
| | |
|---|---|
| Project Title | AN OPEN, TRUSTED FOG COMPUTING PLATFORM FACILITATING THE DEPLOYMENT, ORCHESTRATION AND MANAGEMENT OF SCALABLE, HETEROGENEOUS AND SECURE IOT SERVICES AND CROSS-CLOUD APPS |
| Project Acronym | RAINBOW |
| Grant Agreement No | 871403 |
| Instrument | Research and Innovation action |
| Call / Topic | H2020-ICT-2019-2020 / Cloud Computing |
| Start Date of Project | 01/01/2020 |
| Duration of Project | 36 months |

# D5.2 – RAINBOW Integrated Platform and Unified Dashboard - Early Release

| | |
|---|---|
| Work Package | WP5 – Continuous Integration and Accessibility |
| Lead Author (Org) | Ioannis Avramidis, Alex Bensenousi (INTRASOFT) |
| Contributing Author(s) (Org) | Konstantinos Theodosiou, Giannis Ledakis (UBI); Raphael Schermann (IFAT); Thomas Pusztai (TUW); Demetris Trihinas (UCY); Heini Bergsson Debes (DTU); Stefanos Venios (SUITE 5); Casseti Claudio Ettore (POLITO); Theodoros Toliopoulos (AUTH) |
| Due Date | 30.06.2021 |
| Actual Date of Submission | 07.07.2021 |
| Version | V1.0 |

**Dissemination Level**

| | |
|---|---|
| x | PU: Public (*on-line platform) |
| | PP: Restricted to other programme participants (including the Commission) |
| | RE: Restricted to a group specified by the consortium (including the Commission) |
| | CO: Confidential, only for members of the consortium (including the Commission) |


The work described in this document has been conducted within the project RAINBOW. This project has received funding from the European Union's Horizon 2020 (H2020) research and innovation programme under the Grant Agreement no 871403. This document does not represent the opinion of the European Union, and the European Union is not responsible for any use that might be made of such content.

## Versioning and contribution history

| Version | Date | Author | Notes |
|---|---|---|---|
| 0.1 | 05.05.2021 | Ioannis Avramidis, Alex Bensenousi (INTRASOFT) | Initial ToC |
| 0.4 | 22.05.2021 | Ioannis Avramidis, Alex Bensenousi (INTRASOFT) | First Draft |
| 0.5 | 05.06.2021 | Konstantinos Theodosiou, Giannis Ledakis (UBI); Raphael Schermann (IFAT); Thomas Pusztai (TUW); Demetris Trihinas (UCY); Heini Bergsson Debes (DTU); Stefanos Venios (SUITE 5); Casseti Claudio Ettore (POLITO); Theodoros Toliopoulos (AUTH); | Contributed to components integration status, early release status, and unit testing. |
| 0.6 | 12.06.2021 | Demetris Trihinas (UCY) | 1st Review |
| 0.65 | 13.06.2021 | Giannis Ledakis (UBI) | 2nd Review |
| 0.7 | 22.06.2021 | Ioannis Avramidis, Alex Bensenousi (INTRASOFT) | Consolidation of comments and contributions |
| 0.8 | 25.06.2021 | Ioannis Avramidis, Alex Bensenousi (INTRASOFT) | Additions in Introduction, Exec. Summary, Conclusions |
| 0.9 | 30.06.2021 | Ioannis Avramidis, Alex Bensenousi (INTRASOFT) | 1st complete version released |
| 1.0 | 07.07.2021 | Ioannis Avramidis, Alex Bensenousi (INTRASOFT) | Addressing Reviewers' comments and Final version |

# Table of Contents

# List of tables

# List of figures



# List of acronyms

| Acronym | Full name |
|---|---|
| **AK** | Attestation Key |
| **AOT** | Ahead-Of-Time |
| **API** | Application Programming Interface |
| **CI/CD** | Continuous Integration/Continuous development |
| **CPU** | Central Processing Unit |
| **DAA** | Direct Anonymous Attestation |

| Acronym | Full name |
|---|---|
| **DAG** | Direct Acyclic Graphs |
| **DHT** | Distributed Hypertext |
| **GPS** | Global Positioning System |
| **HTTP** | Hypertext Transfer Protocol |
| **IoT** | Internet of Things |
| **IT** | Integration Testing |
| **IPR** | Intellectual Property Rights |
| **ORA** | Oblivious Remote Attestation |
| **PCR** | Platform Configuration Register |
| **RAM** | Random Access Memory |
| **REST** | Representational state transfer |
| **SDK** | Software Development Kit |
| **SGC** | Service-Graph Chain |
| **SLO** | Service Level Objectives |
| **SLOC** | Source Lines Of Code |
| **S-ZTP** | Secure Zero Touch Provisioning |
| **S-ZTP CIV** | Zero-Touch Configuration Integrity Verification |
| **TPM** | Trusted Platform Module |
| **TSS** | Technology Support Services |
| **UI** | User Interface |
| **URL** | Uniform Resource Locator |
| **UT** | Unit Testing |
| **VCS** | Version control systems |
| **WPx** | Work Package |
| **YAML** | *Ain't Markup Language* |

# Executive Summary

This deliverable is a public report summarizing the implementation approach of the 1st software release of the RAINBOW integrated platform. It presents the status of the components up to M18 and provides an updated version of the architecture. D5.2 comprises the 1st version out of total 3 distinct versions which are scheduled to be submitted with the three different releases of the RAINBOW Platform. The 2nd version will be published in M27 (D5.3 RAINBOW Integrated Platform and Unified Dashboard – Second Release) and the 3rd and final version will be submitted by M36 in D5.4 RAINBOW Integrated Platform and Unified Dashboard – Final Release. Therefore, this is a live document that will be constantly updated to depict the developments and releases of the RAINBOW platform up until the end of the project.

The status of the integrated platform is also provided. For this 1st confidential prototype, partial integration has been achieved to offer basic functionalities of the platform. The goal was to support functionalities as the proper definition of application graphs and their deployment over cloud and edge resources, thus allowing the planning and the execution of the first prototypes of the RAINBOW demonstrators. In addition, this prototype supports scaling of the service graphs based on (Service Level Objectives) SLOs that consider different metrics, collected, and analysed by the Monitoring and Analytics Services of RAINBOW. Furthermore, the attestation security enabler is supported, through which we can attest either the devices that want to join the cluster or the pre-existing ones.

This deliverable also provides an overview of the interaction of a user in the first release of the RAINBOW Dashboard, and the initial results of software quality evaluation. Finally, we provide the plans for the upcoming releases of the RAINBOW platform.

# 1 Introduction

Software integration is a process that involves following several multi-disciplinary approaches during the design and implementation phase. In this document, we provide documentation of the early release of the RAINBOW platform and explain the process of design and implementation, covering the phases of components' development, integration, and testing/evaluation.

## 1.1 Relationship with RAINBOW Deliverables

This deliverable is built on the foundation of the architecture initially defined in D1.2, the integration points defined in D5.1, the technical developments that have been performed in the scope of WP2, WP3 and WP4, and the processes for development, testing and integration as also defined in D5.1. The information presented in this deliverable is used as an addition to the 1st software release of the RAINBOW platform.

## 1.2 Structure of the deliverable

The rest of the deliverable is structured as follows.

- Section 2 is the core part of the document, which presents an updated version of the architecture, details the components, and provides an overview of platform status for this first release.
- Section 3 presents the first results of the testing and quality assurance processes.
- Section 4 presents the plans for the upcoming releases of the RAINBOW platform.
- Finally, Section 5 concludes the document.

# 2 RAINBOW Integrated Platform Overview

In this section, we provide the documentation regarding the first release of the platform, covering the updates we had to introduce to the architecture, the status of the components, and the status of the integrated platform in this first release.

## 2.1 RAINBOW Architecture

The architecture or RAINBOW has been initially presented in D1.2 [1] and was updated in the scope of D5.1 [2]. In this document, we provide the architecture as a starting point for the presentation of the RAINBOW integrated platform, and in addition, we provide some updates that better reflect the actual integrated platform. The changes in the architecture as presented in the Figure 1 below.

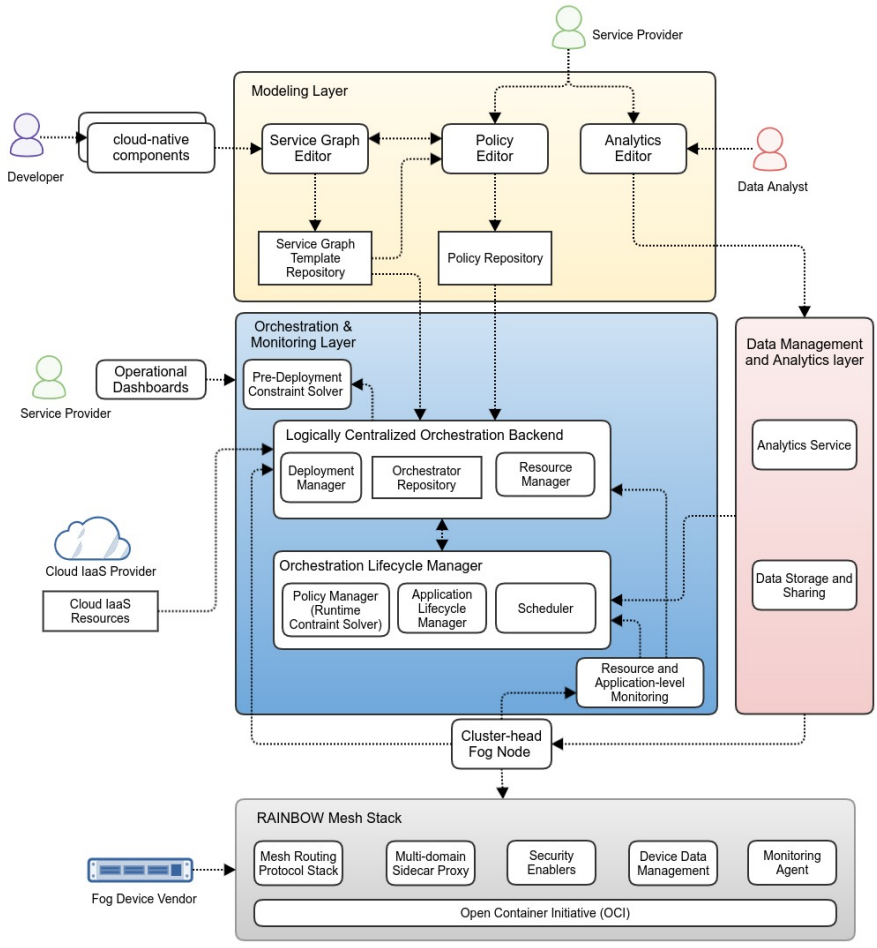


*Figure 1 RAINBOW Reference Architecture*

The changes were mainly in the Orchestration and Monitoring layer; two main packages of loosely coupled components are provided with the Logically Centralized Orchestration Backend and the Orchestration Lifecycle Manager. A new subcomponent called Scheduler being also introduced. Also, the Analytics Service, together with the newly added Data

Storage and Sharing component, build a new layer, the Data Management and Analytics layer. Finally, the Monitoring Agent is also included as part of the Mesh Stack.

In the next section, the components of these main entities, and where applicable, their subcomponents, are listed, explaining the provided functionalities in this first release and their integration as part of the platform. Furthermore, additional details are provided regarding the packaging and the setup of the components.

## 2.2 RAINBOW Components

### 2.2.1 Logically Centralized Orchestrator Backend

The RAINBOW Logically Centralized Orchestrator Backend enables the UI Editors to persist the necessary data to the database and authenticate and authorize the user to access them. It is part of the Orchestration and Monitoring Layer. Also, as this component host the repository, it helps the other RAINBOW components send and depict anything essential to the appropriate UI tool.

The Logically Centralized Orchestrator Backend is part of the first RAINBOW Platform release and it consists of three loosely coupled components: the resource manager, the deployment manager, and the *orchestrator repository,* as depicted in the figure below.

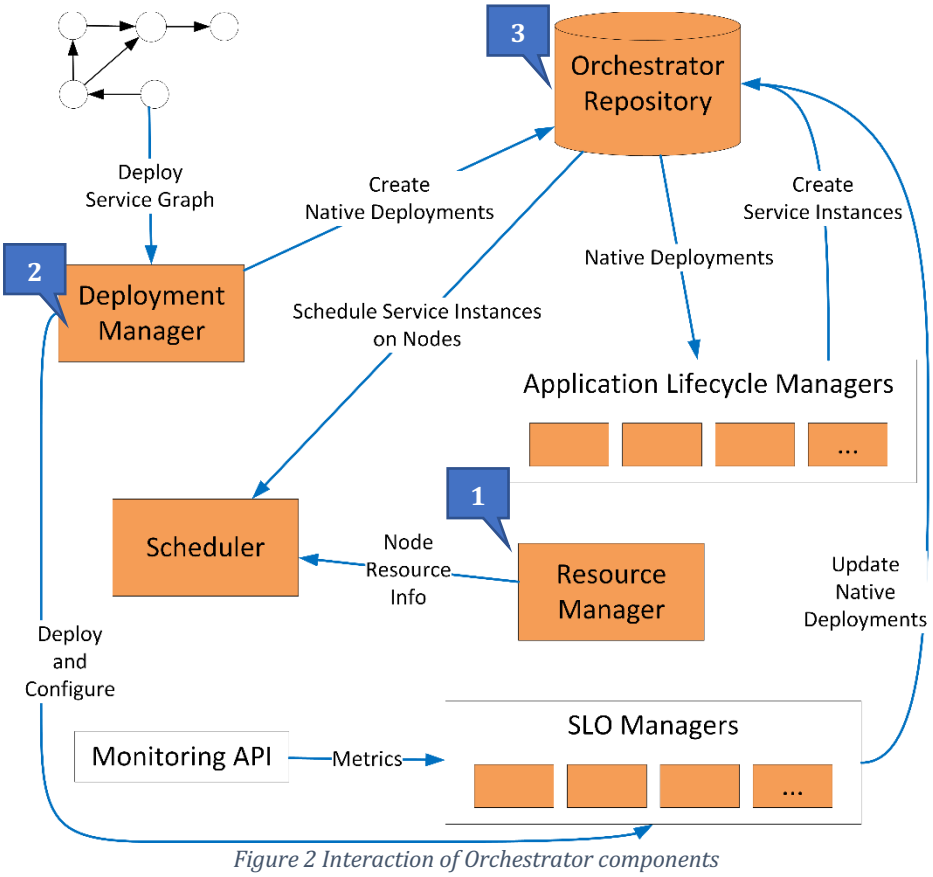


*Figure 2 Interaction of Orchestrator components*

#### 2.2.1.1 *Resource Manager*

Based on the state-of-the-art analysis and requirements definition performed in WP1, we chose Kubernetes as Resource Manager of the RAINBOW platform. In specific, for this release of RAINBOW, we use native Kubernetes V1.21 [3]. For the second and the final release, the handling of fog-specific resources (e.g., GPS or cameras) will be improved.

#### 2.2.1.2 *Deployment Manager*

The Deployment Manager is implemented as a Kubernetes controller for handling Service Graphs. It is implemented in Go [4] using kubebuilder [5]. In the first release, this controller can create and update Kubernetes-native deployments based on submitted Service Graphs and configure Service Level Objectives (SLOs) and basic monitoring.

This component relies on Service Graph Editor and the Kubernetes distribution used as Resource Manager. The integration of the Deployment Manager to the RAINBOW Orchestrator is complete for this first release, thus allowing the proper execution of deployment flows, as presented in section 2.3.2.

For the second and the final release, the monitoring configuration (that is added as part of the service graph) will be expanded, and information about the current deployment status will be added to the Service Graph.

#### 2.2.1.3 *Orchestrator Repository*

This repository is a MySQL database that keeps all the information generated during the deployment and orchestration process.

#### 2.2.1.4 *Integration and Component Dependencies*

This Orchestrator Backend and all its subcomponents depend on the Kubernetes distribution that is used. Kubernetes (v1.21 for this first release) must be configured appropriately so that Rainbow Orchestrator components can create and delete the deployments of the Service Graphs, as it also depends on a database that the templates of the Service Graphs and other related information are stored.

The implementation and integration of this component will continue until the next RAINBOW release.

#### 2.2.1.5 *Component Packaging and Distribution*

The Logically Centralized Orchestrator Backend is packaged as a single Docker container and made available through the RAINBOW container registry [6] that features a private Docker Image Hub.

#### 2.2.1.6 *Component Installation & Deployment*

A container image is provided for this component, which will be installed by using a provided deployment YAML that is compatible with the kubectl[7] tool used for Kubernetes deployments.
The deployment must be done to a control plane/master node, which must be of x86 CPU architecture.

### 2.2.2 Orchestration Lifecycle Manager

The Orchestration Lifecycle Manager is part of the Orchestration and Monitoring layer, and consists of three loosely coupled components, namely the *Scheduler*, the *Policy Managers* and the *Lifecycle Managers*:

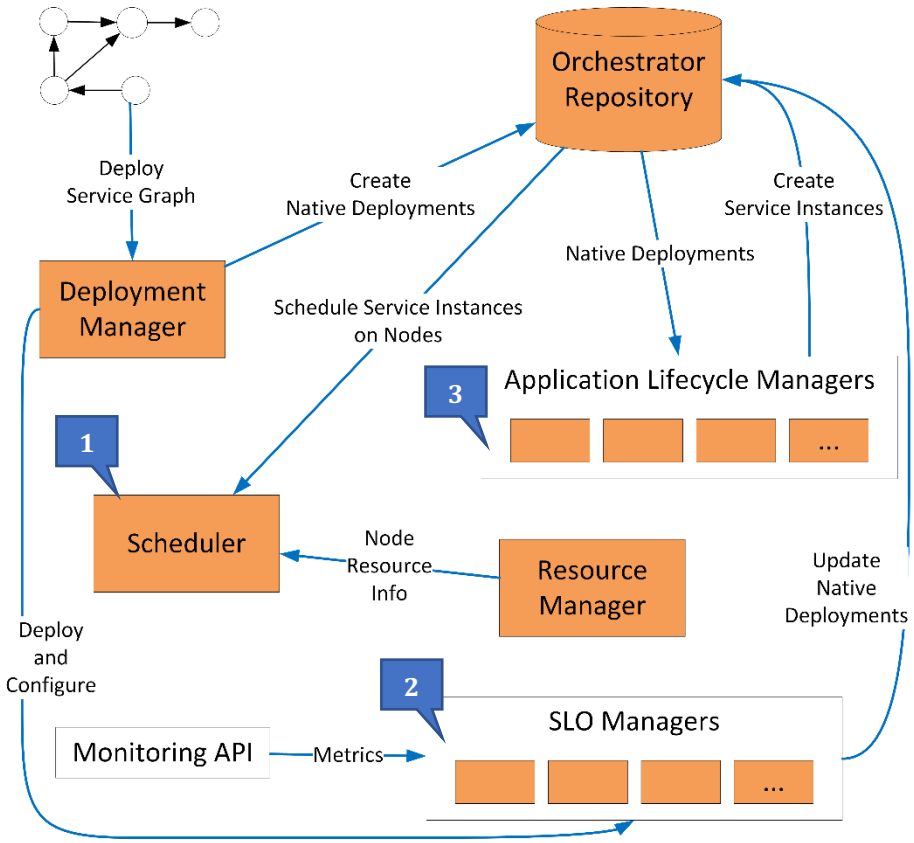


*Figure 3 Interaction of Orchestrator components*

#### 2.2.2.1 *Scheduler*

The Scheduler determines on which node a service should be placed. It is written in Go and is based on the Kubernetes Scheduling Framework [8]. Its integration with the RAINBOW Orchestrator is complete. A subset of the planned scheduling plugins is complete, the rest will be implemented for the final release.

#### 2.2.2.2 *SLO Policy Managers*

The Policy Managers monitor SLO compliance and trigger elasticity strategies upon violations. They are written in TypeScript using the Polaris/SLOC framework [9]. The managers for the average CPU usage SLO and the hazard detected SLO (for use case 2) have been implemented and integrated. The rest will follow for the second and the final release.

#### *2.2.2.3 Application Lifecycle Managers*

The Lifecycle Managers manage the service instances and execute the elasticity strategies. Service instances management is provided by Kubernetes natively. Elasticity strategies are custom Kubernetes controllers written in Go using kubebuilder. The elasticity strategies for horizontal scaling and service migration between cluster nodes are implemented and integrated.

#### *2.2.2.4 Integration and Component Dependencies*

This Lifecycle Manager and all its subcomponents depend on the Kubernetes distribution that is used. Kubernetes (v1.21 for this first release) must be configured appropriately so that Rainbow Orchestrator and the Lifecycle Manager works properly. It also depends on a database that the templates of the Service Graphs and other related information are stored.

#### *2.2.2.5 Component Packaging and Distribution*

For the packaging of this component, container images have been created and hosted in the projects docker registry.

#### *2.2.2.6 Component Installation & Deployment*

A container image is provided for this component, which will be installed by using a provided deployment YAML that is compatible with the kubectl tool used for Kubernetes deployments.
The deployment must be done to a control plane/master node, which must be of x86 CPU architecture.

### 2.2.3 Pre-deployment Constraint Solver

The Pre-deployment Constraint Solver a component of the Orchestration and Monitoring Layer that is under implementation and is available as a Kubernetes Admission Webhook [10] for Service Graphs. It will use Optaplanner [11] constraint solver to solve the issue of resource optimization. It is planned for the second platform release.

#### *2.2.3.1 Integration and Component Dependencies*

At this first release, the pre-deployment constraint solver has not been integrated with other components. However, it depends on the Service Graph Editor and the Kubernetes distribution used by RAINBOW (installed during RAINBOW setup).

#### *2.2.3.2 Component Packaging and Distribution*

For the packaging of this component, container images will be created and hosted in the projects docker registry.

#### *2.2.3.3 Component Installation & Deployment*

For the deployment of this component, dedicated YAML files and the kubectl[7] tool will be used.

### 2.2.4 Service Graph Editor & Analytics Editor

The RAINBOW Service Graph Editor & Analytics Editor is a component of the modelling layer, responsible to author and maintain application templates of cloud-native components, as also to save and send the instantiation of them. This can be done through an abstract way by formulating abstracted direct acyclic graphs (DAG) representations of the cloud-native applications, which are compatible with the industrial format (i.e., docker-compose, Kubernetes YAML, helm-charts).

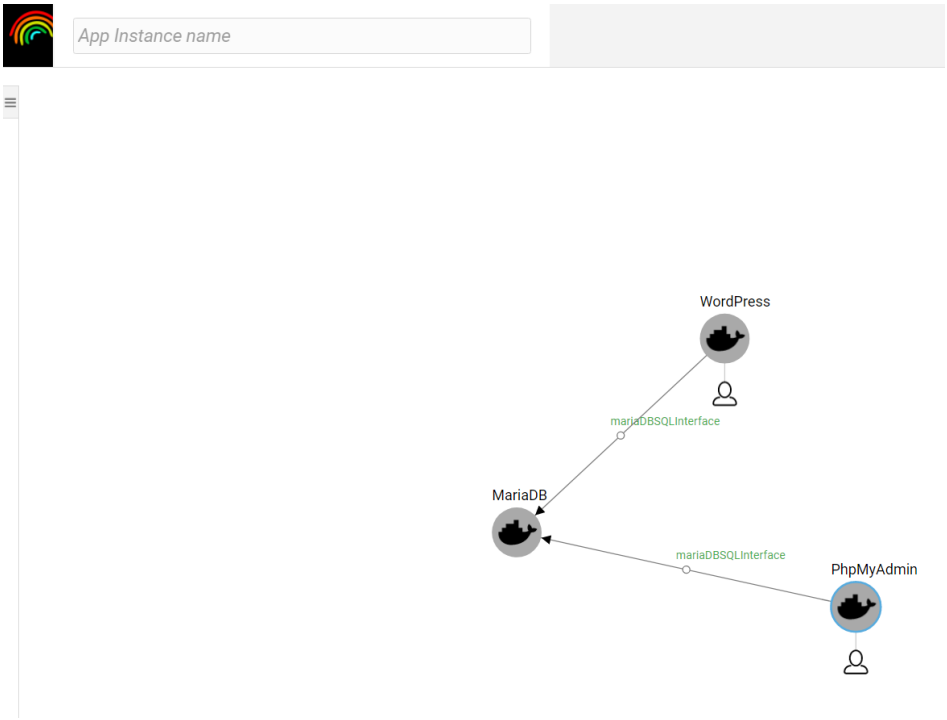


*Figure 4 RAINBOW Graph Editor*

Adding to that, the Service Graph Editor can help add deployment constraints, resource constraints, operation constraints and security constraints to the instantiation of the Service Graphs.

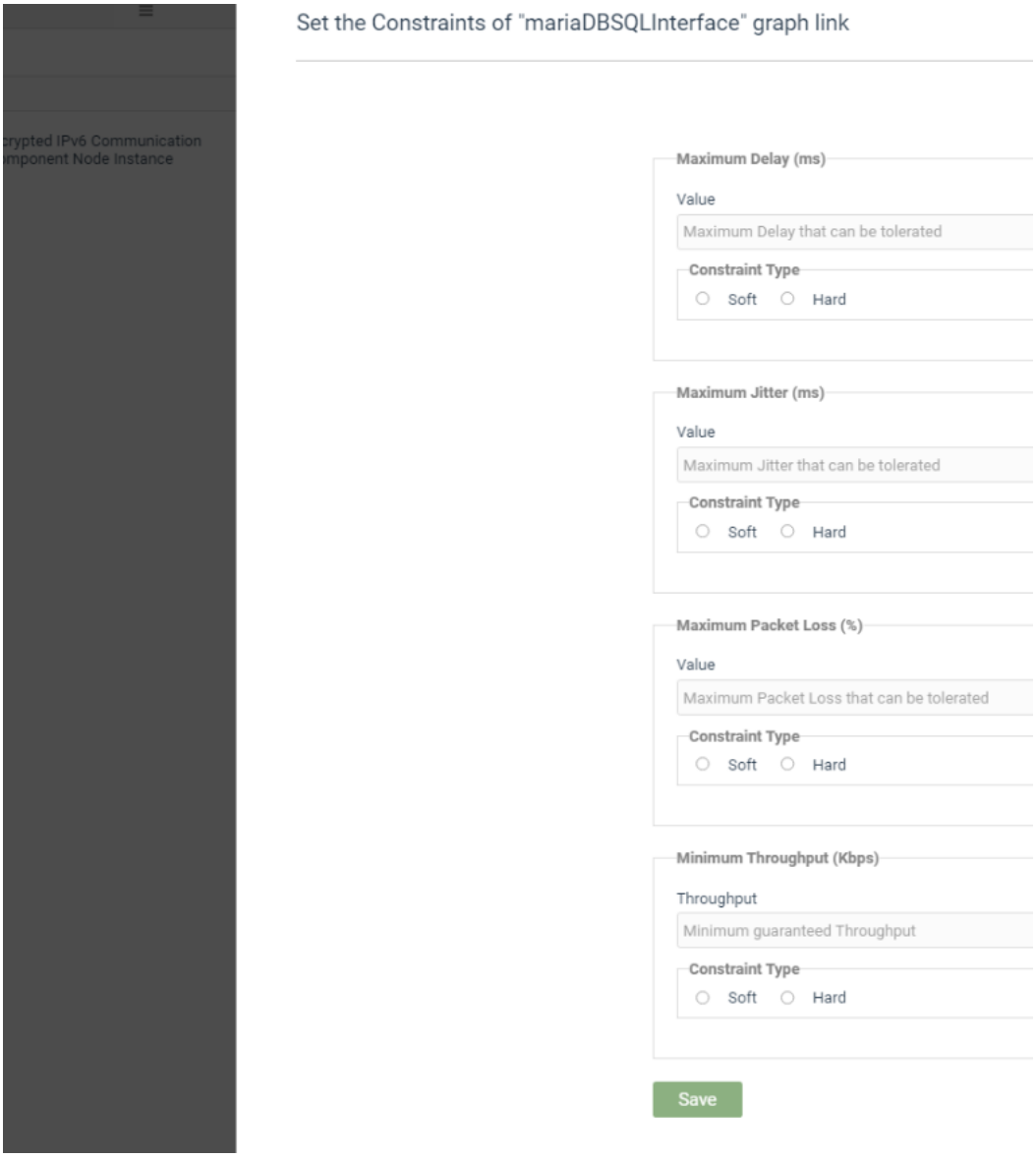


*Figure 5 Setting constraints through the RAINBOW Graph Editor*

The Service Graph Editor is part of the first RAINBOW Platform release. There are no deviations in the implementations, and everything that is implemented is fully integrated with the needed RAINBOW components. The component will be further updated with the rest of the functionalities until the second and the final release.

The analytics part of the editor will be used for the creation or edit of analytic queries and the declaration of various optimization strategies and constraints regarding query execution and data movement and is currently under implementation.

#### *2.2.4.1 Integration and Component Dependencies*

The Service Graph Editor & Analytics Editor depends on the Logically Centralized Orchestrator Backend to save the Service Graphs as also to deploy them. Also, it depends on the Analytics Service to create and send analytic queries.

#### 2.2.4.2 *Component Packaging and Distribution*

The Service Graph Editor & Analytics Editor is packaged as a Docker container along with the Policy Editor are available through the RAINBOW Artifactory that features a private Docker Image Hub.

#### 2.2.4.3 *Component Installation & Deployment*

For the installation process, a container image is provided that can be used for deployment of the component; however, a deployment YAML file that can be used with the kubectl command to deploy it at Kubernetes will also be provided in the components' repository. As far as the configuration, at the deployment YAML file, the URL of the Logically Centralized Orchestrator Backend need to be specified as an environmental variable.

The deployment must be done to a control plane/master node, which must be x86 CPU architecture.

### 2.2.5 Mesh Routing Protocol Stack

At the routing layer, the Mesh routing protocol stack provides a node with secure-layer-3 connectivity to an existing mesh topology without having to statically configure its IP address or the IP address of one of its adjacent nodes. Moreover, it automates the process of binding to a 'logically centralized' Kubernetes cluster. To avoid reinventing the wheel, RAINBOW forked an open source DHT-based routing protocol called CJDNS in order to satisfy the raised functional requirements. Hence specific extensions have been implemented to make the admission control more secure and the process of cluster-head selection completely autonomic.

#### 2.2.5.1 *Integration and Component Dependencies*

The Mesh Routing protocol stack has two major three major dependencies. These are:

- It can operate on physical devices that support 802.11s wireless link protocols which is essential for layer 2 connectivity of the mesh nodes.
- It requires the existence of the Security Enablers (see 2.2.11) since the admission control is performed upon a formal attestation process that requires access to TPM functional crypto-primitives.
- It can operate x86/ARM processors.

#### 2.2.5.2 *Component Packaging and Distribution*

The entire stack is developed using Quarkus Framework which is an Ahead-Of-Time (AOT) compilation framework for Java using also linked C++ binaries along the rest of the edge components. The entire stack is packaged as a container and hosted to the project's CI registry.

#### 2.2.5.3 *Component Installation & Deployment*

The installation of the Mesh Routing protocol stack is performed manually by the owner/administrator of the edge device.

### 2.2.6 Multi-domain sidecar proxy

The sidecar proxy is an umbrella component of the routing layer, that wraps all edge-related components that need to operate on the edge node. These include the Mesh Routing component (2.2.5), the raw device management component, the attestation handlers, the kublet management and the L7 control plane component.

#### 2.2.6.1 *Integration and Component Dependencies*

As already mentioned, this umbrella component encapsulates five internal modules as depicted on the figure below. The component acts as a single point of reference for any API call that is routed to these internal modules.

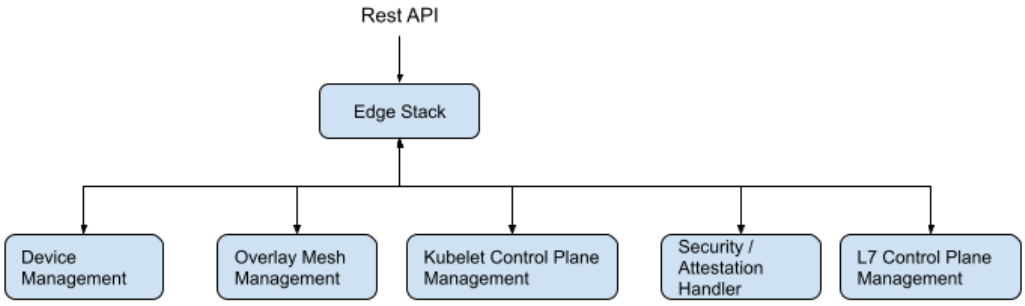


*Figure 6 Dependencies of the sidecar proxy*

#### 2.2.6.2 *Component Packaging and Distribution*

The mesh Overlay Mesh Management and the Attestation/Security Enablers are autonomously packaged in their own containers as explained on 2.2.5 and 2.2.11. The other three components are shipped as a single container since they belong to the same codebase. They are developed in Java using the Quarkus Framework. They are packaged as ARM containers and x86 containers with the project's CI registry.

#### 2.2.6.3 *Component Installation & Deployment*

The installation of the side-car proxy is performed manually by the owner/administrator of the edge device. The edge device must support Open Container Interface and must be equipped with either x86 or ARM processor.

### 2.2.7 Resource & Application-level Monitoring

RAINBOW introduces a comprehensive and extensible stack for automating the monitoring process of IoT services deployed through containerized execution environments in geo-distributed fog settings. The architecture of the RAINBOW Monitoring follows an agent-based architecture that embraces the producer-consumer

paradigm. This approach provides interoperable, scalable and real-time monitoring for extracting both infrastructure and application behaviour data from deployed IoT services. The RAINBOW Monitoring runs in a non- intrusive and transparent manner to underlying fog environments as neither the metric collection process nor the metric distribution and storage are dependent to underlying platform APIs (e.g., fog-node specific) and communication mechanisms. The following figure introduces a high-level overview of the RAINBOW Monitoring.

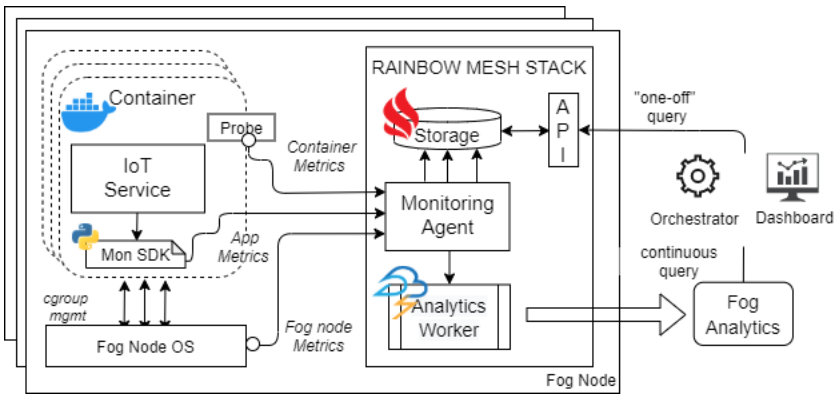


*Figure 7 Resource and Application Monitoring in RAINBOW*

On each RAINBOW-enabled fog node, a Monitoring Agent is deployed to collect system-level data regarding the resource utilization of both the fog node and the deployed containerized services through Monitoring Probes created by RAINBOW Developers. Several Monitoring Probes are already made available by RAINBOW and include a Netdata metric collector and a Docker Container Probe. In turn, users can take advantage of the RAINBOW Monitoring SDK to develop their own custom Monitoring Probes, so that app-level performance metrics are also "pushed" to the Monitoring Agent. This enables users to have one unified environment to view performance data and interact with, rather than having to deal with multiple monitoring tools. All monitoring data is exported by the Monitoring Agent to the local Storage Agent so that users can query for both real-time data and historical data persistently stored across the Storage Fabric created on top of the overlay mesh network inter-connecting the user's fog nodes.

Monitoring is part of the RAINBOW Platform first release, and there are no deviations from the DoW timeline.

#### *2.2.7.1 Integration and Component Dependencies*

Monitoring does not feature any dependencies with other RAINBOW components. On the other hand, the Storage Agent, which is in charge of exposing the Monitoring API, depends on the resource and application monitoring to provide for each fog node monitoring data.

#### *2.2.7.2 Component Packaging and Distribution*

Monitoring is packaged as a single Docker container and made available through the RAINBOW container registry.

#### 2.2.7.3 *Component Installation & Deployment*

No installation effort is required by the users, as the Monitoring is part of the mandatory services of the RAINBOW Mesh Stack and thus, upon opting to deploy an application on fog offerings the Monitoring will be automatically installed. Nonetheless, users are free (and are highly suggested!) to deviate from the default parameterization of the monitoring process to set the periodicity of metrics to be collected, enable logging and the level of reporting, and give a name and tags to the Monitoring Agent to ease readability and association when performing monitoring queries via the RAINBOW Dashboard. The configuration can be done either through the YAML configuration file of each Agent or through the RAINBOW Dashboard.

Monitoring is part of the RAINBOW Mesh Stack and thus, the Monitoring Agent is both deployed and run on the fog node offerings that are reserved by the user for his/her application deployment.

### 2.2.8 Policy Editor

The RAINBOW Policy Editor is part of the Modelling Layer and used by the Service Graph provider to apply instructions/guidelines regarding how the overall application should behave prior to deployment and during runtime. These instructions are addressed as Policies and based on their properties affect either the initial deployment of the Service Graph or the overall runtime behavior. Depending on the state of the Service Graph that affect, the policies can be addressed as Design-Time policies or Runtime policies accordingly.

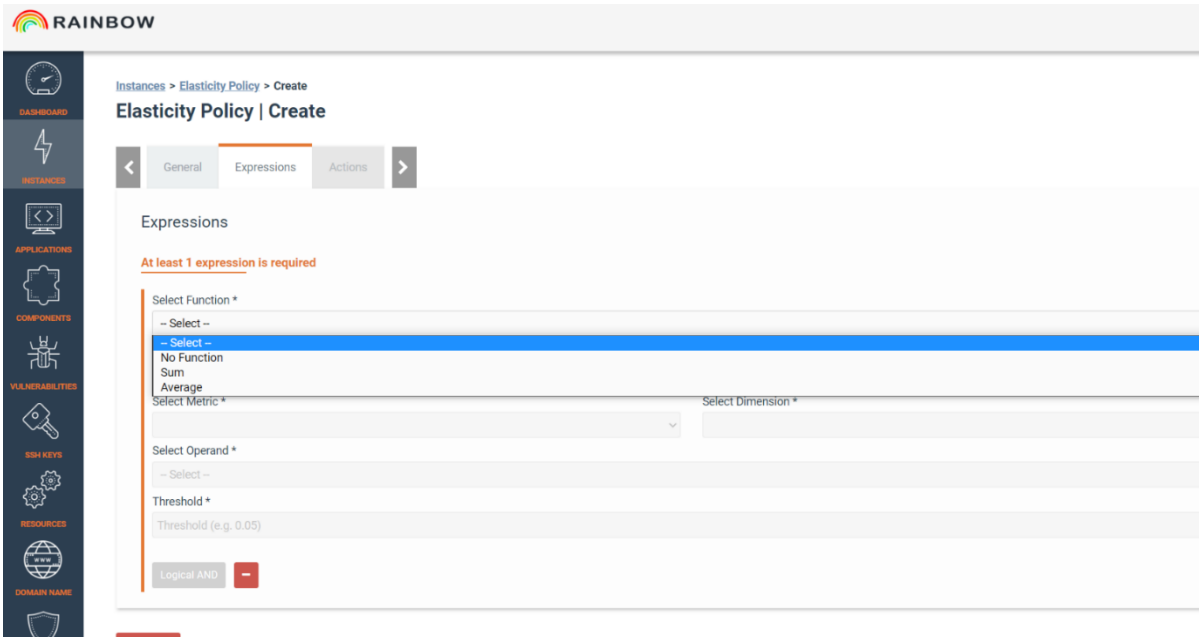


*Figure 8 Policy Editor*

The Policy Editor is part of the first RAINBOW Platform release, there are no deviations in the implementation, and no deviations are foreseen. As for now, the Policy Editor UI is

implemented for the most part, with some features remaining to be added in the second release. As far as the integration is concerned, some indicative integrations have been already done, but further will be needed for the rest of the features that will be implemented in the second release.

#### *2.2.8.1 Integration and Component Dependencies*

This component depends on the Data Storage and Sharing component to fetch the deployment's exposed metrics and the Logically Centralized Orchestrator Backend to fetch the deployments and save the policies, which are then sent to the appropriate RAINBOW component.

#### *2.2.8.2 Component Packaging and Distribution*

The Policy Editor is packaged as a Docker container together with the Service Graph Editor & Analytics Editor and made available through the RAINBOW Artifactory that features a private Docker Image Hub.

#### *2.2.8.3 Component Installation & Deployment*

For the installation process, a container image will be provided, as also a deployment YAML file that can be used with the kubectl command to deploy it at Kubernetes. As far as the configuration, at the deployment YAML file, the URL of the Logically Centralized Orchestrator Backend need to be specified as an environmental variable.
The deployment must be done to a control plane/master node, which must be of x86 CPU architecture.

### 2.2.9 Data Storage and Sharing

The Data Storage and Sharing component is a component of the Data Management & Analytics layer, implemented in Java on top of the Apache Ignite main-memory database. The data ingestion and extraction services are implemented behind a REST API and are available for usage to write and read data from the database's caches. The functionalities that are provided along with the ingestion/extraction services are:

1. Tuning the persistence of data. Either only in-memory caches or a combination of in-memory and persistent caches can be used.
2. Tuning the eviction rate of persistent data. If persistence is used, the eviction rate is used to delete data that were created before the specified period.
3. Tuning the optional user-application cache. The RAINBOW users can use this option to store in-memory key-value pairs through the ingestion service.

The component will be further upgraded with more filters and functions on the extraction service. Along with the API upgrades, data placement and movement algorithms will be implemented for the second and final release of the project. These algorithms are intended to make decisions based on node information about the data placement and if needed, move data from a fog node to another to decongest unstable nodes. The Data Storage and Sharing component has been integrated and tested with RAINBOW's Analytics Service and Resource & Application-level Monitoring component.

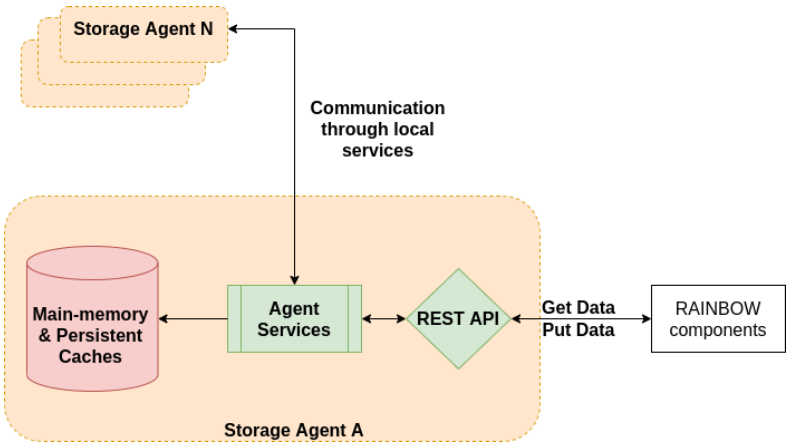


*Figure 9 Overview of Data Storage and Sharing in RAINBOW*

#### 2.2.9.1 *Integration and Component Dependencies*

The component is used by the Resource & Application-level Monitoring component to store the monitored metrics for both the latest and the historical values. The stored data are further used by the Analytics Service and the Orchestrator to answer queries and take decisions on the service placement.

#### 2.2.9.2 *Component Packaging and Distribution*

The component is implemented in Java using the Maven management tool. Using the "mvn clean compile test assembly: single" command, the code is compiled, tested against the implemented unit tests, and packaged in a single jar file that contains all the necessary dependencies, e.g., Ignite. Furthermore, a Docker image is created with the resulting package that initiates an Ignite instance whenever a container is started.

#### 2.2.9.3 *Component Installation & Deployment*

The component instances can be configured to change the status of the optional functionalities, e.g., user-application cache, and change the type of Ignite instance. Each instance can either be a Server, where data are stored, or a Client for mass data extraction.

The component is deployed using the Docker image in every fog node of the RAINBOW's environment. Starting with a single Server instance that initializes the Ignite cluster, the rest of the Server instances can enter the cluster by being deployed to every fog node available that runs RAINBOW's Side-Car proxy. Finally, one or more Client instances can be deployed on the cloud nodes for usage.

### 2.2.10 Analytics Service

The RAINBOW Analytics Service is a component of the Data Management & Analytics layer, responsible for the RAINBOW ecosystem's needs for data processing so that real-time analytic insights can be extracted from the vast amounts of monitoring data collected from both the underlying fog resources and performance indicators from deployed IoT applications. To this end, the service provides a completely distributed solution with the data processing performed -in place- right where the data is generated

so that analytic insights are extracted with low-latency and with the collected data never leaving the overlay mesh network interconnecting the collaborating fog nodes. The Distributed Data Processing service builds upon Apache Storm with our aim being to not implement yet another distributed data processing engine but rather to design novel scheduling algorithms that are decoupled from the underlying engine and acknowledge the unique settings found in the majority of geo-distributed environments that IoT applications are deployed in.
Three are the key internal components of the RAINBOW Analytics Service. The first is the *Analytics Enabler*, which is the Orchestration Service that accepts requests from the orchestrator and manages the distributed processing environment across the fog realm. The second component is *StreamSight* [13], which is the framework accepting user-designed analytic queries in a high-level and declarative format. StreamSight is responsible for translating these high-level queries into continuous queries that will be deployed for execution across the Analytics Workers reserved on the fog offerings. The third component, which are many instances of the same component, are the *Analytics Workers* which execute the analytic tasks of the submitted jobs and these workers are deployed on the fog nodes reserved by the user for his/her application.

The Analytics Service is part of the RAINBOW Platform first release, and there are no deviations from the DoW timeline.

#### *2.2.10.1 Integration and Component Dependencies*

The Analytics Workers upon instantiation expect to find two key RAINBOW services. First, the Analytics Enabler, which is the service managing the Workers; and second, the RAINBOW Storage Fabric, so that the monitoring data that analytics will be derived, can be extracted.

#### *2.2.10.2 Component Packaging and Distribution*

All components of the Analytics Service are packaged as Docker containers. In turn, the Analytics Enabler and StreamSight are made available through the RAINBOW Artifactory that features a private Docker Image Hub. On the other hand, the Analytics Worker is the vanilla Apache Storm Supervisor, and the Docker Image for this container is made available through the public Docker Hub.

#### *2.2.10.3 Component Installation & Deployment*

No installation effort is required by the users. Specifically, both the Analytics Enabler and StreamSight are part of the RAINBOW Orchestration Services and thus, will be automatically installed upon the deployment of RAINBOW on the application's Cluster Head. Nonetheless, users are free to configure the analytics scheduling process by opting declare, upon analytics job submission, what optimization method should be used during runtime (e.g., optimize job for latency, performance, data quality, etc). In turn, no installation effort is required for the deployment of the Analytics Workers are they are part of the RAINBOW Mesh Stack. However, as analytics are an optional feature made available to users, the user will have to select during the configuration of the Mesh Stack that an Analytics Worker should be installed on each fog node.

The Analytics Workers are part of the RAINBOW Mesh Stack and thus, are deployed on the fog node offerings that are reserved by the user for his/her application deployment. On the other hand, the Analytics Enabler, in charge of managing the analytics process and handling requests by the Orchestrator, is part of the RAINBOW Orchestration layer and "sits" on the Cluster head node where the Orchestrator is located as well. In turn, StreamSight, the framework accepting user-designed analytic queries in a high-level and declarative format, is also located on the cluster node as well.

### 2.2.11 Security Enablers

The RAINBOW secure enrolment service, comprised of the Zero-Touch Configuration Integrity Verification (S-ZTP CIV) and Remote Attestation Variants (including the Enhanced Direct Anonymous Attestation (DAA) to be integrated with the RAINBOW CJDNS networking mechanism), focuses on the provision of operational assurance and secure device on-boarding prior to the enrolment of a fog/edge node to the overall network. This, in turn, enables the creation of trust-aware service graph chains. As described in D2.2 [14] and D2.3 [15], the architecture followed by the RAINBOW attestation agents follows a decentralized architecture in which the RAINBOW orchestrator can act as the *verifier* for attesting the secure state of a device requesting access to the overall system. The architecture is depicted in the diagram below: The service comprises two interdependent components: the *prover* component, which is a Docker container instance running on fog/edge nodes which exposes a set of functionalities through a REST API that enable it to be securely enrolled, and the RAINBOW *Orchestrator* component (acting as the *verifier*), which is a centralized Docker container instance exposing a REST API that, upon enrolment requests from the Dashboard, executes the secure enrolment of a specified fog/edge node (*prover*) based on the secure Attestation Key (AK) establishment, certifiable measurements (update), and Oblivious Remote Attestation (ORA). Both components are implemented in C++ and utilize IBM's TSS and software TPM as the underlying trusted component.

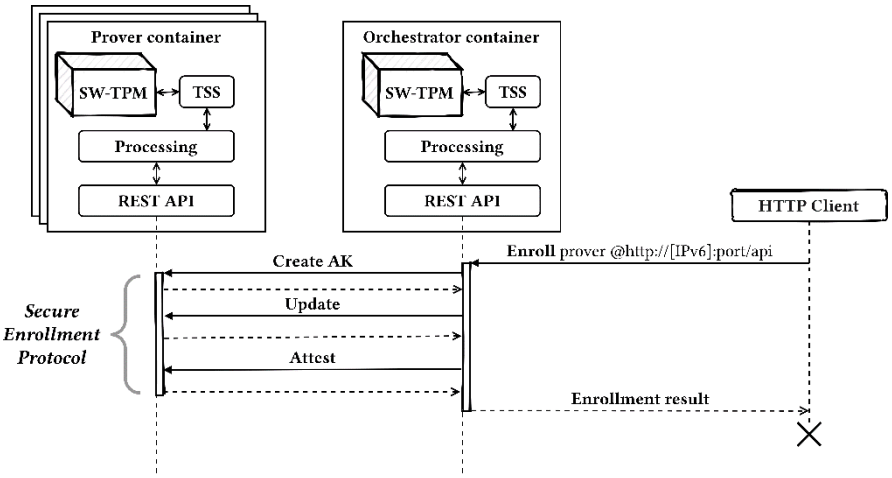


*Figure 10 Secure Enrolment of devices*

The current implementation provides the functionality of *Attestation by Quote* [] for enabling the secure establishment of trust between deployed fog/edge nodes wanting to

access the network. The attached SW-based TPM authenticates the evidence of the integrity of the state of the service binary images running inside the devices. Key features provided include the: (i) possibility to distinguish which device is compromised so that not trust relationship can be stablished with the other nodes in the network, (ii) the possibility for low-level fine-grained tracing capability, and (iii) S-ZTP capability for privacy-preserving attestation. The former is a significant feature because, once a device is compromised, it can be immediately retracted and replaced by the Orchestrator without affecting the entire Service-Graph Chain (SGC), thus, catering to efficient SGC management and flexible slicing. The latter enables the integrity verification of a designated device without conveying any information on its configuration to other nodes/entities that may be acting as the *verifier*. This is of paramount importance in fog-based environments, such as the one envisioned in the smart manufacturing use case, where the role of the verifier can be enacted by the cluster head of each manufacturing floor; in this case, *each cluster head should be able to have verifiable evidence on the correctness of a device without the need to know its exact configuration*.

In a nutshell, once a attestation request is received (either by the administrator through the Dashboard for attesting a specific device or set of devices or by a new device wanted to be enrolled in the overall network), the Orchestrator initiates the attestation process. In the first step, the Orchestrator proactively determines the device's expected configuration state by accumulating the artificial vPCR construct of the corresponding state that a device needs to be prior to be enrolled in the network (essentially, a whietelist of service binaries that need to have been loaded in the device). The Orchestrator then requests the device to similarly accumulate its PCRs to reflect potential changes. This update request contains only the PCR index i that must be updated and a configuration file identifier, ID, to measure. Upon receiving such update requests, the device then invokes the RAINBOW Monitoring Agent (Section 3.2.1.2.5) to measure the requested file(s) and subsequently invokes the SW-based TPM to extend PCR i with the new measurement. The extracted PCR Quote is then sent back to the orchestrator for verification.

#### *2.2.11.1 Integration and Component Dependencies*

The components (available in the RAINBOW attestation GitLab repository) are self-contained and include Dockerfiles which ensure that the necessary dependencies are installed in the Docker images to ensure correct execution. The RAINBOW Attestation service relies on the correct tracing and extraction of the configuration properties to be attested (e.g., list of loaded service binaries). This output is provided by the RAINBOW Monitoring Agent. Furthermore, as aforementioned, the RAINBOW Orchestrator is also a core part of the entire attestation process since this entity is acting as the *verifier*.

#### *2.2.11.2 Component Packaging and Distribution*

The components are packaged as separate Docker containers and include Dockerfiles and docker-compose files for installation.

#### *2.2.11.3 Component Installation & Deployment*

Both components (the *prover* and *Orchestrator*) are built by executing either docker build or docker-compose build. The components are started by executing either docker run or docker-compose up.
The *prover* is deployed on fog/edge nodes, whereas the centralized *Orchestrator* component is deployed as a centralized instance at the Orchestrator-side.

## 2.3 Early Release Status

For this first release of RAINBOW, a prototype of the platform has been deployed, composed of the components developed so far and with partial integration of them to achieve basic functionalities of the platform.

The focus of the early release was to support functionalities as the proper definition of application graphs and their deployment over cloud and edge resources, thus allowing the planning and the execution of the first prototypes of the RAINBOW demonstrators.

Also, besides the basic functionality of deployment, this early release supports scaling of the service graphs based on SLOs that consider different metrics, such as CPU utilization, RAM, etc. Furthermore, the attestation security enabler is supported, through which we can attest either the devices that want to join the cluster or the pre-existing ones. This will aid us to accomplish the Secure Enrolment of the new devices in the second release. Furthermore, in the Early Release, we provide the Analytics Service, which helps us process data to extract real-time analytic insights from all the huge amount of monitoring data collected from the resources and the deployed applications. This is achieved by utilizing the Analytic workers deployed in each node as also the Data Storage and Sharing service that has also been provided in the early release.

Finally, the deletion of the deployed service graphs has already been supported in order to allow the clean-up of the Kubernetes cluster, thus avoiding any unneeded costs.

### 2.3.1 Interface's implementation Status

The following table provides a recap of the integration status among the interfaces provided by the platform components.

*Table 1 Interfaces Status*

| Reference Code | Name | Responsibilities | Status |
|---|---|---|---|
| DSS_01 | Data Ingestion | AUTH | Interface is available for usage |
| DSS_02 | Data Extraction | AUTH | Interface is available for usage |
| CS_01 | *Pre-deployment Constraint Solver* | TUW | Planned for the second and final releases |

| Reference Code | Name | Responsibilities | Status |
|---|---|---|---|
| *DM_01* | *Service Graph Deployment Template Interface* | TUW | Integrated |
| *DM_02* | *Policy Enforcement Interface* | TUW | Planned for the second and final releases |
| *OLM_01* | *Scheduler* | TUW | Integrated |
| *OLM_02* | SLO Manager | TUW | Integrated |
| *OLM_03* | SLO Controller | TUW | Integrated |
| *OLM_04* | Elasticity strategy Controller | TUW | Integrated |
| *RM_01* | *Resources Registry* | TUW | Planned for the second and final releases |
| *MON_01* | *Monitoring Interface* | UCY/AUTH | Completed. Monitoring API for extracting real-time and historical monitoring data from a fog node is available through a REST API exposed by the Storage Agent of each fog node. API described in detail in D4.1 |
| *FSA_01* | *Fog Service Analytics Interface* | UCY | Completed. Despite the fact that users can submit analytic queries and entire jobs for execution through the RAINBOW Dashboard, a REST API is also provided for Service Developers. |
| *MRP_01* | *Mesh Routing Interface* | UBI | Integrated |
| *SP_01* | *Sidecar Proxy Interface* | UBI | Integrated |
| *SE_01* | *Secure Enrolment Agent* | DTU | Completed. A centralized REST API is made available which executes the secure enrolment of fog/edge nodes as described in D2.3a. |
| *SE_02* | *Control-Flow Attestation Agent* | *UBI* | Planned for the second and final releases |
| *SE_03* | *Multi-level Detailed Tracing Agent* | *UBI/DTU* | Tracing for the monitoring of device configuration |

| Reference Code | Name | Responsibilities | Status |
|---|---|---|---|
| | | | properties is implemented in the first release. Tracing capabilities focusing on the introspection of the execution of a device (to be used for the Control-flow Attestation), is planned for the final relase |
| *SE_04* | *Direct Anonymous Attestation Agent* | *DTU* | Code is described in D2.3a, but integration with RAINBOW is planned for the second release |
| *SE_05* | *Key Management Interface* | *IFAT* | Code is described in D2.3a. Further, DAA uses key management functionality. For CJDNS we have to do some additional research, e.g. if the keys have to be stored within the TPM. Integration with the Components is planned for the second release. |

### 2.3.2 Integrated Orchestration Flow

With this section, we want to assist the reader of this document and the users of RAINBOW in understanding how the integrated platform works.

At this early release, the Service Graph Editor part of the Dashboard has been created. This is fully integrated with the rest of the platform and supports most of the current features of the other components that are needed. The deployment lifecycle is implemented, and all the needed components for the first release are integrated from the Dashboard to the actual orchestrator. To complete a deployment lifecycle, the User creates the Service Graph through the Dashboard; once the deployment button is pressed, the service graph is sent to the backend, which in turn sends it to the Deployment manager for now. From then on, the Orchestration Lifecycle Manager start the spawning of the Service Graph nodes using the Kubernetes (Resource Manager).

The abovementioned interaction is also presented in the sequence diagram depicted in Figure 11.

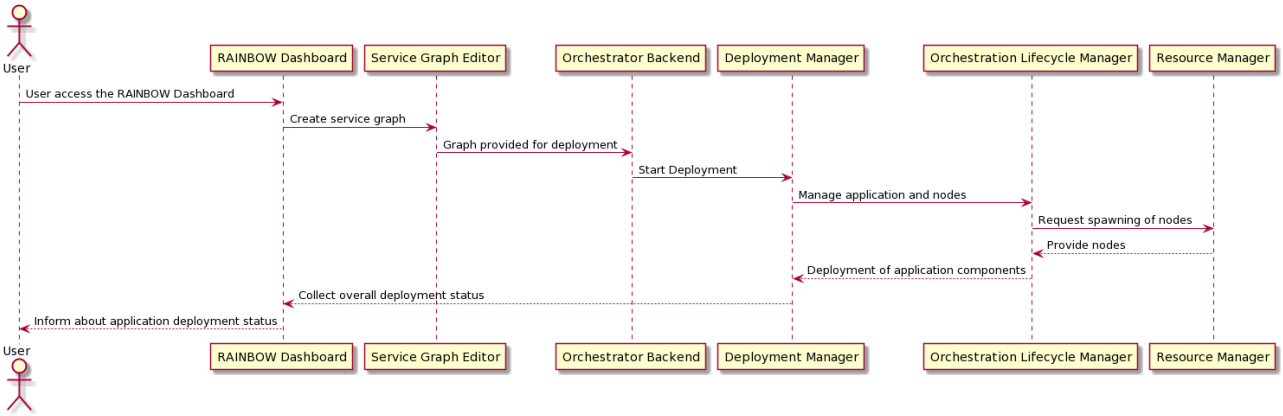


*Figure 11 Sequence diagram for RAINBOW Deployment*

After the deployment succeeds, the monitoring mechanisms collect any potential application-level metrics as also continues to collect the system-level metrics and stores them in the Data Storage and Sharing component. Those metrics are later used from other components for decision making. SLOs are supported, which helps to scale the Service Graph. Currently, for the SLOs we support the scalability of the application based on some metrics that the Data Storage and Sharing component are fetching, but although the Policy Editor is partly implemented, it is not integrated with the rest of the platform. So, currently, the SLO works by manually applying the SLO descriptors through YAML files.

The abovementioned interaction is also presented in the sequence diagram of Figure 12.

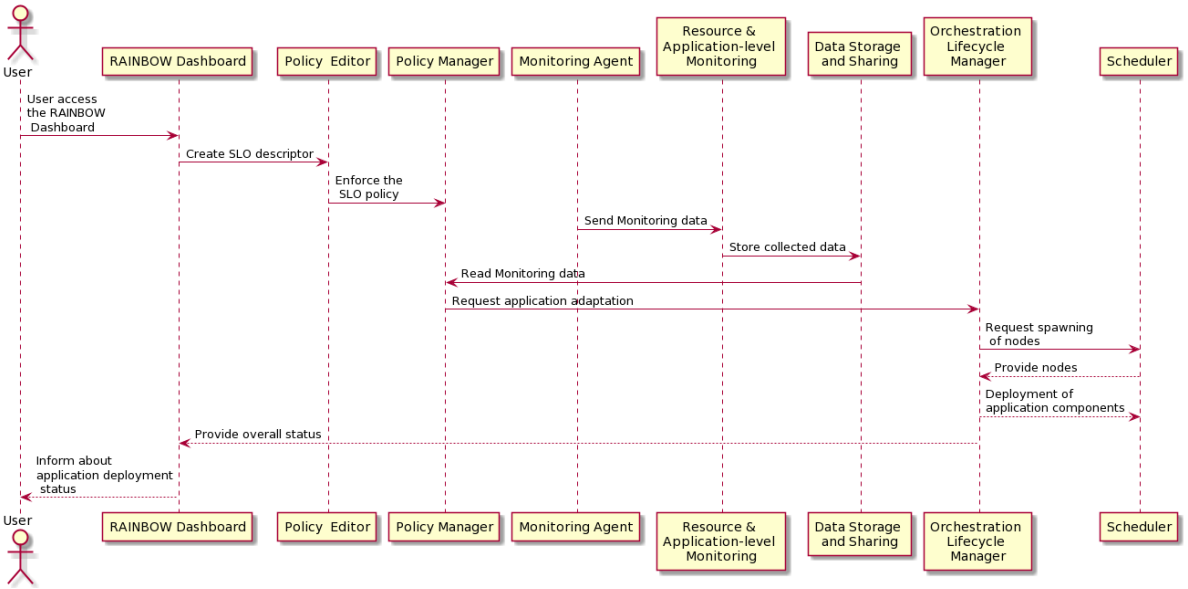


*Figure 12 Sequence diagram for RAINBOW scaling*

Undeployment function also supported. More specifically, through the RAINBOW Dashboard the user can select which Service Graph to undeploy. The Dashboard forwards the undeployment command to the backend, which then interacts with Kubernetes (Resource Manager) to undeploy the specified Service graph.

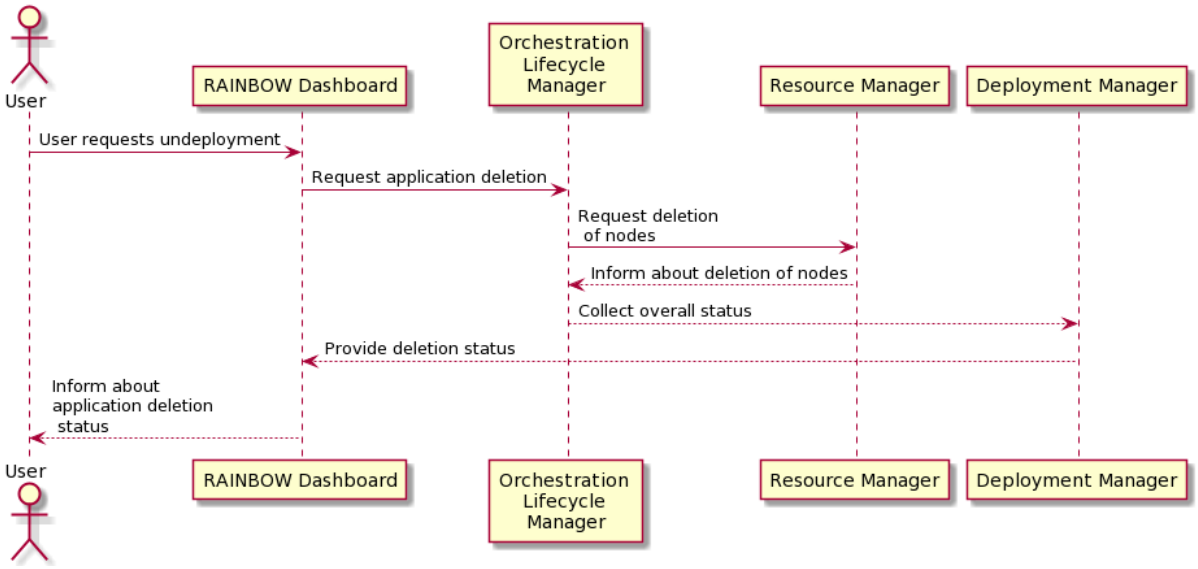


*Figure 13 Sequence diagram for undeployment of an application deployed with RAINBOW*

### 2.3.3 Rainbow Unified Dashboard

RAINBOW Platform, including the Unified Dashboard, have been integrated and deployed as a confidential prototype. Still, for the sake of clarity of the reader, we present some basic parts of the Dashboard implemented for this first release, with a flow presenting how an application is defined as a graph, configured, deployed and how it scales.

At first, a user that visits the RAINBOW Dashboards views an overview of the available resources and monitoring information from deployed application, as depicted in Figure 14.

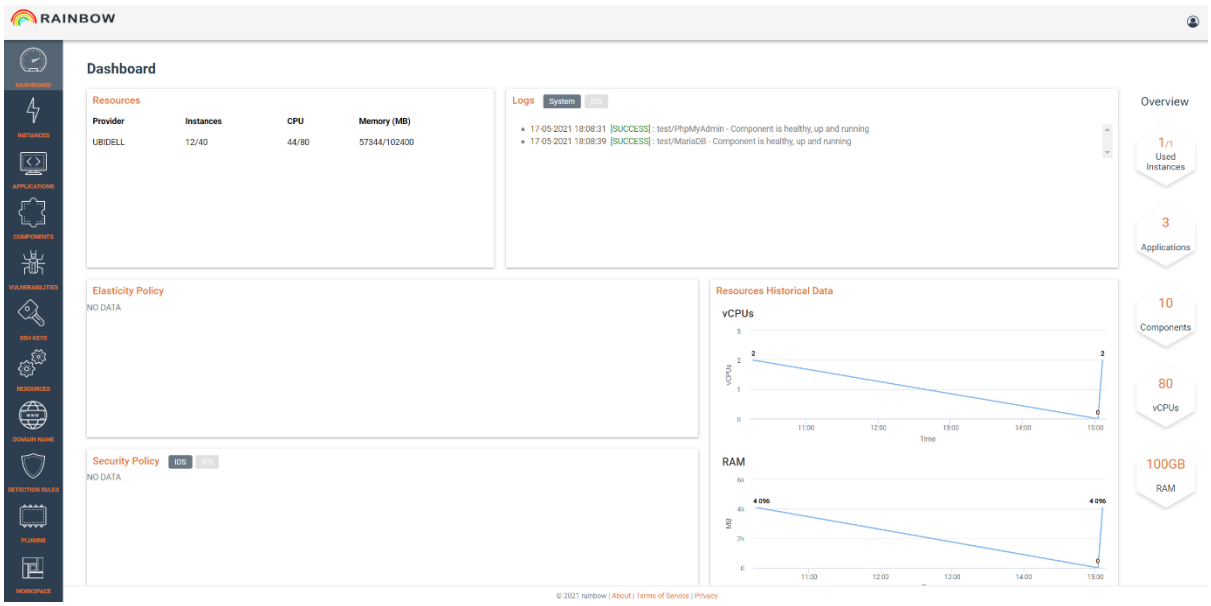


*Figure 14 Main page of the RAINBOW Dashboard*

The user is provided with the available components as depicted in the figure below.

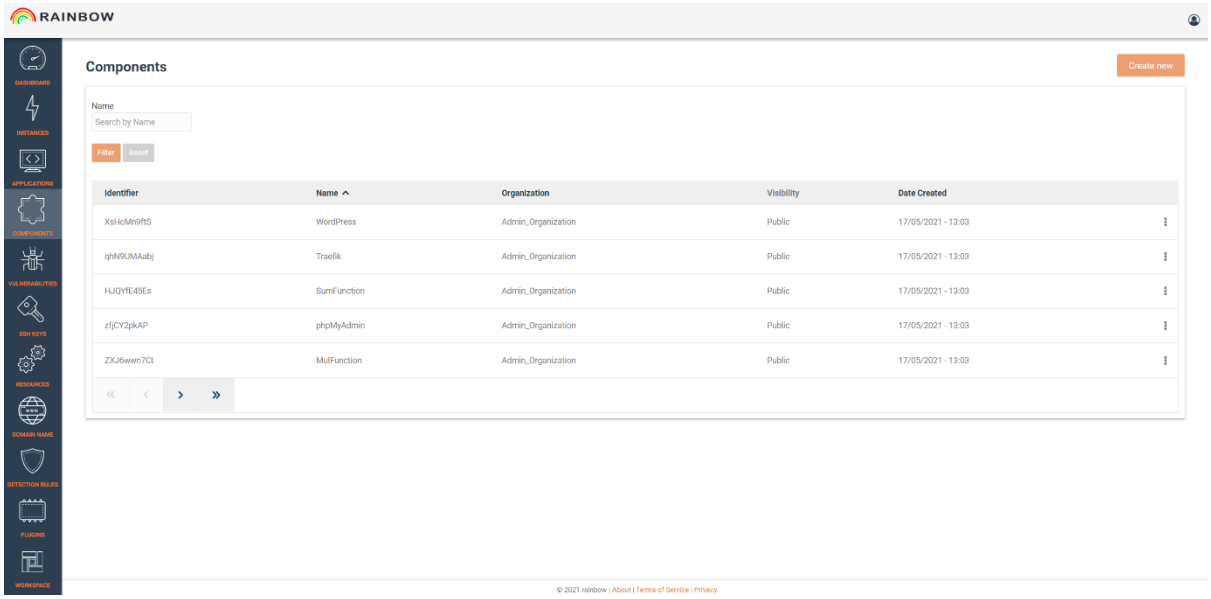


*Figure 15 Components List in the RAINBOW Dashboard*

A user can add or edit a component, as depicted in Figure 16 and Figure 17 below.

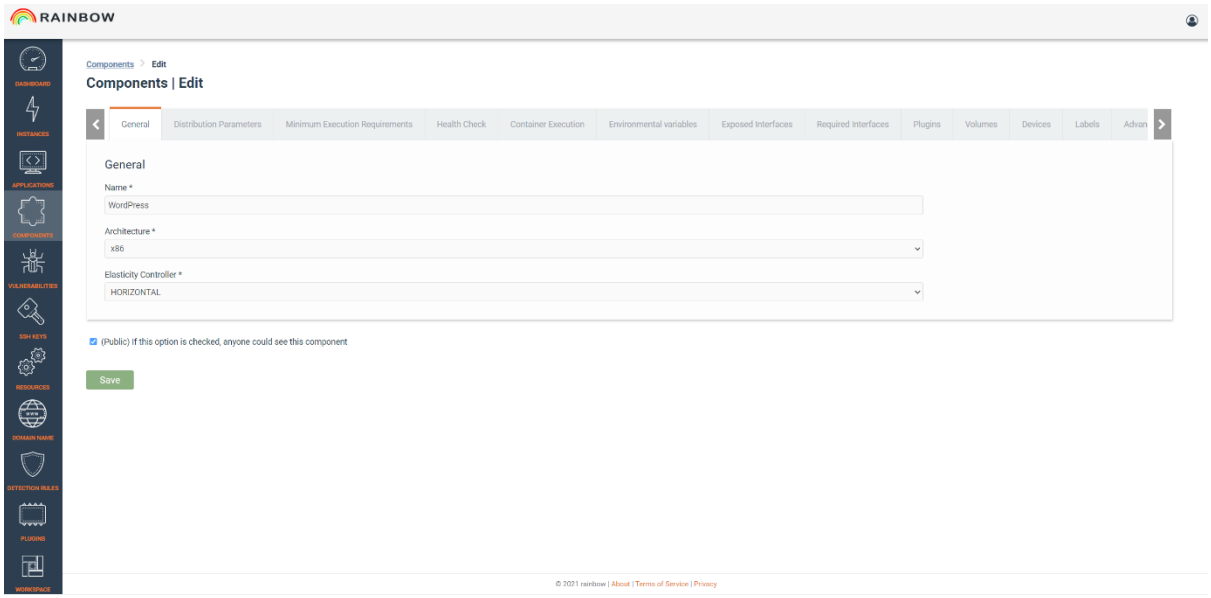


*Figure 16 Create or edit components (defining architecture)*

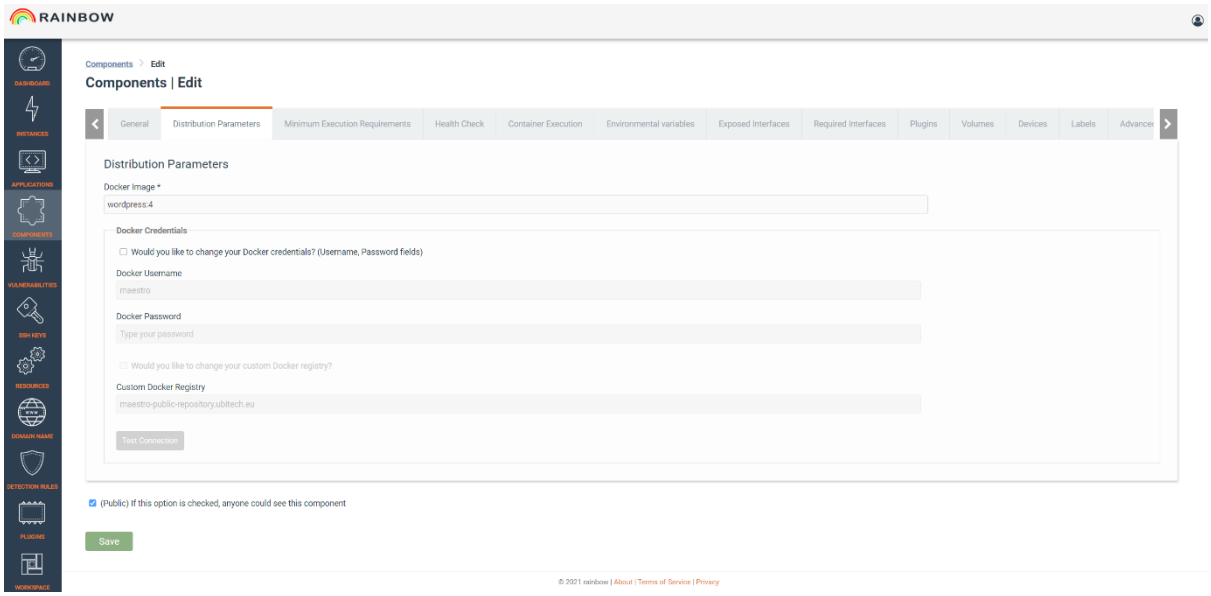


*Figure 17 Create or edit components (providing distribution parameters)*

With components defined, the user can create the application graphs (see figure below) that are then stored as templates.

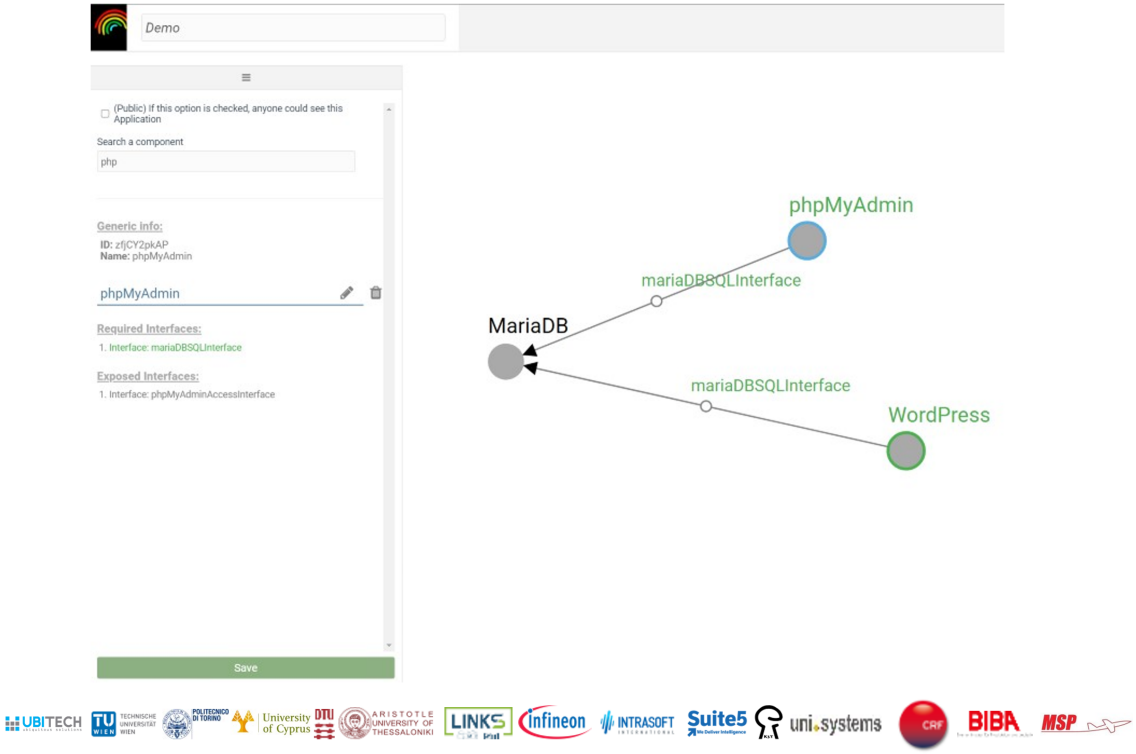


*Figure 18 The graph editor of RAINBOW Dashboard*

The application template can then be used for the creation of a new deployment, as depicted in the figure below.

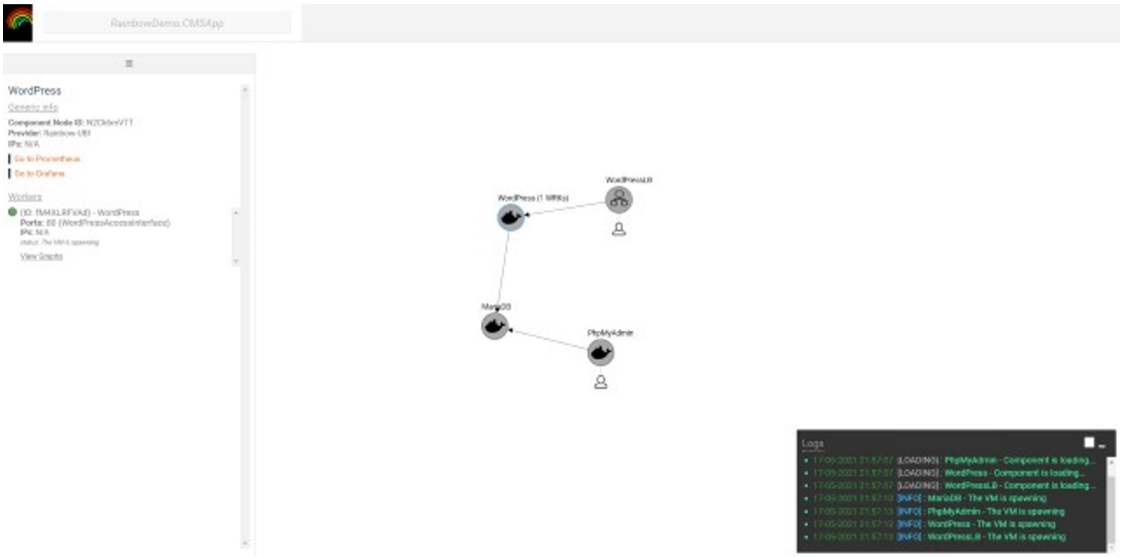


*Figure 19 A deployment in process in the RAINBOW Dashboard*

Once the application is deployed, it is also shown in the list of application instances, as depicted in the figure below.

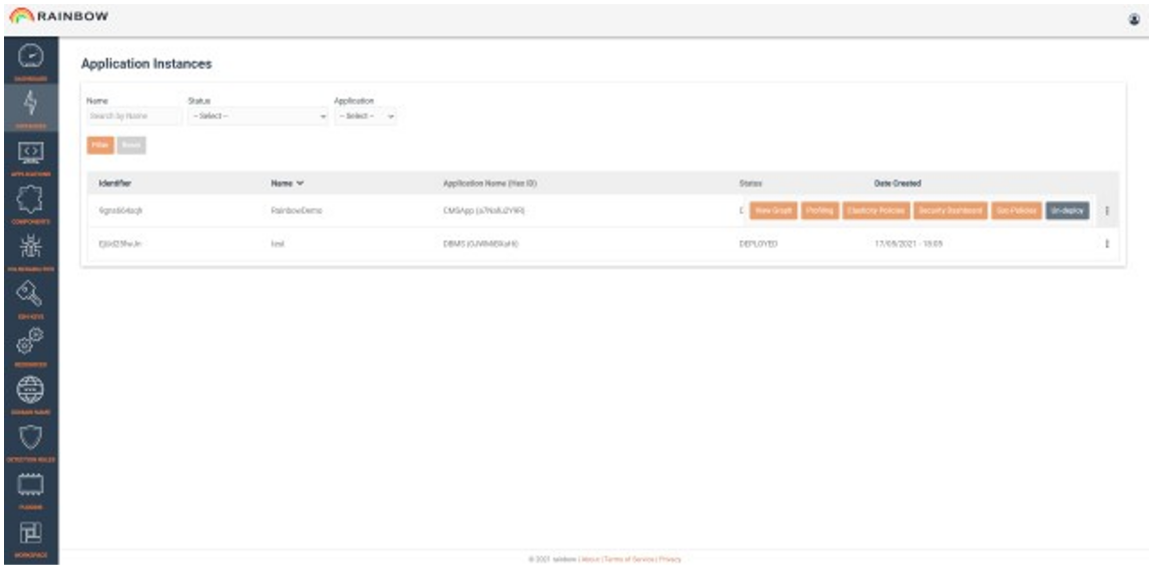


*Figure 20 Completed deployments list in RAINBOW Dashboard*

User can define the elasticity policies that desire once the application is deployed, using the dedicated editor, as depicted in the figure below.

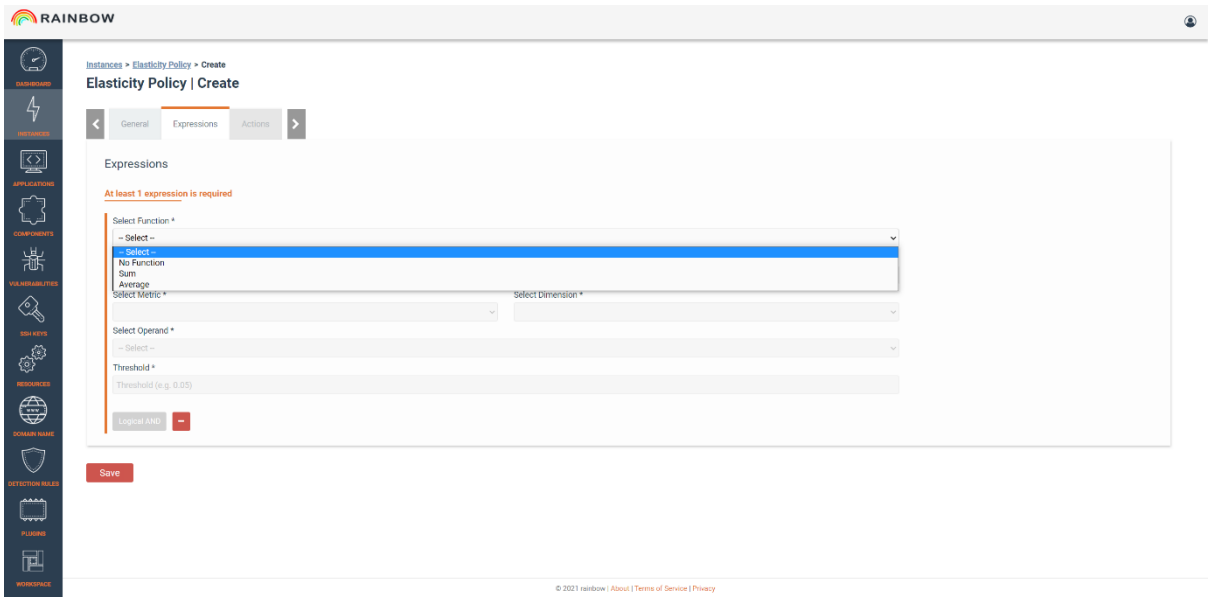


*Figure 21 Defining policies in the RAINBOW Dashboard*

Finally, if a policy is in place, the scaling can happen once the SLO is violated. The scaling results are also presented in the Dashboard, as depicted in the figure below.

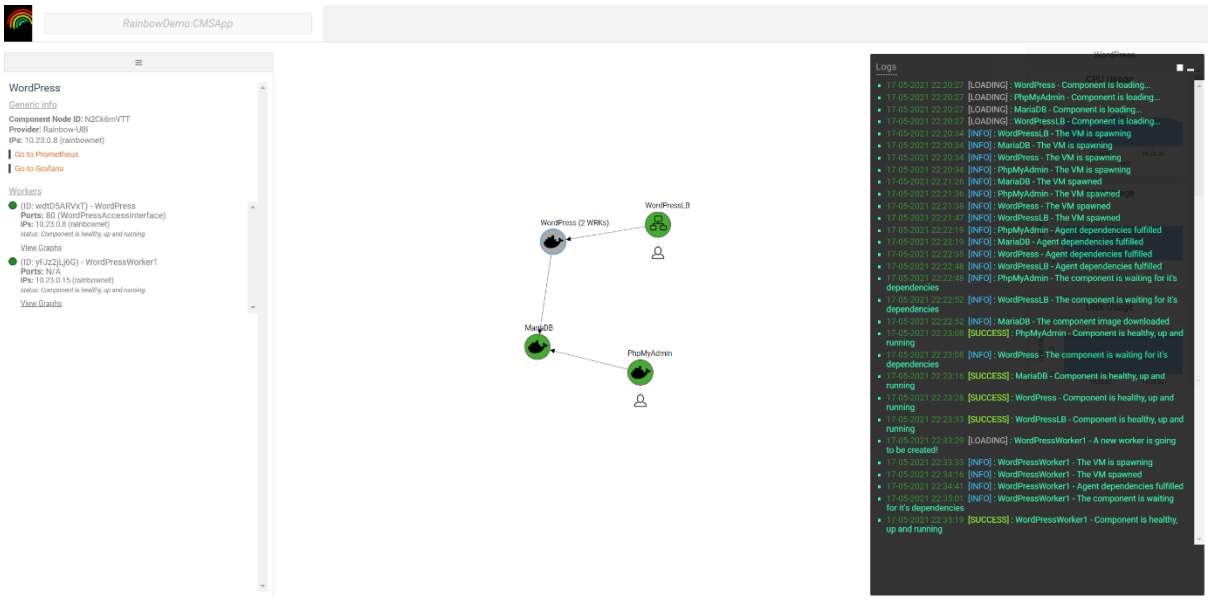


*Figure 22 A scaling performed and shown in the RAINBOW Dashboard*

# 3 Technical Evaluation and Quality Assurance

In this section, we provide an overview of the work performed so far for the achievement of the first platform release, in terms of the development and integration process followed, the software quality assessment process, and the testing procedures.

## 3.1 Continuous Integration and Quality Assurance

The flow the consortium will follow for the Continuous Integration and Quality Assurance of the developed platform has been presented in D5.1. This section provides a snapshot of the process, focusing on the tools used for enabling the CI part of RAINBOW.

### 3.1.1 Version Control System – Gitlab

As already presented above, the consortium has selected Gitlab as the primary VCS system. The Gitlab group that has been created and hosts all components' repositories is depicted in Figure 23.

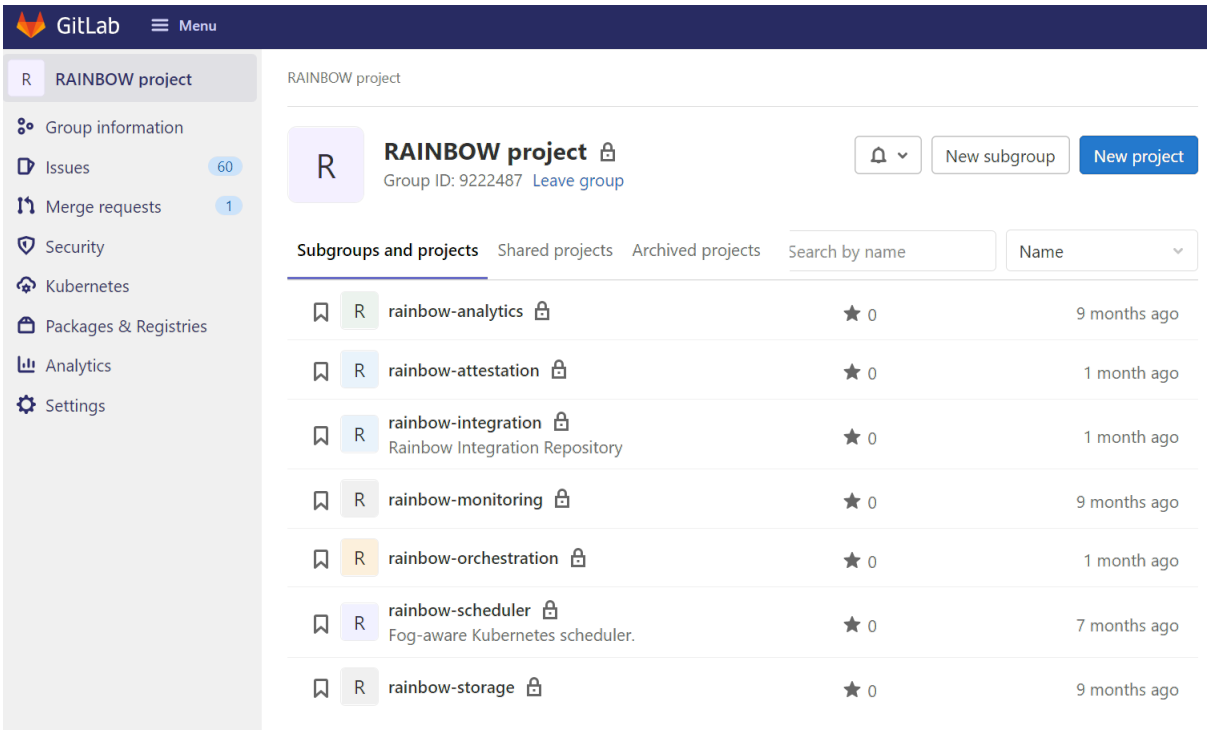


*Figure 23 Gitlab group and repositories for RAINBOW project*

The RAINBOW project source code is currently organised in the following repositories:

- **rainbow-scheduler** contains all source code of the RAINBOW Scheduler,
- **rainbow-orchestration** contains all source code of the RAINBOW Orchestrator,
- **rainbow-analytics** contains all source code of the RAINBOW analytics service,
- **rainbow-monitoring** contains the source code of the RAINBOW monitoring applications,

- **rainbow-storage** contains all the source code and configurations of the RAINBOW storage mechanisms,
- and **rainbow-attestation,** which contains all the source code of the RAINBOW attestation mechanisms.

Currently, the Gitlab group and all repositories are private with access only to consortium members. After the finalization of IPR related agreements within the consortium, the group and some of the repositories (that will offer components with open-source licence) will be made public.

### 3.1.2 Container Registry

For the distribution of components, as explained in D5.1, we use docker registry. This registry is hosted in the project's GitLab group, as depicted in the figure below.

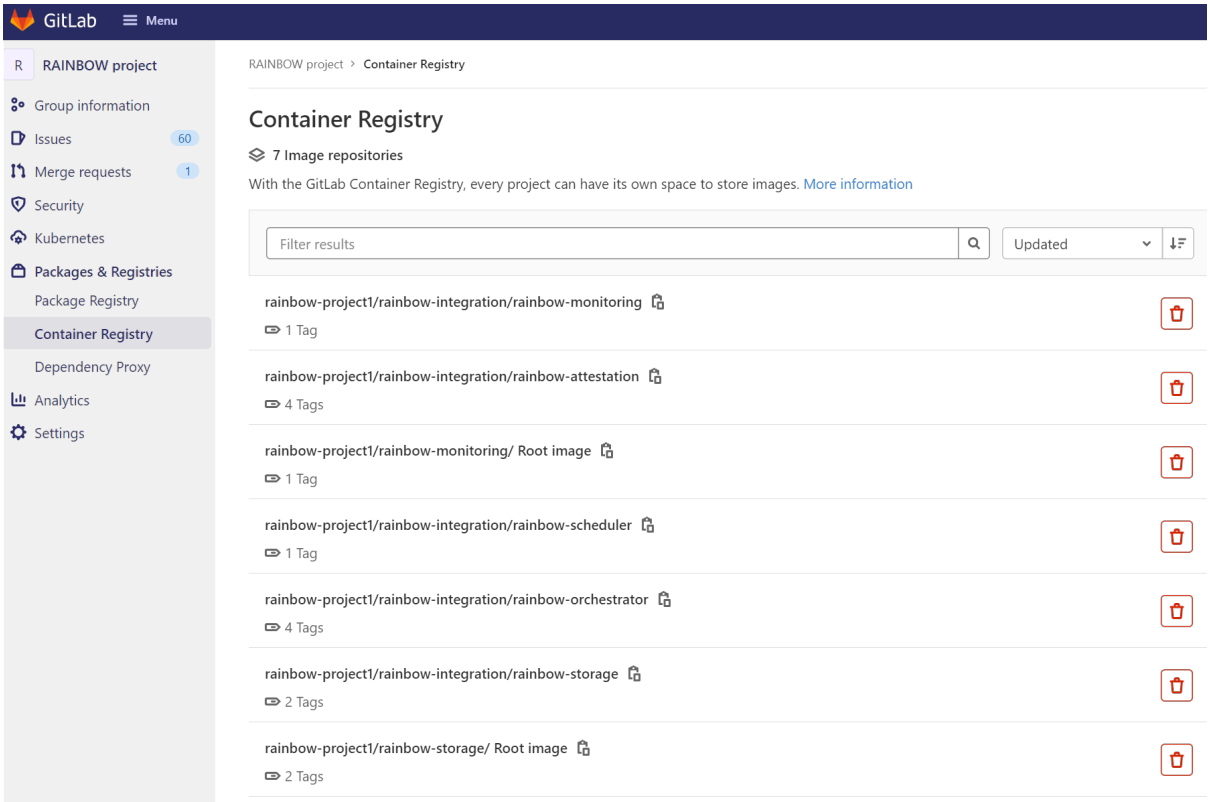


*Figure 24 Container Images available at the project's Container Registry*

### 3.1.3 Issue Tracking – Gitlab

GitLab Issues is the issue/bug tracking toolset that RAINBOW project uses. The GitLab issues of the RAINBOW Project are located at https://gitlab.com/groups/rainbow-project1/-/issues (see Figure 25), whose access is limited to the consortium developers for the time being.

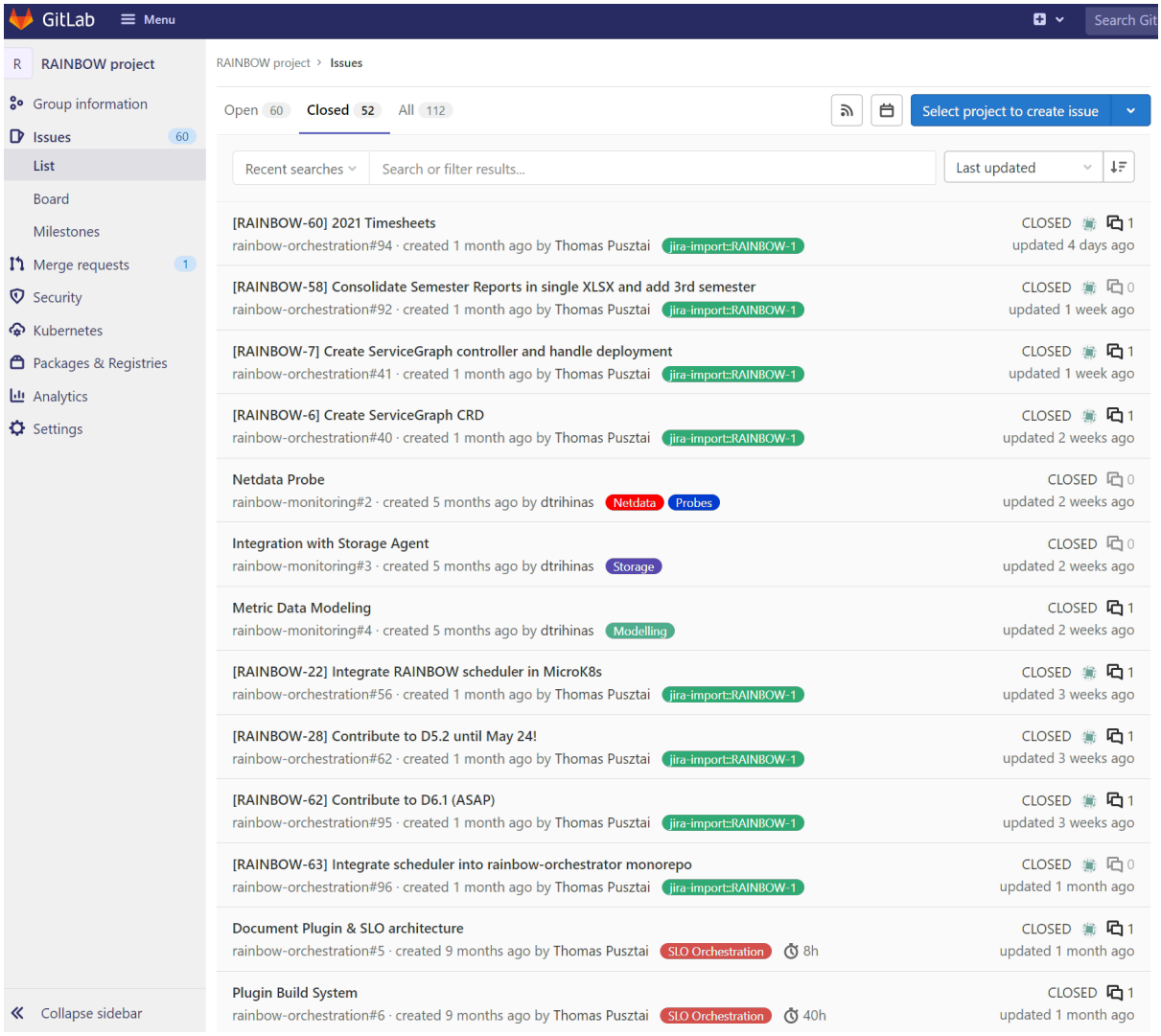


*Figure 25 Issues at the project's GitLab group*

### 3.1.4 Software Quality Evaluation

As explained in D5.1, we planned for the usage of SonarQube for the software quality evaluation part of the platform, as part of the CI process of the project. During the development of the components for this first release we integrated SonarQube mainly for the components of the orchestration layer, as depicted in Figure 26, but more components will be part of the softquare quality evaluation in the next months.

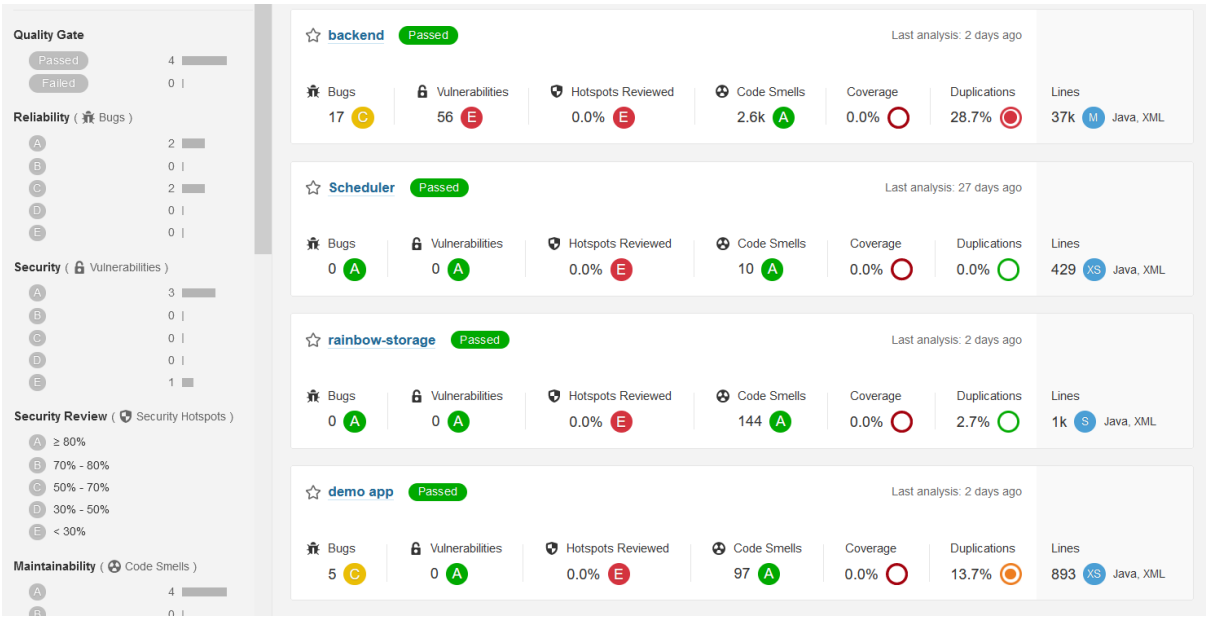


*Figure 26 SonarQube Results of major components*

## 3.2 Testing Procedures of the RAINBOW Early Release

In this section, we collect the results related to the unit and integration testing, as performed so far. We expect more integration testing to take place in the upcoming releases of the platform. Also, as part of the testing plan, user testing will be executed once the RAINBOW platform reaches a better maturity level.

### 3.2.1 Unit Testing

Unit tests are the tool to test the functional modules of the developed software. Therefore, the developer of each component needs to test the components utilizing unit tests before integrating them into the complete application. These unit tests will run in parallel with the integration testing.

### 3.2.2 Integration Testing

The tests performed (manually in most cases) to assure the proper integration of the components were designed based on the interfaces used and are presented below.

*Table 2 Integration test for receiving deployment graphs*

| | |
|---|---|
| **Name** | *Service Graph Deployment* |
| **Description** | *This test is used to test if deployment graphs are successfully provided to the orchestrator backend* |
| **Reference Code** | *IT_01* |
| **Interfaces tested** | DM_01 |
| **Preconditions** | *A predefined application graph at the Editor* |
| **Components involved** | *Orchestration Lifecycle Manager, Orchestration Repository* |
| **Status** | *Test Implemented, executed successfully* |

*Table 3 Integration Test for the assignment of pods to nodes*

| | |
|---|---|
| **Name** | *Schedule pods to nodes* |
| **Description** | *This test is used to test if the Scheduler is able to assign the pods to the nodes that are available* |
| **Reference Code** | *IT_02* |
| **Interfaces tested** | *OLM_01* |
| **Preconditions** | *A Service Graph deployment* |
| **Components involved** | *Deployment Manager, Resource manager* |
| **Status** | *Test Implemented, executed successfully* |

*Table 4 Integration Test for the proper send/receive of SLO violation*

| | |
|---|---|
| **Name** | *SLO check and violation trigger* |
| **Description** | *This test is used to test if the SLO controller is able to check the status of the deployment and trigger actions in case of any violation based on the applied SLO* |
| **Reference Code** | *IT_03* |
| **Interfaces tested** | *OLM_03* |
| **Preconditions** | *A deployed Service Graph and an applied SLO* |
| **Components involved** | *SLO Manager, Elasticity Strategy Controller* |
| **Status** | *Test Implemented, executed successfully* |

*Table 5 Integration Test for the execution corrective elasticity action*

| | |
|---|---|
| **Name** | *Elasticity Strategy Controller counteractions* |
| **Description** | *This test is used to validate that the Elasticity Strategy Controller takes corrective actions that result from the violation of an SLO* |
| **Reference Code** | *IT_04* |
| **Interfaces tested** | *OLM_04* |
| **Preconditions** | *A deployed Service Graph, an applied SLO and an SLO violation trigger* |
| **Components involved** | *SLO Controller* |
| **Status** | *Test Implemented, executed successfully* |

*Table 6 Integration Test for the extraction of monitoring data from the nodes*

| | |
|---|---|
| **Name** | *Monitoring Data extraction* |
| **Description** | *This test is used to validate that real-time and historical data can be extracted correctly from the nodes.* |
| **Reference Code** | *IT_05* |
| **Interfaces tested** | *MON_01* |
| **Preconditions** | *A deployed Service Graph* |
| **Components involved** | Orchestrator,<br>Analytics Service,<br>Storage Service |
| **Status** | *Test Implemented, executed successfully* |

*Table 7 Integration Test for the secure enrolment of devices*

| | |
|---|---|
| **Name** | *Secure Enrolment attestation action* |
| **Description** | *This test is used to validate that a node is secure in order to be enrolled on the cluster by attesting it and then verifying it by the orchestrator* |
| **Reference Code** | *IT_06* |
| **Interfaces tested** | *SE_01* |
| **Preconditions** | *N/A* |
| **Components involved** | Sidecar proxy Interface,<br>Orchestrator (acting as the Verifier) |
| **Status** | *Test Implemented, executed successfully* |

# 4 Plans for Upcoming Releases

RAINBOW follows a specific approach to implement the mechanisms that constitute the RAINBOW framework. RAINBOW development is a continuous process that contains all required discrete steps that re-assure quality during the entire lifetime of the project. However, given the time plan of work in the various WPs, the following integration time plan is followed:

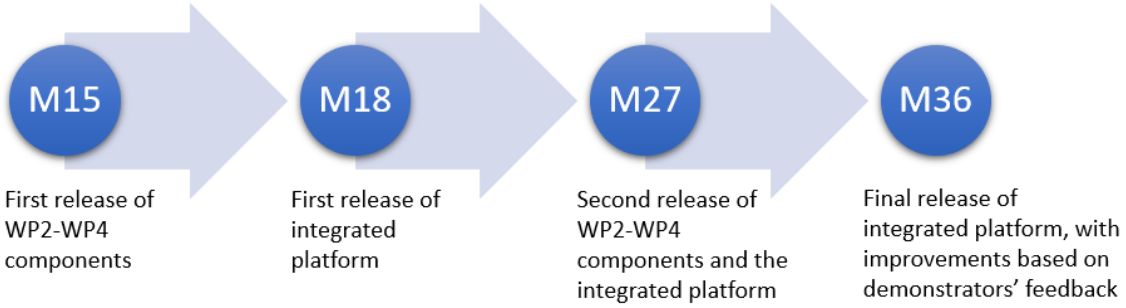


*Figure 27 Roadmap for RAINBOW Development*

## 4.1 Second Release

For the second release, we expect that all functionalities will be available for usage. Some highlights include.

1 Policy Editor will be updated and integrated with the rest of the platform so that policies and SLOs can be created and used through it.
2 Pre-deployment constraint solver to be implemented and integrated to allocate specific compute nodes depending on the needs and optimize the placement of the deployments.
3 Integration of the CJDNS with the key-management and the secure enrolment to onboard new compute nodes in the cluster and verify that they are safe to use.
4 Overall optimization of the dashboard depending on the feedback gained from the Users.
5 Extend the support to more complex SLOs.

## 4.2 Final Release

With all major functionalities in place from the second release, for the final integrated version to be delivered at M36, we will provide improvements based on the feedback derived from demonstrators (WP6). Also, the final release will focus on allowing the easy deployment of the complete RAINBOW platform.

# 5 Conclusions

In this deliverable, we presented the implementation approach and the status of 1st the RAINBOW integrated platform. For creating this first prototype, we used the architecture and integration plan as presented in D5.1. However, as the components were implemented and integrated into a single platform, some updates were made in the architecture; these were presented at the beginning of section 2 to better reflect the actual work performed. Then we provided more details about the components of the platform, regarding their status, their integration, and their deployment; at this first release, the components are deployed as standalone services, with single setup options being planned for the interim and the final release of the platform.

In addition to this initial technical presentation, we tried to provide the reader with a clear view of the platform's status by explaining the provided functionalities in this first release. For this purpose, we also explained how basic flows, such as deployment and scaling, are implemented in terms of components' interactions and from the user perspective by using RAINBOW Dashboard.

Finally, we presented the initial results regarding the software testing and the adoption of the CI process we had given in D5.1, along with the plans for the upcoming releases of the RAINBOW platform.

This report will be continuously updated to reflect the development progress and document the next two releases of the RAINBOW platform in M27 (D5.3) and M36 (D5.4).

# 6 References

1. D5.1 Technical Integration and Testing Plan
2. D1.2 RAINBOW Reference Architecture
3. https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.21.md#v1210
4. https://golang.org/
5. https://github.com/kubernetes-sigs/kubebuilder
6. Rainbow Container Registry: https://gitlab.com/rainbow-project1/rainbow-integration/container_registry
7. Kubectrl tool: https://kubernetes.io/docs/tasks/tools/#kubectl
8. https://kubernetes.io/docs/concepts/scheduling-eviction/schedulingframework/
9. https://slocloud.github.io
10. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/
11. https://www.optaplanner.org/
12. https://gitlab.com/groups/rainbow-project1/-/container_registries
13. Z. Georgiou, M. Symeonides, D. Trihinas, G. Pallis and M. D. Dikaiakos, "StreamSight: A Query-Driven Framework for Streaming Analytics in Edge Computing," 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC), 2018, pp. 143-152, doi: 10.1109/UCC.2018.00023.
14. D2.2 RAINBOW Collective Attestation Policy Enablers Design
15. D2.3 RAINBOW Collective Attestation & Runtime Verification - Version 1

# Annex I: Unit Tests for Early Release

| | |
|---|---|
| **Name** | *Deploy Service Graph* |
| **Description** | *This test submits a RAINBOW Service Graph without SLOs to the Deployment Manager and checks whether it creates the expected deployment resources.* |
| **Reference Code** | *UT_01* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Deployment Manager* |
| **Input** | *Service Graph (YAML)* |
| **Output** | *Set of Kubernetes Deployment resources that should be created from the Service Graph* |
| **Status** | *Implemented with the Go testing package* |

| | |
|---|---|
| **Name** | *Update Service Graph* |
| **Description** | *This test submits an updated Service Graph for an application that is already deployed and checks whether the existing deployment resources are updated accordingly.* |
| **Reference Code** | *UT_02* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Deployment Manager* |
| **Input** | *Service Graph (YAML)* |
| **Output** | *Set of Kubernetes Deployment resources that should be updated, based on the Service Graph* |
| **Status** | *Implemented with the Go testing package* |

| | |
|---|---|
| **Name** | *Service Graph SLO Mapping Creation* |
| **Description** | *This test submits a Service Graph with an SLO to the Deployment Manager and ensures that the corresponding SLO Mapping is created.* |
| **Reference Code** | *UT_03* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Deployment Manager* |
| **Input** | *Service Graph (YAML)* |
| **Output** | *SLO Mapping Custom Resource Definition (CRD) instance* |
| **Status** | *Implemented with the Go testing package* |

| | |
|---|---|
| **Name** | *Service Graph SLO Mapping Update* |
| **Description** | *This test submits a Service Graph with an updated SLO for an application that is already deployed and checks whether the existing SLO Mapping is updated accordingly.* |
| **Reference Code** | *UT_04* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Deployment Manager* |
| **Input** | *Service Graph (YAML)* |
| **Output** | *SLO Mapping Custom Resource Definition (CRD) instance* |
| **Status** | *Implemented with the Go testing package* |

| | |
|---|---|
| **Name** | *Scale out/in through the Horizontal Elasticity Strategy* |
| **Description** | *This test submits a Horizontal Elasticity Strategy CRD that is supposed to trigger a scale out/scale in and checks if the scaling operation is performed accordingly.* |
| **Reference Code** | *UT_05* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Orchestration Lifecycle Manager – Lifecycle Manager* |
| **Input** | *Horizontal Elasticity Strategy CRD instance* |
| **Output** | *Updated Scale sub-resource of the deployment object* |
| **Status** | *Implemented with the Go testing package* |

| | |
|---|---|
| **Name** | *ServiceGraph Scheduler Plugin* |
| **Description** | *This test triggers the ServiceGraph plugin of the RAINBOW Kubernetes scheduler an ensures that it loads the Service Graph of the application that the current pod belongs to.* |
| **Reference Code** | *UT_06* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Orchestration Lifecycle Manager – Scheduler* |
| **Input** | *The pod to be scheduled* |
| **Output** | *The correct Service Graph should be available in the pod's scheduling context* |
| **Status** | *Implemented with the Go testing package* |

| | |
|---|---|
| **Name** | *NetworkQoS Scheduler Plugin* |
| **Description** | *This test triggers the NetworkQoS plugin of the RAINBOW Kubernetes scheduler an ensures that cluster nodes that do not meet the pod's requirements are filtered out.* |
| **Reference Code** | *UT_07* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Orchestration Lifecycle Manager – Scheduler* |
| **Input** | *Set of cluster nodes, the pod to be scheduled, and a Service Graph* |
| **Output** | *List of nodes that satisfy the network QoS requirements.* |
| **Status** | *Implemented with the Go testing package* |

| | |
|---|---|
| **Name** | *SLO Control Loop* |
| **Description** | *This test ensures that the SLO Control Loop periodically executes all active SLOs and that it handles errors within an SLO properly.* |
| **Reference Code** | *UT_08* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Orchestration Lifecycle Manager – Policy Managers* |
| **Input** | *Set of configured SLO instances* |
| **Output** | *Every SLO should be evaluated once per evaluation interval and errors in one SLO should not prevent other SLOs from being evaluated* |
| **Status** | *Implemented with Jest* |

| | |
|---|---|
| **Name** | *Watch Manager* |
| **Description** | *This test configures the Watch Manager to observe instances of a particular SLO Mapping CRD and ensures that additions/changes/deletions of a CRD instance trigger the correct event handlers.* |
| **Reference Code** | *UT_09* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Orchestration Lifecycle Manager – Policy Managers* |
| **Input** | *SLO Mapping Type and respective CRD instances* |
| **Output** | *Method calls to the registered event handlers* |
| **Status** | *Implemented with Jest* |

| | |
|---|---|
| **Name** | *Transformation Service* |
| **Description** | *This test ensures that the Transformation Service used for converting between Kubernetes resources and SLO Controller resource instances transforms the objects properly.* |
| **Reference Code** | *UT_10* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Orchestration Lifecycle Manager – Policy Managers* |
| **Input** | *Kubernetes CRDs and SLO Controller Objects* |
| **Output** | *The transformed SLO Controller objects or Kubernetes CRDs respectively* |
| **Status** | *Implemented with Jest* |

| | |
|---|---|
| **Name** | *CPU Usage SLO Controller* |
| **Description** | *This test triggers evaluations the CPU Usage SLO, with input causing it to report SLO fulfilment, SLO violation with more resources needed, and SLO violation with fewer resources needed.* |
| **Reference Code** | *UT_11* |
| **Responsibilities** | *Implementation: TUW* |
| **Component** | *Orchestration Lifecycle Manager – Policy Managers* |
| **Input** | *SLO Mapping and CPU monitoring data* |
| **Output** | *A Horizontal Elasticity Strategy CRD that reflects the compliance state of the SLO* |
| **Status** | *Implemented with Jest* |

| | |
|---|---|
| **Name** | *Cache Creation* |
| **Description** | *This test is used to verify the Ignite instance deployment and the creation of the necessary caches.* |
| **Reference Code** | *UT_12* |
| **Responsibilities** | *Implementation: AUTH* |
| **Component** | *Data Storage and Sharing* |
| **Input** | - |
| **Output** | *Created cache names* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Add metric* |
| **Description** | *This test is used to verify the correctness of the Ingestion interface through a POST request for data storing.* |
| **Reference Code** | *UT_13* |
| **Responsibilities** | *Implementation: AUTH* |
| **Component** | *Data Storage and Sharing* |
| **Input** | *A Json file of the data to be ingested* |
| **Output** | *A message that the ingestion was successful* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Get latest Metric* |
| **Description** | *This test is used to verify the correctness of the Extraction interface through a POST request that includes a stored metric.* |
| **Reference Code** | *UT_14* |
| **Responsibilities** | *Implementation: AUTH* |
| **Component** | *Data Storage and Sharing* |
| **Input** | *A Json file of the data to be extracted with the necessary flag for the latest value* |
| **Output** | *A Json file with the latest value and metadata on the requested data* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Get historical Metric* |
| **Description** | *This test is used to verify the correctness of the Extraction interface through a POST request that includes a stored metric.* |
| **Reference Code** | *UT_15* |
| **Responsibilities** | *Implementation: AUTH* |
| **Component** | *Data Storage and Sharing* |
| **Input** | *A Json file of the data to be extracted with the necessary fields that denote the time period in question* |
| **Output** | *A Json file with the values and metadata on the requested data for the specified time period* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Get non-existent Metric* |
| **Description** | *This test is used to verify the correctness of the Extraction interface through a POST request that includes a metric that is not stored in the database.* |
| **Reference Code** | *UT_16* |
| **Responsibilities** | *Implementation: AUTH* |
| **Component** | *Data Storage and Sharing* |
| **Input** | *A Json file of the data to be extracted* |
| **Output** | *A Json file with no data fields* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Activate Monitoring Agent* |
| **Description** | *This test is used to assess if the Monitoring Agent daemon instance is deployed and activated on the host environment (fog node).* |
| **Reference Code** | *UT_17* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *Monitoring configuration* |
| **Output** | *Success response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Activate Monitoring Probe* |
| **Description** | *This test is used to assess if a requested Monitoring Probe can be instantiated, configured, and attached to the Monitoring Agent* |
| **Reference Code** | *UT_18* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *Probe id, name, and periodicity* |
| **Output** | *Success response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Attempt to Activate Non-Existent Monitoring Probe* |
| **Description** | *This test is used to assess if a request to activate a Monitoring Probe that does not exist can be handled gracefully* |
| **Reference Code** | *UT_19* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *Probe id* |
| **Output** | *Failed response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Get Monitoring Probe's Metrics* |
| **Description** | *This test is used to assess if a Monitoring Probe is collecting metrics and the metrics are of the correct type* |
| **Reference Code** | *UT_20* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *None* |
| **Output** | *Success response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Change Monitoring Probe Periodicity* |
| **Description** | *This test is used to assess if a Monitoring Probe's periodicity can be altered at runtime* |
| **Reference Code** | *UT_21* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *Periodicity* |
| **Output** | *Success response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Get Monitoring Probes* |
| **Description** | *This test is used to assess if a Monitoring Agent can access its monitoring probes* |
| **Reference Code** | *UT_22* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *None* |
| **Output** | *Success response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Export Monitoring Data* |
| **Description** | *This test is used to assess if a Monitoring Agent can correctly format and export monitoring data* |
| **Reference Code** | *UT_23* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *Exporter (e.g., RAINBOW-Storage)* |
| **Output** | *Success response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Deactivate Monitoring Probe* |
| **Description** | *This test is used to assess if a Monitoring Agent can gracefully deactivate a requested monitoring probe* |
| **Reference Code** | *UT_24* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *Monitoring probe id* |
| **Output** | *Success response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Deactivate Monitoring Agent* |
| **Description** | *This test is used to assess if a Monitoring Agent can be deactivated* |
| **Reference Code** | *UT_25* |
| **Responsibilities** | *UCY* |
| **Component** | *RAINBOW Monitoring* |
| **Input** | *Deactivate status* |
| **Output** | *Success response code* |
| **Status** | *Implemented with python unit test* |

| | |
|---|---|
| **Name** | *Compile Correct StreamSight Query into Analytic Job* |
| **Description** | *This test is used to assess if a correctly given StreamSight query can compile into a Storm job.* |
| **Reference Code** | *UT_26* |
| **Responsibilities** | *UCY* |
| **Component** | *StreamSight (RAINBOW Distributed Processing Engine and Fog Service Analytics Service)* |
| **Input** | *StreamSight query* |
| **Output** | *Storm job* |
| **Status** | *Implemented with java junit* |

| | |
|---|---|
| **Name** | *Handle incorrect StreamSight Query* |
| **Description** | *This test is used to assess if a given incorrect query can be gracefully handled by StreamSight.* |
| **Reference Code** | *UT_27* |
| **Responsibilities** | *UCY* |
| **Component** | *StreamSight (RAINBOW Distributed Processing Engine and Fog Service Analytics Service)* |
| **Input** | *Non-compliant StreamSight query* |
| **Output** | *Bad response code* |
| **Status** | *Implemented with java junit* |

| | |
|---|---|
| **Name** | *Attach Data Stream* |
| **Description** | *This test is used to assess if a -denoted in a query- data source can be attached to a Storm job* |
| **Reference Code** | *UT_28* |
| **Responsibilities** | *UCY* |
| **Component** | *StreamSight (RAINBOW Distributed Processing Engine and Fog Service Analytics Service)* |
| **Input** | *StreamSight query* |
| **Output** | *Success response code* |
| **Status** | *Implemented with java junit* |

| | |
|---|---|
| **Name** | *Attach Insight Output Stream* |
| **Description** | *This test is used to assess if a -denoted in a query- output destination can receive insights from a Storm job* |
| **Reference Code** | *UT_29* |
| **Responsibilities** | *UCY* |
| **Component** | *StreamSight (RAINBOW Distributed Processing Engine and Fog Service Analytics Service)* |
| **Input** | *StreamSight query* |
| **Output** | *Success response code* |
| **Status** | *Implemented with java junit* |

| | |
|---|---|
| **Name** | *AMQP Broker Server* |
| **Description** | *This test is used to monitor the flow of messages through the AMQP Broker server and to measure the end-to-end latency* |
| **Reference Code** | *UT_30* |
| **Responsibilities** | *Implementation: POLITO* |
| **Component** | *AMQP Broker (Demonstrator #2)* |
| **Input** | *CAM and DENM incoming message flow* |
| **Output** | *CAM and DENM message end-to-end latency* |
| **Status** | *Implemented with Python* |

| | |
|---|---|
| **Name** | *City Aggregator Hazardous Events Localizer* |
| **Description** | *This is a qualitative test of the hazardous events localizer functionality of the City Aggregator platform* |
| **Reference Code** | *UT_31* |
| **Responsibilities** | *Implementation: POLITO* |
| **Component** | *City Aggregator platform (Demonstrator #2)* |
| **Input** | *CAM and DENM incoming message flow* |
| **Output** | *Hazardous events localization displayed on the City Aggregator platform* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *City Aggregator Vehicle Localizer* |
| **Description** | *This is a qualitative test of the vehicle localization functionality of the City Aggregator platform* |
| **Reference Code** | *UT_32* |
| **Responsibilities** | *Implementation: POLITO* |
| **Component** | *City Aggregator platform (Demonstrator #2)* |
| **Input** | *CAM and DENM incoming message flow* |
| **Output** | *Live vehicles position displayed on the City Aggregator platform* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Create Attestation Key* |
| **Description** | *This test is used to check that an Attestation Key was created successfully on a remote system.* |
| **Reference Code** | *UT_33* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrolment Agent* |
| **Input** | *Policy Digest (TPM2B_DIGEST), Object Attributes (TPMA_OBJECT)* |
| **Output** | *Attestation Key Certificate (CertifyCreation_Out)* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Verify Attestation Key Creation* |
| **Description** | *This test is used to verify that a remotely created Attestation Key was created correctly by inspecting and validating its Attestation Key Certificate.* |
| **Reference Code** | *UT_34* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrollment Agent* |
| **Input** | *Attestation Key Certificate (CertifyCreation_Out), Attestation Key Public (TPM2B_PUBLIC), Signing Key Public (TPM2B_PUBLIC), Policy Digest (TPM2B_DIGEST), Object Attributes (TPMA_OBJECT)* |
| **Output** | *Boolean true if: (1) the Attestation Key Certificate's internal certifyInfo (TPM2B_ATTEST) structure was created in a TPM, signed by the inverse of the supplied Signing Key Public, and is over the supplied Attestation Key Public, and (2) the supplied Attestation Key Public is bound to the supplied policy digest and carries the supplied properties. False otherwise.* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Add normal PCR* |
| **Description** | *This test is used to register a normal Platform Configuration Register (PCR) on a remote system.* |
| **Reference Code** | *UT_35* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrollment Agent* |
| **Input** | *PCR Index (UINT32)* |
| **Output** | *Not applicable* |
| **Status** | *Implemented with JavaScript* |

| | |
|---|---|
| **Name** | *Add NV-based PCR* |
| **Description** | *This test is used to register and verify the registration of a non-Volatile-based Platform Configuration Register on a remote system.* |
| **Reference Code** | *UT_36* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrollment Agent* |
| **Input** | *NV PCR Index (UINT32), Attributes (TPMA_NV), Policy Digest (TPM2B_DIGEST), Initial Value (TPM2B_MAX_NV_BUFFER)* |
| **Output** | *NV Certificate (NV_Certify_Out)* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Verify NV-based PCR Creation* |
| **Description** | *This test is used to verify that a remotely created non-Volatile-based Platform Configuration Register was created correctly by inspecting and validating its NV Certificate.* |
| **Reference Code** | *UT_37* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrolment Agent* |
| **Input** | *Expected NV Public Area (TPMS_NV_PUBLIC), NV Certificate (NV_Certify_Out), Signing Key Public (TPM2B_PUBLIC), Expected NV-based PCR Value (TPM2B_MAX_NV_BUFFER)* |
| **Output** | *Boolean true if the NV Certificate's internal certifyInfo (TPM2B_ATTEST) structure: (1) is created in a TPM, (2) is signed by the inverse of the supplied Signing Key Public, (3) is over the supplied Expected NV Public Area, and (4) contains the Expected NV-based PCR Value. False otherwise.* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Remeasure Configuration* |
| **Description** | *This test is used to request a remote system to remeasure its configuration.* |
| **Reference Code** | *UT_38* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrolment Agent* |
| **Input** | *PCR Index (UINT32), NV-based (Bool), Fully Qualified Path Name (FQPN), Policy Digest (TPM2B_DIGEST), Signed Policy Digest (TPM2B_DIGEST), Policy Digest Signature (TPMT_SIGNATURE)* |
| **Output** | *Session Audit Digest (GetSessionAuditDigest_Out)* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Verify Configuration Remeasurement* |
| **Description** | *This test is used to verify that a remote system remeasured its configuration and extended the correct PCR by inspecting and validating the Session Audit Digest.* |
| **Reference Code** | *UT_39* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrolment Agent* |
| **Input** | *Session Audit Digest (GetSessionAuditDigest_Out), Signing Key Public (TPM2B_PUBLIC), PCR Index (UINT32), NV-based (Bool), Correct FQPN Digest (TPM2B_DIGEST)* |
| **Output** | *Boolean true if the Session Audit Digest's internal auditInfo (TPM2B_ATTEST) structure: (1) is created in a TPM, (2) is signed by the inverse of the supplied Signing Key Public, (3) contains the expected audit digest, which is calculated by simulating the internal updates to the session's audit digest (see Section 35 of Part 1 of TCG's TPM 2.0 documentation) when running the command TPM2_PCR_Extend (if NV-based is true) or TPM2_NV_Extend (if NV-based is false) and supplying the Correct FQPN Digest as an input parameter. False otherwise.* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Remote Attestation* |
| **Description** | *This test is used to request a remote system to attest its state.* |
| **Reference Code** | *UT_40* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrollment Agent* |
| **Input** | *Not applicable* |
| **Output** | *Nonce Signature (TPMT_SIGNATURE)* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Validate Remote Attestation* |
| **Description** | *This test is used to verify the state of a remote system using only its Attestation Key Public.* |
| **Reference Code** | *UT_41* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrollment Agent* |
| **Input** | *Nonce Digest (TPM2B_DIGEST), Nonce Signature (TPMT_SIGNATURE), Attestation Key Public (TPM2B_PUBLIC)* |
| **Output** | *Boolean true if the Nonce Signature: (1) is signed by the inverse of the supplied Attestation Key Public, and (2) is over the supplied Nonce Digest. False otherwise.* |
| **Status** | *Implemented with Junit* |

| | |
|---|---|
| **Name** | *Remove PCR* |
| **Description** | *This test is used to notify a remote system to remove a Platform Configuration Register (PCR) from consideration. For a Non-Volatile (NV)-based PCR, the remote system is also presented with a Signed Policy Digest (authorization) to allow the deletion the policy-protected NV-based PCR using the TPM2_NV_UndefineSpaceSpecial command (see Part 1 and 3 of TCG's TPM 2.0 documentation for details).* |
| **Reference Code** | *UT_42* |
| **Responsibilities** | *Implementation: DTU* |
| **Component** | *Secure Enrollment Agent* |
| **Input** | *PCR Index (UINT32), NV-based (Bool), cpHashA (TPM2B_DIGEST), aHash Signature (TPMT_SIGNATURE), Policy Digest (TPM2B_DIGEST), Policy Digest Signed (TPM2B_DIGEST), Policy Digest Signature (TPMT_SIGNATURE)* |
| **Output** | *Not applicable* |
| **Status** | *Implemented with Junit* |