ...
    </Contents>
    ...
</ListBucketResult>
```

[9] Source: https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html

## 4.9 Service Assurance and Remediation

### 4.9.1 Description

Performance-related anomaly detection in geographically distributed systems is a very active research topic. It is even more so in the case of edge/cloud heterogeneous systems where performance-related anomalies have a higher impact. In the following section, we will use the term events and anomalies seemingly interchangeably. However, we should note that the methods used for detecting anomalies are applicable in the case of events. The main difference lies in the fact that anomalies pose an additional level of complexity because of their sparse nature. Some anomalies might have an occurrence rate well under 0.01%. Event and anomaly detection can be split up into several categories based on the methods and the characteristics of the available data. The simplest form of anomalies are point anomalies which can be characterised by only one metric (feature). These types of anomalies are fairly easy to detect by applying simple rules (i.e., CPU is above 70%). Other types of anomalies are more complex but ultimately yield a much deeper understanding about the inner workings of a monitored exascale system or application. These types of anomalies are fairly common in complex systems.

Contextual anomalies are extremely interesting in the case of complex systems. These anomalies happen when a certain constellation of feature values is encountered. In isolation, these values are not anomalous but when viewed in context they represent an anomaly. These anomalies represent application bottlenecks, imminent hardware failures, or software misconfigurations. The last major types of relevant anomalies are temporal or sometimes sequential anomalies, where a certain event occurs out of order or at the incorrect time. These anomalies are significant in systems with a strong spatio-temporal relationship between features, which is very much the case for exascale metrics.

#### *4.9.1.1 Architecture*

The Service Assurance and Remediation (SAR) components (Figure 52) are tasked not only with the detection of any performance-related anomalies but also with the analysis of the root cause of the anomalous events along with the notification of other SERRANO components, such as the Resource Orchestrator.

The Event Detection Engine (EDE) is the main component of SAR; it has several sub-components that are based on lambda-type architecture, where we have a speed, batch, and serving layer. Because of the heterogeneous nature of most modern computing systems (including exascale and mesh networks) and the substantial variety of solutions that could constitute a monitoring service, the data ingestion component must contend with fetching data from a plethora of systems. Connectors are implemented such that they serve as adapters for each solution. Furthermore, this component also can load data directly from a static file (HDF5, CSV, JSON, or even raw format).

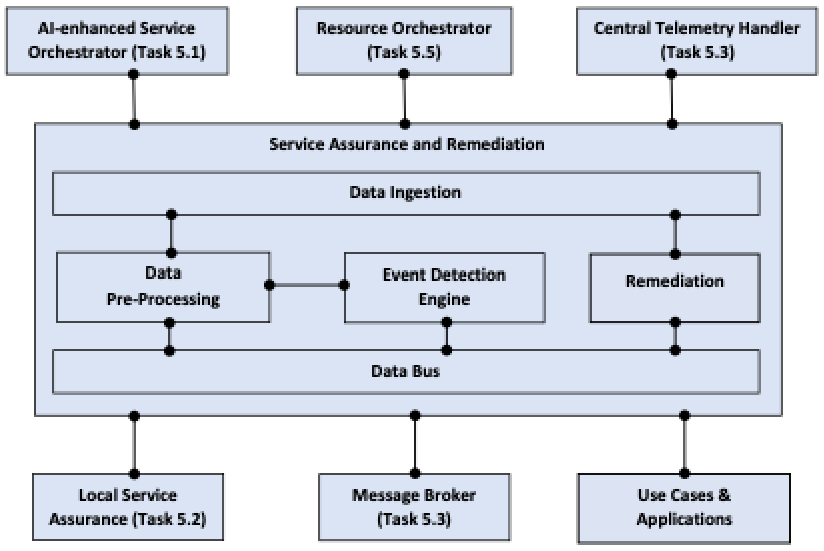

AI-enhanced Service Orchestrator (Task 5.1)
Resource Orchestrator (Task 5.5)
Central Telemetry Handler (Task 5.3)
Service Assurance and Remediation
Data Ingestion
Data Pre-Processing
Event Detection Engine
Remediation
Data Bus
Local Service Assurance (Task 5.2)
Message Broker (Task 5.3)
Use Cases & Applications

**Figure 52: Service Assurance and Remediation general architecture**

This aids in fine-tuning event and anomaly detection methods. We can also see that data ingestion can be done directly via query from the monitoring solution or streamed directly from the queuing service (after ETL if necessary), see Figure 53. This ensures that we have the best chance of reducing the time between the event or anomaly happening and it being detected.

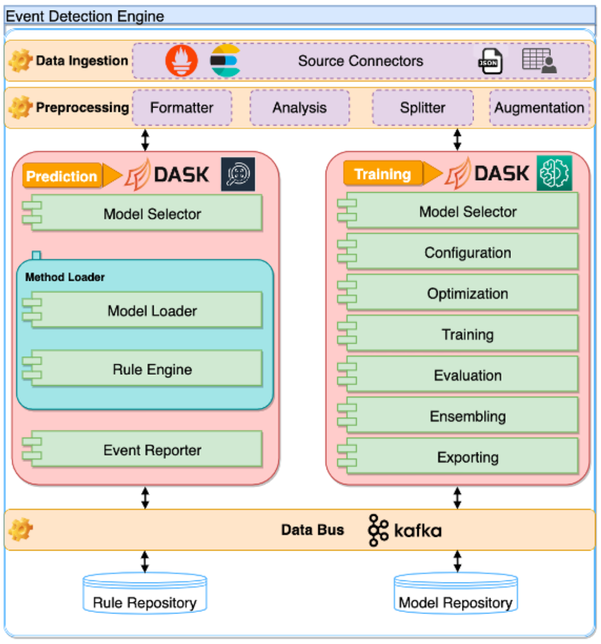


**Figure 53: EDE Architecture**

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 and undersampling).

The training component (batch layer) is used to instantiate and train methods 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 optimised but to define the objectives of the optimization. More specifically, users can define what should be optimised including but not limited to predictive performance, transprecise objectives (inference time, computational limitations, model size etc.).

Evaluation of the created predictive model on a holdout set is also handled in this component. Current research and rankings of machine learning competitions show that creating an ensemble of different methods may yield statistically better results than single model predictions. Because of this, ensemble capabilities have to be included.

Finally, the trained and validated models have to be saved in such a way that enables them to be easily instantiated and used in a production environment. Several predictive model formats have to be supported, such as; PMML, ONNX, HDF5, JSON.

It is important to note at this time that the task of event and anomaly detection can be broadly split into two main types of machine learning tasks; classification and clustering. Classification methods such as Random Forest, Gradient Boosting, Decision Trees, Naive Bayes, Neural Networks, and Deep Neural Networks are widely used in anomaly and event detection. While in the case of clustering, we have methods such as IsolationForest, DBSCAN, and Spectral Clustering. Once a predictive model is trained and validated, it is saved inside a model repository. Each saved model has to have metadata attached to it denoting its performance on the holdout set as well as other relevant information such as size, throughput etc.

The prediction component (speed layer) is in charge of retrieving the predictive model from the model repository and feeding metrics from the monitored system. When an event or anomaly is detected, EDE is responsible for signalling this to both the Monitoring service reporting component and to other tools such as the Resource Orchestrator or any other decision support system.

Once inference is complete, EDE will analyse of each prediction using Shapely values. This allows us to determine the root cause of what features caused any anomalous events. This information, coupled with the timestamp where anomalous events have been detected, will be sent via the SERRANO Data Broker to any SERRANO services that require this information (i.e., Resource Orchestrator).

### 4.9.2 Integration details and REST APIs

SAR-EDE is designed around the utilisation of a YAML-based configuration scheme. This allows the complete configuration of the tool by the end user with limited to no intervention in the source code. It should be mentioned that some of these features are considered unsafe as they allow the execution of arbitrary code.

The configuration file is split up into several categories:

- **Connector** - Deals with connection to the data sources
- **Mode** - Selects the mode of operation for EDE
- **Filter** - Used for applying filtering on the data
- **Augmentation** - User defined augmentations on the data
- **Training** - Settings for training of the selected predictive models
- **Detect** - Settings for the detection using a pre-trained predictive model
- **Point** - Settings for point anomaly detection
- **Misc** - Miscellaneous settings

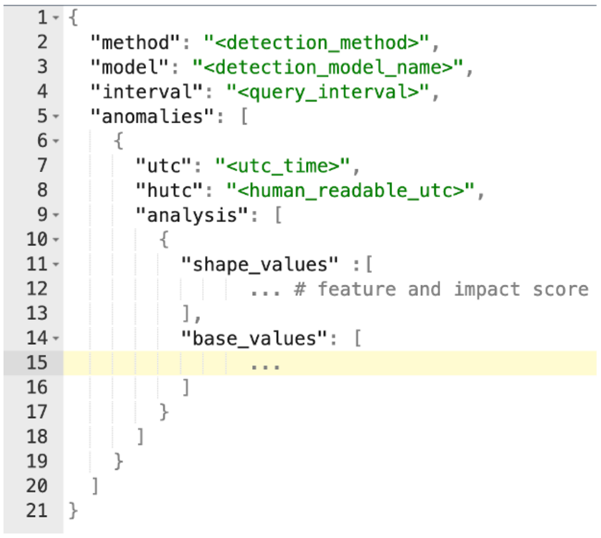
For the sake of brevity, we will not go into detail about all configuration options. The official SERRANO repository[10] contains a comprehensive user guide. Bellow we can find an example configuration:

```
Connector:
  PMDS:
    Endpoint: 'http://pmds.services.cloud.ict-serrano.eu'
    Cluster_id: 7628b895-3a91-4f0c-b0b7-033eab309891
    Start: '-2h'
    End: ''
    Groups:
      - general
      - cpu
      - memory
      - network
      - storage
    Namespace: uvt-aspataru
  Dask:
    SchedulerEndpoint: local
    Scale: 3
    SchedulerPort: 8787
    EnforceCheck: false
  KafkaEndpoint: <Serrano Message Broker>
  KafkaPort: 9092
  KafkaTopic: edetopic
  GrafanaUrl: 'http://85.120.206.26:32000'
  GrafanaToken: <token>
  GrafanaTag: ede_test
  MetricsInterval: 1m
  QSize: 0
  Index: time
  QDelay: 10s
Augmentation:
  Scaler:
    StandardScaler:
      copy: true
Mode:
  Training: true
  Validate: false
  Detect: true
Training:
  Type: clustering
```

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

```
  Method: isoforest
  Export: sr_isolationforest_1
  MethodSettings:
    n_estimators: 100
    max_samples: 10
    contamination: 0.07
    verbose: true
    bootstrap: true
Detect:
  Method: isoforest
  Type: clustering
  Load: sr_isolationforest_1
  Scaler: StandardScaler
  Analysis:
    Plot: true
```

The above configuration sets EDE up by defining the PMDS endpoint, query, DataBroker, Grafana, and Dask Cluster. Next, we select the predictive model training options, including data augmentation (standard scaler). Once training is complete, the predictive model, trained using IsolationForest, is instantiated and executed based on user options. The output of the analysis has the following form:



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

The above analysis results will be pushed to a particular Kafka topic in the SERRANO Data Broker, where any other SERRANO service, including the Resource Orchestrator, can consume it. A complete overview of the integration details can be found in Table 13.

**Table 13: Integration details of SAR-EDE**

| | |
|---|---|
| **IP(s)/Port(s)** | EDE Inference Service:<br>● https://sar.ede.services.cloud.ict-serrano.eu |
| **Publicly accessible (y/n and other details)** | The service is publicly accessible when it comes to inference, while training and validation should be done offline. |
| **Type of API** | REST and CLI. |
| **Associated host names** | https://sar.ede.services.cloud.ict-serrano.eu |

| | |
|---|---|
| **API documentation** | *https://github.com/ict-serrano/service-assurance-ede* |
| **Location of integration tests** | *https://serrano-sonarqube.rid-intrasoft.eu/dashboard?id=service-assurance-ede* |

### *4.9.2.1 REST-API*

EDE is also accessible as a REST service. We should note that training is not possible using this service, only inference. This was done mainly because of the difficulty in training and validating predictive models for anomaly detection and the requirement for large scale historical data.

The REST service provides the capability to select pre-trained models. It uses a loosely coupled asynchronous control architecture. Users can start background workers capable of performing jobs pushed into a distributed queue (currently Redis). Each job represents an EDE instance running in prediction mode. These instances can be queried via the REST API, where each background worker connects to the EDE instance and reports the current status. This way, even if the REST service fails, all background jobs can still run and can be reattached after the service restart.

In the following paragraphs, we detail the complete REST API of this service as it is the primary interaction mechanism between SAR-EDE and other SERRANO components.

```
GET /v1/config
```

Returns the current version of the configuration file.

```
PUT /v1/config
```

Uploads a new configuration file in YAML format. See EDE Configuration[11] for more details.

```
GET /v1/config/augmentation
```

```
{
  "Scaler": {
    "StandardScaler": {
      "copy": true,
      "with_mean": true,
      "with_std": true
    }
  }
}
```

Returns the current augmentation configuration.

[11] https://github.com/ict-serrano/service-assurance-ede#utilization

```
PUT /v1/config/augmentation
```

Modifies the augmentation part of the configuration.

```
GET /v1/config/connector

{
  "Dask": {
    "EnforceCheck": false,
    "Scale": 3,
    "SchedulerEndpoint": "local",
    "SchedulerPort": 8787
  },
  "Index": "time",
  "KafkaEndpoint": "10.9.8.136",
  "KafkaPort": 9092,
  "KafkaTopic": "edetopic",
  "MPort": 9200,
  "MetricsInterval": "1m",
  "PREndpoint": "194.102.62.155",
  "QDelay": "10s",
  "QSize": 0,
  "Query": {
    "query": "{__name__=~\"node.+\"}[1m]"
  }
}
```

Returns the current connector configuration.

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

Modifies connector part of the configuration.

```
GET /v1/config/filter
```

```
{
  "DColumns": {
    "Dlist": "..data/low_variance.yaml"
  },
  "Dropna": true,
  "Fillna": true
}
```

Returns the current filter configuration.

```
PUT /v1/config/filter
```

Modifies the filter part of the configuration.

```
GET /v1/config/inference
```

```
{
  "Analysis": {
    "Plot": true
  },
  "Load": "cluster_y2_v3",
  "Method": "IForest",
  "Scaler": "StandardScaler",
  "Type": "clustering"
}
```

Return current inference configuration.

```
PUT /v1/config/inference
```

Modifies inference part of the configuration.

```
GET /v1/data
```

```
{
  "files": [
    "serrano_test_cluster.csv"
  ]
}
```

Returns a list of local datafiles. Currently only, txt, csv, xlsx and json files are supported.

```
GET /v1/data/{data_file}
```

This resource fetches the datafile denoted by the *data_file* parameter.

```
PUT /v1/data/{data_file}
```

This resource allows external files to be uploaded to the EDE service. Currently only, txt, csv, xlsx and json files are supported. We should note that the *data_file* parameter must be the same as the name of the file being uploaded.

```
POST /v1/inference
```

Starts the inference job using EDE based on the current configuration file.

```
GET /v1/logs
```

Returns EDE Service logs.

The following REST resources are used to control Redis Queue (RQ) workers that wrap individual EDE instances. Each EDE instance can use a DASK cluster (local or remote).

```
GET /v1/service/jobs
```

```
{
    "failed": [],
    "finished": [
        "a9784914-165c-488a-b5a6-7c58c6b421e6"
    ],
    "queued": [],
    "started": []
}
```

Returns information about jobs from the services. It contains the unique ids for 4 types of jobs; failed, finished, queued, started.

An example response can be seen bellow:

```
GET /v1/service/jobs/{job_id}
```

```
{
    "finished": true,
    "meta": {
        "progress": "Finished inference"
    },
    "status": "finished"
}
```

Returns information about a specific job denoted by its unique id. Some meta information is also contained in the response as reported by the background process. This resource can be used to check the current status of background jobs.

```
GET /v1/service/jobs/worker
```

```
{
    "workers": [
        {
            "id": "e3b0c442-98fc-11e7-8f38-2b66f5e7a637",
            "pid": 1,
            "status": "idle"
        },
    ]
}
```

Returns a list of workers from the current service instance. The list also includes workers who are no longer active; see the status from the response. Other information about the workers are their unique ID and PID from the operating system.

```
POST /v1/service/jobs/worker
```

```
{
  "status": "workers started"
}
```

A background worker will be started every time a POST request is issued to this resource. The maximum number of workers depends on the number of physical CPU cores available and a modifier set as an environment variable called WORKER_THREASHOLD, which is set to 2 by default.

If the maximum number of workers has been reached the following response will be given:

```
{
  "warning": "maximum number of workers active!",
  "workers": 4
}
```

```
DELETE /v1/service/jobs/worker
```

This resource enables the halting of workers. This resource needs to be accessed each time a worker needs to be stopped.

Figure 54 presents the interaction between SAR and other SERRANO components. A typical workflow will entail setting up EDE to analyse the monitoring data from the SERRANO telemetry framework. Each anomalous event detected will be sent to a specialised Kafka topic inside the Stream Handler.

From there, services such as the Resource Orchestrator can fetch the detected anomalies together with the root cause analysis results represented by feature ranking computed using Shapely values. This will allow for remediation actions to be taken as the SERRANO telemetry framework names each metric/feature in such a way that the exact location can be easily determined.

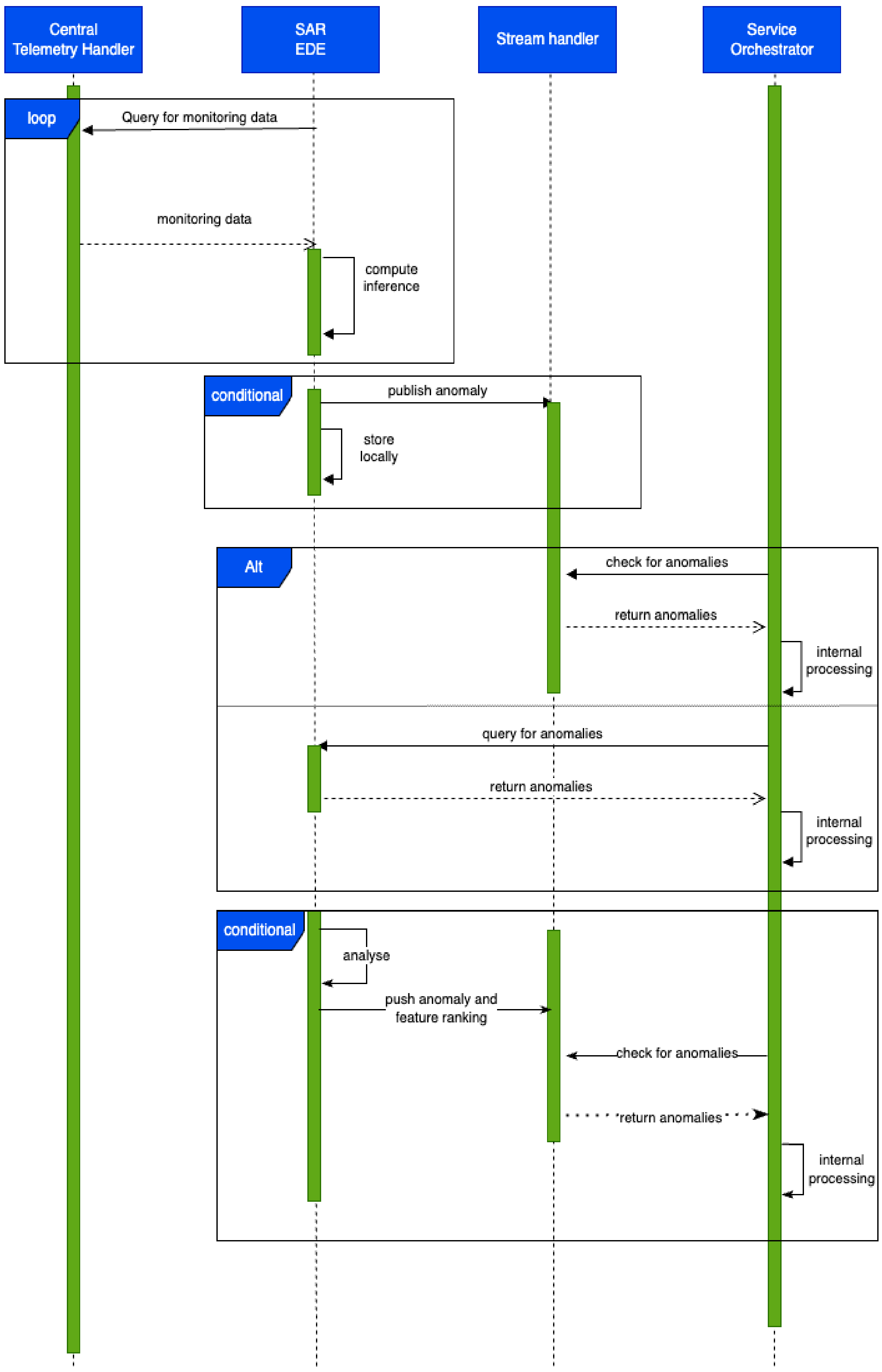


**Figure 54: Sequence Diagram of SAR Interactions**

## 4.10 Secure Storage Use Case Integrated Functionality

The Secure Storage Use Case has been described in detail in Deliverable 2.4. Here, we only included a brief description extracted from the aforementioned document, focusing on its integration of SERRANO platform features and services. The material in this section is an update of the version previously submitted in Deliverable 6.3. The changes from D6.3 are mostly technical in nature as the general direction of the Use case has not changed. We have added more details on how the integration with the different SERRANO platform services has been performed. The evaluation of the use case is presented in Deliverable 6.8.

This UC focuses on providing secure and high-performance storage of files with lower latency than a purely cloud-based approach. We have achieved 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 involves the SERRANO platform's orchestration and telemetry services in two meaningful ways. First, the SERRANO telemetry mechanism collects data from the edge storage locations regarding their status, availability, cost, latency, etc. Similar data is collected about cloud storage resources, published by the On-premises storage gateway (Gateway henceforth). Second, the UC relies on the orchestration mechanisms to deploy the client applications that will utilize the storage services. The SERRANO orchestration mechanisms receive the user intent, which is then translated into a storage policy describing how and where the application's data should be stored. The translation is done using algorithms implemented by the Resource Optimization Toolkit, utilizing telemetry data regarding cloud and edge storage locations. The application can then use the automatically created storage policy when creating an S3 bucket in a manner similar to Amazon Web Services' *LocationConstraint* mechanism.

The Gateway is a performance-critical component, given its role in processing file operations. Hence, the designed solution uses 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 reduces some of the load on the CPU and may increase the number of concurrently supported connections when the CPU is the bottleneck. In addition, it uses FPGAs to accelerate erasure coding. This leads to further CPU offloading as well as a reduction of processing time. We should note that as part of WP4, a GPU-accelerated version of the encryption algorithm has also been developed. We have not included this in the Gateway due to its somewhat limited practical benefit.

### 4.10.1 Integration details and REST APIs

The Storage Policy API allows the platform's users and the SERRANO Resource Orchestrator to manage storage policies. Figure 55 presents the list of supported endpoints that allow creating, modifying, deleting, listing, and retrieving storage policies. A storage policy is like a recipe. It is the result of the translation of an application's storage task's requirements to a concrete storage resource allocation. Storage policies can also be created directly using the REST API for more advanced service users. Each S3 bucket has a storage policy applied to it. Thus, application developers can freely create several types of policies and buckets depending on their requirements. For example, an analytics application that processes a large amount of ephemeral data may have a non-encrypted, low redundancy fast edge-based policy for its inputs and an encrypted, cloud-based high redundancy policy for its outputs.

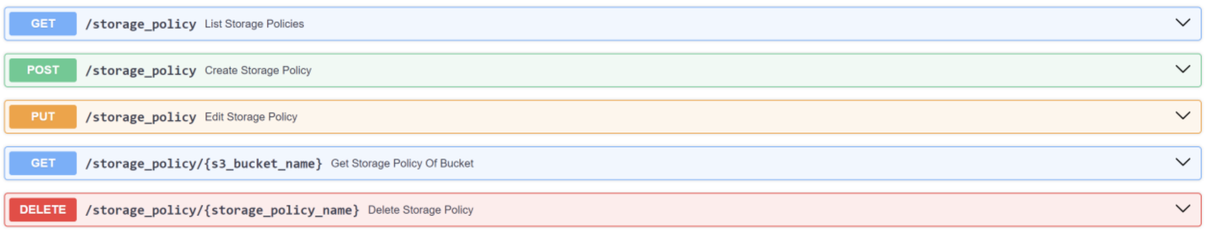

GET /storage_policy List Storage Policies
POST /storage_policy Create Storage Policy
PUT /storage_policy Edit Storage Policy
GET /storage_policy/{s3_bucket_name} Get Storage Policy Of Bucket
DELETE /storage_policy/{storage_policy_name} Delete Storage Policy

**Figure 55: Storage Policy API REST endpoints**

The ARDIA framework developed in Work Package 5 contains both the definitions of the Application Model used to express the intent and the Unified Resource Model used to express the characteristics of the storage resources.

| | |
|---|---|
| **IP(s)/Port(s)** | https://on-premise-storage-gateway.services.cloud.ict-serrano.eu/storage_policy |
| **Publicly accessible (y/n and other details)** | The IP is publicly accessible, and authentication is performed using the same credentials as the Secure Storage API. Chocolate Cloud has made credentials available to all partners who wished to access the service. |
| **Type of API** | REST, JSON requests and responses |
| **API documentation** | https://github.com/ict-serrano/On-Premise-Storage-Gateway/blob/master/openapi.json, https://on-premise-storage-gateway.services.cloud.ict-serrano.eu/docs |

Figure 56 presents an example of a storage policy describing which storage locations to use (referred to in the JSON request as *backends* for cloud storage locations and *edge_devices* for SERRANO edge devices) and what encryption and erasure coding schemas to apply. This example shows a hybrid policy that utilizes 3 SERRANO edge devices deployed to the UVT K8s cluster and a single cloud location in a 3+1 erasure-coded configuration. For encryption, AES-256 is used. Compression is performed using the DEFLATE algorithm at level 7.

```
storage_policy = {
  "name": "Hybrid-storage-policy",
  "description": "This is a hybrid storage policy that utilizes 3 SERRANO edge
      devices deployed to the UVT K8s cluster and a single cloud location, in a
      3+1 erasure coded configuration. AES-256 encryption and DEFLATE compression
      are configured.",
  "edge_devices": [1,2,3],
  "backends": [144],
  "redundancy": {
     "scheme": "RLNC",
     "redundant_packets": 1
  },
  "compression": {
     "scheme": "DEFLATE”,
     "level": 7
  },
  "encryption": {
     "scheme": "AES",
     "key_size": 256,
     "block_mode": "GCM"
  }
}
```

**Figure 56: 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 (through the Telemetry Service) 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. Figure 57 shows the two endpoints used to list cloud and edge storage locations.

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

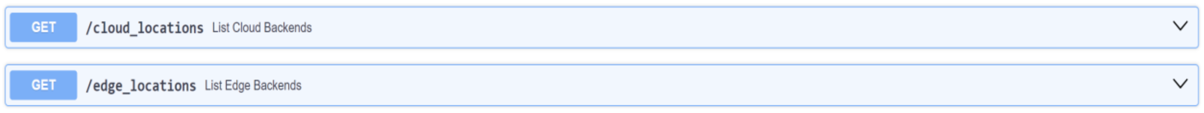
- provider name: Google, Amazon, ….
- geographic location – GPS coordinates
- country/city
- GDPR compliance
- storage cost in $ / GB / month
- ingress cost in $ / GB
- egress cost in $ / GB

Several dynamically measured parameters have also been added:

- upload errors in the last 12 months,
- download errors in the last 12 months,
- time taken to download 1B, used to estimate read latency,
- time taken to upload 1B, used to estimate write latency,
- time taken to download 1MB, used to estimate read throughput
- time taken to upload 1MB, used to estimate write throughput.

The dynamic parameters are measured using a custom solution developed as part of SERRANO and described in greater detail in Section 4.8.1.1. The measurement results are the same as those published to the general public on Chocolate Cloud's website https://www.skyflok.com/backend-performance/.

| | |
|---|---|
| **IP(s)/Port(s)** | https://on-premise-storage-gateway.services.cloud.ict-serrano.eu/ |
| **Publicly accessible (y/n and other details)** | The IP is publicly accessible, no authentication is performed. |
| **Type of API** | REST, JSON requests and responses |
| **API documentation** | https://github.com/ict-serrano/On-Premise-Storage-Gateway/blob/master/openapi.json, https://on-premise-storage-gateway.services.cloud.ict-serrano.eu/docs |


GET /cloud_locations List Cloud Backends

GET /edge_locations List Edge Backends

**Figure 57: Resource and Telemetry API (exposed by On-premises Storage Gateway) REST endpoints**

A second endpoint lists the static characteristics of SERRANO edge devices. These are used by the AI-enhanced Service Orchestrator and Resource Orchestrator to distinguish between the different SERRANO edge devices they manage in the K8s clusters they are deployed in:

- unique identifier used in storage policies,
- name,
- description,
- cluster identifier,
- storage URL of S3 endpoints,
- team identifier,
- S3 region.

The SERRANO telemetry framework monitors the dynamic characteristics of these edge resources. The custom MinIO instances that run the SERRANO edge devices expose these characteristics through a Prometheus-compatible API. A wide range of metrics are collected, including free capacity, storage use, number of objects stored, health metrics, and error rates. It is also possible to set up alerts whenever an abnormal event occurs.

| | |
|---|---|
| **IP(s)/Port(s)** | Accessed withing K8s cluster through a domain name that is specified by a StatefulSet. A suffix of /minio/metrics/v2 is used. |
| **Publicly accessible (y/n and other details)** | The MinIO instances are only accessible within the K8S cluster. Monitoring information is available to all services within the cluster but can easily be restricted as needed. |
| **Type of API** | REST, JSON requests and responses. Message format defined by Prometheus: https://prometheus.io/docs/prometheus/latest/querying/api/ |
| **API documentation** | https://min.io/docs/minio/linux/operations/monitoring/collect-minio-metrics-using-prometheus.html<br>https://github.com/minio/minio/blob/master/docs/metrics/prometheus/README.md |

#### 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/
```

```
[
    {
        “redundancy”: {
            “redundant_packets”: 1,
            “scheme”: “RLNC”
        },
        “description”: “Used to run measurements for D6.5. Edge-only.”,
        “team_id”: 535,
        “compression”: null,
        “encryption”: {
            “block_mode”: “GCM”,
            “key_size”: 256,
            “scheme”: “AES”
        },
        “edge_devices”: [
            4,
            6,
            7,
            8
        ],
        “name”: “measurements-edge-only”,
        “id”: 6216908948897792
    },
    {
        "redundancy": {
            "redundant_packets": 1,
            "scheme": "RLNC"
        },
        "description": "Used to run measurements for D6.5. Cloud-only.",
        "team_id": 535,
        "compression": null,
        "encryption": {
            "block_mode": "GCM",
            "key_size": 256,
            "scheme": "AES"
        },
        "backends": [
            22,
            112,
            39,
            125
        ],
        "name": "measurements-cloud-only",
        "id": 6238940956721152
    }
…
]
```

We also provide a sample for fetching the cloud locations.

Request:

```
GET /cloud_locations/
```

```
[
   {
      "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/,
      "upload_errors_in_last_12_months": 0,
      "download_errors_in_last_12_months": 0,
      "rtt_upload_1B_ms": [252, 248, … 230],
      "rtt_upload_1MB_ms": [1556, 2167, …1474],
      "rtt_download_1B_ms": [618, 652, … 626],
      "rtt_download_1MB_ms": [1160, 1214, … 1162]
   },
 …
   {
      "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/,
      “upload_errors_in_last_12_months”: 0,
      “download_errors_in_last_12_months”: 0,
      “rtt_upload_1B_ms”: [112, 108, … 209],
      “rtt_upload_1MB_ms”: [450, 595, … 694],
      “rtt_download_1B_ms”: [100, 117, … 139],
      “rtt_download_1MB_ms”: [248, 252, … 349]
   }
]
```

#### *4.10.1.2 Integration with acceleration features developed in WP3 and WP4*

Whenever a Nvidia Data Processing Unit (DPU) is available, the On-premises Storage Gateway can perform the TLS encryption directly on this resource for outgoing connections. This CPU-offloading technique increases the Secure Storage Service’s performance in scenarios with a CPU bottleneck. Integration has been achieved by building a custom version of the OpenSSL 3.0.0 library. This library is loaded into the container and automatically detects the available HW resources. If an appropriate DPU is detected and HW TLS offloading is enabled, computations related to TLS encryption are performed on the DPU. To more accurately measure and better separate CPU load associated with TLS encryption, the OpenSSL library is loaded by nginx, rather than the Gateway’s Python application. Nginx acts as a TLS termination proxy, forwarding incoming HTTPS requests to the Python application as HTTP requests through the host's loopback interface.

Figure 58 shows an overview of the deployment used to evaluate the integration. Host machine 512 has the Gateway in two flavours: one with SSL and a baseline one with no SSL, both running inside Docker containers[12]. Host machine 513 represents the clients and is able to simulate a large number of parallel requests. Both machines are equipped with an Nvidia DPU.

[12] Dockerfiles used to run the Gateway with TLS offloading and without: https://github.com/ict-serrano/On-Premise-Storage-Gateway/blob/master/Dockerfile_custom_openssl_kTLS
https://github.com/ict-serrano/On-Premise-Storage-Gateway/blob/master/Dockerfile_custom_openssl_noTLS

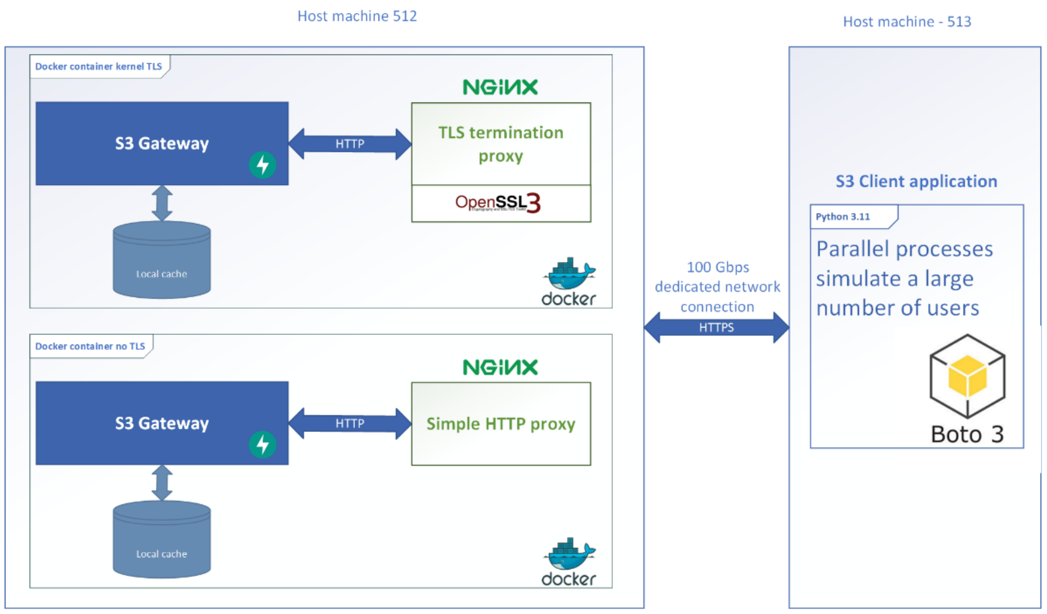


**Figure 58: Overview of the integration of TLS offloading into the On-premises Storage Gateway**

The integration of the TLS encryption is done by utilizing 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 to the developer, who, simply by employing the provided libraries, can leverage the advantages of TLS offloading.

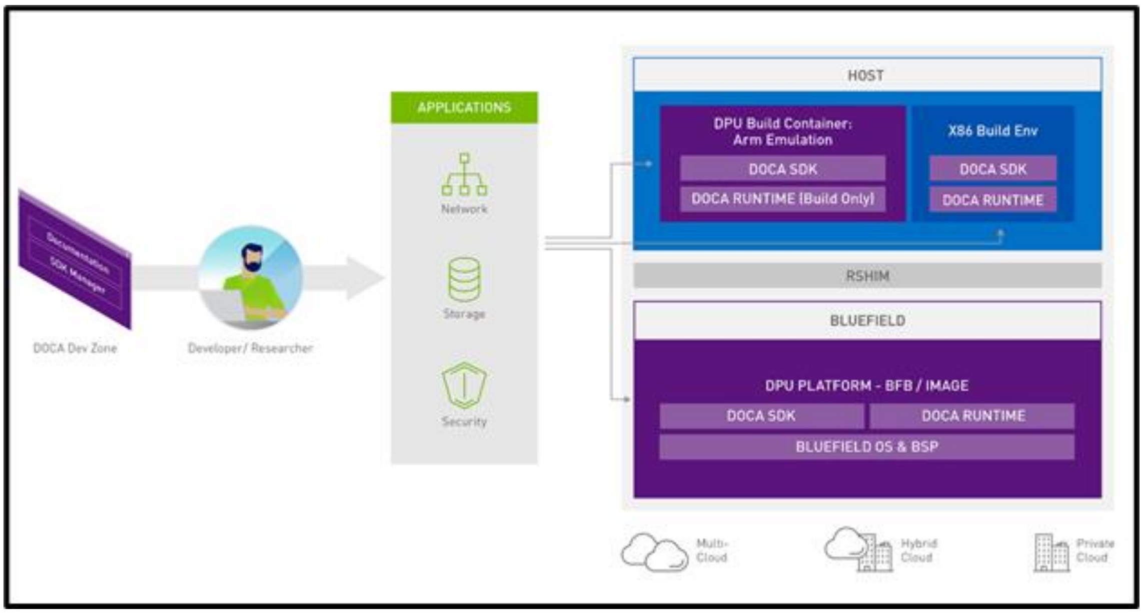


**Figure 59: DOCA DPU utilization in SERRANO**

The use case involves the movement of a large amount of data, in the form of uploaded and downloaded files. The Gateway needs 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 employs the techniques developed as part of Work Package 4. The integration of FPGA-accelerated erasure coding has been achieved through a pluggable dynamically loaded software library[13]. To be able to call the library from the Gateway's Python code, a set of bindings has been developed using ctypes[14]. This provides a very efficient solution that avoids unnecessary copying of data between the two language environments by using C compatible data types. Given the technical limitations of the Xilinx runtime, we have created a custom Docker image[15].

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 is 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 deployed in the SERRANO's infrastructure. Additional details regarding the acceleration of the algorithms of this use case can be found in D4.1. GPU-accelerated encryption algorithms have not been integrated with the Gateway. Given that the UC is I/O heavy and modern CPUs have dedicated functionality related to AES encryption, they have limited practical benefit in the specific context of the UC.

#### 4.10.1.3 *Interfaces created to aid in evaluating the use case*

Throughout the development of the use case, we strived to make its features work seamlessly and transparently for the end user. However, to aid in the automation of measurements and to provide low-level remote control of features such as TLS offloading, we have also added a set of REST endpoints, as shown in Figure 60. The Gateway's code has been instrumented to measure the time taken by different processes in the file upload and download workflows. It also interacts with the underlying OS through a series of Bash scripts that make it possible, for example, to change the number of nginx workers, write and read measurement data to and from local storage, change whether the kernel TLS HW TLS-offloading is enabled on the Network Interface Card. We have also added a pair of endpoints to select whether a CPU-based or an FPGA-based erasure coded library should be loaded dynamically.

| | |
|---|---|
| **IP(s)/Port(s)** | https://on-premise-storage-gateway.services.cloud.ict-serrano.eu/ |
| **Publicly accessible (y/n and other details)** | The IP is publicly accessible, no authentication is performed. |

[13] FPGA-accelerated erasure coding library used by the Gateway: https://github.com/ict-serrano/On-Premise-Storage-Gateway/tree/master/auth_erasure_coding_library
[14] Python bindings using ctypes: https://docs.python.org/3/library/ctypes.html
[15] Dockerfile used to run the Gateway with FPGA-accelerated erasure coding: https://github.com/ict-serrano/On-Premise-Storage-Gateway/blob/master/Dockerfile_AUTH

| | |
|---|---|
| **Type of API** | REST, simple text-based requests and responses, some measurement results as CSV files |
| **API documentation** | https://github.com/ict-serrano/On-Premise-Storage-Gateway/blob/master/openapi.json, https://on-premise-storage-gateway.services.cloud.ict-serrano.eu/docs |

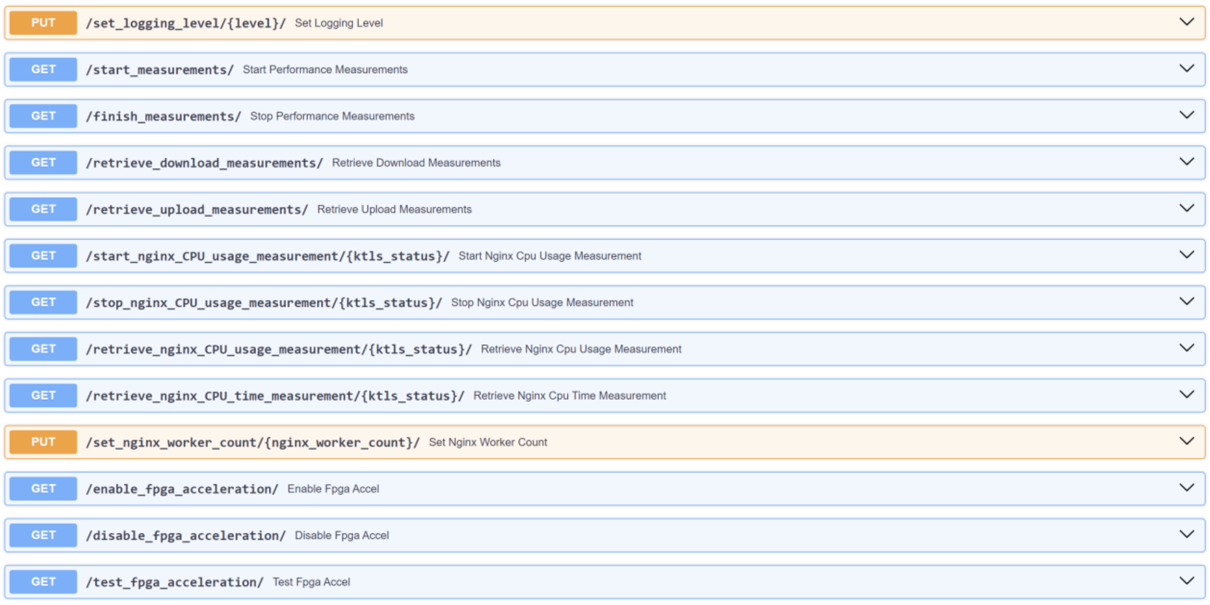

PUT /set_logging_level/{level}/ Set Logging Level
GET /start_measurements/ Start Performance Measurements
GET /finish_measurements/ Stop Performance Measurements
GET /retrieve_download_measurements/ Retrieve Download Measurements
GET /retrieve_upload_measurements/ Retrieve Upload Measurements
GET /start_nginx_CPU_usage_measurement/{ktls_status}/ Start Nginx Cpu Usage Measurement
GET /stop_nginx_CPU_usage_measurement/{ktls_status}/ Stop Nginx Cpu Usage Measurement
GET /retrieve_nginx_CPU_usage_measurement/{ktls_status}/ Retrieve Nginx Cpu Usage Measurement
GET /retrieve_nginx_CPU_time_measurement/{ktls_status}/ Retrieve Nginx Cpu Time Measurement
PUT /set_nginx_worker_count/{nginx_worker_count}/ Set Nginx Worker Count
GET /enable_fpga_acceleration/ Enable Fpga Accel
GET /disable_fpga_acceleration/ Disable Fpga Accel
GET /test_fpga_acceleration/ Test Fpga Accel

**Figure 60: List of REST endpoints used in the evaluation of the Secure Storage Use Case**

Results of the integrations as well as a detailed description of the measurement setups can be found in Deliverable 6.8 (M36).

We provide a sample request-response for retrieving the CPU time utilized by the nginx workers.

Request:

```
GET /storage_policy/
```

Response:

```
9647 (nginx) S 40 40 40 0 -1 4194624 237 0 0 0 0 0 0 0 0 -20 1 0 650130789 1123532
8 781 18446744073709551615 94616585252864 94616589957457 140732034551456 0 0 0 0 1
073745920 402745863 1 0 0 17 13 0 0 0 0 0 94616590894512 94616591263776 9461662130
5856 140732034554756 140732034554762 140732034554762 140732034555881 0
9646 (nginx) S 40 40 40 0 -1 4194624 378 0 0 0 2 12 0 0 0 -20 1 0 650130789 112353
28 1563 18446744073709551615 94616585252864 94616589957457 140732034551456 0 0 0 0
1073745920 402745863 1 0 0 17 19 0 0 0 0 0 94616590894512 94616591263776 946166213
05856 140732034554756 140732034554762 140732034554762 140732034555881 0
9645 (nginx) S 40 40 40 0 -1 4194624 237 0 0 0 0 0 0 0 0 -20 1 0 650130789 1123532
8 781 18446744073709551615 94616585252864 94616589957457 140732034551456 0 0 0 0 1
073745920 402745863 1 0 0 17 14 0 0 0 0 0 94616590894512 94616591263776 9461662130
5856 140732034554756 140732034554762 140732034554762 140732034555881 0
9644 (nginx) S 40 40 40 0 -1 4194624 451 0 0 0 16 47 0 0 0 -20 1 0 650130789 11526
144 1774 18446744073709551615 94616585252864 94616589957457 140732034551456 0 0 0
```

```
0 1073745920 402745863 1 0 0 17 26 0 0 0 0 0 94616590894512 94616591263776 9461662
1305856 140732034554756 140732034554762 140732034554762 140732034555881 0
9647 (nginx) S 40 40 40 0 -1 4194624 239 0 0 0 0 0 0 0 0 -20 1 0 650130789 1123532
8 781 18446744073709551615 94616585252864 94616589957457 140732034551456 0 0 0 0 1
073745920 402745863 1 0 0 17 13 0 0 0 0 0 94616590894512 94616591263776 9461662130
5856 140732034554756 140732034554762 140732034554762 140732034555881 0
9646 (nginx) S 40 40 40 0 -1 4194624 378 0 0 0 2 12 0 0 0 -20 1 0 650130789 112353
28 1563 18446744073709551615 94616585252864 94616589957457 140732034551456 0 0 0 0
1073745920 402745863 1 0 0 17 50 0 0 0 0 0 94616590894512 94616591263776 946166213
05856 140732034554756 140732034554762 140732034554762 140732034555881 0
9645 (nginx) S 40 40 40 0 -1 4194624 239 0 0 0 0 0 0 0 0 -20 1 0 650130789 1123532
8 781 18446744073709551615 94616585252864 94616589957457 140732034551456 0 0 0 0 1
073745920 402745863 1 0 0 17 46 0 0 0 0 0 94616590894512 94616591263776 9461662130
5856 140732034554756 140732034554762 140732034554762 140732034555881 0
9644 (nginx) S 40 40 40 0 -1 4194624 470 0 0 0 34 125 0 0 0 -20 1 0 650130789 1155
8912 1820 18446744073709551615 94616585252864 94616589957457 140732034551456 0 0 0
0 1073745920 402745863 1 0 0 17 59 0 0 0 0 0 94616590894512 94616591263776 9461662
1305856 140732034554756 140732034554762 140732034554762 140732034555881 0
```

The response contains the recorded stats of all nginx worker processes as provided by the Linux kernel in /proc/{PID}/stat. For each process, there is a pair of entries corresponding to the time right before and right after a measurement cycle. The example shows four processes.

## 4.11 Fintech Analysis Use Case Integrated Functionality

The specifics of this use case are thoroughly outlined in deliverables D2.4 (M16) and D6.3 (M18). Here, a concise synopsis is provided, emphasizing the incorporation of SERRANO platform features and services.

InbestMe (INB) creates automated investment portfolios utilizing financial instruments like stocks, ETFs, and bonds. The asset allocation in these portfolios is informed by a blend of historical and current market data, aiming to meet specific investment goals within a set timeframe.

The Fintech use case (UC) showcases InbestMe's Dynamic Portfolio Optimization (DPO) application, where investment profiles are dynamically adjusted. This optimization hinges on comprehensive market analysis paired with strategic investment planning to determine the precise distribution of assets.

The SERRANO project serves as a cornerstone in enhancing the investment management UC, offering a robust framework that greatly eases the deployment, management, and monitoring of DPO applications. Through SERRANO, the complexities of DPO are streamlined, facilitating a more efficient and user-friendly experience for investment managers. Moreover, SERRANO's contribution is pivotal in transforming investment management into a service (SaaS), particularly with its secure storage extension designed to safeguard third-party data. This security feature ensures that all client information is protected, fostering trust and reliability in the system. Furthermore, the UC stands to gain from SERRANO's ability to expedite a variety of computationally demanding tasks through cloud-based acceleration. This not only enhances performance but also optimizes the responsiveness of the DPO process, thereby underscoring the invaluable support that SERRANO provides in the sophisticated realm of investment management.

Furthermore, the SERRANO platform is a key advantage for InbestMe, as it will cut cloud costs and boost the quality of services provided. With SERRANO, InbestMe can easily set up multiple versions of its investment management platform on various cloud resources. This capability not only saves money but also simplifies operations.

Additionally, SERRANO enables more precise analysis, which is crucial for building investment portfolios with less risk and more potential for profit. At present, InbestMe faces challenges in analysing all the available data thoroughly. With SERRANO, InbestMe can look into more data and use algorithms for prediction and forecasting, namely Savitzky–Golay and Kalman filter, Wavelet transform, and Black–Scholes model with improved precision and accuracy, thus making better-informed decisions for portfolio management.

The following Figure 61 illustrates how the DPO application is integrated with select SERRANO components to meet the objectives and requirements of the use case as detailed in Deliverable 2.4.

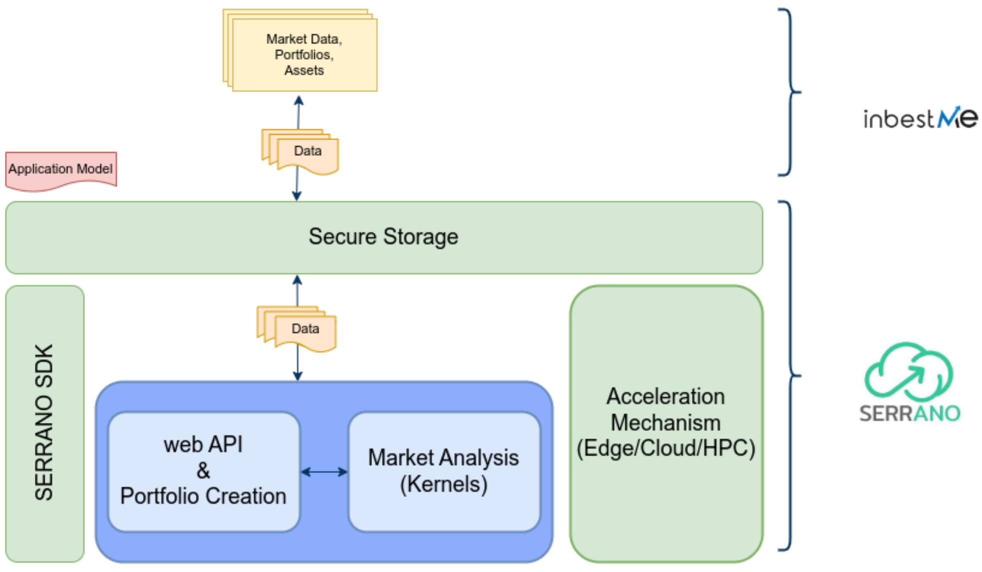


**Figure 61: DPO application structure and integration with SERRANO components**

### 4.11.1 Integration details and REST APIs

The DPO service utilizes the SERRANO SDK for implementation, encapsulating the application within containers. This is supported by a Jenkins pipeline for continuous integration and delivery, all hosted on the Kubernetes server:

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

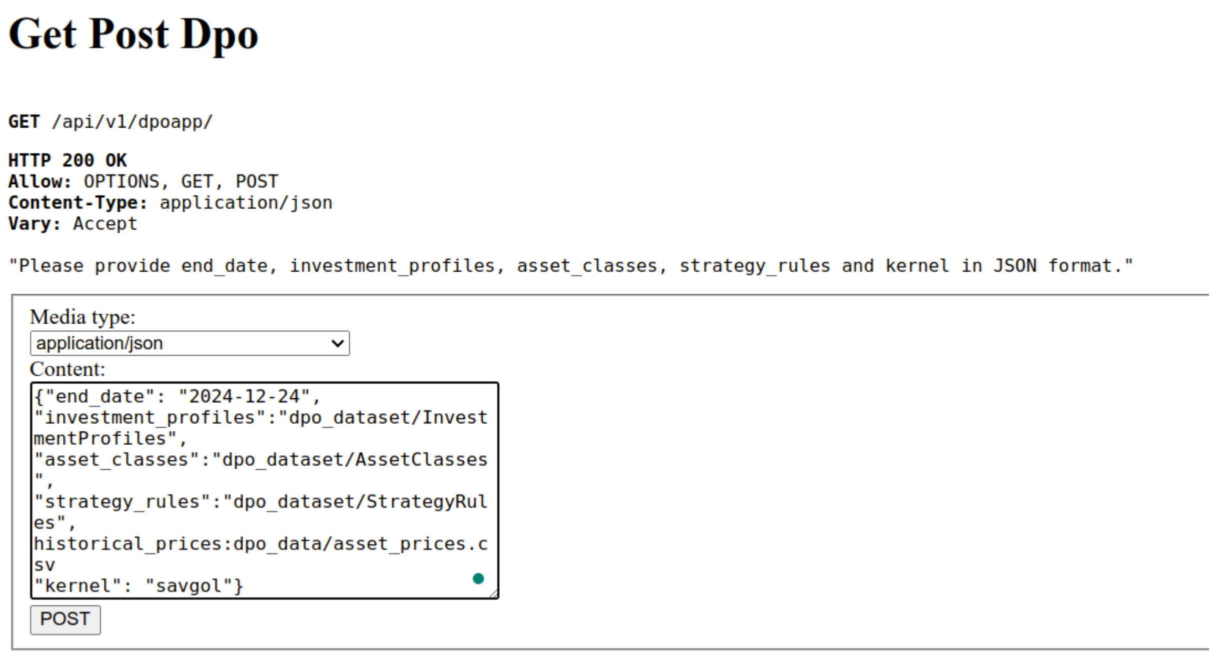
To ensure smooth deployment and oversight of services/microservices, the necessary details of execution prerequisites are conveyed to the AI-enhanced Service Orchestrator using the Application Model, complemented by the deployment description in YAML format. The orchestration tools within SERRANO, namely the AI-enhanced Service Orchestrator and the Resource Orchestrator, will interpret these specifications into runtime parameters tailored for the SERRANO infrastructure and align them with the optimal resources via the Resource Optimization Toolkit.

Moreover, the developed DPO services use the SERRANO acceleration mechanisms in edge/cloud and HPC environments, facilitating access to SERRANO's accelerated kernels over heterogeneous computational resources. These mechanisms provide enhanced performance and optimization of the kernels used by the UC (Savgol, Wavelet, Kalman, Black Scholes). Additionally, the Secure Storage service, compatible with S3, is utilized for storing essential data such as historical asset prices, investment profile details, strategy rules, and asset class information. This approach ensures third-party data protection and simplifies access to large datasets.

The DPO features a single API which is publicly accessible via the provided link:

```
https://dpoapp.services.cloud.ict-serrano.eu/api/v1/dpoapp/
```

By navigating to the provided link, users are directed to the DPO's cloud service, where they can initiate the application by entering the required information in JSON format into the Content box, as depicted in Figure 62. Descriptions for each input value are provided below in the box.



Get Post Dpo

```
GET /api/v1/dpoapp/

HTTP 200 OK
Allow: OPTIONS, GET, POST
Content-Type: application/json
Vary: Accept

"Please provide end_date, investment_profiles, asset_classes, strategy_rules and kernel in JSON format."
```

Media type: application/json

Content:

```
{"end_date": "2024-12-24",
"investment_profiles":"dpo_dataset/InvestmentProfiles",
"asset_classes":"dpo_dataset/AssetClasses",
"strategy_rules":"dpo_dataset/StrategyRules",
historical_prices:dpo_data/asset_prices.csv
"kernel": "savgol"}
```

**Figure 62: DPO Landing Page and example input**

```
Input JSON
{
   end_date:              DateTime
   investment_profiles:   Url1ToSkyFlokStorage: String
   asset_classes:         Url2ToSkyFlokStorage: String
   strategy_rules:        Url3ToSkyFlokStorage: String
   historical_prices:     Url4ToSkyFlokStorage: String
   kernel:                String
}

- EndDate:                Historical Asset Data look up to date

- investment_profiles:    Path to Historical Asset Data file stored in SkyFlock
cloud storage

- asset_classes:          Path to Asset Classes Data file stored in SkyFlock cloud
storage

- strategy_rules:         Path to Strategy Rules Data file stored in SkyFlock cloud
storage

- historical_prices:      Path to Historical Price Data file stored in SkyFlock
cloud storage

- kernel: kernel construct portfolios of DPO – [all, wavelet, savgol, kalman,
blackscholes]
```

Once the "POST" button is clicked with all the required information filled in the Content box, the DPO application begins its operation. Upon completion, the user is presented with a page that includes a link to download a zip folder. This folder contains three distinct files, each offering valuable insights: Backtesting Results, Asset Distributions for each Investment Profile, and Different scenarios for each profile with corresponding asset recommendations.

Additionally, the results page provides detailed execution information, such as which computational kernel was utilized, the number of assets analyzed, and the total duration of the execution process.

## 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 (Figure 63) to analyse real-time signals from the ball-screw sensors and check for anomalies, detect anomalous behaviours that may affect the part quality, and predict imminent failures. The 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 the 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*: 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 (vibration data).

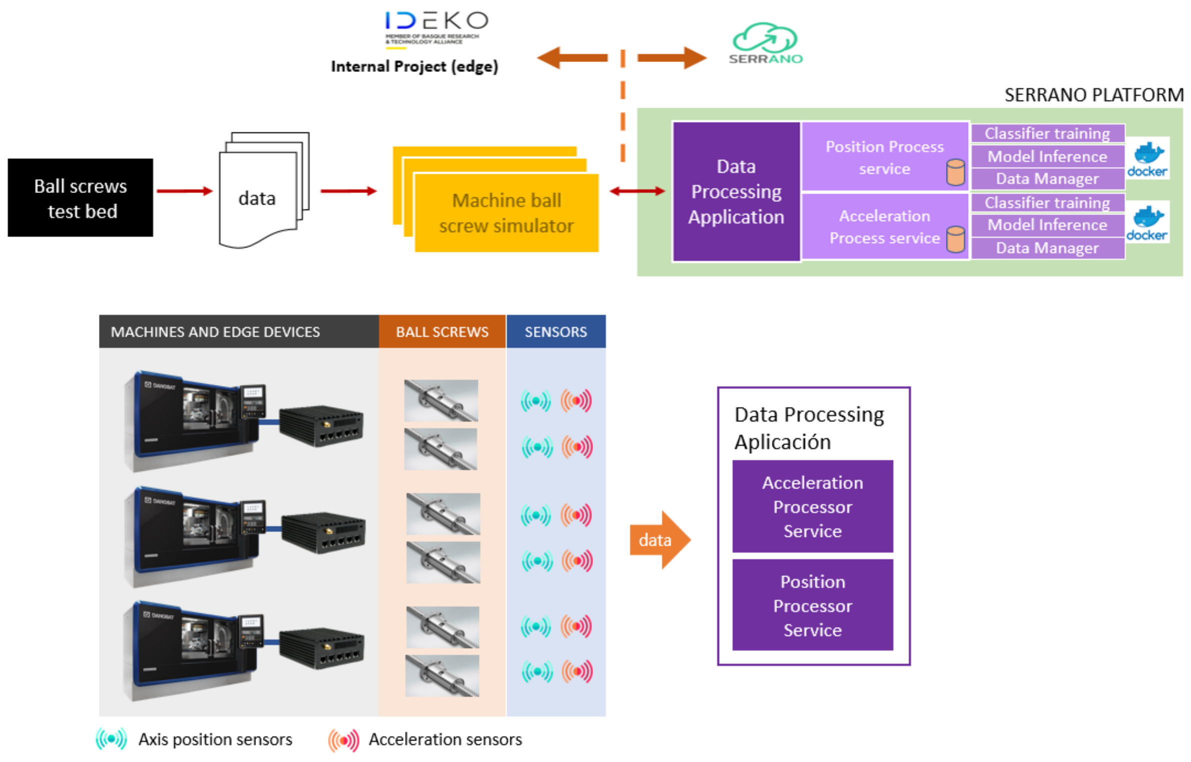


**Figure 63: Developed Data Processing application**

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

- *Model Inference:* Loads the trained classifier model and classifies the new incoming stream data, predicting whether the data is anomalous or not.
- *Data Manager:* Manages the streaming data from the ball screw sensors and the predictions that the Model Inference microservice makes. The application ensures that a limited number of data and predictions are stored as historical data.
- *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).

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 [84]).

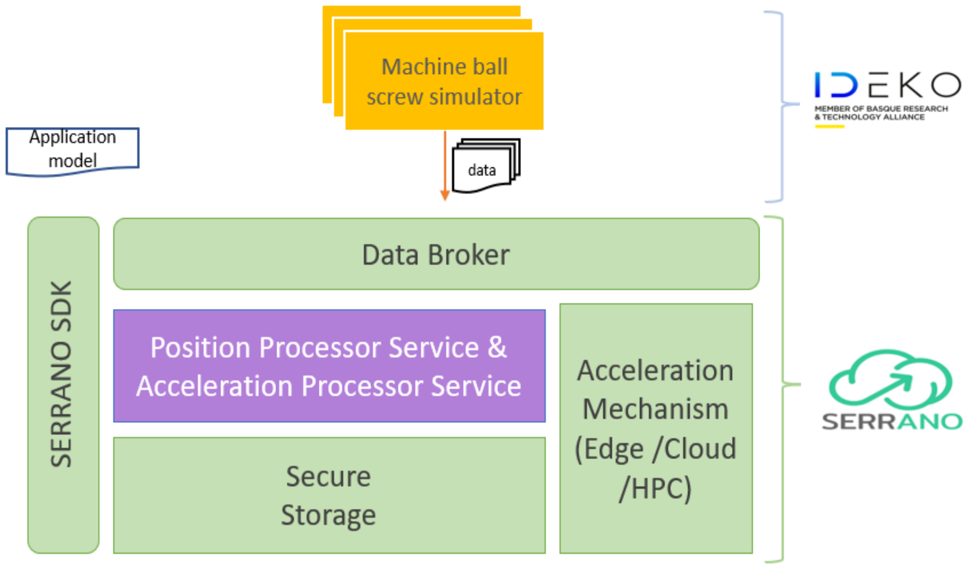

Machine ball screw simulator
Application model
data
IDEKO
MEMBER OF BASQUE RESEARCH & TECHNOLOGY ALLIANCE
Data Broker
SERRANO SDK
Position Processor Service & Acceleration Processor Service
Acceleration Mechanism (Edge /Cloud /HPC)
Secure Storage
SERRANO

**Figure 64: Interactions between the use case developed services and core components of the SERRANO platform**

The services developed by IDEKO for anomaly detection have been deployed in container-based applications on the SERRANO platform through the *Allien4Cloud* platform **Error! Reference source not found.**, more specifically using the SERRANO Orchestrator plugin developed to interpret the TOSCA [80] language and interact with the SERRANO framework. The SERRANO extension to the TOSCA specification models Application intent elements (part of the *ARDIA framework*), and IDEKO describes the services' constraints, high-level requirements, and the deployment configuration of the services/microservices (aka deployment descriptor YAML). Static deployment descriptors have been provided during the project, and this information has been abstracted in the SERRANO TOSCA extension. The Orchestrator plugin can dynamically generate these descriptors based on the selected location. For example, configuration parameters are generated based on which instance of a Data Broker is used (a local one or the SERRANO-provided instance). The plugin allows for deployment using either a vanilla Kubernetes cluster (for example, during the development phase) or the SERRANO continuum (during the operations phase). When deploying on the SERRANO Continuum, the Orchestrator plugin prepares the intent for the *AI-enhanced Service Orchestrator* (AISO), which uses historical telemetry data and machine learning models to select and rank different placements for the services and provides this ranking to the *Resource Orchestrator*. The latter, along with the *Resource Optimization Toolkit (ROT)***,** allocates the platform resources so that they can satisfy the services' requirements. The *Orchestration*

*Driver* handles the actual deployment of the application in the selected platform based on the high-level deployment description from the Resource Orchestrator.

The streaming data integration with the SERRANO platform is done through the Data Broker component (detailed in Section 4.4). *Data Broker* provides 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 leverage the SERRANO SDK to facilitate their seamless access to the SERRANO accelerated kernels. The SDK abstracts the integration with the SERRANO *hardware acceleration mechanisms in edge/cloud* (Section 4.7) and *HPC* (Section 4.6). These SERRANO developments provide better performance and optimization of the highly computational intensive kernels used (e.g., DTW, KMeans, KNN, or FTT) by the Model Inference and Classifier Training services. Moreover, the *S3-compatible Secure Storage* (detailed in Section 4.9) interface is used to store the last N streaming data received through the Data Broker. This way, the required data is stored and accessible by all SERRANO components and the use case services.

The idea is to reduce the classifier training time and the needed time to make a new prediction through the streaming data. This enables 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. The SERRANO platform is needed since the current techniques and resources available at the edge cannot support the above operations.

### 4.12.1 Integration details and REST APIs

In order to execute the developed services/microservices and obtain the ball-screws' status assessment without stopping the machine in real time scenarios, this use case capitalizes on several SERRANO platform services (Figure 65). This approach ensures service quality in terms of latency, constantly adapting to the current demand of resources.

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

Thus, before starting the analysis of the streaming data sent through the Machine Ball Screw Simulators, the Position and Acceleration Processor Service microservices must be subscribed to their corresponding topics for every ball screw of every machine. To this end, MQTT wildcards are used to subscribe to all topics that match the necessary pattern.

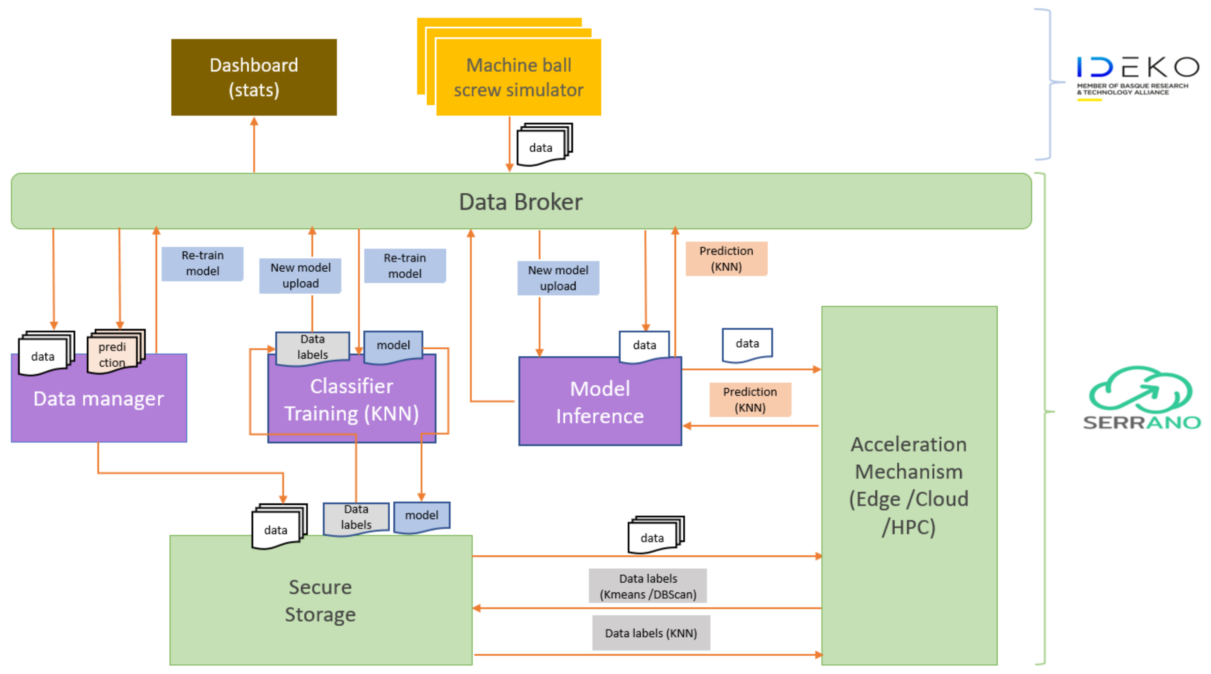


**Figure 65: Integration with SERRANO components**

On the one hand, the Model Inference microservice subscribes to three topics: *"training flag"*, *"model uploaded flag"*, and *"cycle batch data"*. On the other hand, Classifier Training subscribes just to the "training flag" topic. Last but not least, the Data Manager subscribes to *"prediction"* and *"cycle batch data"* topics. To achieve this, they subscribe to specific MQTT topic patterns, where the special character '**+**' represents anything. These subscriptions are designed to identify a specific machine and ball screw. In order to differentiate between topics related to the Position and Acceleration Processor Services, the topic patterns specify the nature of data using the placeholder **{nature of data}**. The previously mentioned topics are translated into the following topic patterns:

- **Training flag:** data/+/+/ **{nature of data}** /training/flag
- **Model uploaded flag:** data/+/+/ **{nature of data}** /training/model_upload
- **Cycle batch data:** data/+/+/ **{nature of data}** /cycle
- **Prediction:** data/+/+/ **{nature of data}** /inference/prediction

The Data Manager, which is subscribed to the Data Broker, receives the data batch and saves it as a CSV file in a volume. If this file exists, it 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. Thus, at the end, a sliding 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. The CSV file in the volume is uploaded into the secure storage through the following method of the SERRNO SDK, replacing the previous file if it exists, when a new model needs to be trained. This happens when the percentage of anomalous signals exceeds the configured threshold.

```
// SERRANO's Secure Storage mechanism (Store data into a defined bucket)
result =secureStorage.upload_object(bucket_name, file_with_abs_path, data)
```

At the same time, the Model Inference microservice, also subscribed to the MQTT Broker, reads streaming data (*input_file*) and loads the classifier model (*position and labels*) from a known path. Using that pre-trained model classifies the data as anomalous or normal. Before the service can request the on-demand execution of SERRANO-accelerated kernels, it has to describe the required input data to the SERRANO platform services. To this end, the SERRANO SDK provides a set of methods that facilitate the seamless move of input data to SERRANO storage services. The following example shows the corresponding method for describing the input data for the SERRANO-accelerated KNN kernel.

```
// SERRANO's acceleration mechanisms
// Provide required input data for the SERRANO-accelerated KNN kernel
payload = serrano_kernels.knn_pack_data(position = dataset_file,
                                       labels = labels_file,
                                       input_file = parsed_data)
```

The Model Inference service can classify either using typical edge resources or by leveraging the SERRANO's acceleration mechanisms through their exposed API to reduce execution time. In the latter case, it uses the SERRANO SDK method to request the on-demand execution of a SERRANO accelerated kernel. For this operation, it provides the name of the kernel (e.g., *kernelNames.SERRANO_KERNEL_KNN*) along with the input data description that was returned from the previous method (e.g., *payload*). Next, the service calls the *serrano_faas_kernel_results()* method to retrieve the results.

```
// SERRANO's acceleration mechanisms
// Request the execution of the SERRANO-accelerated KNN Kernel
req = serrano_kernels.serrano_faas_kernel(kernelNames.SERRANO_KERNEL_KNN,
                                          payload)

// Get the prediction result from the kernel execution
result = json.loads(serrano_kernels.serrano_faas_kernel_results(req["uuid"]))
```

After making the classification (*result*), the service publishes the prediction to the Data Broker, and the Data Manager service saves that prediction. As it happens with the cycle batch data, the Data Manager stores a sliding window of the last N number predictions. At this point, the Model Inference microservice also publishes performance metrics (the time needed to make the prediction) that is then displayed on the internal statistics Grafana Dashboard (Figure 68).

After saving the prediction, the Data Manager checks if the anomalous percentage value exceeds 80% of the last number of predictions. If not, it repeats the described process, but if it is, the data manager uploads the sliding window of the last N number of data to the secure storage and publishes a retrain flag in the Data Broker.

```
// SERRANO's Secure Storage mechanism (Store data into a defined bucket)
result = secureStorage.upload_object(bucket_name, file_with_abs_path, data)
```

Moreover, the Data Manager microservice also publishes performance metrics, such as the percentage of the anomalous and normal cycle batches, that is then displayed on the use case Dashboard. The Classifier Training, which is subscribed to the training flag, receives a "1" as a flag value that triggers the training pipeline. During this training pipeline, the acceleration mechanisms at the selected resources retrieve the label data (*labels*) using the corresponding Secure Storage service methods provided by the SERRANO SDK. Also, the acceleration mechanism retrieves the last stored filename in S3 (*position*).

```
// SERRANO's Secure Storage mechanism (Get data into a defined bucket)
position = secureStorage.get_object(bucket_name, file_with_abs_path)

// SERRANO's Secure Storage mechanism (Get data into a defined bucket)
labels = secureStorage.get_object(bucket_name, file_with_abs_path)

// SERRANO's acceleration mechanisms (Retrieve data from S3 secure storage)
payload = serrano_kernels.kmeans_pack_data(position=file_path,labels=labels_file)
```

The Secure Storage returns the historical data and the acceleration mechanisms label the retrieved data (*payload*), training a cluster using KMEANS based on DTW metric accelerated kernel (*kernelNames.SERRANO_KERNEL_KMEANS*).

```
// SERRANO acceleration mechanisms (training a cluster using KMEANS based on DTW)
d_labels = serrano_kernels.serrano_faas_kernel(kernelNames.SERRANO_KERNEL_KMEANS,
                                               payload)
```

After labelling the data, the service securely stores the computed labels in the Secure Storage service and publishes performance metrics to the Data Broker.

```
// SERRANO Secure Storage mechanism (Store data into a defined bucket)
Result = secureStorage.upload_object(bucket_name, file_with_abs_path, d_labels)

// SERRANO acceleration mechanisms (Publish results in Data Broker)
results = serrano_kernels.serrano_faas_kernel_results(req["request_uuid"])
```

Upon completion of model training and evaluation, the model upload flag is published. At this point, the Classifier Training microservice also publishes relevant performance data, such as the time required for model training), which is then presented on the internal Dashboard.

The Model Inference, subscribed to the model upload flag, receives a "1" as a flag value, which triggers the initiation of a new model creation process. To achieve this, the service downloads the historical data and labels from the Secure Storage service, utilizing the provided methods. The retrieved data are then utilized to build up the KNN classifier.

```
// SERRANO's Secure Storage mechanism (Get data into a defined bucket)
labels = secureStorage.get_object(bucket_name, file_with_abs_path)
```

Figure 66 shows a detailed workflow for integrating the use case services within the SERRANO platform components.

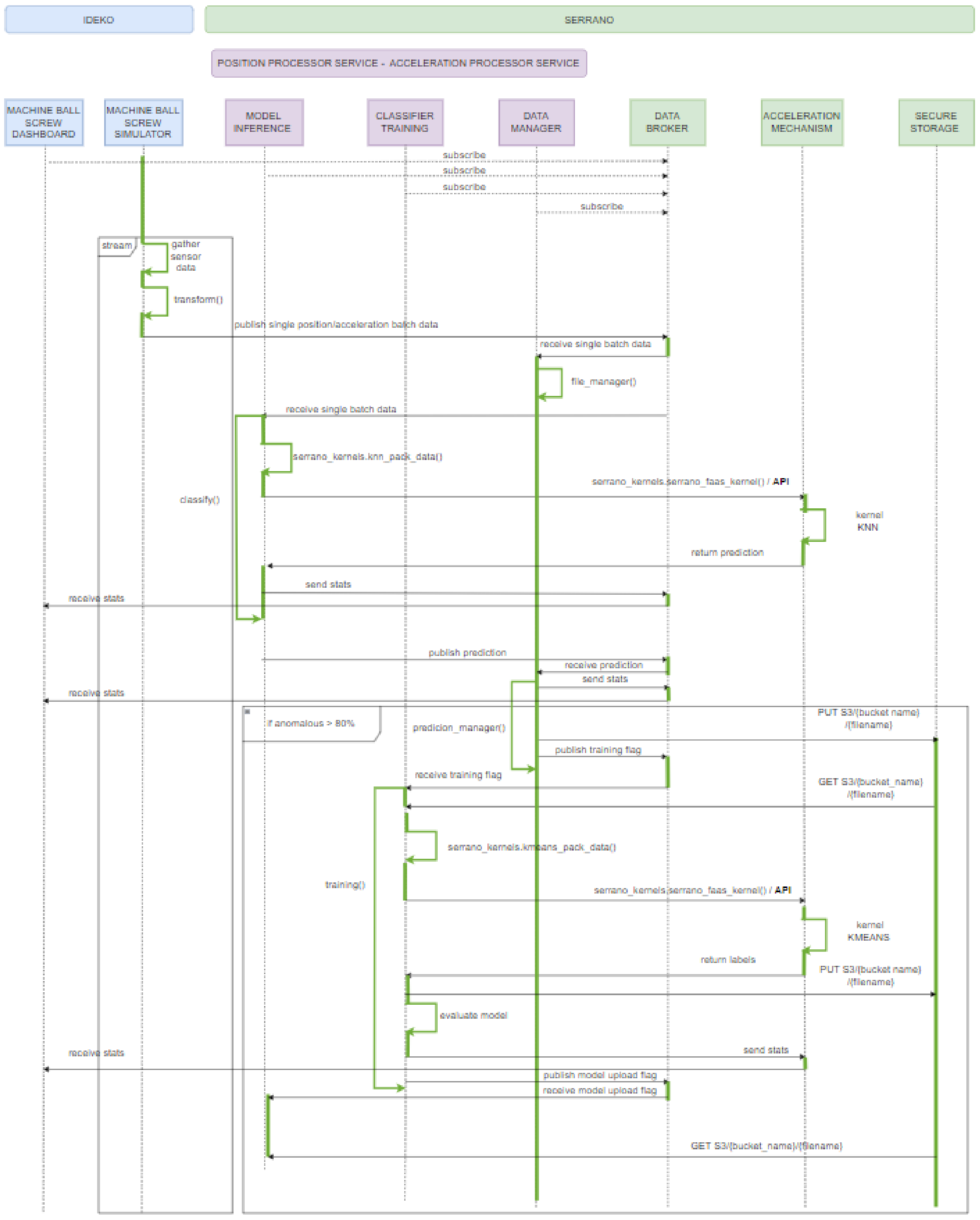
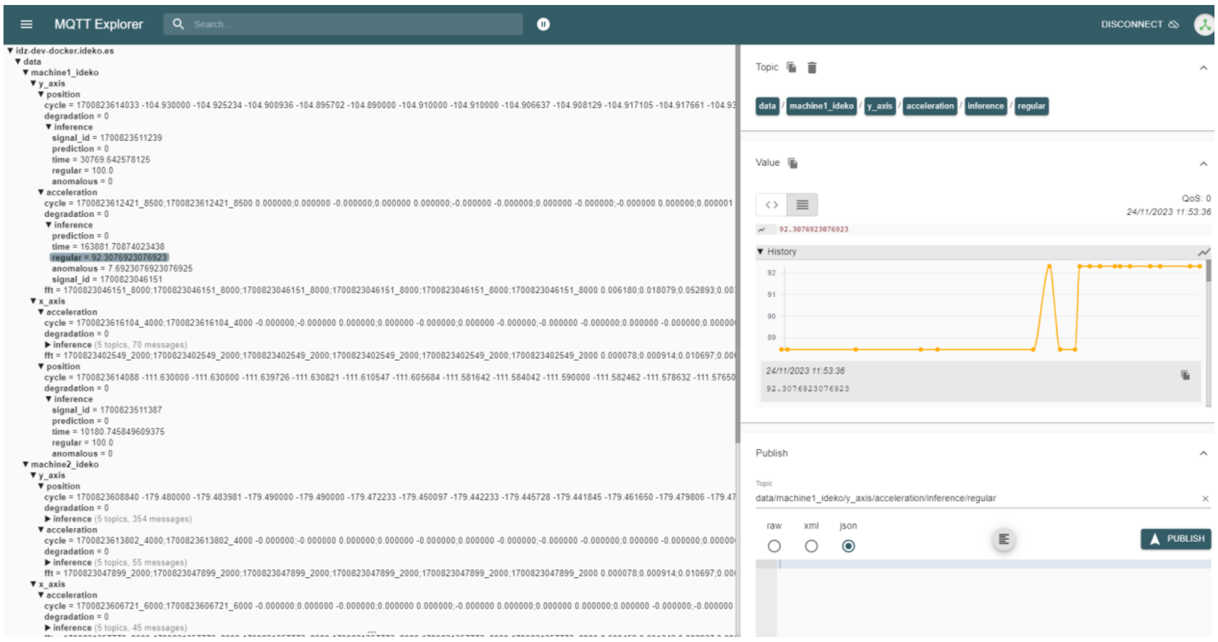

IDEKO
SERRANO
POSITION PROCESSOR SERVICE - ACCELERATION PROCESSOR SERVICE
MACHINE BALL SCREW DASHBOARD
MACHINE BALL SCREW SIMULATOR
MODEL INFERENCE
CLASSIFIER TRAINING
DATA MANAGER
DATA BROKER
ACCELERATION MECHANISM
SECURE STORAGE
subscribe
subscribe
subscribe
subscribe
stream
gather sensor data
transform()
publish single position/acceleration batch data
receive single batch data
file_manager()
receive single batch data
serrano_kernels.knn_pack_data()
serrano_kernels.serrano_faas_kernel() / API
classify()
kernel KNN
return prediction
send stats
receive stats
publish prediction
receive prediction
send stats
receive stats
if anomalous > 80%
predicion_manager()
PUT S3/{bucket name}/{filename}
publish training flag
receive training flag
GET S3/{bucket_name}/{filename}
serrano_kernels.kmeans_pack_data()
training()
serrano_kernels.serrano_faas_kernel() / API
kernel KMEANS
return labels
PUT S3/{bucket name}/{filename}
evaluate model
send stats
receive stats
publish model upload flag
receive model upload flag
GET S3/{bucket_name}/{filename}

**Figure 66: Integration workflow**

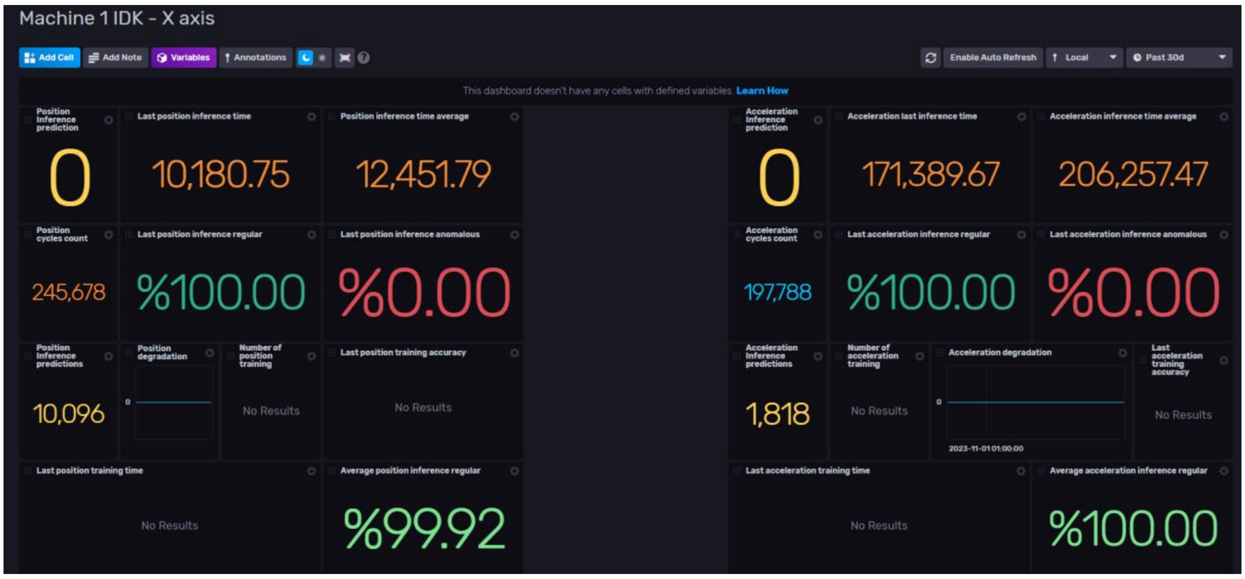
As an example of the tests carried out internally at IDEKO, the topics generated for communication between the microservices are shown through the MQTT Broker (Figure 67), as well as the published and received data from the streaming data generated in the test bench.



```
▼ idz-dev-docker.ideko.es
  ▼ data
    ▼ machine1_ideko
      ▼ y_axis
        ▼ position
          cycle = 1700823614033 -104.930000 -104.925234 -104.900936 -104.895702 -104.890000 -104.910000 -104.910000 -104.906637 -104.908129 -104.917105 -104.917661 -104.93
          degradation = 0
          ▼ inference
            signal_id = 1700823511239
            prediction = 0
            time = 30769.642578125
            regular = 100.0
            anomalous = 0
        ▼ acceleration
          cycle = 1700823612421_8500;1700823612421_8500 0.000000;0.000000 -0.000000;0.000000 0.000000;-0.000000 -0.000000;0.000000 -0.000000;-0.000000 0.000000;0.000001
          degradation = 0
          ▼ inference
            prediction = 0
            time = 163881.70874023438
            regular = 92.3076923076923
            anomalous = 7.6923076923076925
            signal_id = 1700823046151
          fft = 1700823046151_8000;1700823046151_8000;1700823046151_8000;1700823046151_8000;1700823046151_8000;1700823046151_8000 0.006180;0.018079;0.052893;0.00
      ▼ x_axis
        ▼ acceleration
          cycle = 1700823616104_4000;1700823616104_4000 -0.000000;-0.000000 0.000000;0.000000 -0.000000;0.000000 -0.000000;-0.000000 -0.000000;0.000000 -0.000000;0.00000
          degradation = 0
          ▶ inference (5 topics, 70 messages)
          fft = 1700823402549_2000;1700823402549_2000;1700823402549_2000;1700823402549_2000;1700823402549_2000;1700823402549_2000 0.000078;0.000914;0.010697;0.00
        ▼ position
          cycle = 1700823614088 -111.630000 -111.630000 -111.639726 -111.630821 -111.610547 -111.605684 -111.581642 -111.584042 -111.590000 -111.582462 -111.578632 -111.5765
          degradation = 0
          ▼ inference
            signal_id = 1700823511387
            prediction = 0
            time = 10180.745849609375
            regular = 100.0
            anomalous = 0
    ▼ machine2_ideko
      ▼ y_axis
        ▼ position
          cycle = 1700823608840 -179.480000 -179.483981 -179.490000 -179.490000 -179.472233 -179.450097 -179.442233 -179.445728 -179.441845 -179.461650 -179.479806 -179.47
          degradation = 0
          ▶ inference (5 topics, 354 messages)
        ▼ acceleration
          cycle = 1700823613802_4000;1700823613802_4000 -0.000000;-0.000000 0.000000;0.000000 -0.000000;0.000000 -0.000000;-0.000000 -0.000000;0.000000 -0.000000;0.00000
          degradation = 0
          ▶ inference (5 topics, 55 messages)
          fft = 1700823047899_2000;1700823047899_2000;1700823047899_2000;1700823047899_2000;1700823047899_2000;1700823047899_2000 0.000078;0.000914;0.010697;0.00
      ▼ x_axis
        ▼ acceleration
          cycle = 1700823606721_6000;1700823606721_6000 -0.000000;0.000000 -0.000000;0.000000 0.000000;-0.000000 0.000000;0.000000 0.000000;0.000000 -0.000000;-0.000000
          degradation = 0
          ▶ inference (5 topics, 45 messages)
```

**Figure 67: Internal topics generated for communication between the microservices through the MQTT Broker**

In addition, a Grafana Dashboard (Figure 68) 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.



Position Inference prediction: 0
Last position inference time: 10,180.75
Position inference time average: 12,451.79
Acceleration Inference prediction: 0
Acceleration last inference time: 171,389.67
Acceleration inference time average: 206,257.47
Position cycles count: 245,678
Last position inference regular: %100.00
Last position inference anomalous: %0.00
Acceleration cycles count: 197,788
Last acceleration inference regular: %100.00
Last acceleration inference anomalous: %0.00
Position Inference predictions: 10,096
Position degradation: 0
Number of position training: No Results
Last position training accuracy: No Results
Acceleration Inference predictions: 1,818
Number of acceleration training: No Results
Acceleration degradation: 0, 2023-11-01 01:00:00
Last acceleration training accuracy: No Results
Last position training time: No Results
Average position inference regular: %99.92
Last acceleration training time: No Results
Average acceleration inference regular: %100.00

**Figure 68: Internal website Dashboard for aggregated results and stats**

### 4.12.2 UC Integration with Data Broker

The use case leverages the capabilities of the Data Broker component through the exposed publish-subscribe interface (Figure 69). More specifically, it uses the MQTT protocol (i.e., the standard for IoT messaging) to forward the data generated by the use case machines to the services deployed within the SERRANO platform. To this end, a connector has been created in the machine ball screw simulation application that publishes the data in the SERRANO Data Broker component, from where the services/components consume the data through the corresponding subscriptions, as explained in the previous Section 4.12.1.

The results (e.g., predictions, time, data, etc.) also are exchanged through the MQTT protocol provided by the Data Broker to be stored internally and then used by a dedicated Grafana Dashboard (in order to visualize and check the status of the ball screw in real time.

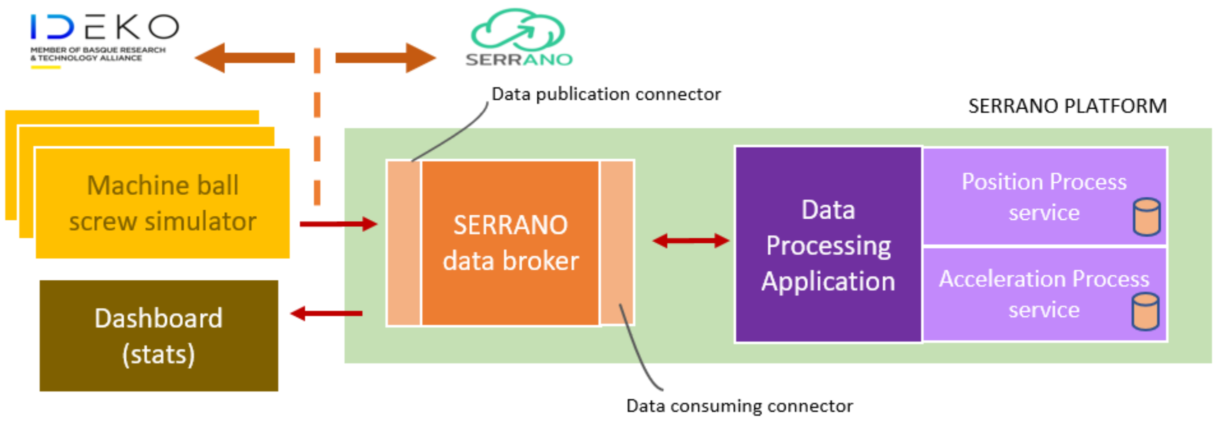

IDEKO
MEMBER OF BASQUE RESEARCH & TECHNOLOGY ALLIANCE
SERRANO
Data publication connector
SERRANO PLATFORM
Machine ball screw simulator
SERRANO data broker
Data Processing Application
Position Process service
Acceleration Process service
Dashboard (stats)
Data consuming connector

**Figure 69: Communication between the sending of streaming data through the MQTT protocol (Data Broker) to the SERRANO platform and how the results obtained are stored in an internal Dashboard**

Below, some examples of the subscriptions of the different microservices to the topics generated within the Data Broker are represented.

- Model Inference (Position Processor Service):

```
// 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):

```
// 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):

```
// 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

*For completeness, some sections that existed in D6.3 have been repeated in the contents of this chapter. The below information has been updated according to the current status.*

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 that 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. Continuous Deployment aim to release applications to end users faster and more cost-effectively by managing small, incremental software changes which pass through the entire pipeline.

At a 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 communication and collaboration and automate the integration of security at every phase of the 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**.**

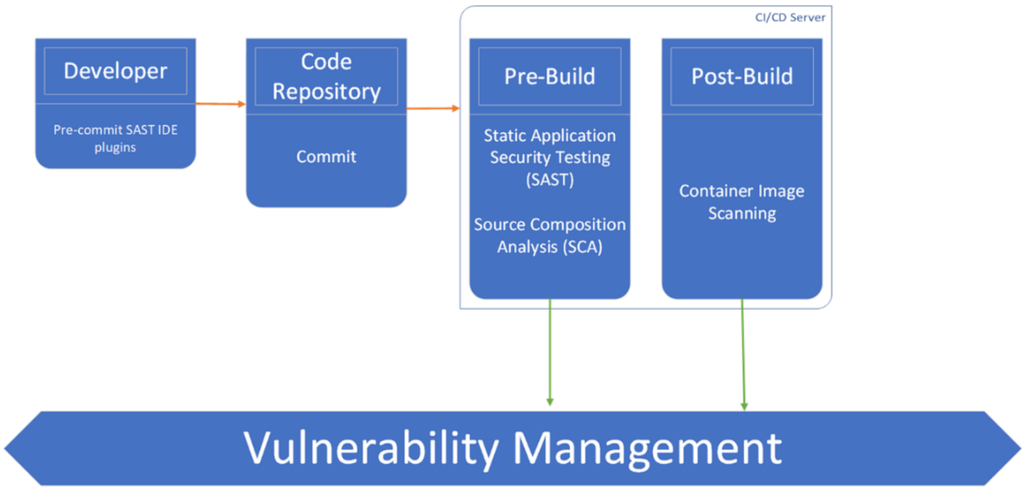
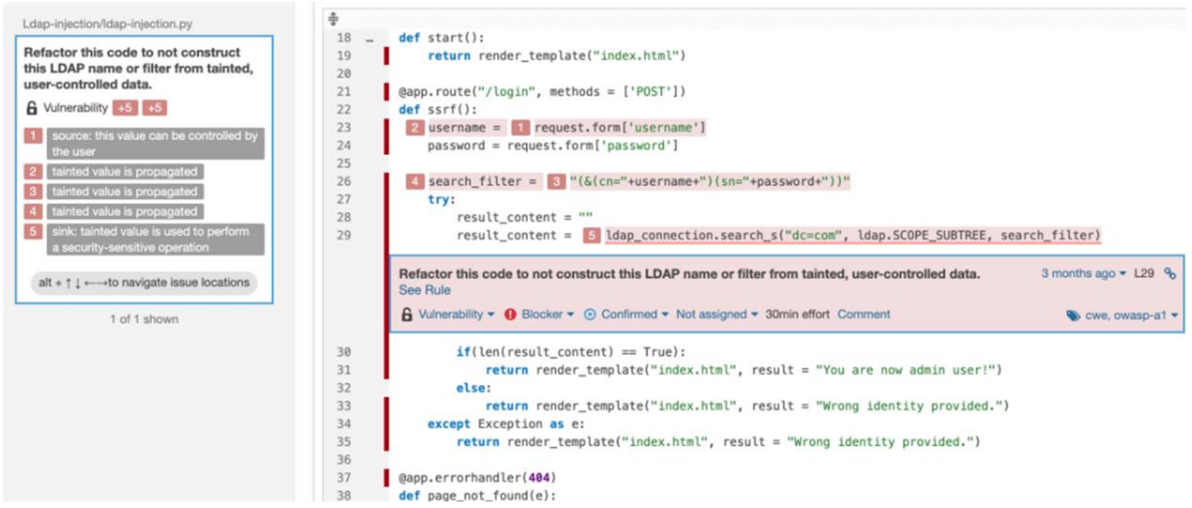


**Figure 70: Vulnerability management in CI/CD**

## 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.



Ldap-injection/ldap-injection.py

**Refactor this code to not construct this LDAP name or filter from tainted, user-controlled data.**

Vulnerability +5 +5

1 source: this value can be controlled by the user
2 tainted value is propagated
3 tainted value is propagated
4 tainted value is propagated
5 sink: tainted value is used to perform a security-sensitive operation

alt + ↑ ↓ ←→ to navigate issue locations

1 of 1 shown

```
def start():
    return render_template("index.html")

@app.route("/login", methods = ['POST'])
def ssrf():
    2 username = 1 request.form['username']
    password = request.form['password']

    4 search_filter = 3 "(&(cn="+username+")(sn="+password+"))"
    try:
        result_content = ""
        result_content = 5 ldap_connection.search_s("dc=com", ldap.SCOPE_SUBTREE, search_filter)
```

**Refactor this code to not construct this LDAP name or filter from tainted, user-controlled data.** 3 months ago L29
See Rule
Vulnerability Blocker Confirmed Not assigned 30min effort Comment cwe, owasp-a1

```
        if(len(result_content) == True):
            return render_template("index.html", result = "You are now admin user!")
        else:
            return render_template("index.html", result = "Wrong identity provided.")
    except Exception as e:
        return render_template("index.html", result = "Wrong identity provided.")

@app.errorhandler(404)
def page_not_found(e):
```

**Figure 71: SAST by SonarQube**

SAST tools such as SonarQube [34] 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 [35] 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 [36].

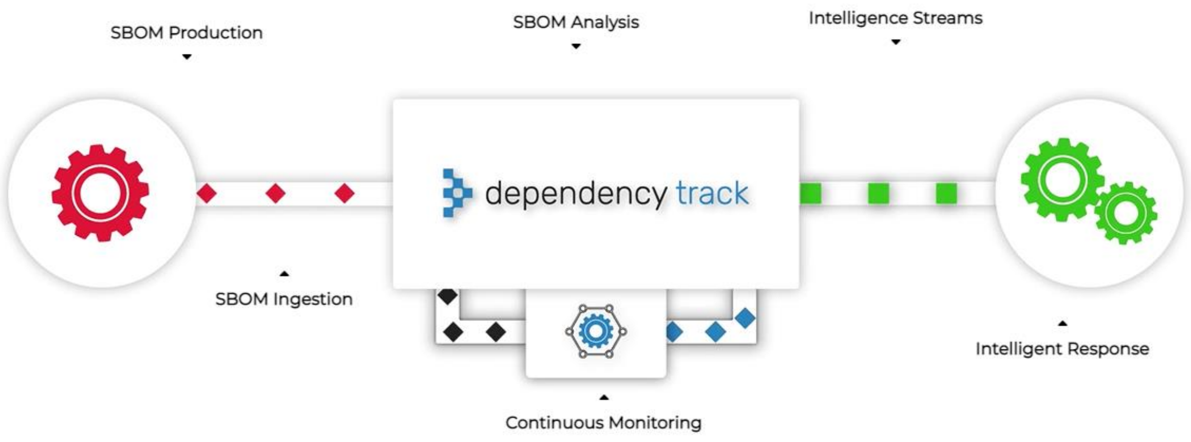


**Figure 72: SBOM Operations using Dependency-Track**

An example of the operations that the SBOM is involved, is visible in Figure 72. 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 [47] 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.

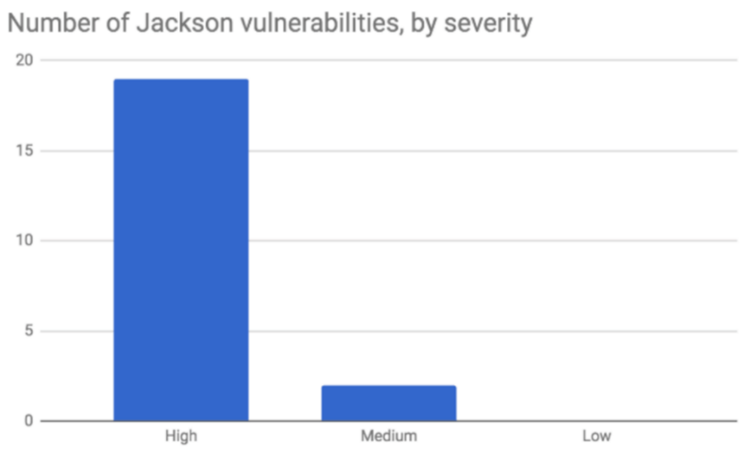


**Figure 73: Severity assessment of Jackson vulnerabilities**

### 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 [43], 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 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 74.

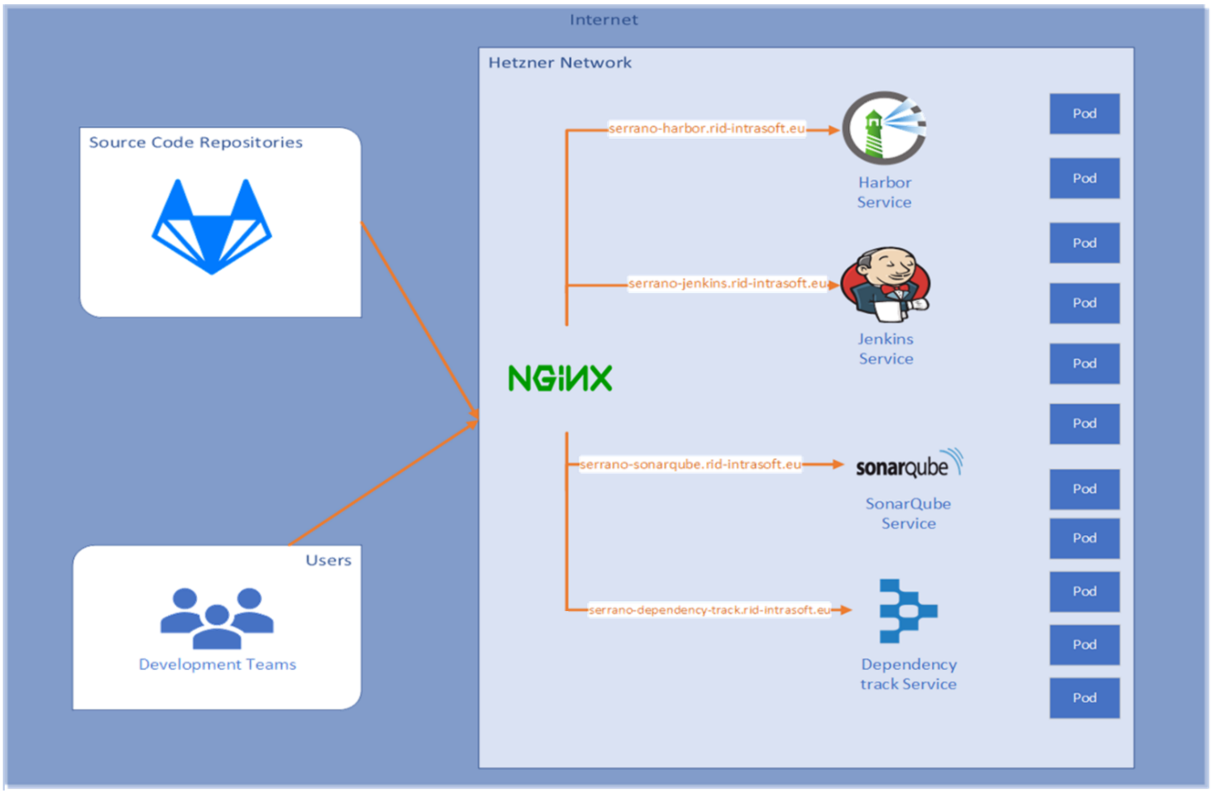


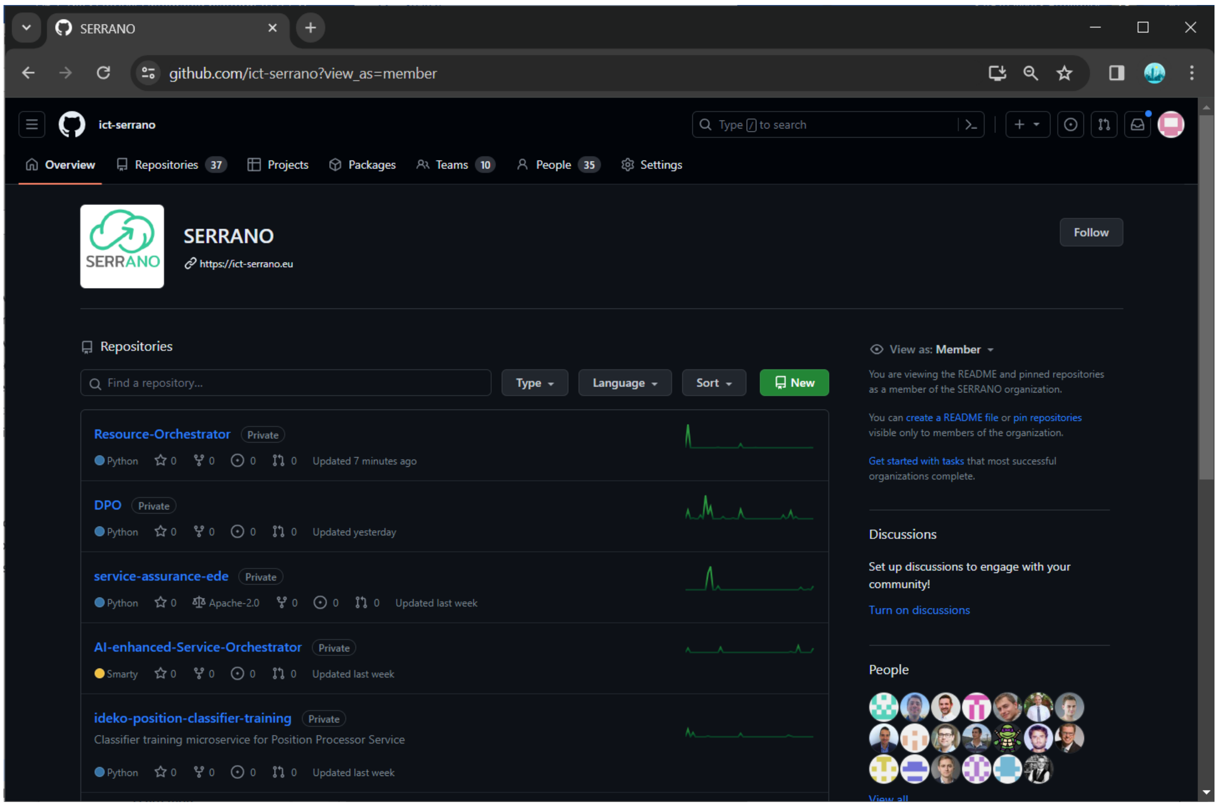
**Figure 74: SERRANO CI/CD components**

### 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.

GitHub [37] 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 GitHub repository.

For the needs of the SERRANO project, a dedicated and private GitHub organization named "SERRANO" has been built, as depicted in Figure 75. Under this group, whose URL is https://github.com/ict-serrrano, 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.


SERRANO

github.com/ict-serrano?view_as=member

ict-serrano

Type / to search

Overview | Repositories 37 | Projects | Packages | Teams 10 | People 35 | Settings

SERRANO
https://ict-serrano.eu

Follow

Repositories

Find a repository... | Type | Language | Sort | New

Resource-Orchestrator Private
Python ☆ 0 | 0 | 0 | 0 | Updated 7 minutes ago

DPO Private
Python ☆ 0 | 0 | 0 | 0 | Updated yesterday

service-assurance-ede Private
Python ☆ 0 | Apache-2.0 | 0 | 0 | 0 | Updated last week

AI-enhanced-Service-Orchestrator Private
Smarty ☆ 0 | 0 | 0 | 0 | Updated last week

ideko-position-classifier-training Private
Classifier training microservice for Position Processor Service
Python ☆ 0 | 0 | 0 | 0 | Updated last week

View as: Member

You are viewing the README and pinned repositories as a member of the SERRANO organization.

You can create a README file or pin repositories visible only to members of the organization.

Get started with tasks that most successful organizations complete.

Discussions

Set up discussions to engage with your community!

Turn on discussions

People

View all

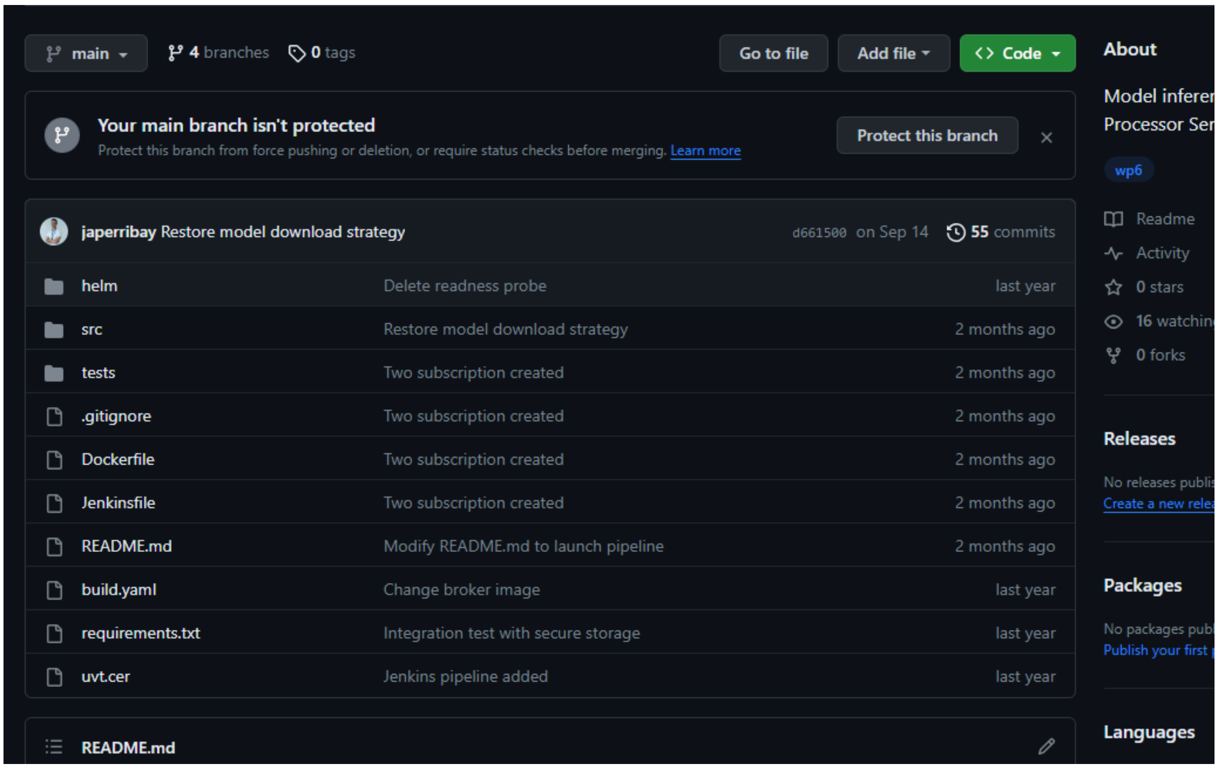
**Figure 75: The SERRANO GitHub organization and its contents**

Several GitHub user teams were generated to control access to the source code and scripts related to the SERRANO software components. Each team has read/write access to repositories that are named after the SERRANO individual tools. Each of these repositories 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 [51] chart is not generally applicable.

GitHub [37] 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.



Your main branch isn't protected
Protect this branch from force pushing or deletion, or require status checks before merging. Learn more

Model inferer
Processor Ser
wp6

No releases publis
Create a new relea

No packages publ
Publish your first

japerribay Restore model download strategy d661500 on Sep 14 55 commits

| | | |
|---|---|---|
| helm | Delete readness probe | last year |
| src | Restore model download strategy | 2 months ago |
| tests | Two subscription created | 2 months ago |
| .gitignore | Two subscription created | 2 months ago |
| Dockerfile | Two subscription created | 2 months ago |
| Jenkinsfile | Two subscription created | 2 months ago |
| README.md | Modify README.md to launch pipeline | 2 months ago |
| build.yaml | Change broker image | last year |
| requirements.txt | Integration test with secure storage | last year |
| uvt.cer | Jenkins pipeline added | last year |

README.md

**Figure 76: Structure of a SERRANO component on GitHub**

## 5.2.2 Continuous Integration – Jenkins

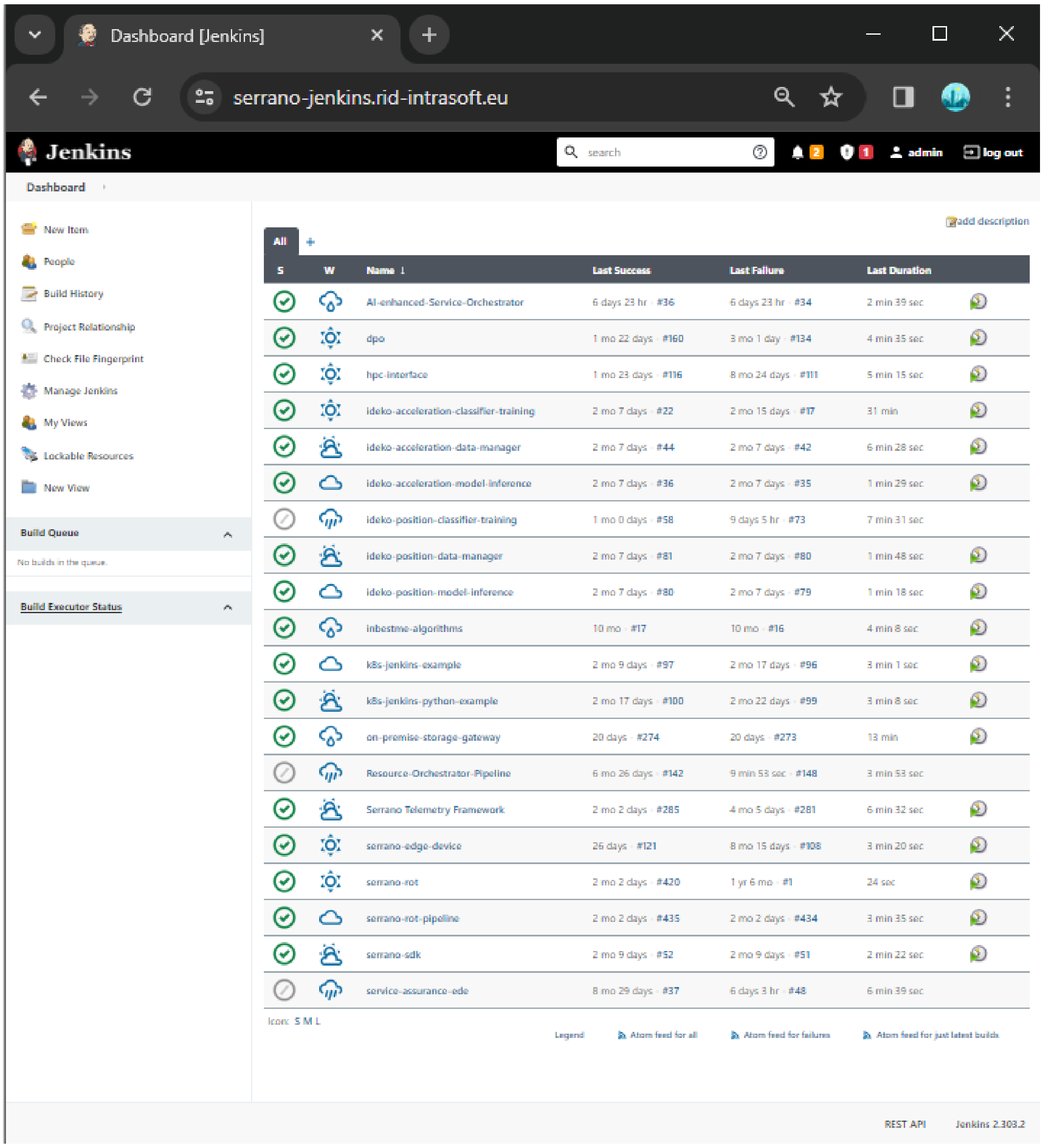
Jenkins [38] was selected as the Continuous Integration server of the CI/CD stack for SERRANO. It operates as a Kubernetes deployment upon Kubernetes workers running on dedicated servers on Hetzner [39] infrastructure and its URL https://serrano-jenkins.rid-intrasoft.eu.

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.



| S | W | Name ↓ | Last Success | Last Failure | Last Duration |
|---|---|---|---|---|---|
| ✓ | 🌦 | AI-enhanced-Service-Orchestrator | 6 days 23 hr · #36 | 6 days 23 hr · #34 | 2 min 39 sec |
| ✓ | ☀ | dpo | 1 mo 22 days · #160 | 3 mo 1 day · #134 | 4 min 35 sec |
| ✓ | ☀ | hpc-interface | 1 mo 23 days · #116 | 8 mo 24 days · #111 | 5 min 15 sec |
| ✓ | ☀ | ideko-acceleration-classifier-training | 2 mo 7 days · #22 | 2 mo 15 days · #17 | 31 min |
| ✓ | ⛅ | ideko-acceleration-data-manager | 2 mo 7 days · #44 | 2 mo 7 days · #42 | 6 min 28 sec |
| ✓ | ☁ | ideko-acceleration-model-inference | 2 mo 7 days · #36 | 2 mo 7 days · #35 | 1 min 29 sec |
| ⊘ | 🌧 | ideko-position-classifier-training | 1 mo 0 days · #58 | 9 days 5 hr · #73 | 7 min 31 sec |
| ✓ | ⛅ | ideko-position-data-manager | 2 mo 7 days · #81 | 2 mo 7 days · #80 | 1 min 48 sec |
| ✓ | ☁ | ideko-position-model-inference | 2 mo 7 days · #80 | 2 mo 7 days · #79 | 1 min 18 sec |
| ✓ | 🌦 | inbestme-algorithms | 10 mo · #17 | 10 mo · #16 | 4 min 8 sec |
| ✓ | ☁ | k8s-jenkins-example | 2 mo 9 days · #97 | 2 mo 17 days · #96 | 3 min 1 sec |
| ✓ | ⛅ | k8s-jenkins-python-example | 2 mo 17 days · #100 | 2 mo 22 days · #99 | 3 min 8 sec |
| ✓ | 🌦 | on-premise-storage-gateway | 20 days · #274 | 20 days · #273 | 13 min |
| ⊘ | 🌧 | Resource-Orchestrator-Pipeline | 6 mo 26 days · #142 | 9 min 53 sec · #148 | 3 min 53 sec |
| ✓ | ⛅ | Serrano Telemetry Framework | 2 mo 2 days · #285 | 4 mo 5 days · #281 | 6 min 32 sec |
| ✓ | ☀ | serrano-edge-device | 26 days · #121 | 8 mo 15 days · #108 | 3 min 20 sec |
| ✓ | ☀ | serrano-rot | 2 mo 2 days · #420 | 1 yr 6 mo · #1 | 24 sec |
| ✓ | ☁ | serrano-rot-pipeline | 2 mo 2 days · #435 | 2 mo 2 days · #434 | 3 min 35 sec |
| ✓ | ⛅ | serrano-sdk | 2 mo 9 days · #52 | 2 mo 9 days · #51 | 2 min 22 sec |
| ⊘ | 🌧 | service-assurance-ede | 8 mo 29 days · #37 | 6 days 3 hr · #48 | 6 min 39 sec |

**Figure 77: 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 organization in GitHub, are connected to a Jenkins Pipeline job, like the ones depicted in the Figure 77 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 occurs as a result of source code changes on the GitHub 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 [40] 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 78 represents the basic parts of the Docker architecture.

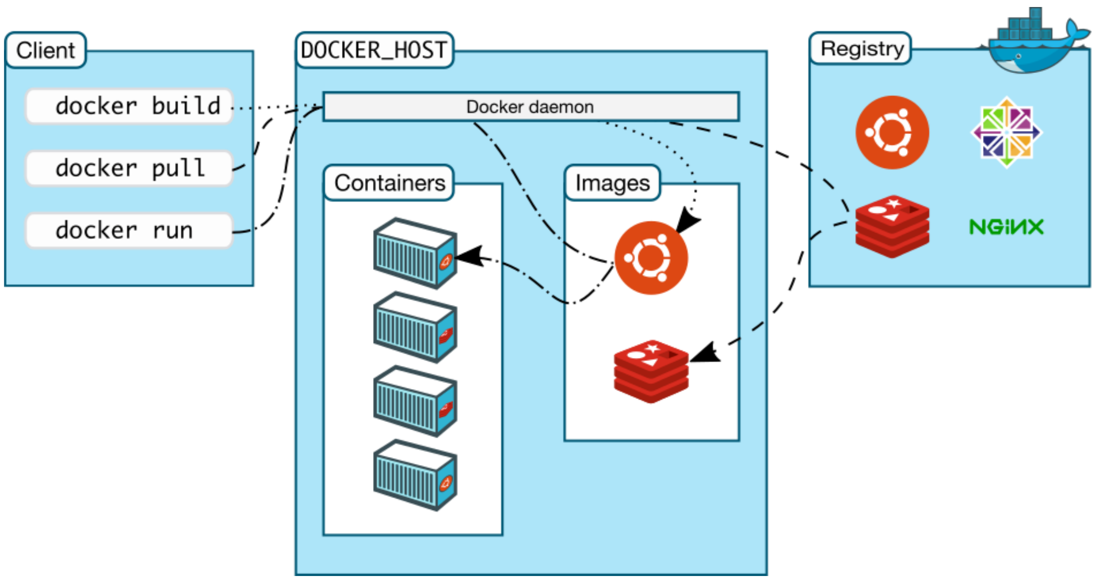


**Figure 78: Basic parts of the Docker architecture**

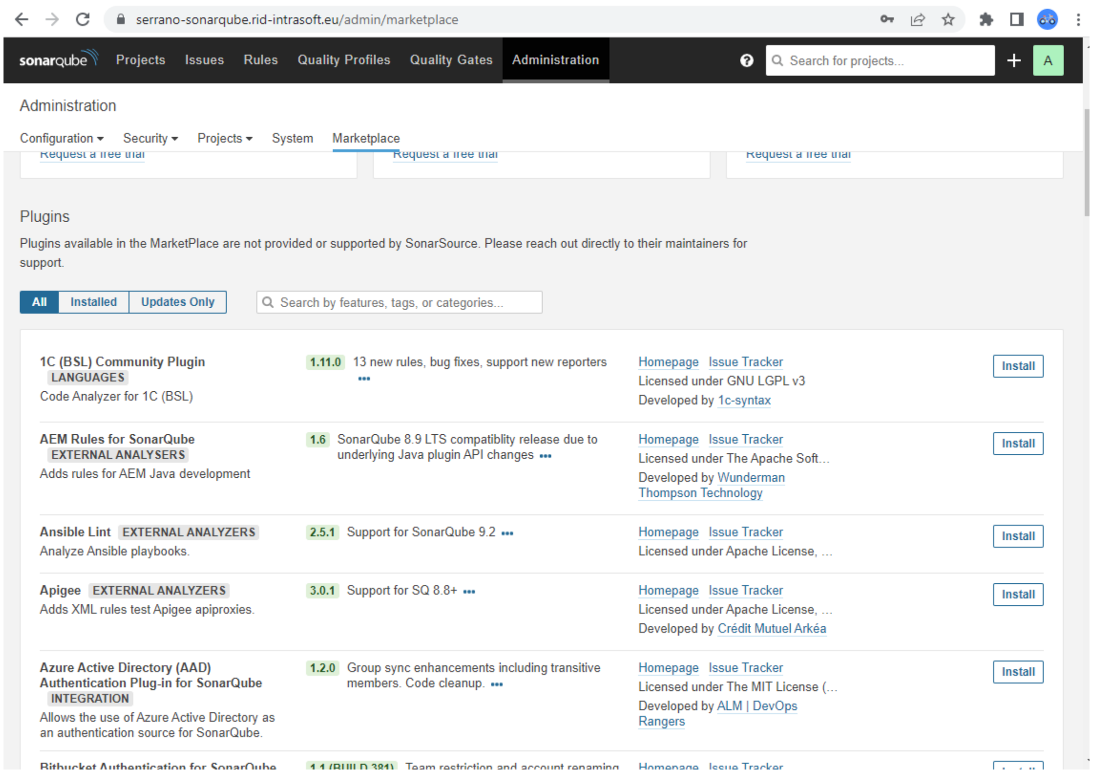
The Docker client and daemon can be both executed on the same machine, or a Docker client can be linked to a Docker daemon that is located elsewhere. A REST API, UNIX sockets, or a network interface are used by the Docker client and daemon to communicate. Docker Compose is another Docker client that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

### 5.2.4 SonarQube

SonarQube [34] collects and analyses 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. In SERRANO, SonarQube performs both security and quality assurance tests. The URL to the SonarQube service in SERRANO is the following:

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



serrano-sonarqube.rid-intrasoft.eu/admin/marketplace

Administration

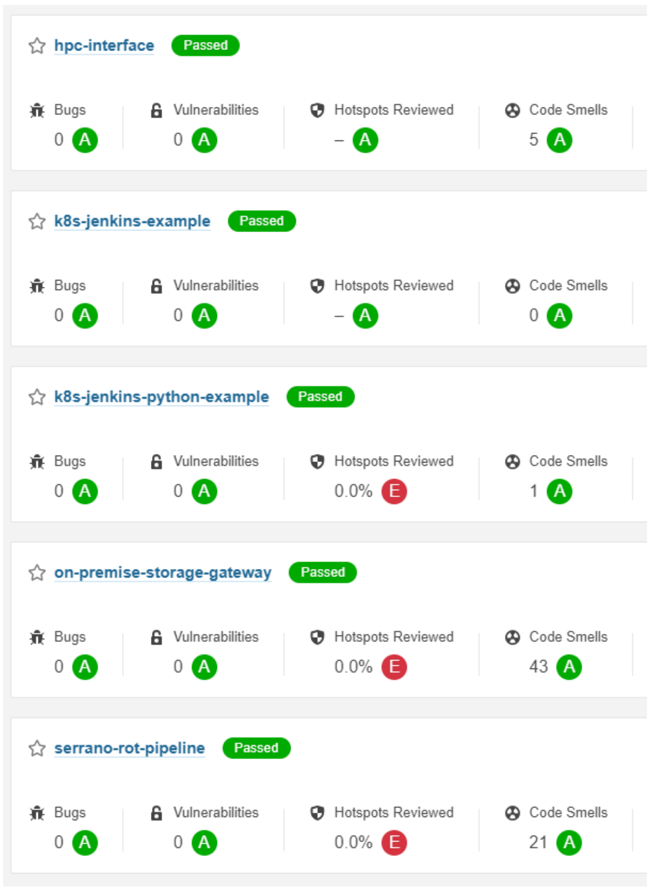
Request a free trial | Request a free trial | Request a free trial

Plugins

Plugins available in the MarketPlace are not provided or supported by SonarSource. Please reach out directly to their maintainers for support.

| | | | |
|---|---|---|---|
| 1C (BSL) Community Plugin LANGUAGES<br>Code Analyzer for 1C (BSL) | 1.11.0 13 new rules, bug fixes, support new reporters ••• | Homepage Issue Tracker<br>Licensed under GNU LGPL v3<br>Developed by 1c-syntax | Install |
| AEM Rules for SonarQube EXTERNAL ANALYSERS<br>Adds rules for AEM Java development | 1.6 SonarQube 8.9 LTS compatiblity release due to underlying Java plugin API changes ••• | Homepage Issue Tracker<br>Licensed under The Apache Soft...<br>Developed by Wunderman Thompson Technology | Install |
| Ansible Lint EXTERNAL ANALYZERS<br>Analyze Ansible playbooks. | 2.5.1 Support for SonarQube 9.2 ••• | Homepage Issue Tracker<br>Licensed under Apache License, ... | Install |
| Apigee EXTERNAL ANALYZERS<br>Adds XML rules test Apigee apiproxies. | 3.0.1 Support for SQ 8.8+ ••• | Homepage Issue Tracker<br>Licensed under Apache License, ...<br>Developed by Crédit Mutuel Arkéa | Install |
| Azure Active Directory (AAD) Authentication Plug-in for SonarQube INTEGRATION<br>Allows the use of Azure Active Directory as an authentication source for SonarQube. | 1.2.0 Group sync enhancements including transitive members. Code cleanup. ••• | Homepage Issue Tracker<br>Licensed under The MIT License (...<br>Developed by ALM \| DevOps Rangers | Install |
| Bitbucket Authentication for SonarQube | 1.1 (BUILD 381) Team restriction and account renaming | Homepage Issue Tracker | |

Figure 79: SonarQube in SERRANO CI/CD



| Project | Status | Bugs | Vulnerabilities | Hotspots Reviewed | Code Smells |
|---|---|---|---|---|---|
| hpc-interface | Passed | 0 A | 0 A | – A | 5 A |
| k8s-jenkins-example | Passed | 0 A | 0 A | – A | 0 A |
| k8s-jenkins-python-example | Passed | 0 A | 0 A | 0.0% E | 1 A |
| on-premise-storage-gateway | Passed | 0 A | 0 A | 0.0% E | 43 A |
| serrano-rot-pipeline | Passed | 0 A | 0 A | 0.0% E | 21 A |

Figure 80: 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 [42] has been chosen to be deployed as a reverse proxy in front of SERRANO CI/CD services. As depicted in Figure 74, 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

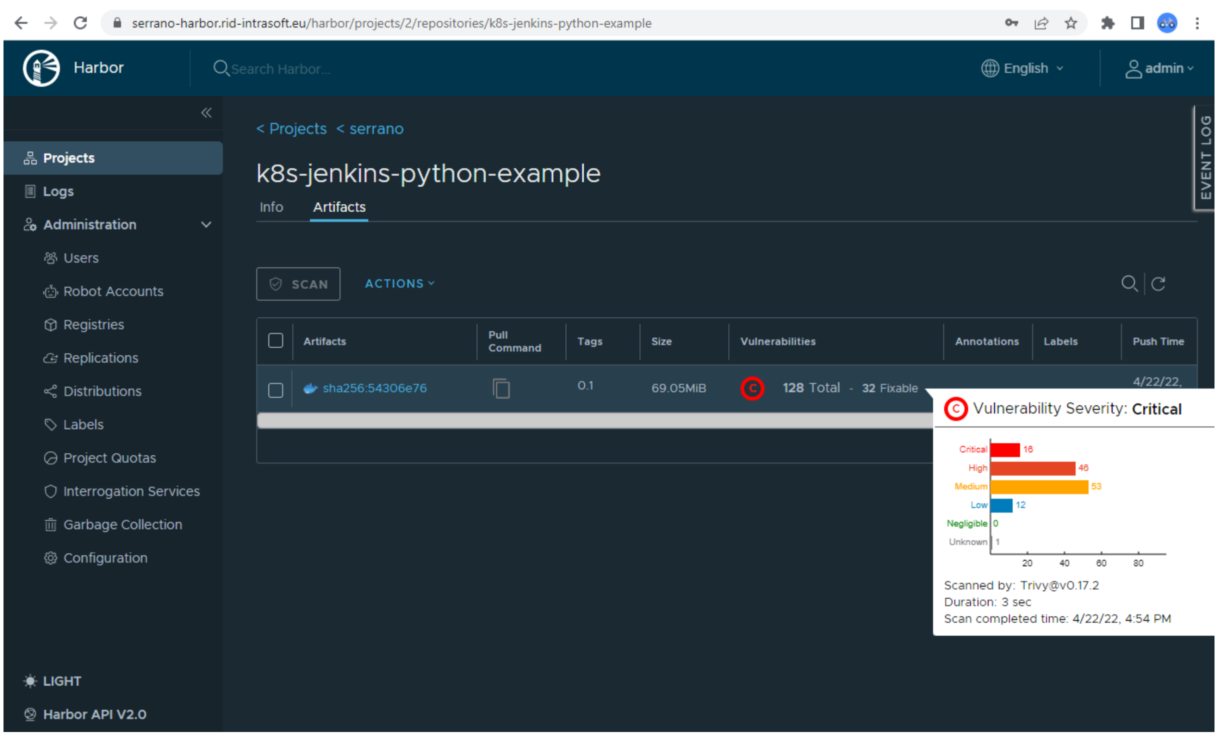
Harbor [41] is an open-source registry that secures artifacts with policies and role-based access control, ensures images are scanned and free from vulnerabilities, and signs images as trusted. Harbor delivers compliance, performance, and interoperability towards securely managing artifacts across cloud 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 [43] 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 [44], RHEL [45], CentOS [46], 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 81 and Figure 82, 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.



serrano-harbor.rid-intrasoft.eu/harbor/projects/2/repositories/k8s-jenkins-python-example

< Projects < serrano

k8s-jenkins-python-example

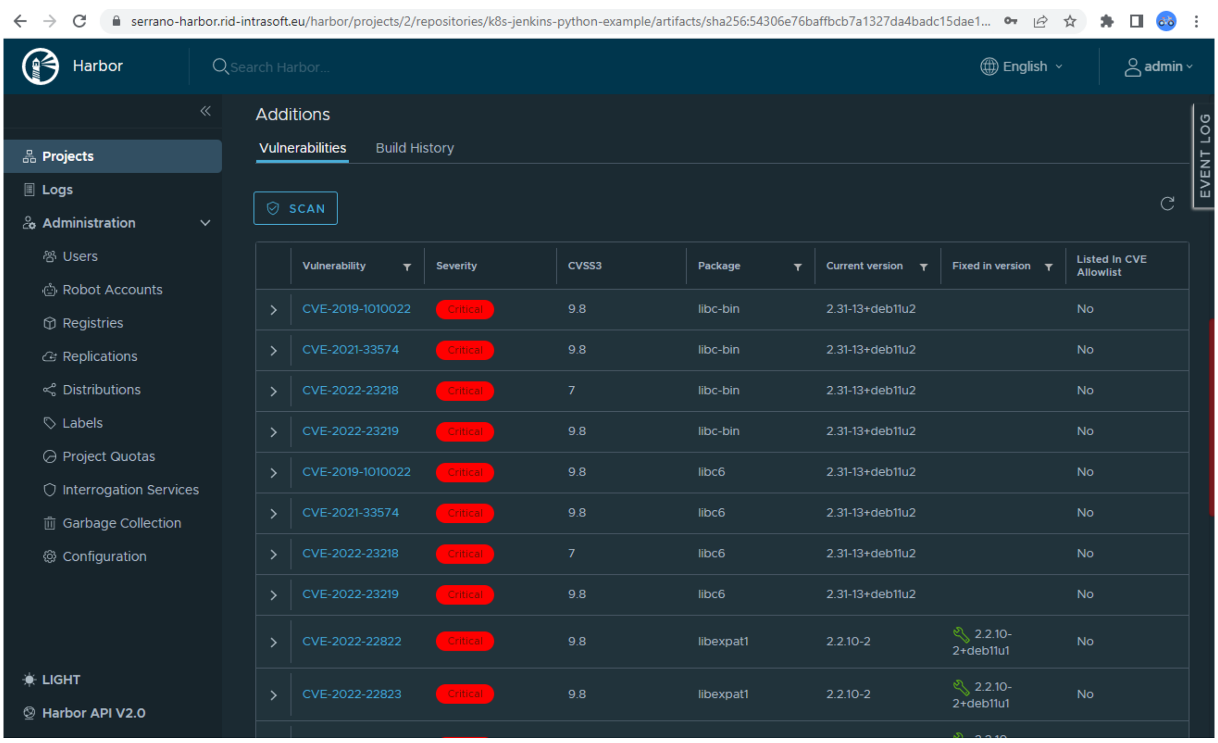
| Artifacts | Pull Command | Tags | Size | Vulnerabilities | Annotations | Labels | Push Time |
|---|---|---|---|---|---|---|---|
| sha256:54306e76 | | 0.1 | 69.05MiB | 128 Total · 32 Fixable | | | 4/22/22, |

Vulnerability Severity: Critical

| | |
|---|---|
| Critical | 16 |
| High | 46 |
| Medium | 53 |
| Low | 12 |
| Negligible | 0 |
| Unknown | 1 |

Scanned by: Trivy@v0.17.2
Duration: 3 sec
Scan completed time: 4/22/22, 4:54 PM

Figure 81: Trivy scan result overview on Harbor



serrano-harbor.rid-intrasoft.eu/harbor/projects/2/repositories/k8s-jenkins-python-example/artifacts/sha256:54306e76baffbcb7a1327da4badc15dae1...

Additions

| Vulnerability | Severity | CVSS3 | Package | Current version | Fixed in version | Listed In CVE Allowlist |
|---|---|---|---|---|---|---|
| CVE-2019-1010022 | Critical | 9.8 | libc-bin | 2.31-13+deb11u2 | | No |
| CVE-2021-33574 | Critical | 9.8 | libc-bin | 2.31-13+deb11u2 | | No |
| CVE-2022-23218 | Critical | 7 | libc-bin | 2.31-13+deb11u2 | | No |
| CVE-2022-23219 | Critical | 9.8 | libc-bin | 2.31-13+deb11u2 | | No |
| CVE-2019-1010022 | Critical | 9.8 | libc6 | 2.31-13+deb11u2 | | No |
| CVE-2021-33574 | Critical | 9.8 | libc6 | 2.31-13+deb11u2 | | No |
| CVE-2022-23218 | Critical | 7 | libc6 | 2.31-13+deb11u2 | | No |
| CVE-2022-23219 | Critical | 9.8 | libc6 | 2.31-13+deb11u2 | | No |
| CVE-2022-22822 | Critical | 9.8 | libexpat1 | 2.2.10-2 | 2.2.10-2+deb11u1 | No |
| CVE-2022-22823 | Critical | 9.8 | libexpat1 | 2.2.10-2 | 2.2.10-2+deb11u1 | No |

Figure 82: Trivy scan result details on Harbor

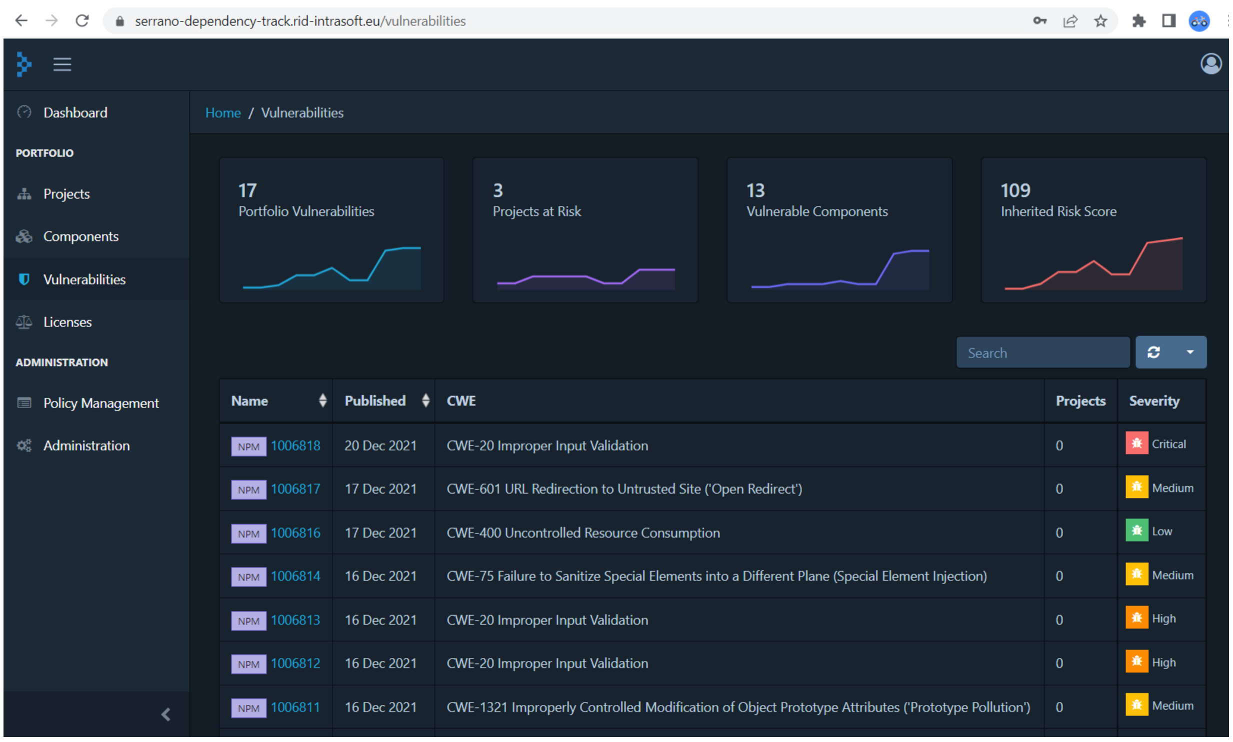
## 5.2.7 Dependency-Track

Dependency-Track [47] 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/



17 Portfolio Vulnerabilities | 3 Projects at Risk | 13 Vulnerable Components | 109 Inherited Risk Score

| Name | Published | CWE | Projects | Severity |
|---|---|---|---|---|
| NPM 1006818 | 20 Dec 2021 | CWE-20 Improper Input Validation | 0 | Critical |
| NPM 1006817 | 17 Dec 2021 | CWE-601 URL Redirection to Untrusted Site ('Open Redirect') | 0 | Medium |
| NPM 1006816 | 17 Dec 2021 | CWE-400 Uncontrolled Resource Consumption | 0 | Low |
| NPM 1006814 | 16 Dec 2021 | CWE-75 Failure to Sanitize Special Elements into a Different Plane (Special Element Injection) | 0 | Medium |
| NPM 1006813 | 16 Dec 2021 | CWE-20 Improper Input Validation | 0 | High |
| NPM 1006812 | 16 Dec 2021 | CWE-20 Improper Input Validation | 0 | High |
| NPM 1006811 | 16 Dec 2021 | CWE-1321 Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution') | 0 | Medium |

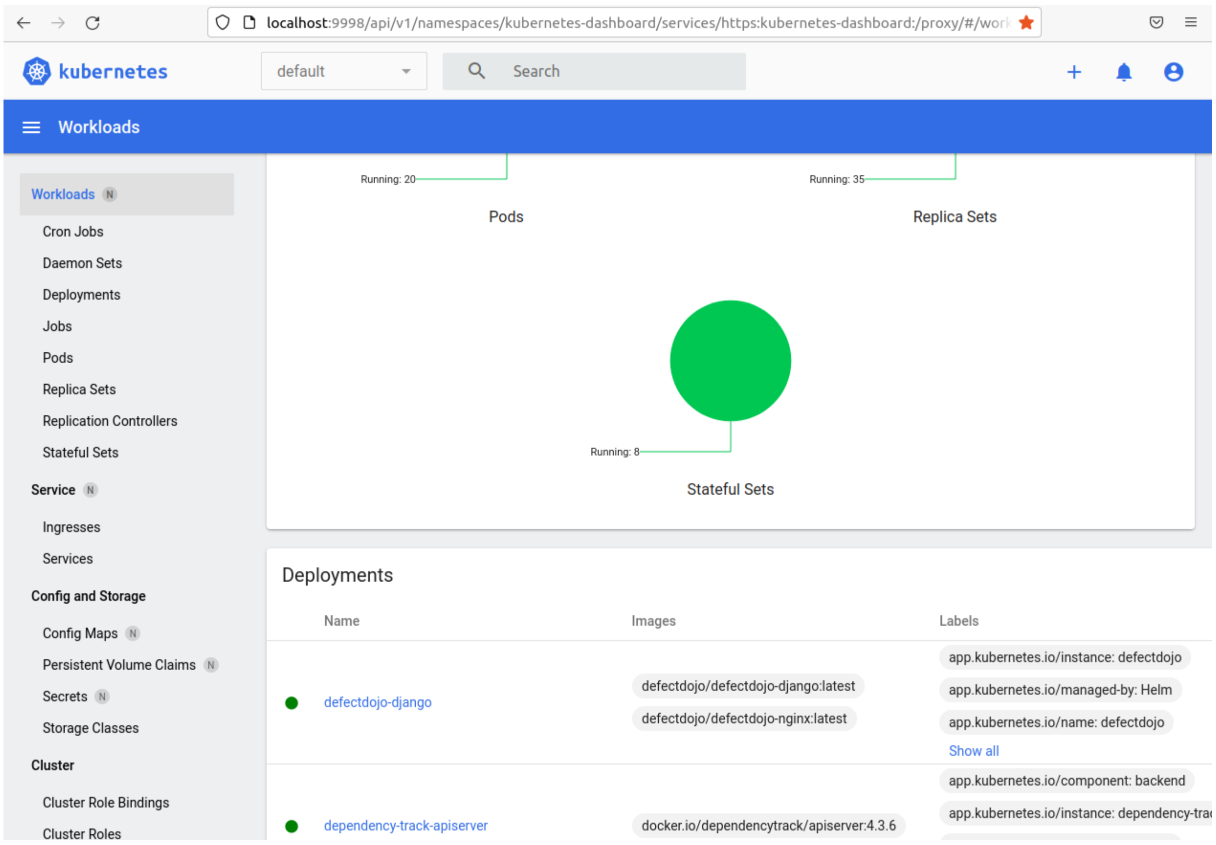
**Figure 83: Dependency-Track in SERRANO CI/CD**

## 5.2.8 Kubernetes

Kubernetes [49], 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 [50] (CNCF).



Deployments

| | Name | Images | Labels |
|---|---|---|---|
| ● | defectdojo-django | defectdojo/defectdojo-django:latest<br>defectdojo/defectdojo-nginx:latest | app.kubernetes.io/instance: defectdojo<br>app.kubernetes.io/managed-by: Helm<br>app.kubernetes.io/name: defectdojo<br>Show all |
| ● | dependency-track-apiserver | docker.io/dependencytrack/apiserver:4.3.6 | app.kubernetes.io/component: backend<br>app.kubernetes.io/instance: dependency-tra |

**Figure 84: Kubernetes Dashboard in SERRANO CI/CD**

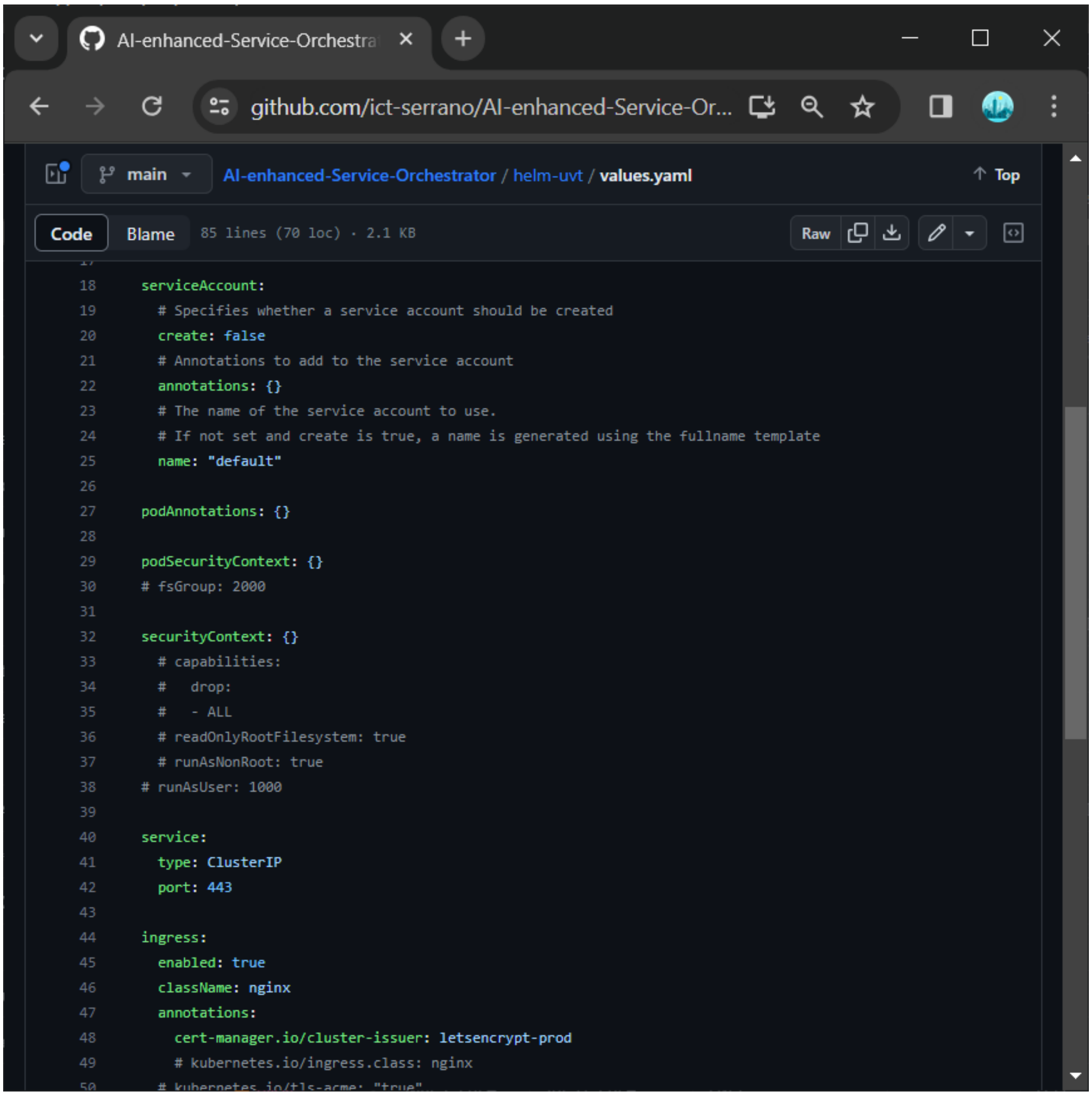
### *5.2.8.1 Helm*

Helm [51] 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.


AI-enhanced-Service-Orchestra
github.com/ict-serrano/AI-enhanced-Service-Or...
main
AI-enhanced-Service-Orchestrator / helm-uvt / values.yaml
Top
Code
Blame
85 lines (70 loc) · 2.1 KB
Raw

```yaml
serviceAccount:
  # Specifies whether a service account should be created
  create: false
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname template
  name: "default"

podAnnotations: {}

podSecurityContext: {}
# fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
# runAsUser: 1000

service:
  type: ClusterIP
  port: 443

ingress:
  enabled: true
  className: nginx
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    # kubernetes.io/ingress.class: nginx
```

Figure 85: Helm charts in SERRANO on GitHub

# 6 Software Deployment Specifications and Validation

## 6.1 Continuous Integration/Continuous Deployment Processes

Figure 86 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 GitHub, which triggers 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 are deployed to the K8s clusters of the integration and operational environments.

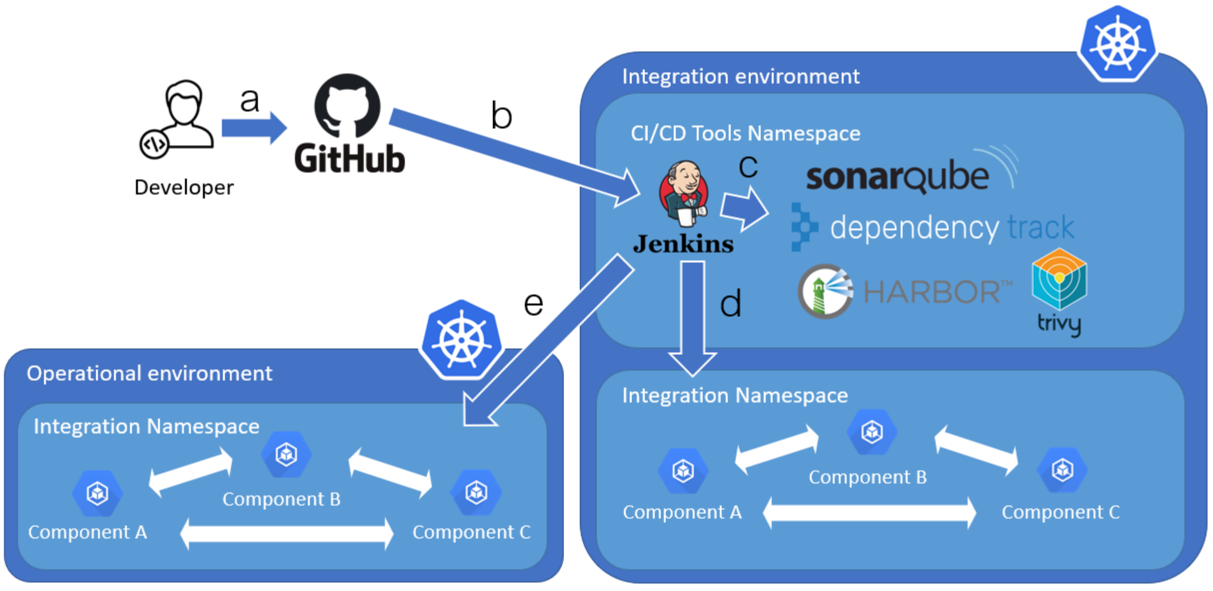


**Figure 86: Procedure for developing and releasing the software components**

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 GitHub https://github.com/ict-serrano
- 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/
- Step 9: Deploy component to integration environment
    - Sample helm charts are available in https://github.com/ict-serrano/k8s-jenkins-python-example/tree/main/helm and https://github.com/ict-serrano/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://github.com/ict-serrano/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://github.com/ict-serrano/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 GitHub 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 *.services.cloud.ict-serrano.eu.

Also, similar instructions are provided to partners that do not intend to push the source of their solutions to the SERRANO GitHub 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 GitHub https://github.com/ict-serrano
- Step 6: Check image scanning report coming from Trivy on Harbor https://serrano-harbor.rid-intrasoft.eu/
- Step 7: Deploy component to integration environment
  - Sample helm charts are available in https://github.com/ict-serrano/k8s-jenkins-python-example/tree/main/helm and https://github.com/ict-serrano/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://github.com/ict-serrano/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://github.com/ict-serrano/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 GitHub 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 ***.services.cloud.ict-serrano.eu** 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 [52] is one such tool, which can automatically generate a user interface (UI) from the OpenAPI specification.

Developers in the SERRANO easily communicate through exchanging usage examples of the API in the form of Postman [53] or Insomnia [54] collections. These tools also provided the ability to exchange full request examples in addition to Swagger.

### 6.2.2 Development using containers

Components developed or used in SERRANO that require a runtime environment for their execution provide a Docker image for their deployment. Libraries and other components that can be dynamically imported usually do not require the creation of a container, but they can be stored in suitable repositories, such as PyPI  [55].

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.

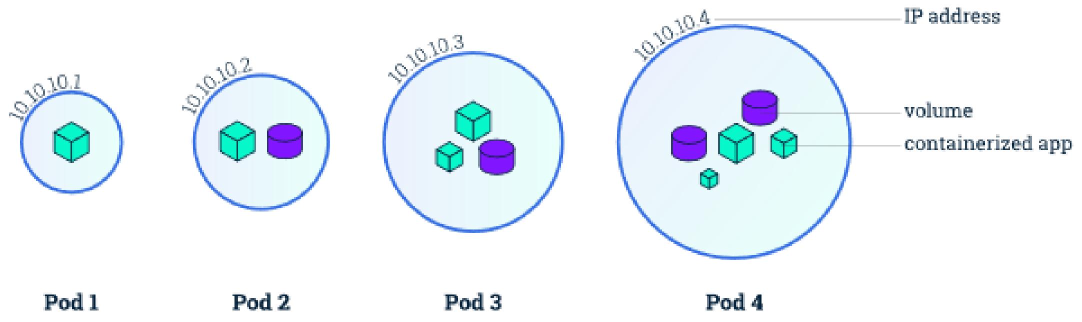
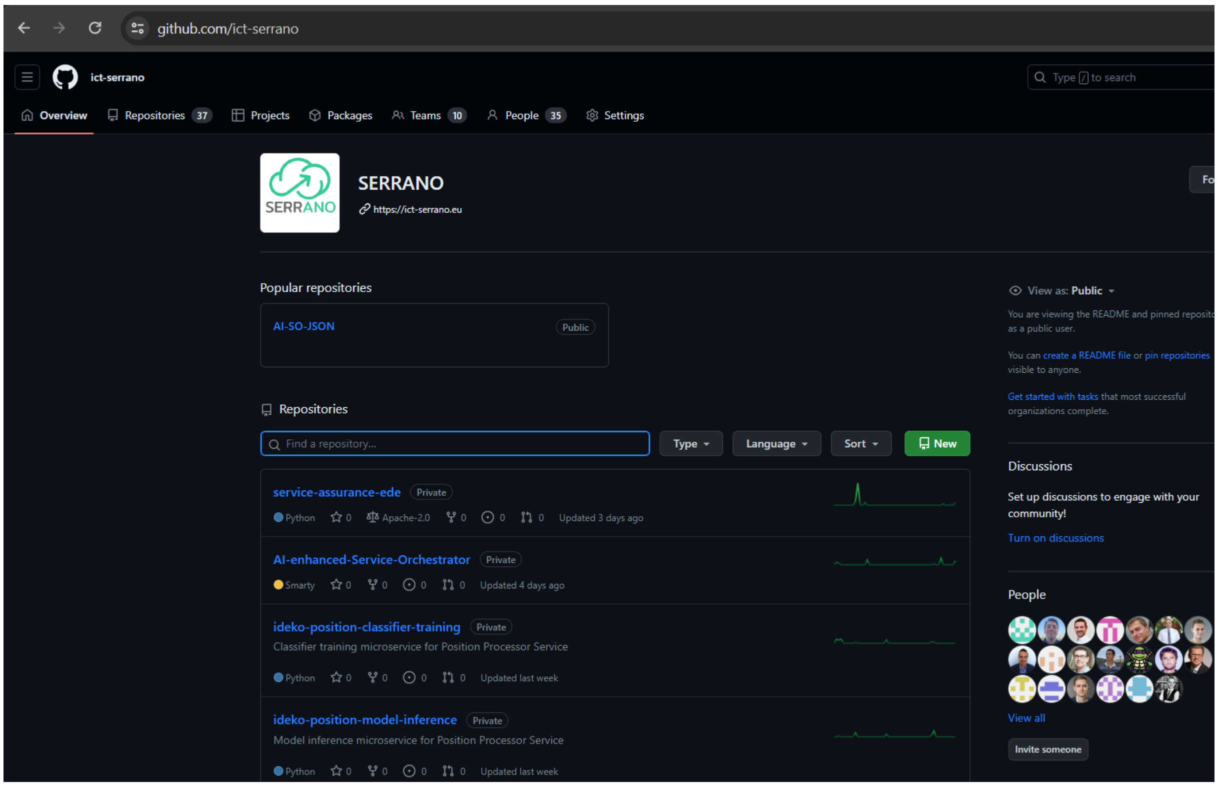


**Figure 87: Containers in Kubernetes pods**

## 6.2.3 Code and Deployment configuration on GitHub

GitHub is the code repository of choice for every component. The code is placed in the repositories of ***"ict-serrano"*** project, as shown in Figure 88.



SERRANO
https://ict-serrano.eu

Popular repositories

AI-SO-JSON Public

Repositories

service-assurance-ede Private
Python 0 Apache-2.0 0 0 0 Updated 3 days ago

AI-enhanced-Service-Orchestrator Private
Smarty 0 0 0 0 Updated 4 days ago

ideko-position-classifier-training Private
Classifier training microservice for Position Processor Service
Python 0 0 0 0 Updated last week

ideko-position-model-inference Private
Model inference microservice for Position Processor Service
Python 0 0 0 0 Updated last week

View as: Public
You are viewing the README and pinned reposit
as a public user.
You can create a README file or pin repositories
visible to anyone.
Get started with tasks that most successful organizations complete.

Discussions
Set up discussions to engage with your community!

People

**Figure 88: GitHub repositories**

## 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. In addition, the earlier version should also contain a number of unit and integration tests that prevent developers from breaking functionality that worked in the last version before new changes were introduced.

Automated unit, integration, and functional security tests were 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 approac. Special focus is put on the capability of SonarQube to analyse 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 Results

All SERRANO components that are part of the platform or the use cases have been 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 14: 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<br>● Components receive notifications<br>● Setup execution requests for ROT<br>● Handle ROT response<br>● Preparation of deployments<br>● Orchestration Manager and Orchestration Drivers interaction | 48 | ● Orchestration Controller REST<br>● Orchestration Manager REST<br>● Orchestration Drivers REST<br>● ROT REST<br>● Telemetry REST | 38 |

| | | | | |
|---|---|---|---|---|
| | for new deployments<br>● Testing interaction with K8s for the actual deployment<br>● Testing integration with K8s for kernel execution<br>● Creation and management of secure storage policies<br>● Testing integration with HPC for kernel executions | | | |
| Resource Optimization Toolkit (ROT) | ● Execution Engine registration to Controller<br>● Valid and invalid execution requests to Controller<br>● Valid and invalid termination requests to Controller<br>● Interaction with telemetry framework<br>● Assignment of requests to Execution Engine<br>● Load and execute selected algorithm<br>● Forward results to Controller<br>● Detect performance issues on engines<br>● Execute the three integrated algorithms with valid and invalid input parameters | 24 | ● ROT REST<br>● AMQP publisher<br>● AMQP publisher<br>● Telemetry REST | 17 |
| Message Broker | MQTT service:<br>● topic creation<br>● topic subscription<br>● publish messages<br>● receive messages<br>AMQP service:<br>● setup basic publisher, | 26 | ● MQTT connector<br>● MQTT publisher<br>● MQTT consumer<br>● AMQP connector<br>● AMQP publisher<br>● AMQP consumer | 23 |

| | | | | |
|---|---|---|---|---|
| | consumer, exchange messages<br>● setup topic publisher, consumer, exchange messages<br>● setup fanout publisher, consumer, exchange messages<br>● setup route publisher, consumer, exchange messages | | | |
| Telemetry Framework | ● Collection of resource characteristics and monitoring data<br>● Monitoring probes registration to agents<br>● Update probes' configuration<br>● Detect events related to operational state of telemetry components<br>● Store collected information to operational databases<br>● Forward telemetry data to Central Handler<br>● Provision of monitoring data to orchestration mechanisms<br>● Support of streaming telemetry<br>● Store telemetry data to PMDS<br>● Retrieve historical telemetry data from PMDS | 41 | ● Monitoring probes REST<br>● Enhanced Telemetry Agent REST<br>● Central Telemetry Handler REST<br>● PMDS REST<br>● Stream Handler | 34 |
| AI-enhanced Service Orchestrator | ● Can identify application requirements and deployment description | 18 | ● Resource Orchestrator REST API calls | 12 |

| | | | | |
|---|---|---|---|---|
| | • Ensures that Application Model Constraints are properly formed<br>• Application Constraints are properly translated to Resource Constraints based on mapping rules specified<br>• Possible Application Deployment Scenarios are being developed<br>• Telemetry Data is being dynamically retrieved<br>• Application is being properly deployed through Resource Orchestrator | | • Central Telemetry Handler REST API calls<br>• Incoming requests from Alien4Cloud | |
| HPC system hardware interface | • List of HPC services<br>• Infrastructure model creation<br>• Infrastructure telemetry<br>• Job submission<br>• Job status retrieval<br>• Data movement between S3 and HPC infrastructure for input data and job results | 40 | • SSH communication between the Gateway and HPC infrastructure<br>• REST API calls<br>• S3 API calls | 38 |
| HW Acceleration abstractions | • Static API function call<br>• Dynamic API application port<br>• Generic/debug plugin functionality<br>• Custom CPU/FPGA/GPU plugins | 6 | • VM application execution with hardware accelerator | 2 |
| Trusted Virtualization | • Boot process<br>• K8s integration | 2 | • Signed kernel booting & node joining k8s cluster | 1 |
| Secure storage on-premises gateway | These features are provided by the On- | 38 | These features are implemented through integration of the On- | 101 |

| | | | | |
|---|---|---|---|---|
| | premises Storage Gateway:<br>● File caching functionality<br>● HTTP Range header, as defined by IETF RFC 7233 | | premises Storage Gateway with the Skyflok.com backend and the SERRANO edge devices:<br>● Bucket creation, deletion, list contents,<br>● Object creation, deletion, retrieval<br>● Multipart upload process: create, complete, abort, upload part, list parts, list multipart uploads,<br>● AWS Boto3 end-to-end file upload and download features. | |
| TLS offloading | ● TLS handshake procedure validation<br>● HTTP packet information inclusion<br>● TLS status signalling | 10 | The features are integrated into libraries that are placed under the DOCA DPU SDK for developer utilization. | 22 |
| Service assurance | ● SA configuration parameters validation<br>● SA subcomponent distributed cluster configuration validation<br>● SA subcomponent inference functional tests (validation, detection, model selection etc.)<br>● SA subcomponent data bus connector validation (stream handler connection related tests) | 27 | ● SA Connector API<br>● SA Inference API<br>● SA Data bus Connector API | 32 |

**Table 15: 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 has been performed as part of the testing for the Secure storage on-premises gateway. | 38 | • Component integration testing for the Secure Storage Use Case has been performed as part of the testing for the Secure storage on-premises gateway. | 101 |
| Fintech Analysis Use Case Integrated Functionality | • Activity completion for data retrieval from storage<br>• Activity completion for investment strategy application<br>• Activity completion for the creation of investment profiles<br>• Activity completion for portfolio creation | 5 | • GPU/FPGA acceleration<br>• Secure Storage API interface interaction | 17 |
| Anomaly Detection in Manufacturing Settings Integrated Functionality | • Validate the correct sending of data streaming from the ball screw sensors to the MQTT Broker.<br>• Validate the services and microservices developed for the detection of anomalies in ball screw. Triggering different functionalities using specific topics.<br>• 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. | 12 | • Data Broker connection (Message Broker through. MQTT protocol)<br>• GPU/FPGA acceleration<br>• Secure Storage API interface interaction | 25 |

# 7 Conclusions

Deliverable 6.7 reports on the work performed in WP6 to develop the final SERRANO integrated platform. The WP6 activities related to D6.7 aim to unify the outcomes of the developed components and services in WP3-5 to release the integrated SERRANO platform.

In particular, the deliverable presents an overview of the SERRANO platform, including the final release description, 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 final platform release described in this deliverable aims to provide the fully fletched functional prototype that supports the core functionalities of SERRANO's three use cases. The final evaluation of these use cases is reported on deliverable D6.8 "Final version of business, end user and technical evaluation" (M36).

# 8 References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[60] Eclipse Paho: https://eclipse.dev/paho/index.php?page=clients/python/index.php

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

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

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

[64] gRPC- high performance, open-source universal RPC framework: https://grpc.io

[65] https://vaccel.org.

[66] https://www.qemu.org/

[67] https://firecracker-microvm.github.io/

[68] https://www.cloudhypervisor.org/

[69] https://katacontainers.io/

[70] https://github.com/cloudkernels/qemu-vaccel/tree/vaccelrt

[71] https://github.com/cloudkernels/firecracker/tree/vaccel-0.23

[72] https://docs.vaccel.org/python_bindings/

[73] https://docs.vaccel.org/tensorflow_bindings/

[74] https://docs.vaccel.org

[75] https://www.openfaas.com/

[76] https://alien4cloud.github.io/

[77] RedHat - https://cloud.redhat.com/blog/signing-and-verifying-container-images

[78] https://docs.sigstore.dev/policy-controller/overview/

[79] https://github.com/sigstore/cosign

[80] Binz, Tobias, et al. "Portable cloud services using tosca." IEEE Internet Computing 16.3 (2012): 80-85.

[81] L. Smith, "Architectures for secure computing systems," MITRE CORP BEDFORD MASS, 1975.

[82] Sigstore - https://www.sigstore.dev

[83] Deliverable D2.3 “SERRANO architecture” – SERRANO Consortium

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

[85] Deliverable D2.5 “Final version of SERRANO architecture” – SERRANO Consortium

[86] Deliverable D3.4 “Final release of SERRANO Secure Infrastructure Layer” – SERRANO Consortium

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

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

[89] Deliverable D4.4 “Final Release of the SERRANO Cloud and Edge Acceleration Platforms and Tools” – SERRANO Consortium

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

[91] Deliverable D5.3 “Resource orchestration, telemetry and lightweight virtualization mechanisms” – SERRANO Consortium

[92] Deliverable D5.4 “Intelligent service and resource orchestration mechanisms” – SERRANO Consortium