



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

D5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Work Package	WP5 – Continuous Integration and Accessibility
Lead Author (Org)	Ioannis Avramidis, Alex Bensenousi (INTRASOFT)
Contributing Author(s) (Org)	Alex Vasileiou, Konstantinos Theodosiou, Giannis Ledakis (UBI); Raphael Schermann (IFAT); Thomas Pusztai (TUW); Moysis Symeonidis (UCY); Heini Bergsson Debes (DTU); Stefanos Venios (SUITE 5); Caseti Claudio Ettore (POLITO); Theodoros Toliopoulos (AUTH)
Due Date	31.12.2022
Actual Date of Submission	30.01.2023
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	15.11.2022	Ioannis Avramidis, Alex Bensenousi (INTRASOFT)	Initial ToC
0.2	25.11.2022	Ioannis Avramidis, Alex Bensenousi (INTRASOFT)	First Draft
0.3	30.11.2022	Konstantinos Theodosiou,(UBI); Thomas Pusztai (TUW); Demetris Trihinas, Moysis Symeonidis (UCY); Stefanos Venios (SUITE 5); Caseti Claudio Ettore (POLITO); Theodoros Toliopoulos (AUTH);	Contributed to components integration status, status, and unit testing.
0.4	10.12.2022	Alex Vasileiou, Giannis Ledakis (UBI); Thomas Pusztai (TUW);	Updating the orchestration part in section 3
0.5	20.12.2022	Ioannis Avramidis, Alex Bensenousi (INTRASOFT), Alex Vasileiou (UBI)	Updated version, including installation instructions (section 4)
0.6	10.01.2023	Ioannis Avramidis, Alex Bensenousi (INTRASOFT)	Updated section 6, ready for review
0.6.1	20.01.2023	Demetris Trihinas (UCY)	1 st Review
0.6.2	23.01.2023	Giannis Ledakis (UBI)	2 nd Review
0.9	27.01.2023	Ioannis Avramidis, Alex Bensenousi (INTRASOFT)	Addressing Reviewers' comments and Final version
1.0	30.01.2023	Christina Stratigaki (UBI)	QA review and Submission

Disclaimer

This document contains material and information that is proprietary and confidential to the RAINBOW Consortium and may not be copied, reproduced, or modified in whole or in part for any purpose without the prior written consent of the RAINBOW Consortium

Despite the material and information contained in this document is considered to be precise and accurate, neither the Project Coordinator, nor any partner of the RAINBOW Consortium nor any individual acting on behalf of any of the partners of the RAINBOW Consortium make any warranty or representation whatsoever, express or implied, with respect to the use of the material, information, method or process disclosed in this document, including merchantability and fitness for a particular purpose or that such use does not infringe or interfere with privately owned rights.

In addition, neither the Project Coordinator, nor any partner of the RAINBOW Consortium nor any individual acting on behalf of any of the partners of the RAINBOW Consortium shall be liable for any direct, indirect, or consequential loss, damage, claim or expense arising out of or in connection with any information, material, advice, inaccuracy or omission contained in this document.



Table of Contents

1. Introduction	9
1.1 Document Purpose and Scope	9
1.2 Relationship with RAINBOW Deliverables	9
1.3 Structure of the deliverable	9
2. RAINBOW Integrated Platform Architecture	10
2.1 Conceptual architecture updates	10
2.2 Technical feedback from the 2nd platform release usage and demonstrators	12
2.2.1 Human Robot Collaboration Demonstrator	12
2.2.2 Digital Transformation of Urban Mobility Demonstrator	12
2.2.3 Power Line Surveillance Demonstrator	13
3 Implementation and Integration Status	15
3.1 Final Release Overview	15
3.1.1 Overall Integration and Component Dependencies	16
3.2 Orchestration Layer Components	16
3.2.1 Logically Centralized Orchestrator	16
3.2.2 Orchestration Lifecycle Manager	21
3.2.3 Pre-deployment Constraint Solver	22
3.2.4 Backend Services	23
3.3 Modeling Layer and Dashboard Components	26
3.3.1 Service Graph Editor & Analytics Editor	26
3.3.2 Policy Editor	28
3.4 Data Management & Analytics Layer Components	29
3.4.1 Data Storage and Sharing	29
3.4.2 Analytics Service	30
3.5 RAINBOW Edge Stack Components	31
3.5.1 Device Management	31
3.5.2 Control Plane Management Module	31
3.5.3 Secure Mesh Routing protocol stack	32
3.5.4 Multi-domain sidecar proxy	33
3.5.5 Storage Agent & Storage Coordination	34
3.5.6 Analytics Worker & Analytics Coordination	34
3.5.7 Resource & Application-level Monitoring Agent	34
3.5.8 Security Enablers	35
4. RAINBOW Platform Installation	37
4.1. Prerequisites	37
4.2. RAINBOW Platform Setup	37
5. RAINBOW Usage Guide	44
6. Technical Evaluation and Quality Assurance	56
6.1. Continuous Integration and Quality Assurance	56
6.1.1. Version Control System – Gitlab	56
6.1.2 Container Registry	56



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

6.1.3 Issue Tracking – Gitlab	57
6.1.4 Software Quality Evaluation	58
6.1.5 Continuous Integration Flow	58
6.2. Testing Procedures of the RAINBOW Final Release	59
6.3. Unit Testing	59
6.4. Integration Testing	59
7. Conclusions	65
References	66
<i>Annex I: Unit Tests for Final Release</i>	67



List of tables

Table 1 Overview of Final Release Functionalities	15
Table 2 Public RAINBOW Orchestrator APIs.....	18
Table 3 Internal RAINBOW Orchestrator API Types	20
Table 4 APIs for User & Infrastructure Management	24
Table 5 APIs for Service Graph, Analytics Interpreter and Policies Interpreter	25
Table 6 APIs for Data Storage & Sharing.....	29
Table 7 APIs for Analytics Service	30
Table 8 APIs for Mesh Routing Protocol Stack	31
Table 9 APIs for Mesh Routing Protocol Stack	32
Table 10 APIs for Mesh Routing Protocol Stack.....	32
Table 11 APIs for Mesh Routing Protocol Stack.....	33
Table 12 APIs for Monitoring Agent	35
Table 13 APIs for Mesh Routing Protocol Stack.....	35
Table 14 Analytic Stack master node installation variables	39
Table 15 Analytic Stack worker node installation variables	41

List of figures

Figure 1 Roadmap for RAINBOW Development	10
Figure 2 Roadmap for RAINBOW Development	15
Figure 3 Logically Centralized Orchestrator Components and Interactions	17
Figure 4 Orchestration Lifecycle Manager Components and Interactions.....	21
Figure 5 Affinity/Anti-affinity rules button.....	27
Figure 6 Create an affinity rule	27
Figure 7 Analytics Editor update.....	28
Figure 8 Policy Editor update	28
Figure 9 Monitoring Agent overview	34
Figure 10 Core platform installation scripts	38
Figure 11 Analytics Stack master node installation files.....	39
Figure 12 Analytics Scheduler Configuration File.....	40
Figure 13 Analytics Stack worker node installation files.....	40
Figure 14 Monitoring configuration file.....	41
Figure 15 Dashboard installation scripts.....	42
Figure 16 Login page of the RAINBOW Dashboard	44
Figure 17 Main page of the RAINBOW Dashboard	45
Figure 18 Components' list.....	45
Figure 19 Component configuration.....	46
Figure 20 Application creation	46
Figure 21 Applications' list.....	47
Figure 22 Resource creation	47
Figure 23 Application Instance creation	48
Figure 24 Application Instance service graph editor	48
Figure 25 Application Instance service graph editor - Component editing.....	49
Figure 26 Application Instance service graph editor - affinity/anti-affinity	49



Figure 27 Application Instance service graph editor - set affinity rules.....	50
Figure 28 Application Instances list.....	50
Figure 29 Service graph monitoring - part 1.....	51
Figure 30 Service graph monitoring - part 2.....	51
Figure 31 Analytics and SLO Editor	52
Figure 32 Creation of a new analytic	52
Figure 33 Adding expressions on the analytic.....	53
Figure 34 Creation of a new SLO	53
Figure 35 Add metrics in the SLO	54
Figure 36 Add Computations to the SLO.....	54
Figure 37 Add Expressions to the SLO.....	55
Figure 38 RAINBOW's Gitlab group and repositories.....	56
Figure 39 RAINBOW container images.....	57
Figure 40 RAINBOW issues	58
Figure 41 RAINBOW's CI flow	59
Figure 42 Defining request parameters of a REST call.....	60
Figure 43 Defining assertions based on the expected response of a REST call	61
Figure 44 Overview of integration tests in ReadyAPI	62
Figure 45 Integration test results (part 1)	63
Figure 46 Integration test results (part 2)	64



List of acronyms

Acronym	Full name
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IT	Integration Testing
IPR	Intellectual Property Rights
PCR	Platform Configuration Register
RAM	Random Access Memory
REST	Representational state transfer
SDK	Software Development Kit
SLO	Service Level Objectives
TPM	Trusted Platform Module
UI	User Interface
URL	Uniform Resource Locator
UT	Unit Testing
VCS	Version control systems
WPx	Work Package
YAML	<i>YAML Ain't Markup Language</i>



Executive Summary

This deliverable is a public report presenting the final software release of the RAINBOW integrated platform. It presents the functionalities provided by the components as part of the integrated platform and provides the final version of the architecture with highlights on the interface. In total, 3 distinct releases of the RAINBOW Platform were planned with 3 corresponding supporting documents. This document constitutes the version of a live document that was constantly updated to depict the developments of the RAINBOW platform and, which coincides with the final release of the RAINBOW integrated platform.

For this final release full integration has been achieved among all platform components, and minor improvements have been made, thus providing a homogenized user experience. A complete flow of platform usage is part of the document, along with updated instructions for the installation of the platform as a whole. Finally, updated results of the technical evaluation and quality assurance are provided, including integration testing that has been executed along with the delivery of last release of the RAINBOW integrated platform.



1. Introduction

1.1 Document Purpose and Scope

This document has as a purpose to accompany RAINBOW's final platform release. The following chapters describe the final steps taken with regards to the integration of RAINBOW's components towards a complete, stable, functional, and user-friendly framework as well as the guidelines for the final prototype installation and utilization to facilitate its future evolvement, use and exploitation.

1.2 Relationship with RAINBOW Deliverables

Like its previous releases, this deliverable uses RAINBOW's outcomes such as the reference architecture, integration approach, and overall, the platform evolution as documented in several deliverables like D1.2, D5.1, 5.2 and 5.3. In the same way, this document also consolidates the technical developments of the different components under WP2, WP3 and WP4, and presents the results on the testing and integration procedures and actual work as reported in the respective first and second release deliverables. Finally, for this document we also utilized feedback resulted from the final releases of the demo reports i.e., D6.3, D6.5, and D6.7 as input for further fixes and improvements. That being so, all information provided herein is used as support material for the final release of the RAINBOW platform.

1.3 Structure of the deliverable

The rest of the deliverable is structured as follows.

- Section 2 presents updates on the architecture based on the feedback that was collected from the second release
- Section 3 provides an overview of the functionalities provided and integration points of RAINBOW's final release.
- Section 4 adumbrates the last version of the platform's installation and set-up guides.
- Section 5 presents how the RAINBOW platform is used.
- Section 6 provides the latest results of the technical evaluation
- Section 7 concludes the document.

2. RAINBOW Integrated Platform Architecture

2.1 Conceptual architecture updates

The goal of this deliverable is to report on the fine-grained APIs of the finalized architecture. The architecture per se has not been amended when compared to previous deliverables since the componentization represents the final code-level structure. However, in the frame of this deliverable, 18 concrete groups of APIs have been abstracted to make the presentation of the platform more comprehensive. Figure 1 depicts the final architecture along the code names of the reported APIs. The notation that has been selected is *LayerX.GroupY-API*.

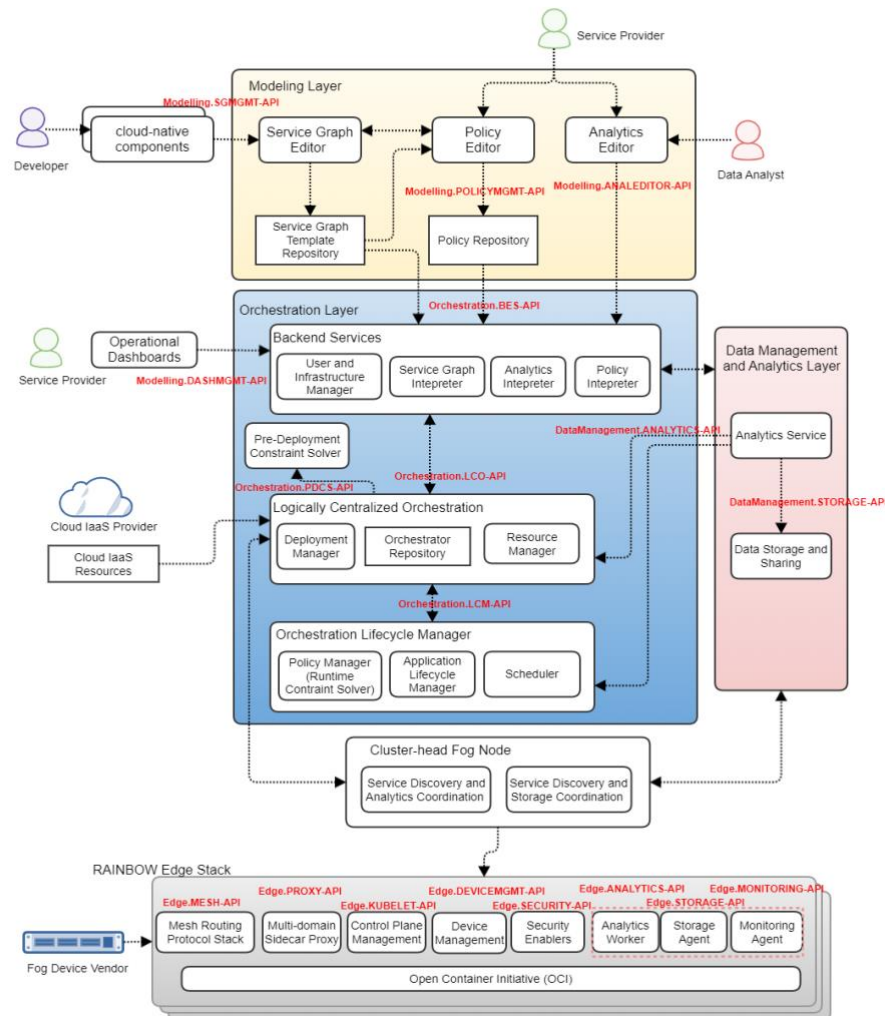


Figure 1 Roadmap for RAINBOW Development

Initially, the **Modelling.SGMGMT-API** (Service Graph Management API for Editor & Repository) is responsible for exposing all methods that author service graphs. This API includes not only valid service graphs, but also metadata that are used during initial



deployment and runtime reconfiguration. Moreover, the **Modelling.POLICYMGMT-API** (Policy Editor & Repository API) is responsible for exposing methods that are attaching (adding/removing) deployment-time and run-time policies.

The **Orchestration.BES-API** (Backend Services API) is the 'thin layer' between the modelling tools and the logically centralized orchestrator. It is the cornerstone API through which instances of service graphs and policies are interpreted. On the other hand, the **Orchestration.LCM-API** (Orchestration Lifecycle Manager API) exposes the public state of executable service graphs and manages this state through low-level orchestration commands. Regarding the initial state of a service graph deployment, the **Orchestration.PDCS-API** (Pre-Deployment Constraint Solver) is responsible for finding the optimal solution for placement; thus, making use of all soft and hard constraints that are expressed through the 'attached' policies.

The binding on the runtime orchestration elements with the K8S runtime is provided by the **Orchestration.LCO-API** (Logically Centralized Orchestration API). The project took the decision to conceptually comply with Kubernetes (K8S) i.e., to use the extensibility mechanisms for job scheduling, placement and management. These low-level bindings are exposed by the LCO-API. The high-level interaction of a DevOps user with the Orchestrator is performed through the **Modelling.DASHMGMT-API** (Dashboard API). Beyond the 'logically centralized part', RAINBOW relies on an edge 'bundle' which is addressed as RAINBOW Edge Stack. The submodules that comprise this bundle are 8, which include:

- **Edge.DEVICEMGMT-API** (Device Management API) for capabilities exposure (sensors, actuators, TPMs)
- **Edge.MESH-API** (Mesh Routing Protocol Stack API) for materialized the overlay onboarding
- **Edge.KUBELET-API** (Control Plane Management Module API) for materializing the joining to the logical centralized k8s cluster
- **Edge.PROXY-API** (Multi-domain Sidecar Proxy API) for configuring envoy service HTTP/GRPC proxies
- **Edge.SECURITY-API** (Security Enablers/Attestation API) for performing integrity verification tests
- **Edge.MONITORING-API** (Resource and Application Monitoring Agent API) for low level monitoring stream extraction
- **Edge.STORAGE-API** (Storage Agent API) that support analytic pipeline edge storage requirements
- **Edge.ANALYTICS-API** (Analytics Workers API) that support analytic pipeline edge execution requirements

Finally, a set of complementary APIs are exposed in order to perform Analytics tasks. These includes the **Modelling.ANAEDITOR-API** (Analytics Editor API) for authoring pipelines and **DataManagement.STORAGE-API** (Data Storage and Sharing API) and **DataManagement.ANALYTICS-API** (Analytics Service API) for storage and execution respectively.

All the aforementioned APIs are thoroughly discussed in the frame of the current deliverable.



2.2 Technical feedback from the 2nd platform release usage and demonstrators

Towards improving the final release of the rainbow platform, we took advantage of the feedback gained from the use case demonstrators while installing, configure and using second release's different functions in a physical demo set up. The value of these tests was that they allowed us to get our hand on the behaviour of the system in real conditions. More specifically, the integration team had the chance to compare the platform's deployment and usage against the 1st release, and evaluate the monitoring, configuration of service level objectives and metrics, as well as the usage of different platform components. Overall, the deployment and use of the 2nd release of RAINBOW platform presented no significant issues, while the platform was found to be more user-friendly and feature-rich by the demo partners. On the other hand, as a result of this testing activity, RAINBOW's technical partners also received bug reports as well as points where the platform's behaviour is open to further improvements as can be found bellow.

2.2.1 Human Robot Collaboration Demonstrator

In this pilot, a human-robot collaboration system was tested with multiple applications such as robot motion tracking, personnel localization and collision prediction, requiring these applications to have (i) scalability, (ii) easy management, (iii) analytics, and (iv) Quality of Service. RAINBOW's orchestration, deployment, management and analytics functionalities were used to accomplish these goals. After the running of this demonstrator the following bugs fixing/feature request were suggested:

Deployment recommendations:

- All the remnants of any previous RAINBOW installation were removed before starting the installation of the 2nd release
 - [feature request, for easier upgrades as part of a commercial solution].
- A more dynamic management (through the dashboard UI) of the SLO policies and analytics will be considered an UI enhancement
 - [feature request, taken into account for the improvements in final version of SLO UI].

Configuration of Service Level Objective and Metrics:

- Adding SLOs in this use case requires other metrics such as network and custom metrics such as the queue properties of the used RabbitMQ queues
 - [Feature request, taken into account and network metrics have been provided as part of the final release of RAINBOW].

2.2.2. Digital Transformation of Urban Mobility Demonstrator

In this case, the goal was to demonstrate how RAINBOW can contribute to fulfil a real-time geo-referenced notification system about a hazardous situation for vehicles travelling in urban areas while also acting in the vehicle communication field. This is a scenario where the optimum balancing between MEC and Fog Node in terms of energy



consumption, bandwidth occupancy, accuracy and latency is required, while a secure data collection and distribution of messages to enable a scalable bidirectional communication is equally important. RAINBOW's modelling layer, orchestration, secure enrolment, privacy preserving exchange of messages were used to achieve these goals. After the running of this demonstrator the following bugs fixing/feature request were suggested:

Deployment recommendations:

- On the deployment of the 2nd release, Nvidia Xavier was used without a reset of the machine to avoid the kernel setup that was a complex procedure already for the 1st release.
 - [Nonissue, but a limitation in the current version of the OS of the Nvidia Jetson/Xavier device. We expect better support on newer OS versions.]

2.2.3. Power Line Surveillance Demonstrator

The role of the physical demonstrator was to simulate real conditions during inspection missions along power lines by implementing a distributed GCS that will govern a swarm of drones to optimize their operations and increase the swarm's range, autonomy while eliminating the execution of fail-safe procedures due to interference or interruptions in the radio link. RAINBOW's service graph and policy editor, pre-deployment container solver, analytics service, and mesh routing protocol stack were employed to address these challenges. After the running of this demonstrator the following bugs fixing/feature request were suggested:

Deployment recommendations:

- Since the worker nodes were based on the Jetson TX2 computers, an additional preparatory step was required. The Linux kernel had to be recompiled to ensure that options required by the mesh networking stack are enabled. This was a bit challenging since the Jetson modules were mounted on carrier boards that required non-standard kernels
 - [Non issue, but a limitation in the current version of the OS of the Nvidia Jetson/Xavier device. We expect better support on newer OS versions].
- It is advisable to develop a tool that will allow users to assess the state of the RAINBOW cluster so that they can check whether the installation has succeeded and whether all components are working as expected with just a single command to save a considerable amount of time
 - [Feature request. Currently users have to check various component for the status, feature should be part of any commercial solution based on RAINBOW].
- Verification of the analytic stack required lots of effort and a bit of experimentation
 - [Feature request. Already included UI parts along the installation of the analytics stack for the final release, more improvements could be part of any commercial solution based on the analytic stack].



Configuration of Service Level Objective and Metrics:

- Extending the dashboard should be done after the project becomes more widely adopted and it will be possible to determine what SLO definitions are most frequently used
 - [Feature request. A interesting idea for improving the user experience, feature could be part of any commercial solution based on RAINBOW].

3 Implementation and Integration Status

In this section, we describe the status of the overall RAINBOW platform for the final release. To be in this position, we followed the development and integration plan presented in D5.1 and the feedback received from the physical set up demos to provide a platform that can be characterized as stable. RAINBOW followed a standard approach to implement the RAINBOW framework mechanisms by adopting from early on a continuous process that contained a set of discrete steps that re-assured its high quality. In specific, the following integration time plan is followed:

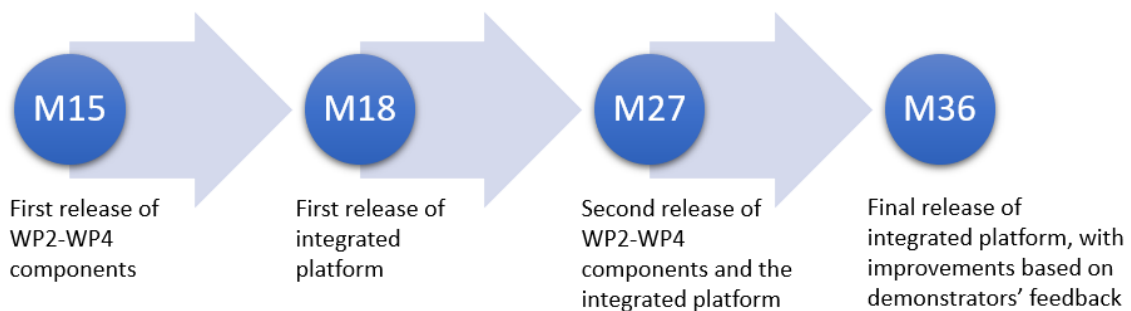


Figure 2 Roadmap for RAINBOW Development

In the following subsections we provide the status for the platform as whole (section 3.1), and then we proceed to more detailed description of the improvements and updates done per component. To facilitate the reading process, we separate the various components per layer; in section 3.2 we present the current state of implementation and integration of Orchestration layer, in 3.3 we present the components of the Modelling Layer, in section 3.4 the components of Data Management and Analytics Layer and in section 3.5 we present the edge stack.

3.1 Final Release Overview

Table 1 Overview of Final Release Functionalities

Tool - Service	Participants	API Specification Codename
Orchestration Layer		
Orchestration Lifecycle Manager	TUW - UBI - UCY - AUTH	Orchestration.LCM-API
Pre-Deployment Constraint Solver	TUW - UBI - UCY	Orchestration.PDCS-API
Logically Centralized Orchestration	TUW	Orchestration.LCO-API
Backend Services	UBI - SUITE5	Orchestration.BES-API
Modeling Layer and Dashboard		
Service Graph Editor & Repository	SUITE5 - UBI - TUW	Modelling.SGMGMT-API
Analytics Editor	SUITE5 - UBI - UCY - AUTH	Modelling.ANAEDITOR-API
Policy Editor & Repository	SUITE5 - UBI - UCY - TUW	Modelling.POLICYMGMT-API



Dashboard	SUITE5 -UBI	Modelling.DASHMGMT-API
Data Management & Analytics Layer		
Data Storage and Sharing	AUTH - UCY - K3Y - SUITE5	DataManagement.STORAGE-API
Analytics Service	UCY - SUITE5 - TUW - AUTH - UNIS	DataManagement.ANALYTICS-API
Edge Stack		
Mesh Routing Protocol Stack	UBI - IFAT - DTU - POLITO	Edge.MESH-API
Multi-domain Sidecar Proxy	UBI - UCY - INTRA	Edge.PROXY-API
Security Enablers	POLITO - IFAT- DTU UBI- K3Y	Edge.SECURITY-API
Storage Agent	AUTH - UCY - K3Y - SUITE5	Edge.STORAGE-API
Resource and Application Monitoring Agent	UCY - TUW - UNIS - K3Y - AUTH	Edge.MONITORING-API
Analytics Workers	UCY	Edge.ANALYTICS-API
Device Management	UBI	Edge.DEVICEMGMT-API
Control Plane Management Module	UBI	Edge.KUBELET-API

3.1.1 Overall Integration and Component Dependencies

The integration status of high-level interfaces between the different RAINBOW components was reported in previous deliverables. For the final release since all dependencies have been resolved in previous versions, some final adjustments on APIs are reported below.

3.2 Orchestration Layer Components

As its name suggests, the orchestration Layer includes all components responsible for the deployment and orchestration of the applications using RAINBOW. The overall architecture of this layer remains the same as for the second release of RAINBOW.

3.2.1 Logically Centralized Orchestrator

The RAINBOW Logically Centralized Orchestrator is responsible for managing the deployment and resources of RAINBOW applications, as well as storing metadata about them. It receives service graphs that have been deployed by the Backend Services and provides status information about them.

The RAINBOW Logically Centralized Orchestrator consists of the three loosely coupled components highlighted in Figure 1, i.e., the Resource Manager, the Deployment Manager, and the Orchestrator Repository.

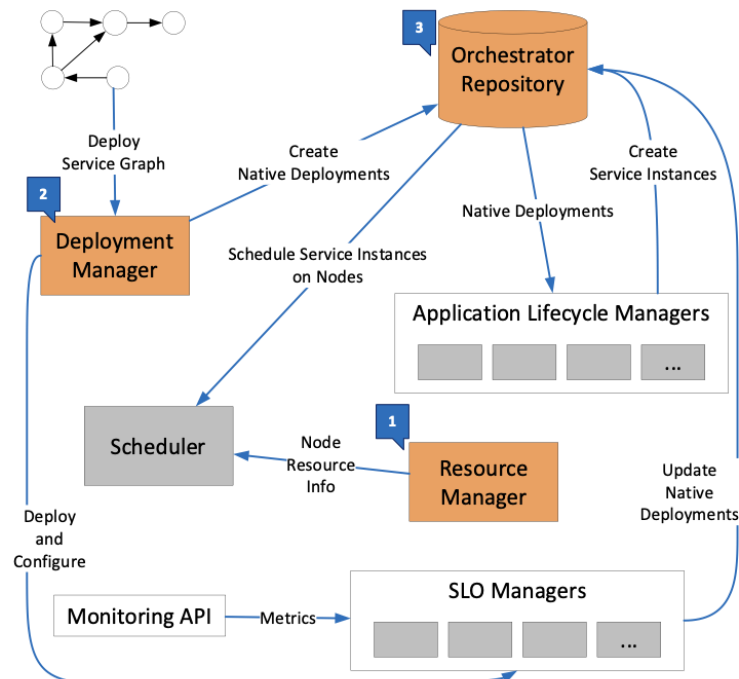


Figure 3 Logically Centralized Orchestrator Components and Interactions

Resource/Container Manager

Based on the state-of-the-art analysis and requirements elicitation performed in WP1, the RAINBOW platform uses Kubernetes as its Resource Manager. Specifically, this release of RAINBOW is built on top of the vanilla Kubernetes distribution v1.21.

RAINBOW relies on a fixed schema for handling fog specific resources, such as GPS sensors or cameras, using the extended resources and labels mechanisms provided by Kubernetes. Non-sharable resources that need to be exclusively assigned to a container, e.g., a video camera, are represented as extended resources, which allow managing quantities. Conversely, sharable resources, which may be used by multiple containers simultaneously, e.g., a GPS sensor, are represented using labels on nodes.

Deployment Manager

The Deployment Manager is implemented as a Kubernetes controller responsible for service graphs. It also provides the Kubernetes Custom Resource Definition (CRD) [4] for service graph objects. It is written in Go [5] and relies on scaffolding and the controller framework provided by kubebuilder [6]. Upon submission of a service graph, the Deployment Manager creates and/or updates Kubernetes-native deployments and RAINBOW Service Level Objective (SLO) configurations. Additionally, it provides status information about the deployments to the UI through the status subresource of each service graph.



This component relies on the Service Graph Editor as UI and the Backend Services as the middleware responsible for all the transformation and the communication services and Kubernetes as the Resource Manager.

The Deployment Manager is able to configure all aspects of an SLO and its elasticity strategy and adds the ability to provide application-specific configuration properties through the service graph, which are implemented as Kubernetes ConfigMaps.

Orchestrator Repository

The Orchestrator Repository is divided between a MySQL database, which stores information about the deployment lifecycle and orchestration process, and an etcd key-value store, which houses all information relevant for Kubernetes.

APIs and Integration Status

All components of the RAINBOW Logically Centralized Orchestrator depend on the underlying Kubernetes distribution (vanilla Kubernetes v1.21 for this release of the RAINBOW platform). Kubernetes must be set up and configured with the RAINBOW Mesh networking components. Afterwards, the orchestrator components can be deployed and take their responsibility of creating, modifying, and deleting Kubernetes-native and RAINBOW-specific resources, based on applications' service graphs.

In Table 2 we present all interfaces exposed by the component, along with their descriptions.

Table 2 Public RAINBOW Orchestrator APIs

Method	Path	Description	Used By
GET	/apis/v1/namespaces	Returns a list of all namespaces registered in the orchestrator	User and Infrastructure Manager, Service Graph Interpreter
GET	/apis/v1/namespaces/<name>	Returns the namespace object with the specified name	User and Infrastructure Manager, Service Graph Interpreter
POST	/apis/v1/namespaces	Creates a new namespace object	User and Infrastructure Manager, Service Graph Interpreter
PUT	/apis/v1/namespaces/<name>	Replaces the namespace object with the specified name. To be successful, the resourceVersion number in the body must match the current version of the object.	User and Infrastructure Manager, Service Graph Interpreter



Method	Path	Description	Used By
DELETE	/apis/v1/namespaces/<name>	Deletes the namespace with the specified name	User and Infrastructure Manager, Service Graph Interpreter
GET	/apis/fogapps.k8s.rainbow-h2020.eu/v1/namespaces/<namespace>/servicegraphs	Returns a list of all Service Graphs in the specified namespace	User and Infrastructure Manager, Service Graph Interpreter
GET	/apis/fogapps.k8s.rainbow-h2020.eu/v1/namespaces/<namespace>/servicegraphs/<name>	Returns the Service Graph object (which includes its deployment status) with the specified namespace and name	User and Infrastructure Manager, Service Graph Interpreter
POST	/apis/fogapps.k8s.rainbow-h2020.eu/v1/namespaces/<namespace>/servicegraphs	Creates a new Service Graph object in the specified namespace	User and Infrastructure Manager, Service Graph Interpreter
PUT	/apis/fogapps.k8s.rainbow-h2020.eu/v1/namespaces/<namespace>/servicegraphs/<name>	Replaces the Service Graph object with the specified namespace and name. To be successful, the resourceVersion number in the body must match the current version of the object.	User and Infrastructure Manager, Service Graph Interpreter
DELETE	/apis/fogapps.k8s.rainbow-h2020.eu/v1/namespaces/<namespace>/servicegraphs/<name>	Deletes the Service Graph with the specified namespace and name.	User and Infrastructure Manager, Service Graph Interpreter
GET	/apis/cluster.k8s.rainbow-h2020.eu/v1/namespaces/default/networklinks	Returns a list of all Network Link objects that are part of the cluster topology graph.	User and Infrastructure Manager, Service Graph Interpreter
GET	/apis/cluster.k8s.rainbow-h2020.eu/v1/namespaces/default/networklinks/<name>	Returns the Network Link object with the specified name	User and Infrastructure Manager, Service Graph Interpreter
POST	/apis/cluster.k8s.rainbow-h2020.eu/v1/namespaces/default/networklinks	Creates a new Network Link object in the cluster topology graph	User and Infrastructure Manager, Service Graph



Method	Path	Description	Used By
	worklinks		Interpreter
PUT	/apis/cluster.k8s.ra inbow- h2020.eu/v1/name spaces/default/net worklinks/<name>	Replaces the Network Link object with the specified name in the cluster topology graph. To be successful, the resourceVersion number in the body must match the current version of the object.	User and Infrastructure Manager, Service Graph Interpreter
DELETE	/apis/cluster.k8s.ra inbow- h2020.eu/v1/name spaces/default/net worklinks/<name>	Deletes the Network Link with the specified name from the cluster topology graph	User and Infrastructure Manager, Service Graph Interpreter

To get a specific API path, the placeholders <GROUP> and <TYPE> need to be replaced with the group and type name values from the list of API types in Table 2.

Table 3 Internal RAINBOW Orchestrator API Types

Object Type	Group	Type Name	Used By
Custom Stream Sight SLO Mapping	slo.k8s.rainbow -h2020.eu	customstreams ightslomapping s	Deployment Manager, SLO Policy Managers
Network QoS SLO Mapping	slo.k8s.rainbow -h2020.eu	networkqoslom appings	Deployment Manager, SLO Policy Managers
Migration Elasticity Strategy	elasticity.k8s.ra inbow- h2020.eu	migrationelasti citystrategies	SLO Policy Managers, Application Lifecycle Managers
OPC UA Message Elasticity Strategy	elasticity.k8s.ra inbow- h2020.eu	opcuaessagee lasticitystrategi es	SLO Policy Managers, Application Lifecycle Managers
Horizontal Elasticity Strategy	elasticity.polari s-slo- cloud.github.io	horizontalelasti citystrategies	SLO Policy Managers, Application Lifecycle Managers
Vertical Elasticity Strategy	elasticity.polari s-slo- cloud.github.io	verticalelasticit ystrategies	SLO Policy Managers, Application Lifecycle Managers
Kubernetes Deployment	apps	deployments	Deployment Manager, Scheduler, Kubernetes
Kubernetes StatefulSet	apps	statefulsets	Deployment Manager, Scheduler, Kubernetes
Kubernetes ConfigMap	core	configmaps	Deployment Manager, Deployed Services, Kubernetes

Object Type	Group	Type Name	Used By
Kubernetes Service	core	services	Deployment Manager, Kubernetes
Kubernetes Ingress Config	networking.k8s.io	ingresses	Deployment Manager, Kubernetes

3.2.2 Orchestration Lifecycle Manager

The Orchestration Lifecycle Manager is part of the Orchestration layer and consists of the three loosely coupled components highlighted in Figure 2, i.e., the Scheduler, the SLO Policy Managers and the Application Lifecycle Managers.

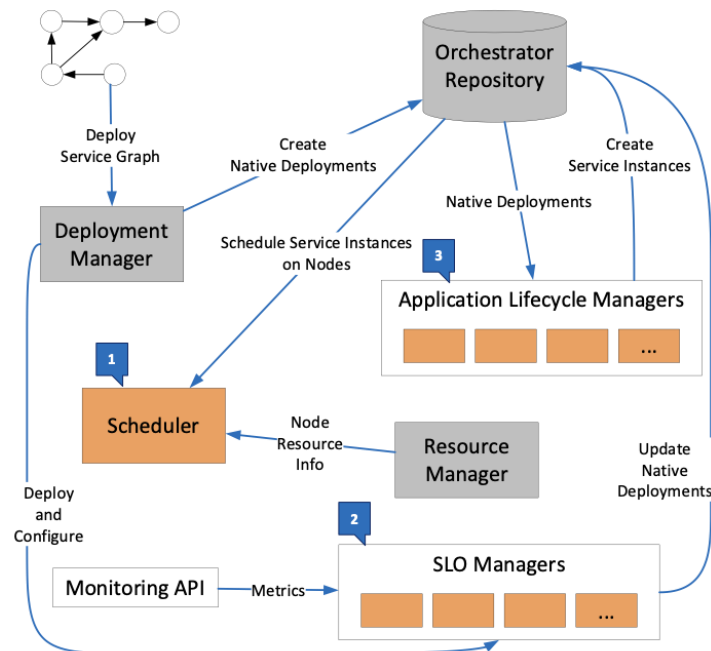


Figure 4 Orchestration Lifecycle Manager Components and Interactions

Scheduler

The Scheduler assigns each service instance to a node for execution, according to its requirements and constraints. It is implemented in Go and built on top of the Kubernetes Scheduling Framework.

The RAINBOW Scheduler plugins enable fog awareness by respecting network Quality of Service (QoS) constraints/network SLOs and fog optimized resource distribution.

SLO Policy Managers

The SLO Policy Managers monitor the SLO compliance of deployed services and trigger elasticity strategies upon violations. They are implemented as Kubernetes controllers in TypeScript using the Polaris framework.



The RAINBOW platform features a generic SLO controller that can be used to build custom SLOs, based on multiple metrics, from the UI.

Application Lifecycle Managers

The Application Lifecycle Managers are responsible for managing the service instances and for executing the elasticity strategies. Service instances management is provided natively by Kubernetes. The elasticity strategies are implemented in TypeScript using the Polaris framework.

The RAINBOW platform features a horizontal elasticity strategy, a vertical elasticity strategy, a migration elasticity strategy to move services from one node to another, and an elasticity strategy to send messages to IoT devices via OPC UA.

APIs and Integration Status

All components of the RAINBOW Orchestration Lifecycle Manager depend on the underlying Kubernetes distribution (vanilla Kubernetes v1.21), the CRDs provided by the components of the RAINBOW Logically Centralized Orchestrator, and its APIs. The Orchestration Lifecycle Manager does not provide an API on its own.

3.2.3 Pre-deployment Constraint Solver

The Pre-deployment Constraint Solver has two major responsibilities: i) validation of a submitted service graph against the corresponding CRD schema, which is handled natively by Kubernetes, and ii) semantic validation of the service graph, e.g., ensuring that it does not contain any loop, which is performed by a custom Admission Webhook [11].

Originally, the Pre-deployment Constraint Solver was planned to be implemented with OptaPlanner [12], but this decision was changed at the second release of the RAINBOW platform, leading to the implementation of the lightweight validation mechanisms describe above. Since the Scheduler is the component that is responsible for finding a placement for each service that satisfies its constraints, it solves the constraints satisfaction problem in an online fashion. The design of the RAINBOW Scheduler ensures, upon the initial deployment or an application, that either all its services are scheduled or none at all. Thus, a duplication of this constraint solving logic from the scheduler in the Pre-deployment Constraint Solver would have provided little additional benefit, while requiring additional processing time for each service graph.

APIs and Integration Status

The Pre-deployment Constraint Solver depends on the underlying Kubernetes distribution (vanilla Kubernetes v1.21) and the CRDs provided by the components of the RAINBOW Logically Centralized Orchestrator. The Pre-deployment Constraint Solver does not provide a public API but is triggered by the Logically Centralized Orchestrator.



3.2.4 Backend Services

The Backend Services of the RAINBOW Orchestration layer acts as the link between the Modelling Layer, the Data Management and Analytics Layer and the Logically Centralized Orchestration and are responsible for all the underlying communication between those layers. The structural elements of the Backend Services are the User and Infrastructure Manager, the Service Graph Interpreter, the Analytics Interpreter and the Policy Interpreter. The User and Infrastructure Manager are finalised during the latest release of RAINBOW and no further updates are implemented. On the other hand, and based on the users feedback, the Service Graph Interpreter, the Analytics Interpreter and the Policy Interpreter receive some new features and updates.

User and Infrastructure Manager

The User and Infrastructure Manager undertakes two main tasks, the user management and the infrastructure management. The user management includes all the operations for users and organizations, such as the registration, authentication, authorization etc. The infrastructure management is responsible for the registration, authentication and authorization of the cloud/fog provider.

Service Graph Interpreter

The Service Graph Interpreter is responsible for the interpretation of an abstract service graph that comes from the Modeling Layer and the delivery to the Logically Centralized Orchestration. Also, it receives and manages all the status updates from the Logically Centralized Orchestration and updates accordingly the Modeling Layer. At the final integration of the RAINBOW platform, we further enhance the service graph in order to support affinity and anti-affinity rules for the components placement. This feature allows constraining components placement on the cluster nodes against other components of the same service graph.

Analytics Interpreter

The Analytics Interpreter receives an analytics query from the Analytics Editor component of the Modeling layer, interprets it to the query language that is used by the Data Management and Analytics Layer and forwards the query to that layer. Then receives back the results from the applied query and interprets them for consumption by the Modeling Layer. During the final integration we implement a separation on the metric type that is used in the analytics query between the component specific metrics (i.e cpu utilisation, ram usage etc) and user custom metrics (i.e frames per second etc) in order to provide a better user experience. This update does not affect the core functionality of the Analytics Interpreter, but it requires the update of the interpretation procedure of the received query into the query language that is used by the Data Management and Analytics Layer.

Policy Interpreter



The Policy Interpreter is responsible for the interpretation of the RAINBOW's Service Level Objective (SLO) configuration from the Policy Repository component of the Modeling layer into the service graph that will be sent to the Logically Centralized Orchestration. The SLOs are part of the service graph object and when a SLO received from the Policy Interpreter, it interprets it, updates the corresponding service graph object and sends the updated service graph to the Logically Centralized Orchestration. During the final integration of the RAINBOW platform, we implement a separation between the component specific metrics (i.e cpu utilisation, ram usage etc) that is used by the SLO and the user custom metrics (i.e frames per second etc), similar to the analytics interpreter update, that is used by the SLO. As in the Analytics Interpreter case, the core functionality of the Policy Interpreter remains unaffected, but an update is required in the way that an incoming SLOs will be interpreted by the Policy Interpreter.

APIs and Integration Status

In the table below we present the most important interfaces provided by the Backend Services Component.

Table 4 APIs for User & Infrastructure Management

Method	Path	Description	Used By
PUT	/api/v1/user	Updates user's information	Operational Dashboards
POST	/api/v1/user	Creates a new user	Operational Dashboards
GET	/api/v1/user/{id}	Retrieves user's information by Id	Operational Dashboards
DELETE	/api/v1/user/{id}	Deletes a user by Id	Operational Dashboards
POST	/api/v1/user/list	Retrieves all users	Operational Dashboards
GET	/api/v1/providertype/{id}	Retrieves the cloud/fog provider type	Operational Dashboards
POST	/api/v1/providertype/list	Retrieves all the available provider types	Operational Dashboards
PUT	/api/v1/provider	Updates provider's information	Operational Dashboards
POST	/api/v1/provider	Creates a new provider	Operational Dashboards
GET	/api/v1/provider/{id}	Retrieves provide's information by Id	Operational Dashboards
DELETE	/api/v1/provider/{id}	Deletes a provider by Id	Operational Dashboards
GET	/api/v1/auth/user	Retrieves the authenticated user	Operational Dashboards



Method	Path	Description	Used By
POST	/api/v1/auth/logout	End user's session/ Logout the user	Operational Dashboards

Table 5 APIs for Service Graph, Analytics Interpreter and Policies Interpreter

Method	Path	Description	Used By
PUT	/api/v1/component	Updates a component	Service Graph Editor
POST	/api/v1/component	Creates a component	Service Graph Editor
GET	/api/v1/component/{id}	Fetches a component by id	Service Graph Editor
DELETE	/api/v1/component/{id}	Deletes a component by id	Service Graph Editor
POST	/api/v1/component/list	Fetches a list of components	Service Graph Editor
POST	/api/v1/component/affinities	Creates component affinity rules	Service Graph Editor
DELETE	/api/v1/component/affinities/ {id}	Delete a component affinity rules by id	Service Graph Editor
GET	/api/v1/component/affinities/ {application_instance_id}	Fetches a component affinity rules by a specific application id	Service Graph Editor
PUT	/api/v1/application	Updates a Service graph	Service Graph Editor
POST	/api/v1/application	Creates a Service graph	Service Graph Editor
GET	/api/v1/application/{id}	Fetches a Service graph by id	Service Graph Editor
DELETE	/api/v1/application/{id}	Deletes a Service graph by id	Service Graph Editor
POST	/api/v1/applicationinstance/{a pplicationInstanceId}/request/ undeployment	Undeploys a Service Graph instance	Service Graph Editor
POST	/api/v1/applicationinstance/{a pplicationInstanceId}/request/ deployment	Deploys a Service Graph instance	Service Graph Editor
POST	/api/v1/applicationinstance/{a pplicationInstanceId}/request/ cancellation	Cancel the deployment of a Service Graph instance	Service Graph Editor
GET	/api/v1/metric/{id}/applicatio ninstance/{applicationInstancel d}	Retrieves the analytics query results for a specific query and application instance	Analytics Editor
DELETE	/api/v1/metric/delete/{id}	Deletes an analytics	Analytics



Method	Path	Description	Used By
		query by Id	Editor
POST	/api/v1/metric/create/{applicationInstanceId}	Create new analytics query for a specific application instance	Analytics Editor
POST	/api/v1/metric/applicationinstance/{applicationInstanceId}/list	Retrieves all analytics queries for a specific application instance id	Analytics Editor
GET	/api/v1/metric/applicationinstance/{applicationInstanceId}/componentnode/{componentNodeHexID}/metrics	Retrieves all the available metrics for a specific application instance and a specific component	Analytics Editor
POST	/api/v1/elasticity/create/{applicationInstanceId}	Create a new elasticity policy for a specific application instance	Policy Editor
GET	/api/v1/elasticity/applicationinstance/{applicationInstanceId}/slo/{sloId}	Retrieves a specific elasticity policy for a specific application	Policy Editor
PUT	/api/v1/elasticity/applicationinstance/{applicationInstanceId}/slo/{sloId}	Updates a specific elasticity policy for a specific application	Policy Editor
DELETE	/api/v1/elasticity/applicationinstance/{applicationInstanceId}/slo/{sloId}	Deletes a specific elasticity policy for a specific application	Policy Editor
POST	/api/v1/elasticity/applicationinstance/{applicationInstanceId}/list	Fetches a list of elasticity policies for a specific application	Policy Editor

More details about this API are provided in the GitLab repository of RAINBOW using the OpenAPI standard.

3.3 Modeling Layer and Dashboard Components

3.3.1 Service Graph Editor & Analytics Editor

The Service Graph Editor & Analytics Editor are implemented as part of the UI and belong to the modeling layer of the RAINBOW's architecture. The Service Graph Editor & Analytics Editor is composed of two main parts, the first part (Service Graph Editor) is responsible for the authoring and maintaining of the application templates of cloud-native components along with the maintenance of the deployment operation. The second part (Analytics Editor) is responsible for the monitoring of the deployed cloud-native components. During the previous releases of RAINBOW, the core functionality of this



component has been finalised and fully integrated with the rest of the RAINBOW components, but during this final integration of the RAINBOW platform, based also on our use-cases feedback, we added some new features and updates.

At the Service Graph Editor, we added a new feature where the users can create affinity and anti-affinity rules. This feature, as depicted in Figure 4 and Figure 5 below, enables the placement of the service graph component in cluster nodes against other components of the service graph based on the user's affinity or anti-affinity rules.

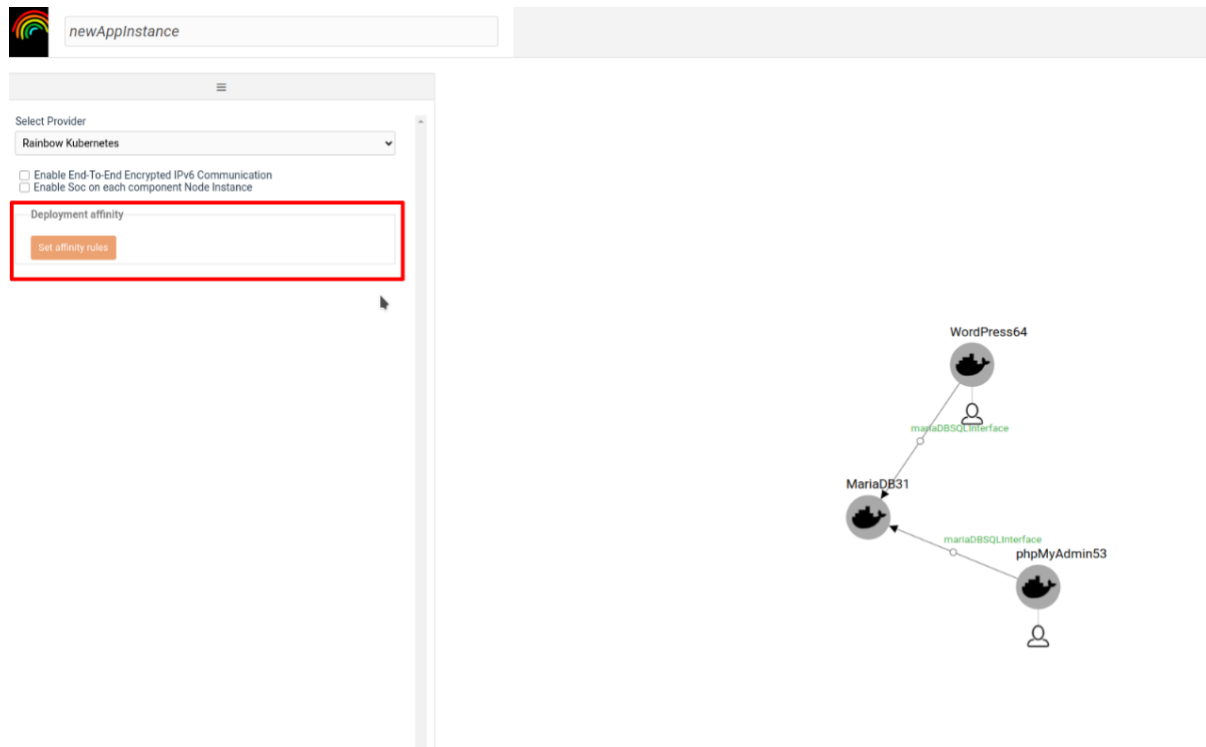


Figure 5 Affinity/Anti-affinity rules button

Set the Pod Affinity

New Affinity

Type: ☒ Affinity ☐ Anti-affinity

Target component: MariaDB31

Related components: phpMyAdmin53

Save

No affinities available.

Figure 6 Create an affinity rule

At the Analytics Editor we updated the editor to create a separation on the metric type that is used in the analytics query between the component specific metrics (i.e., CPU utilisation, ram usage etc) and user custom metrics (i.e., frames per second etc) in order to provide a better user experience. The core functionality remains as was in the previous release, the only change is a radio button in the editor as shown in Figure 7.



Figure 7 Analytics Editor update

3.3.2 Policy Editor

The Policy editor is a component of the Modelling layer that is responsible to apply instructions/guidelines regarding how the overall application should behave prior to deployment and during runtime. These instructions are addressed as SLOs and when created by the user, are sent to the Policy Interpreter for the interpretation and further processing of the RAINBOW platform. The Policy Editor was finalised and fully integrated during the previous release, as far as concerns the final integration the core functionality remains as was and only a minor update took place on the editor. That update concerns the separation of the metrics that are used in the SLOs in component specific metrics (i.e., CPU utilisation, ram usage etc) and user custom metrics (i.e., frames per second etc). The new functionality is highlighted in the Figure 7 below.

Figure 8 Policy Editor update

APIs and Integration Status



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

3.4 Data Management & Analytics Layer Components

3.4.1 Data Storage and Sharing

The Data Storage and Sharing component of the Data Management & Analytics Layer is the main storage unit for the monitoring metrics and any other metadata needed by the RAINBOW components, transparent to the end-user. The component includes two main services for data exchange and more specifically the extraction and ingestion services. Both are available with different variations behind a REST API that is stable since the previous platform release.

The Data Storage and Sharing component also includes a data placement service that replaces the default data replication algorithms of the underline distributed database framework, i.e., Apache Ignite. The data placement algorithm works in the background to replicate data from one storage instance to another in order to reduce extraction latency and data availability. For the final release of the RAINBOW platform, the data placement algorithm has been finalized and evaluated in a real-world scenario.

Table 6 APIs for Data Storage & Sharing

Method	Path	Description	Used By
POST	/nodes	Returns the list of active storage instances with their hostnames and their type of instance.	Analytics Service
POST	/put	Ingestion of monitoring data.	Resource & Application-level Monitoring Agent
POST	/get	Returns monitoring data with their values.	Analytics Service, Policy Editor, Backend Services
POST	/query	Returns an aggregated value from the monitoring data.	Analytics Service, Policy Editor, Backend Services
POST	/list	Returns a list of the monitoring metadata.	Analytics Service
DELETE	/monitoring	Deletes the specified monitoring data.	Resource & Application-level Monitoring Agent
POST	/analytics/put	Ingestion of analytics data.	Analytics Service
POST	/analytics/get	Returns analytics data with their values.	Analytics Service, Policy Editor, Backend Services
DELETE	/analytics	Deletes the specified analytics data.	Analytics Service
POST	/app/put	Ingestion of timestamped data (main-memory).	Analytics Service



Method	Path	Description	Used By
POST	/app/get	Returns timestamped data with their values.	Analytics Service, Policy Editor, Backend Services
DELETE	/app	Deletes the specified timestamped data.	Analytics Service

3.4.2 Analytics Service

The RAINBOW Analytics Service is a part of the Data Management & Analytics layer that helps with data processing for the RAINBOW ecosystem. It allows for real-time analysis of a large amount of data collected from the underlying fog resources and performance indicators from IoT applications. The service is designed to be distributed, meaning that data processing happens where the data is generated, so that analysis can be done quickly with low latency and without the data leaving the network of collaborating fog nodes. It is built on Apache Storm and includes scheduling algorithms that optimize streaming analytic queries and take into account unique factors such as energy consumption, latency, and data quality in the many locations where IoT applications are deployed.

The RAINBOW Analytics Service has three core components, namely the Analytics Enabler, the declarative analytic queries, and the Analytics Workers. The Analytics Enabler is the Orchestration Service that manipulates the distributed processing environment and orchestrates the execution of the analytic tasks. The latter service materialized by the Apache Storm, but we also extended it with novel Fog-enabled scheduling algorithms. The Analytic Stack accepts declarative queries written in StreamSight [cite] language, and StreamSight translates these queries into executables and deploys them on the underlying execution engine (Apache Storm). Finally, the Analytics Workers perform the analytic duties of the submitted jobs and are deployed on the fog nodes that the user has allocated for the deployment.

The interactions and coordination actions between the Analytic Workers and the Analytics Enabler are handled by the Apache Storm cluster. The other components of the RAINBOW communicate with the Analytics Enabler by performing HTTP rest API calls. Specifically, we provided a detailed list of the possible API calls that the components can perform.

In the table below we present the most important interfaces provided by the Analytics Services. We should mention that for the retrieval of monitoring data and for storing generated insights, Analytics Services utilize the Data Storage and Sharing Services.

Table 7 APIs for Analytics Service

Method	Path	Description	Used By
GET	/api/insights/{deployment_id}	Returns the status of the job that has the given {deployment_id}.	Logically Centralized Orchestration, Analytics Editor



Method	Path	Description	Used By
POST	/api/insights/{deployment_id}	Submits the job with an id equal to {deployment_id}. The latter id will be used to collect its status with the appropriate GET request. This API requires as a body the StreamSight queries the user wants the job to execute.	Logically Centralized Orchestration, Analytics Editor
PUT	/api/insights/{deployment_id}	Submits the job with an id equal to {deployment_id}. The latter id will be used to collect its status with the appropriate GET request. This API requires as a body the StreamSight queries the user wants the job to execute.	Logically Centralized Orchestration, Analytics Editor
DELETE	/api/insights/{deployment_id}	Deletes the job that has the given {deployment_id}.	Logically Centralized Orchestration, Analytics Editor

3.5 RAINBOW Edge Stack Components

3.5.1 Device Management

When a device is onboarded on a cluster its capabilities are advertised in the logical centralized orchestrator. Such capabilities include (indicatively) the existence of TPM (for security reasons), the existence of special sensors/actuators etc. The API of the device management component that is installed each device being onboarded is summarized in the table below.

Table 8 APIs for Mesh Routing Protocol Stack

Method	Path	Description	Used By
GET	/device/capabilities	A read-only method that provides the summary of the device capabilities	Logically Centralized Orchestration
PUT	/device/reboot	A put method that forces the device to reboot	Logically Centralized Orchestration
GET	/device/status	A read-only method that fetches the connectivity status of the device	Logically Centralized Orchestration

3.5.2 Control Plane Management Module

Upon a successful onboarding to the mesh environment, an IoT node must join a K8S cluster. This functionality is encapsulated under the family of kubelet-related methods as depicted below. These methods are triggered by the Logically Centralized Orchestration



to commission and decommission the physical node to the cluster. The entire communication is performed using the IPv6 overlay network that is being setup based on the secure mesh routing protocol stack as analysed in the next section.

Table 9 APIs for Mesh Routing Protocol Stack

Method	Path	Description	Used By
POST	/kubernetes/connect	A method that forces a node to attach to a logical centralized k8s master	Logically Centralized Orchestration
DELETE	/kubernetes/disconnect	A method that forces a node to disconnect from a k8s master	Logically Centralized Orchestration
GET	/kubernetes/status	A method that reports the node's connectivity status	Logically Centralized Orchestration

3.5.3 Secure Mesh Routing protocol stack

The purpose of the Secure Mesh Routing stack is to establish and maintain a network of edge nodes which will be used for control-plane and data-plane signaling. Hence, the stack provides the 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 and **automate the process of binding** to a 'logically centralized' Kubernetes cluster. In general, a Mesh network is a type of network where each node in the network may act as an independent (peer) router, regardless of whether it is connected to another network or not.

In a mesh environment network addresses are not statically configured since the risk of conflict is high. Therefore, plain IP assignment protocols cannot work. Hence it is the purpose of the Mesh Protocol Stack to a) **Define automatically an address** within minimum chance of collision; b) **Use this address to join a peer-to-peer network with "limited access"** since the existing trusted network has to attest the new node; c) **Execute the attestation protocol** in order to be accepted in a security-overlay; d) **Take part in the selection process of a cluster-representative** (cluster-head) which will be used to offload several computational tasks.

The following tables summarize the exposed API methods of the respective components that are grouped by.

Table 10 APIs for Mesh Routing Protocol Stack

Method	Path	Description	Used By
PUT	/mesh/alterconnectmode/{modeid}	A method that forces the node to change the layer-2 connectivity mode. The possible modes are BLIND, or ATTESTATION_BASED	Logically Centralized Orchestration
POST	/mesh/joinmesh/{meshid}	A method that attempts to join a node in an existing formulated cluster based on the MODEID	Logically Centralized Orchestration



Method	Path	Description	Used By
DELETE	/mesh/leave mesh/{meshid}	A method that forces a node to leave a cluster unconditionally	Logically Centralized Orchestration
GET	/mesh/lookup/{nodeid}	A method that performs DHT lookup to check if a node exists in the formulated topology	Logically Centralized Orchestration
GET	/mesh/neighborhood	A method that exposes the first degree of connections for a connected node	Logically Centralized Orchestration
GET	/mesh/nodeid	A method that returns the descriptor of a node. The descriptor has all types of information regarding connectivity	Logically Centralized Orchestration
GET	/mesh/public key	A method that retrieves the public key of the k8s paster	Logically Centralized Orchestration
GET	/mesh/routing table	A method that fetches a consolidated version of the routing states	Logically Centralized Orchestration
PUT	/mesh/setgateway/{gatewayid}	A method that announces (in a broadcast mode) the new gateway to the cluster	Logically Centralized Orchestration
GET	/mesh/status	A method that reports the mesh-related connectivity state	Logically Centralized Orchestration

3.5.4 Multi-domain sidecar proxy

Each node that participates in the k8s cluster can host a containerized application. These applications can be controlled by a transparent proxy that can be installed on top of the exposed ports. This stands true in case ports expose HTTP and RPC services. In RAINBOW, the open source 'envoy' component has been utilized. Its low level API is exposed in the following link: <https://www.envoyproxy.io/docs/envoy/latest/api/api>. However, RAINBOW has the obligation to preconfigure the proxy during the deployment process. As such, the high-level API calls that are exposed for such configurations is provided below.

Table 11 APIs for Mesh Routing Protocol Stack

Method	Path	Description	Used By
POST	/sidecar/apply/{nodeid} /{componentid}	A method that configures an envoy proxy on top of an existing component	Logically Centralized Orchestration
PUT	/sidecar/remove/{nodeid} /{componentid}	A method that removes an envoy proxy on top of an existing component	Logically Centralized Orchestration



3.5.5 Storage Agent & Storage Coordination

The APIs related to the Storage agent are presented in section 3.4.2.

3.5.6 Analytics Worker & Analytics Coordination

The APIs related to the Storage agent are presented in section 3.4.2.

3.5.7 Resource & Application-level Monitoring Agent

A Monitoring Agent is enabled on every cluster node, in order to capture Fog-node system metrics like resource utilization, and metrics from the containerized services. To do that, the Monitoring Agent enables probes that are provided by the platform. Moreover, users can expose other metrics by creating new probes extending the Probe interface of the RAINBOW monitoring SDK. Moreover, the monitoring SDK provides functions for application-level metrics extraction, thus users can enable this functionality and the system automatically disseminates metrics to the Monitoring Agent. Through that, users can view and interact with performance data in a single unified environment instead of dealing with different monitoring tools.

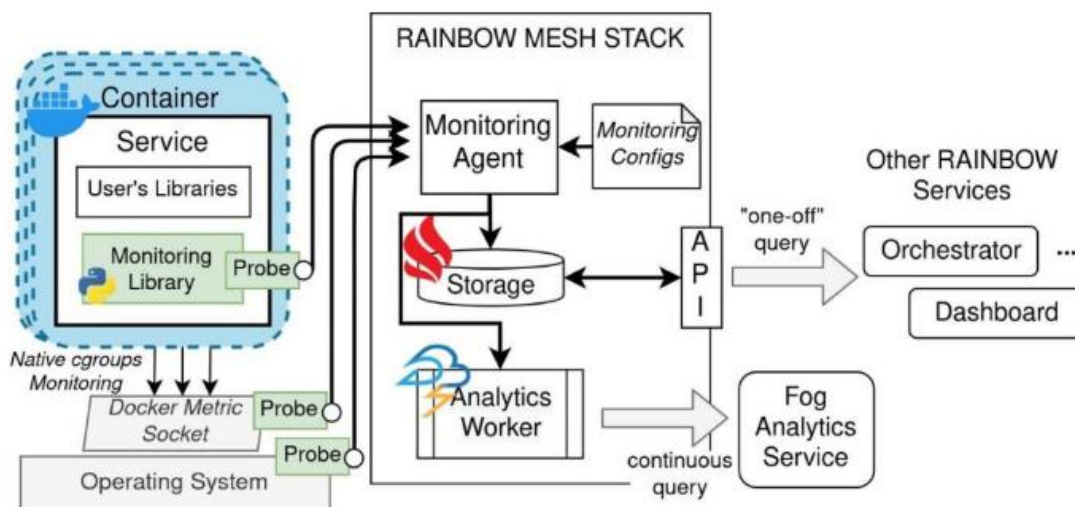


Figure 9 Monitoring Agent overview

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 interconnecting the user's fog nodes. The Monitoring Agent exports all monitoring data to the local Storage Agent, allowing users to query for both real-time and historical data that is persistently stored via the Storage Fabric built on top of the overlay mesh network that connects the user's fog nodes.

We should note that the monitoring agent exposes the monitored metrics to the storage, so do not receive any direct API request from the RAINBOW components. However, the RAINBOW monitoring SDK communicates with the Agent in order to publish the app-level metrics.



Table 12 APIs for Monitoring Agent

Method	Path	Description	Used By
POST	/metrics/	This API call is exposed by the monitoring agent and the MonitoringSDK library can disseminate application-level metrics	User's application via MonitoringSDK

3.5.8 Security Enablers

For a mesh node to join a cluster a 'verification' process has to take place. The verification process is initiated by the Logical Centralized Orchestrator and is addressed as attestation. In the jargon of attestation, the initiator is addressed as Verifier and the entity that is being validated is addressed as prover. The communication among the verifier and the prover is addressed as attestation protocol. The protocol relies on the fact that initial integrity measures (a.k.a. golden hashes) are collected 'offline'.

During runtime, the mesh admission control protocol is requesting the execution of the formal attestation protocol prior to assigning a cryptographic key and an IPv6 address that will be used for the control plane signalling. The API calls that materialize the attestation process are the following:

Table 13 APIs for Mesh Routing Protocol Stack

Method	Path	Description	Used By
POST	/attestation/trigger	Logically Centralized Orchestration acting as a Verifier calls this endpoint to initiate the attestation process by providing a random nonce generated and signed by the Verifier (challenge). After the signature verification, the attestation component returns the triggers internally the attestation process via the Attestation Controller providing the same nonce, the signature of the Verifier and the type of service to be invoked.	Logically Centralized Orchestration
POST	/nodeid}/attestation/response	This endpoint is triggered by the Prover via the control plane to provide the attestation response. In the case of Attestation by Quote a quoted message signed by the AK is returned from the Attestation Controller to the Local Control and Management Framework and finally to the Logically Centralized Orchestrator (Verifier) to verify the signature and the quote with the stored golden hashes. In the case of Attestation by Proof a signature with the AK is returned through the same path to the Verifier to verify the signature representing the correct state of the devices.	Logically Centralized Orchestration



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

Method	Path	Description	Used By
PUT	/{"nodeid"} /attestation/ TRashValues	The method sends a list of hash values (golden hashes stored in the Measurements Database) representing the trusted configuration of the binaries that are known to be correct. This represents the correct state that needs to be considered in the CIV process and is immediately followed by the creation of the Attestation Key (AK) binded to the new expected state as a trusted reference value.	Logically Centralized Orchestration



4. RAINBOW Platform Installation

For the final release of RAINBOW, we further enhance the installation procedure in order to include all the RAINBOW components, provide more advanced configuration options and an easy-to-use installation guide for the whole platform. All the installation instructions are available online and continuously updated based on the users' feedback in a ReadTheDocs page, at <https://rainbow-h2020.readthedocs.io>.

For the purpose of document completeness, in the following subsections we will provide the complete installation instructions.

4.1. Prerequisites

RAINBOW supports a wide variety of Linux capable devices (VMs, Raspberry pi, wearables, drones etc) and the most widespread distributions such as Ubuntu and Debian.

In the previous release of RAINBOW, we provided the installation instructions and prerequisites for the core platform components, in this release we also developed an automated procedure for the Dashboard component that needs a separate node (Bare metal or VM) to operate. For that reason, we separate the prerequisites into the core components prerequisites and the Dashboard prerequisites.

For the core platform components, the minimum execution requirements are at least 4 CPU cores, 8GB of RAM, 40GB of storage and x86 based CPU architecture for the master node and 2 CPU cores, 2GB of RAM, 20 GB of storage and either x86 or ARM based CPU architecture for the worker nodes. For more advanced use cases we propose a master node with at least 4 CPU cores and 16GB of RAM and worker nodes with 2 CPU cores and 4GB of RAM. Moreover, RAINBOW supports GPU enabled devices, as also most of the devices which register under the `/dev` Linux path

For the Dashboard component the minimum execution requirements are 2 CPU cores, 4GB of RAM, 20 GB of storage and x86 based CPU architecture.

4.2. RAINBOW Platform Setup

Compared to the second release of RAINBOW we logically divided the installation procedure into three main stages. The first stage is the core platform setup and consists of the docker engine, the Mesh Network, the Kubernetes cluster and the Rainbow components such as the Logically Centralized Orchestrator along with the Orchestration Lifecycle Manager and their subcomponents. The second stage is the Analytics Stack setup which consists of the Monitoring, Data Storage and Analytic Services components and the third stage is the Dashboard setup which consists of the Dashboard component. All the installation scripts along with instructions are also gathered in a public accessible GitLab repository, at <https://gitlab.com/rainbow-project1/rainbow-installation>.



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

main

rainbow-installation / installation-scripts /

+

History

Find file

Web IDE

Download

Clone





















Name	Last commit	Last update
..		
 calico.yaml	add installation script files	10 months ago
 cjdns.txt	add installation script files	10 months ago
 coreDnsDeployment.json	add installation script files	10 months ago
 init-02-cjdns.sh	feat: add dashboard	6 months ago
 init-03-cjdns-ipv6.py	add installation script files	10 months ago
 init-03-cjdns-worker.py	add installation script files	10 months ago
 init-04-configure-host.py	add installation script files	10 months ago
 init-05-docker-debian.sh	add installation script files	10 months ago
 init-05-docker-ubuntu.sh	add installation script files	10 months ago
 init-06-docker-configure.sh	add installation script files	10 months ago
 init-07-k8s-debian.sh	added missing update step after adding the kubernet...	6 months ago
 init-07-k8s-ubuntu.sh	add installation script files	10 months ago
 init-08-k8s-master.sh	add installation script files	10 months ago
 init-09-k8s-master-configure.py	add installation script files	10 months ago
 init-10-install-rainbow-orchestrator.sh	add installation script files	10 months ago
 init-11-cjdns-master-credentials.py	add installation script files	10 months ago
 init-12-k8s-join-master.py	add installation script files	10 months ago
 k8s-init-config-ipv6.yaml	add installation script files	10 months ago
 rainbow-v3-master.sh	feat: update main installation scripts	2 hours ago
 rainbow-v3-worker.sh	feat: update main installation scripts	2 hours ago

Figure 10 Core platform installation scripts

The first step of the first stage is the setup and configuration of the cluster's master node that can be achieved by the execution of the *rainbow-v3-master.sh* script, as depicted in Figure 10. The only configuration that is required by the user is to provide the necessary docker credentials for the containerized components. So, in order to achieve this the user needs to edit the *rainbow-v3-master.sh* by setting the corresponding docker variables at the beginning of the script and then just execute it.

```
$sudo ./rainbow-v3-master.sh
```

The installation procedure consists of the setup and configuration of the docker engine, the prerequisites of the Mesh Network along with the Mesh Network itself, the Kubernetes Master node and finally the Rainbow components such as the Logically Centralized Orchestrator along with the Orchestration Lifecycle Manager and their subcomponents. After the successful execution of the script it will provide an accomplishment message along with necessary information for the next steps. In case of a failed execution the script will stop the procedure and will display the error that occurred.



The second step of the first stage is the setup and configuration of the cluster's worker nodes that can be achieved by the execution of the *rainbow-v3-worker.sh* script, as depicted in Figure 10, at each one of the worker nodes. In this case the user needs to configure the script with the values that are printed after the successful execution of the master's node script. To accomplish this the user needs to open the *rainbow-v3-worker.sh* script and set the corresponding variables at the beginning of the script and then just execute it.

```
$sudo ./rainbow-v3-worker.sh
```

There are some special occasions in some devices, where the official linux kernel had some flags disabled and the Mesh Network was not able to work properly. In RAINBOW we have addressed that issue and concluded in some pre-configuration steps which downloads the source code of the kernel, enables the necessary flags, recompiles the source code and then installs the new kernel. Since that procedure can cause a lot of issues and need extra attention by the user, we do not offer it as an automated script. All the aforementioned steps are offered as analytic instructions in the form of a Readme file in the public GitLab repository, at <https://gitlab.com/rainbow-project1/rainbow-installation/-/tree/main/xavier-device> and as a special section on the ReadTheDocs page, at [https://rainbow-h2020.readthedocs.io/en/latest/UsageGuide/a rainbow platform installation.html#special-case](https://rainbow-h2020.readthedocs.io/en/latest/UsageGuide/a%20rainbow%20platform%20installation.html#special-case).

Name	Last commit	Last update
..		
storm	update configurations and docker-composes of analytic and mo...	4 months ago
.env	introduce more detailed configurations	4 months ago
docker-compose.yaml	update configurations and docker-composes of analytic and mo...	4 months ago

Figure 11 Analytics Stack master node installation files

The second stage is the installation of the Analytics stack with the first step to be the installation and configuration on the master node. For that step the user needs to configure the variables of the *.env* file, which is depicted in Figure 11. Those variables along with their description are shown in the Table 8 below.

Table 14 Analytic Stack master node installation variables

NODE_IPV6	Node's IPV6
NODE_IPV4	Node's IPV4
PROVIDER_HOSTS	The IPs of the nodes that the system will retrieve its data (all nodes' ips)
NODE_HOSTNAME	Node's hostname/ip
STORM_NIMBUS_CONFIG_FILE	The path of Storm Nimbus configuration file



STORAGE_PLACEMENT	Enables and disables the placement algorithm of the storage. Default is False.
STORAGE_DATA_FOLDER	The folder that the data of the Storage component will be stored persistently

After the variables configuration the user needs just to execute the *docker-compose up* command.

```
$docker-compose up -d
```

The Analytics Stack includes the Apache Storm and Nimbus. Generally, the configuration of Nimbus needs no alteration. However, users can update the provided files from the aforementioned repositories accordingly. Furthermore, users can also add other configurations of Storm Framework (<https://storm.apache.org/releases/current/Configuration.html>). Finally, users can introduce other scheduling strategies (including RAINBOW's strategies) via the provided configuration file. For instance, if users set `storm.scheduler` equals to `ResourceAwareScheduler` and its strategy to be `EnergyAwareStrategy`, the execution will try to minimize the energy consumption. The following Figure 12 depicts a representative RAINBOW-enabled Nimbus configuration file.

```
storm.zookeeper.servers:
  - "cluster-head-IP" # update with master's IPv4
nimbus.seeds: [ "cluster-head-IP" ] # update with master's IPv4
storm.log.dir: "/logs"
storm.local.dir: "/data"
storm.local.hostname: "cluster-head-IP" # update with master's IPv4
supervisor.slots.ports:
  - 6700
  - 6701
  - 6702
  - 6703
nimbus.thrift.max_buffer_size: 20480000
supervisor.thrift.max_buffer_size: 20480000
topology.component.cpu.pcore.percent: 1000.0
topology.component.resources.onheap.memory.mb: 512.0
storm.scheduler: "org.apache.storm.scheduler.resource.ResourceAwareScheduler"
topology.scheduler.strategy: "eu.rainbowh2020.Schedulers.EnergyAwareStrategy"
```

Figure 12 Analytics Scheduler Configuration File

Name	Last commit	Last update
..		
storm	feat: update README.md and add analytic-stack	5 months ago
.env	introduce more detailed configurations	4 months ago
docker-compose-arm32.yaml	fixes on data mngm node compose files	1 month ago
docker-compose-arm64.yaml	fixes on data mngm node compose files	1 month ago
docker-compose.yaml	fixes on data mngm node compose files	1 month ago

Figure 13 Analytics Stack worker node installation files



The second step of that stage is the Analytics stack step on the worker nodes. The procedure is similar to the previous step but in that case the user needs to configure the variables of the `.env` file that are presented and described at Table 9.

Table 15 Analytic Stack worker node installation variables

MONITORING_CONFIGURATION_FILE	The path of monitoring agent configuration file
STORAGE_RAINBOW_HEAD	Cluster head's IPV4/IPV6
STORAGE_NODE_NAME	Node's hostname
STORAGE_PLACEMENT	Enables and disables the placement algorithm of the storage. Default is False.
STORAGE_DATA_FOLDER	The folder that the data of the Storage component will be stored persistently

Furthermore, users can (optionally) configure the parameters of the monitoring agent by providing its configuration file. By default, the RAINBOW monitoring agent captures all utilization metrics from the underlying node and the containerized services (as described in Deliverable D3.2). Through the configuration file, users can enable or disable specific metrics, and apply adaptive monitoring and dissemination techniques in order to minimize the monitoring metrics' size and the monitoring computational footprint. The following image highlights a representative configuration file of the monitoring configurations.

```
node_id: "node_id" # user need to provide a node id (e.g. hostname)

sensing-units:
  general-periodicity: 1s # general sensing rate
  DefaultMonitoring: # Node-level metrics
    periodicity: 1s
    disabled-groups: # metric-groups that the system will not enable
      - "disk"
  metric-groups: # override the sensing preferences on specific groups
    - name: "memory"
      periodicity: 15s # change a static periodicity
    - name: "cpu"
  UserDefinedMetrics: # sensing interface for user-defined metrics
    periodicity: 1s
    sources:
      - "/"
  ContainerMetrics: # Container-level monitoring metrics
    periodicity: 1s

dissemination-units: # users can enable multiple dissemination units
  IgniteExporter:
    hostname: ignite-server
    port: 50000

adaptivity: # optional adaptivity properties
  sensing: # adaptivity in sensing units
    DockerProbe: # e.g., enable adaptivity for the container metrics
      target_name: demo_test|cpu_ptc # target metric
      minimum_periodicity: 1
      maximum_periodicity: 15
      confidence: 0.95
  dissemination: # adaptivity in dissemination
    all: # the system sends adaptively all metrics to the storage
      minimum_periodicity: 1
      maximum_periodicity: 15
      confidence: 0.95
    metric_id: # or it can send adaptively only specific metrics
      - minimum_periodicity: 5s
      - maximum_periodicity: 35s
      - confidence: 95
```

Figure 14 Monitoring configuration file



After the variables configuration the user needs just to execute the *docker-compose up* command on each one of the cluster's worker nodes. It is necessary to mention that, as depicted in Figure 15, there are different docker-compose files in order to support the different architectures of the worker nodes. So, for x86 based CPU architecture, the user needs to execute.

```
$docker-compose up -d
```

For 32-bit ARM based CPU architecture, the user needs to execute.

```
$docker-compose up -f docker-compose-arm32.yaml -d
```

For 64-bit ARM based CPU architecture, the user needs to execute.

```
$docker-compose up -f docker-compose-arm64.yaml -d
```














Name	Last commit	Last update
..		
 .env	feat: rainbow metric exporter	2 months ago
 README.md	feat: dashboard script fixes	4 months ago
 cjdns.txt	feat: add dashboard	5 months ago
 docker-compose.yaml	feat: rainbow metric exporter	2 months ago
 init-02-cjdns.sh	feat: add dashboard	5 months ago
 init-03-cjdns-ipv6.py	feat: dashboard script fixes	4 months ago
 init-03-cjdns-worker.py	feat: add dashboard	5 months ago
 init-04-configure-host.py	feat: add dashboard	5 months ago
 init-05-docker-debian.sh	feat: add dashboard	5 months ago
 init-05-docker-ubuntu.sh	feat: add dashboard	5 months ago
 init-06-docker-configure.sh	feat: add dashboard	5 months ago
 init-13-docker-compose.sh	feat: add dashboard	5 months ago
 rainbow-dashboard.sh	feat: dashboard script fixes	4 months ago

Figure 15 Dashboard installation scripts

The third and final stage of the RAINBOW installation procedure is the Dashboard component setup. The first step includes the configuration and execution of the *rainbow-dashboard.sh* script which is shown in Figure 15. The user needs to configure the script with the values that are printed after the successful execution of the master's node script (first step of the first stage). To achieve this the user needs to open the *rainbow-*



dashboard.sh script and set the corresponding variables at the beginning of the script and then just execute it.

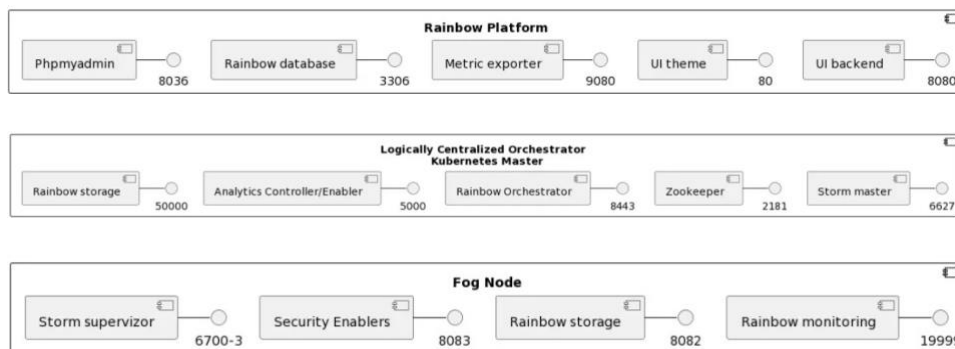
```
$sudo ./rainbow-dashboard.sh
```

The final step of that stage and of the whole installation procedure is the configuration of the *.env* file with just the values of the local IP address and a path to a folder for the persistent volume data. Then the user simply needs to execute the *docker-compose up* command.

```
$docker-compose up -d
```

Following the instructions, users are installing the three main parts for utilizing RAINBOW; a) the central part of the RAINBOW platform, b) the Logically Centralized Orchestrator and the Data Analytics in the Kubernetes Master of the Kubernetes cluster to be used, and c) the fog node.

Below we provide the deployment diagrams for the different parts of the platform.





5. RAINBOW Usage Guide

The Dashboard is the RAINBOW's user interface where the on boarding to the platform begins with the deployment and management of user applications through a user friendly and easy to use interface. The complete instructions of the user interface are available online and continuously updated in a ReadTheDocs page, at <https://rainbow-h2020.readthedocs.io>.

In this section, for the sake of clarity of the reader, we will present some of the basic parts of Dashboard usage that had already been presented in previous releases along with some of the core updates that were implemented on the final release.

The first step for a user is to login to the RAINBOW platform by providing her/his credentials, as depicted in Figure 16.

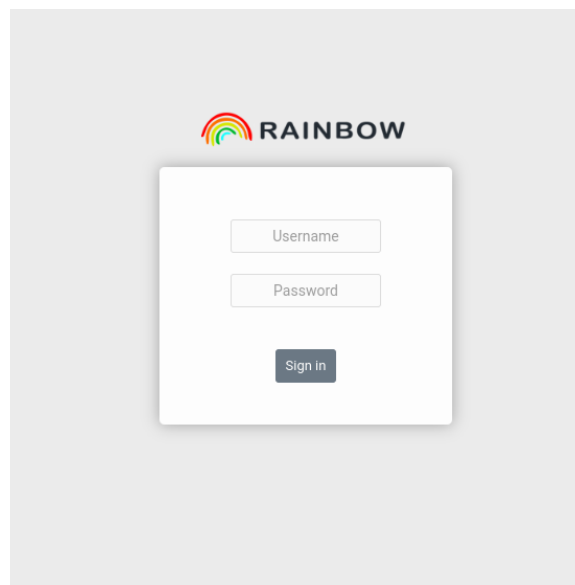


Figure 16 Login page of the RAINBOW Dashboard



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

The landing page of RAINBOW is the main Dashboard view where an overview of the available resources along with monitoring information about the deployed applications are presented, Figure 17.

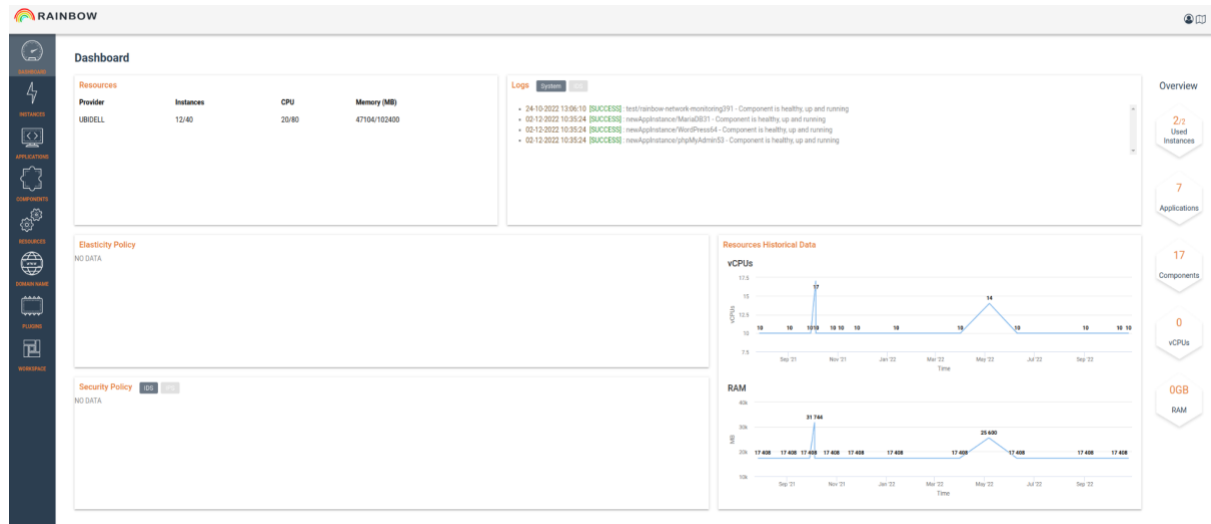


Figure 17 Main page of the RAINBOW Dashboard

The user is provided with a list of all the available components, as also she/he can create a new or edit an existing component by configuring the desired fields as shown in the indicative Figures 18 and 19 below.

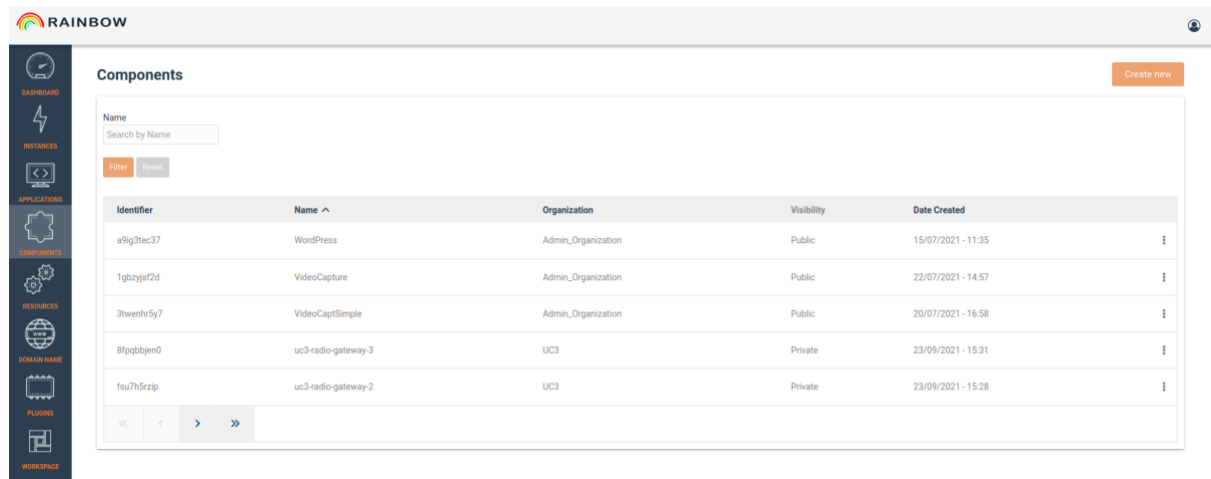


Figure 18 Components' list



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

Components > Create

Components | Create

General Distribution Parameters Minimum Execution Requirements Health Check Container Execution Environmental variables Exposed Interfaces Required Interfaces Plugins Volumes Devices Li

Distribution Parameters

Docker Image *

Type the docker image

Docker Credentials

☐ Use private Docker registry (Username, Password fields)

Docker Username

Type your username

Docker Password

Type your password

☐ Use custom docker registry

Custom Docker Registry

Type the docker registry

Test Connection

☐ (Public) If this option is checked, anyone could see this component

Save

Figure 19 Component configuration

After the component's definition, the user can graphically create the application topology through the Service Graph editor, as depicted in Figure 20.

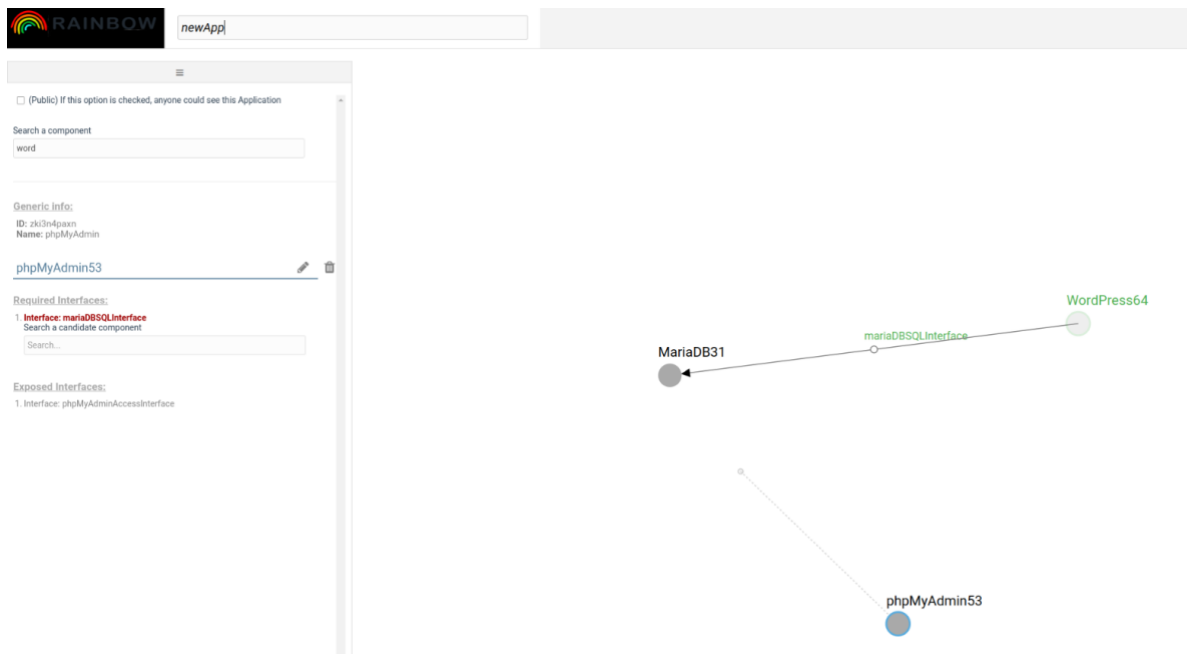


Figure 20 Application creation



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

All the created applications can be viewed as a list on the corresponding view (Figure 21).

Identifier	Name	Organization	Visibility	Date Created
h0wdnw63va	CMSApp	Admin_Organization	Public	15/07/2021 - 11:35
6chex13gnc	DBMS	Admin_Organization	Public	15/07/2021 - 11:35
mzfdzfb4p	FunctionPilot	Admin_Organization	Public	15/07/2021 - 11:35
4jic77jip	HRC-1	UC1	Private	27/09/2021 - 16:44
znf6eeh9sf	HRC_2	UC1	Private	28/09/2021 - 16:42

Figure 21 Applications' list

For deploying an application, a user must register the appropriate resources, as depicted in Figure 22.

Name *

Resource Type *

Configure Connection

☒ Would you like to change the provider's credentials?

Master Url *

Certificate Authority Data *

Client Certificate Data *

Client Key Data *

☒ Network Mode Host - If enabled all components will be at Network Mode Host (for ipv6 providers)

Test Connection

Regions

Or add a new one

Name *

Save

Figure 22 Resource creation



To deploy an application, a user must create an instance from the available applications as shown in the Figure 23 below.

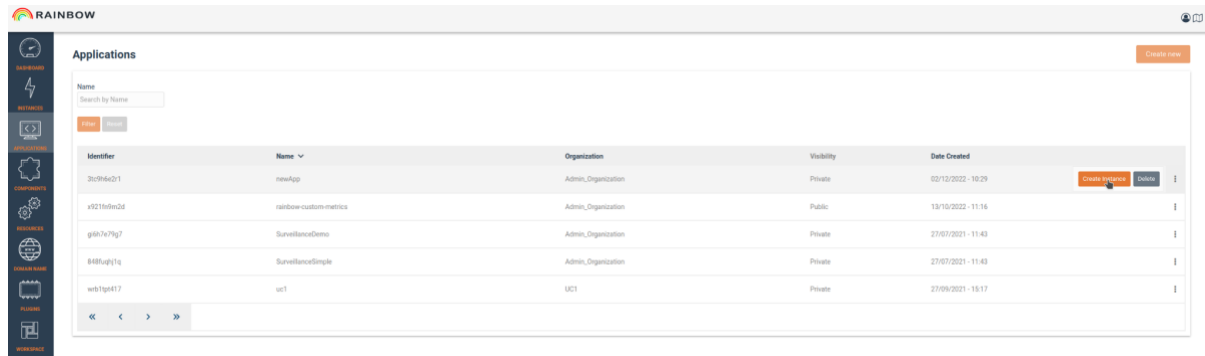


Figure 23 Application Instance creation

In the Application Instance service graph editor, a user can select the desired resource and edit the components configuration as depicted in the following Figures 24 and 25.

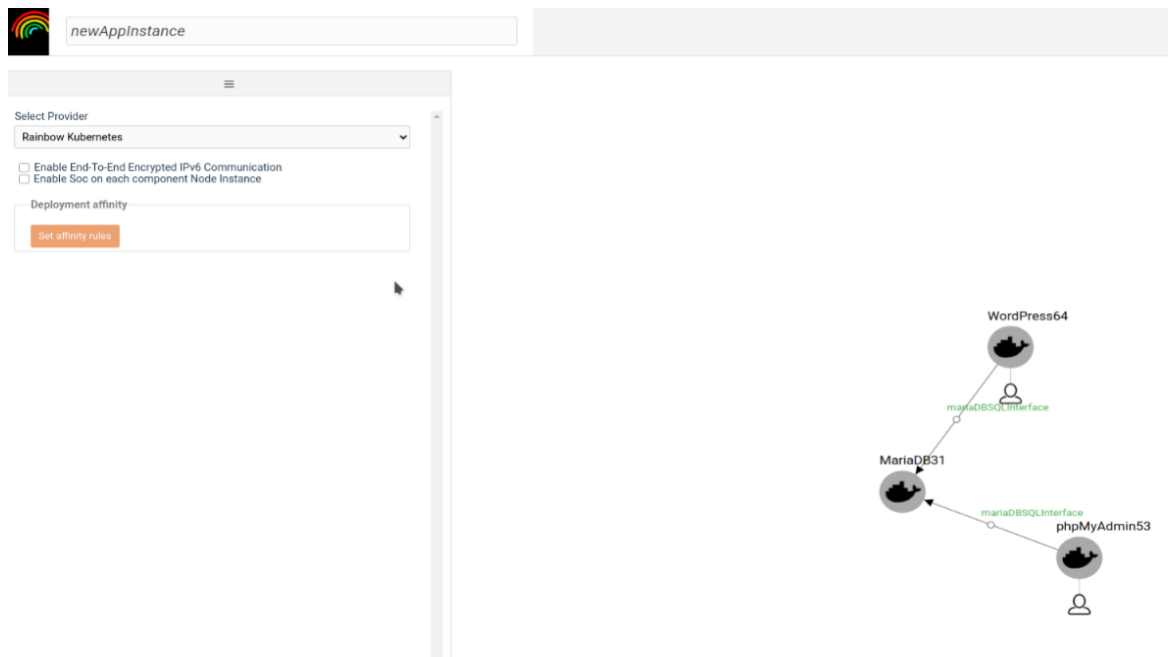


Figure 24 Application Instance service graph editor



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

Configure "WordPress" Component
ID: 2q38qa19dg

Key	Value
WORDPRESS_DB_PASSWORD	wordpress
WORDPRESS_DB_HOST	@MariaDB
WORDPRESS_DB_USER	wordpress

Save

Figure 25 Application Instance service graph editor - Component editing

A new feature in the final release of RAINBOW is the ability to set affinity and anti-affinity rules for the application components through the service graph editor (Figure 26 and Figure 27)

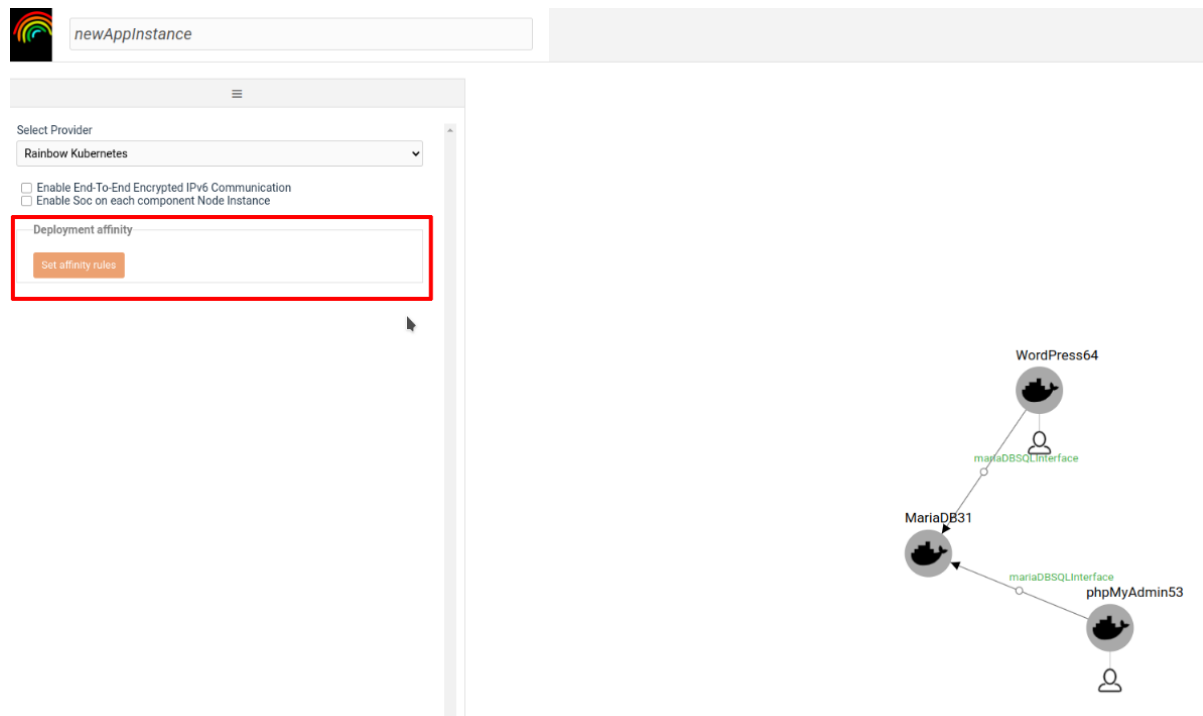


Figure 26 Application Instance service graph editor - affinity/anti-affinity



Set the Pod Affinity

New Affinity

Type
☒ Affinity ☐ Anti-affinity

Target component
MariaDB31

Related components
phpMyAdmin53

Save

No affinities available.

Figure 27 Application Instance service graph editor - set affinity rules

After the successful deployment of an application, it can be found at the list of the Application Instances page (Figure 28).

RAINBOW

Application Instances

Name: Search by Name Status: - Select - Application: - Select -

Cancel Reset

Identifier	Name	Application Name (Hex ID)	Status	Date Created
ghc52w4c0b	newAppInstance	newApp (2b094e2f1)	DEPLOYED	02/12/2022 - 19:30
lzz6g9th9	test	rainbow-custom-metrics (5f219d9e2d)	DEPLOYED	24/10/2022 - 13:05

Figure 28 Application Instances list



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

Correspondingly, after the deployment of an application the user can monitor it through the live service graph by viewing logs and metrics, as shown in Figure 29 and Figure 30 below.

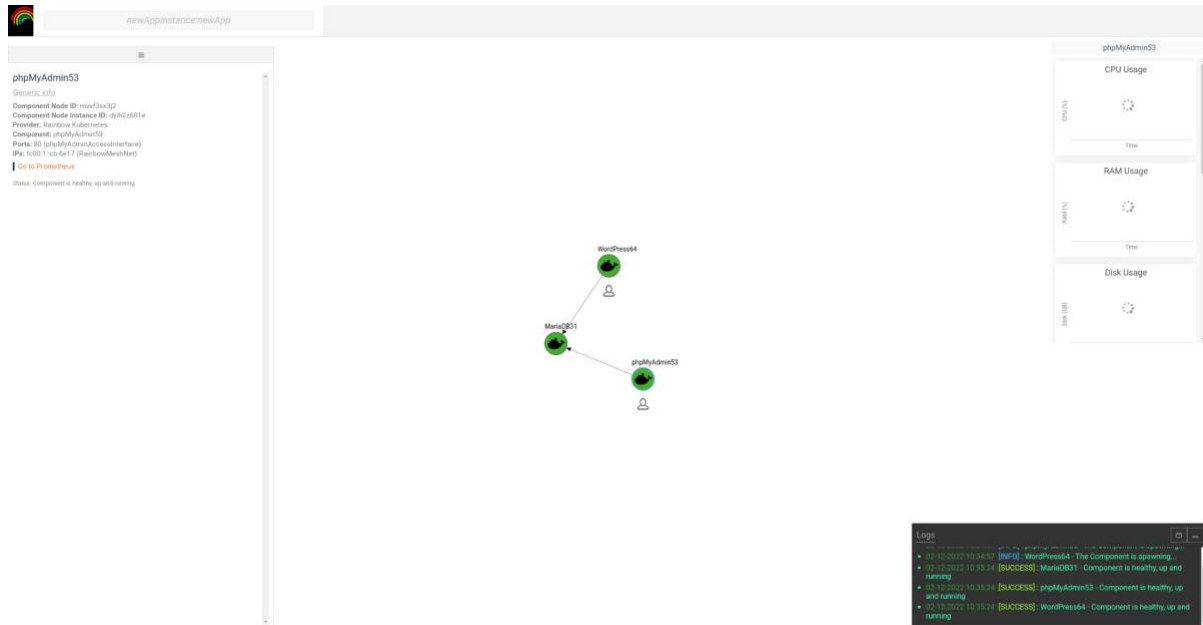


Figure 29 Service graph monitoring - part 1

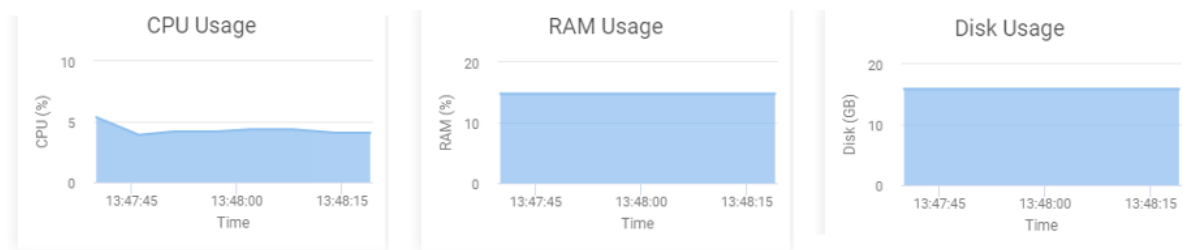


Figure 30 Service graph monitoring - part 2



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

Likewise, a user can define analytics and elasticity policies on a successful deployed application by using the dedicated editors (Figure 31).

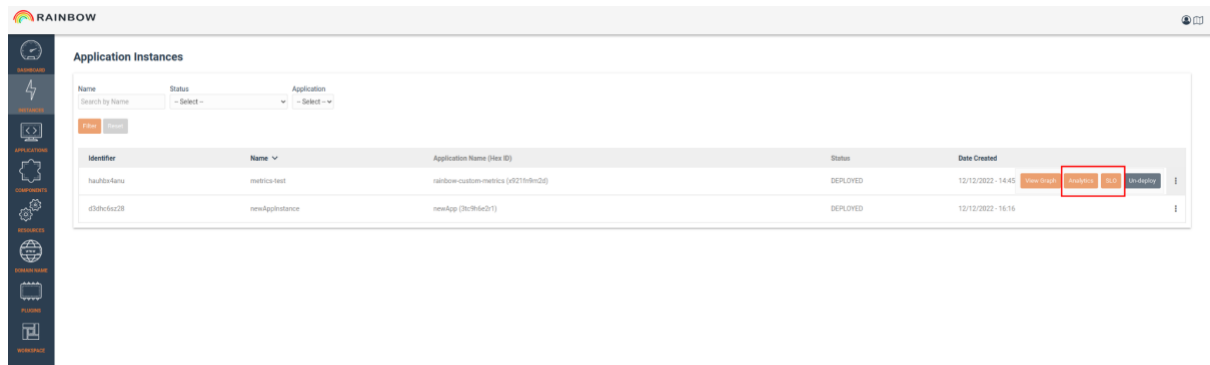


Figure 31 Analytics and SLO Editor

A user is able to create the desired analytics by using the editor, as shown in Figure 32 and Figure 33 below.

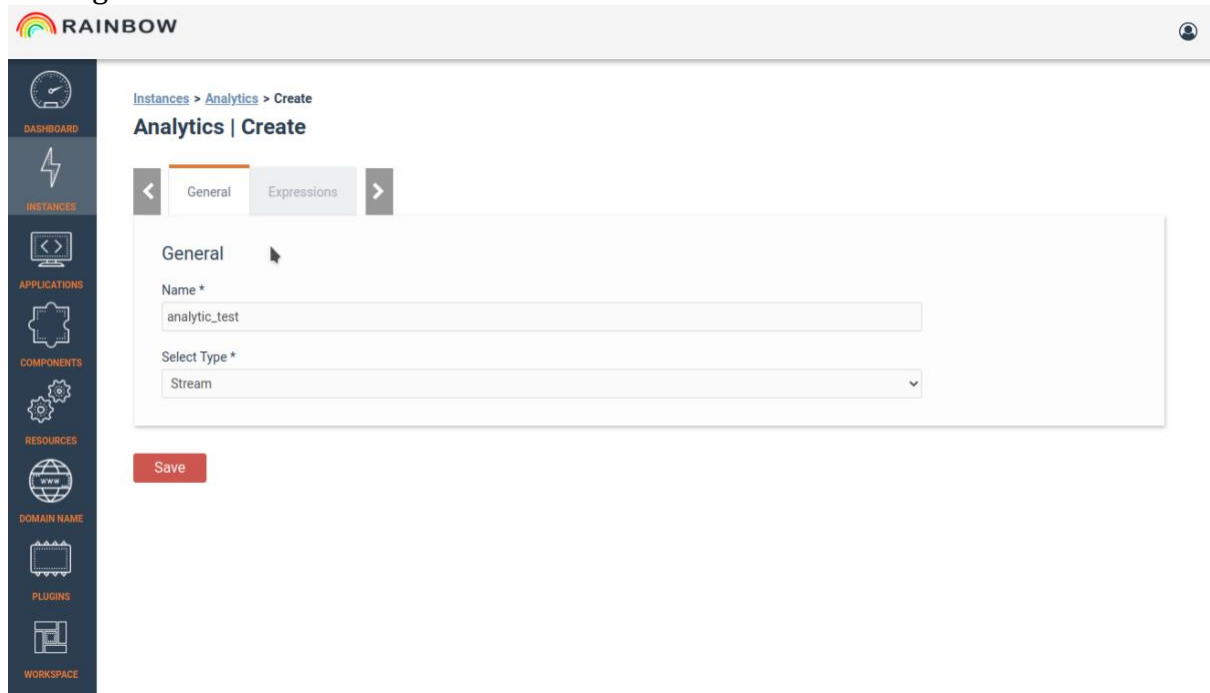


Figure 32 Creation of a new analytic

This screenshot shows the "Analytics | Create" interface in the RAINBOW dashboard. The "Expressions" tab is active, displaying fields for "Period (Seconds)" (set to 20), "Window (Seconds)" (set to 0), and "Select Function" (a dropdown menu). Below these, a red error message states "At least 1 expression is required". The "Metric Type" section has radio buttons for "Component" (selected) and "Custom". Under "Component", there are dropdowns for "Select Component" (set to "Select"), "Select Metric" (set to "Select"), and "Select Operator" (set to "Select"). A "Save" button is located at the bottom left of the form area.

Figure 33 Adding expressions on the analytic

SLOs are added through the SLO editor and depicted in Figure 34 to 37. In the Figure 34 user provide the basic info of an SLO, such as the component it should be applied or the strategy to be used.

This screenshot shows the "SLO | Create" interface in the RAINBOW dashboard, specifically the "General" tab. The form includes fields for "Name" (filled with "mySlo"), "Select Elasticity Strategy" (filled with "VerticalElasticityStrategy"), and "Select Target Component" (filled with "rainbow-network-monitoring391"). A "Save" button is positioned at the bottom left of the form.

Figure 34 Creation of a new SLO

In Figure 35 the user specifies the metrics to be used.



RAINBOW

Instances > SLO > Create

SLO | Create

General Metrics Computations Expressions

Metrics

At least 1 metric is required

Metric Type *
☒ Component ☐ Custom

Name *
cpu

Select Source Component *
rainbow-network-monitoring391

Select Metric *
cpu.pct (cpu percent utilization of the container) (%)

Select Function *
No Function

Window Time *
10

Insert New Metric

Save

Figure 35 Add metrics in the SLO

Then the computations of the SLO are defined, as seen in Figure 36.

RAINBOW

Instances > SLO > Create

SLO | Create

General Metrics Computations Expressions

Computations

At least 1 computation is required

Computation Name *
my_cpu_computation

Compute Every (Seconds) *
15

Type *
Metric

Metric *
cpu

Operand *
+

Add

Insert New Computation Remove Computation

Save

Figure 36 Add Computations to the SLO

Finally, the expressions tab is used to define the target values for the SLO, as seen in Figure 37



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

RAINBOW

Instances > SLO > Create

SLO | Create

General Metrics Computations Expressions

Expressions

At least 1 computation is required

Computation *

my_cpu_computation

Target Value *

50

Tolerance *

5

☒ Higher is better

Insert OR Statement

Insert AND Statement

Remove Expression

Save

Figure 37 Add Expressions to the SLO

After the creation of the policy, the defined elasticity strategy will be applied when the SLO is violated.



6. Technical Evaluation and Quality Assurance

This section summarises the work performed resulting to the RAINBOW's final release, in terms of the development and integration process followed, the software quality assessment process, and the testing procedures.

6.1. Continuous Integration and Quality Assurance

In previous versions of this deliverable, we presented the methodology and activities performed by the integration team in relation to Continuous Integration and Quality Assurance of the developed platform. This section provides a summary of this process, focusing on the tools used during this time frame to enable the CI part of RAINBOW.

6.1.1. Version Control System – Gitlab

RAINBOW has used Gitlab as the primary VCS system. The Gitlab group that has been created and hosts all components' repositories is depicted in Figure 38.

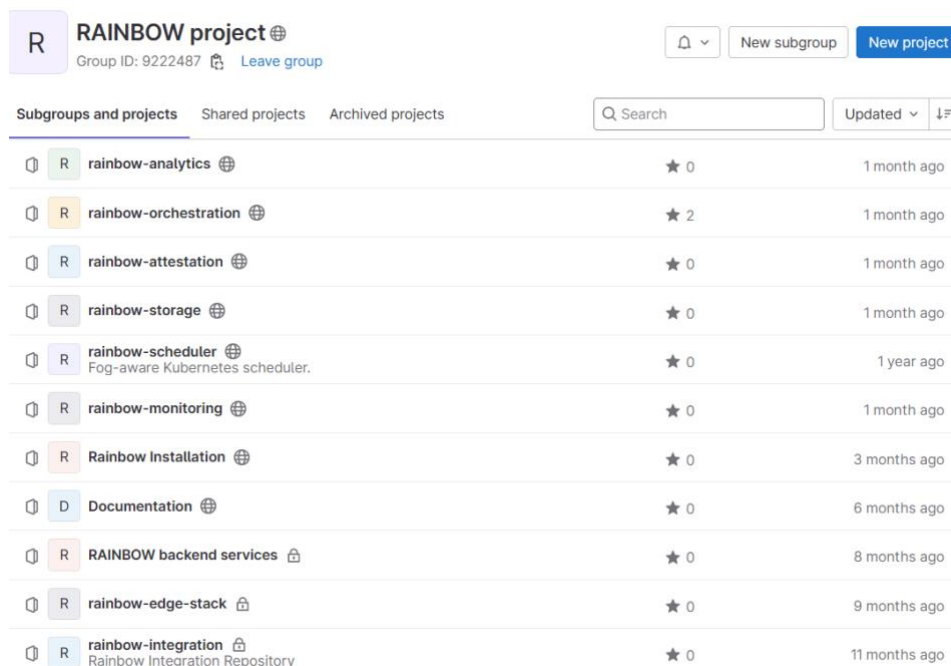


Figure 38 RAINBOW's Gitlab group and repositories

It is worth noting that based on the annotation of RAINBOW components as open source or not in the IPR repository, the corresponding sub-groups were made available to the public considering the corresponding license, as shown in the figure above.

6.1.2 Container Registry

Concerning the distribution of components, RAINBOW continues to use the same approach by using docker registry. This registry is hosted in the project's GitLab group, as depicted in the figure below. Since the last version three more images were added related to orchestration and network monitoring.



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

RAINBOW project > Container Registry

Container Registry

16 Image repositories

Updated ▾

rainbow-integration/opcua-message-estrat-controller	2 Tags	<input type="button" value="🗑"/>
rainbow-integration/rainbow-network-monitoring	1 Tag	<input type="button" value="🗑"/>
rainbow-integration/storm	2 Tags	<input type="button" value="🗑"/>
rainbow-integration/migration-estrat-controller	2 Tags	<input type="button" value="🗑"/>
rainbow-integration/image-throughput-slo-controller	2 Tags	<input type="button" value="🗑"/>
rainbow-integration/rainbow-orchestrator	4 Tags	<input type="button" value="🗑"/>
rainbow-integration/rainbow-storage	13 Tags	<input type="button" value="🗑"/>
rainbow-integration/arm64v8/redis	3 Tags	<input type="button" value="🗑"/>
rainbow-integration/custom-stream-sight-slo-controller	1 Tag	<input type="button" value="🗑"/>
rainbow-integration/rainbow-scheduler	2 Tags	<input type="button" value="🗑"/>

Figure 39 RAINBOW container images

6.1.3 Issue Tracking – Gitlab

RAINBOW project continued to use GitLab Issues for issue/bug tracking toolset. The GitLab issues of the RAINBOW Project are located at <https://gitlab.com/groups/rainbow-project1/-/issues> (see Figure 40). To this point, all of 138 issues raised have been closed. Last but not least, open access to issues is provided to the public for components that have been commented as such.



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU

RAINBOW project > Issues

Open 0	Closed 138	All 138	Select project to create issue
Search or filter results...			
Closed date			
<div>StreamSight Storm Connector</div> <div>rainbow-project1/rainbow-analytics#1 · created 2 years ago by dtrihinas</div> <div>CLOSED 0 closed 1 month ago</div>			
<div>StreamSight Ignite Data Ingestion</div> <div>rainbow-project1/rainbow-analytics#2 · created 2 years ago by dtrihinas</div> <div>CLOSED 0 closed 1 month ago</div>			
<div>StreamSight Scheduling</div> <div>rainbow-project1/rainbow-analytics#3 · created 2 years ago by dtrihinas</div> <div>CLOSED 0 closed 1 month ago</div>			
<div>[RAINBOW-80] Publish Message Elasticity Strategy</div> <div>rainbow-project1/rainbow-orchestration#103 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 1 0 closed 1 month ago</div>			
<div>[RAINBOW-17] Update NetworkLinks with monitoring data</div> <div>rainbow-project1/rainbow-orchestration#51 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 0 closed 1 month ago</div>			
<div>[RAINBOW-20] Network QoS SLO & Elasticity Strategy</div> <div>rainbow-project1/rainbow-orchestration#54 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 0 closed 2 months ago</div>			
<div>[RAINBOW-90] One instance per Node Type SLO (UC3)</div> <div>rainbow-project1/rainbow-orchestration#109 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 0 closed 2 months ago</div>			
<div>[RAINBOW-101] Final SLOs and fine tuning for UC1</div> <div>rainbow-project1/rainbow-orchestration#114 · created 11 months ago by Thomas Pusztai</div> <div>CLOSED 0 closed 2 months ago</div>			
<div>[RAINBOW-100] Event Detection and Power Consumption SLO - final version (UC2)</div> <div>rainbow-project1/rainbow-orchestration#113 · created 11 months ago by Thomas Pusztai</div> <div>CLOSED 0 closed 2 months ago</div>			
<div>[RAINBOW-5] NetworkLink CRD Admission Webhook</div> <div>rainbow-project1/rainbow-orchestration#39 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 1 closed 2 months ago</div>			
<div>[RAINBOW-31] Service Graph Admission Webhook (pre deployment constraints solver)</div> <div>rainbow-project1/rainbow-orchestration#65 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 1 closed 2 months ago</div>			
<div>[RAINBOW-94] NetworkQoS Migration Elasticity Strategy for moving pods to better connected nodes</div> <div>rainbow-project1/rainbow-orchestration#112 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 0 closed 3 months ago</div>			
<div>[RAINBOW-59] SLOC Runtime Unit Tests</div> <div>rainbow-project1/rainbow-orchestration#93 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 1 closed 3 months ago</div>			
<div>[RAINBOW-65] Service Graph Unit Tests</div> <div>rainbow-project1/rainbow-orchestration#97 · created 1 year ago by Thomas Pusztai</div> <div>CLOSED 1 closed 3 months ago</div>			

Figure 40 RAINBOW issues

6.1.4 Software Quality Evaluation

SonarQube was mainly used for the software quality evaluation part of the platform, as part of the CI process of the project while partners for privacy reasons were allowed to host and maintain their own code quality tool setup and make regular quality evaluation over the code. The latest results are presented below.

6.1.5 Continuous Integration Flow

For RAINBOW project, a dedicated Kubernetes group runner has been utilized in order to handle all the CI/CD jobs of the project. The stages included in RAINBOW were (i) build, (ii) package, (iii) analyze and (iv) prepare while the necessary images were used appropriately in each stage. Figure 41 below depicts the CI flow of Rainbow UI backend that involves all the aforementioned stages, with each stage containing a job. The detailed



approach that was relevant till the end of the project can be found in deliverables D5.1, 5.2 and 5.3.

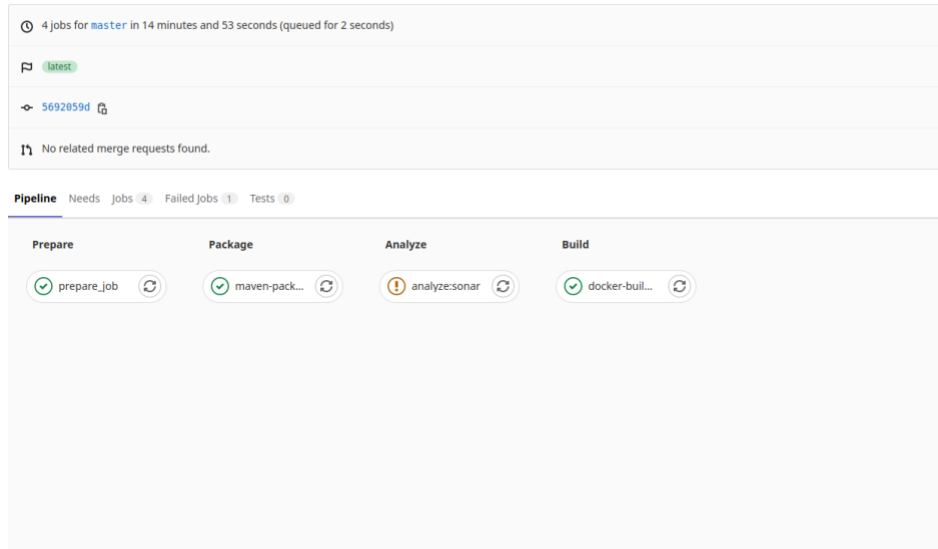


Figure 41 RAINBOW's CI flow

6.2. Testing Procedures of the RAINBOW Final Release

For the development and integration of the final release of RAINBOW, the consortium To deliver a stable final version of RAINBOW, the consortium used Unit tests and integration tests that ensure the proper functioning of the integrated platform. In this section, we collect the results related to unit and integration testing as performed for this final stage.

6.3. 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.

The results of the tests are provided in the Annex I: Unit Tests for Final Release.

6.4. Integration Testing

Integration testing has the purpose of assuring the proper functionality of the interconnected components. For the previous releases of the integrated platform, we utilized manually executed tests and collected the results to ensure the proper functionality. These tests covered in D6.3 included the following:

- Testing of Service Graph Deployment (*IT_01*).
- Testing of the scheduling of pods to be deployed on nodes (*IT_02*).
- Testing of SLO controller is able to check the status of the deployment and trigger actions in case of any violation (*IT_03*).



- Testing the proper counteractions are provided by Elasticity Strategy Controller and applied to the deployment (*IT_04*).

While these tests are still successfully executed, for this final release have also used the ReadyAPI platform¹ that allows the creation and execution of tests based though the usage of the RAINBOW components' REST calls. The REST calls tested usually refer to functionalities from a single component, but we combine the calls as part of a testing single scenario, thus ensuring the proper function of the integrated components.

Also, as OpenAPI standard has been used in for documenting the APIS of RAINBOW components, we able to use the available API calls in ReadyAPI allowing the faster creation of the tests.

For creating tests, we use the calls we need to test as part of a scenario, provide the required input for each of them, as depicted in Figure 42.

Test Step Create

Method: POST Endpoint: http://212.101.173.131:8080/ Resource: /api/v1/provider

Request: Request Raw Outline Form

View request parameters in a table
Use the Projects tool to add request parameters

Media Type: application/json [X] Post QueryString

```
{
  "name": "Rainbow-test-cluster",
  "endpoint": "https://1fc07:d4dc:d430:2029:9c30:45ba:fd96:b3451:6443",
  "username": "LS0tLS1CRUdJTiBDRVJUSUZJODFURSOtLS0tCk1JSUMExND0MhZD0F9SUJBZ0tJC0URBTKJna3Foa2tHOXcvOklFRc0ZBREWFVJN0VPMURWUWFERX0xOw0RKSXSwKY201bGR",
  "password": "",
  "publicKey": "LS0tLS1CRUdJTiBDRVJUSUZJODFURSOtLS0tCk1JSURJVEN0QWdtZ0F9SUJBZ0tJC0URBTKJna3Foa2tHOXcvOklFRc0ZBREWFVJN0VPMURWUWFERX0xOw0RKSXSwKY201bGR",
  "privateKey": "LS0tLS1CRUdJTiBDRVJUSUZJODFURSOtLS0tCk1JSUMExND0MhZD0F9SUJBZ0tJC0URBTKJna3Foa2tHOXcvOklFRc0ZBREWFVJN0VPMURWUWFERX0xOw0RKSXSwKY201bGR",
  "domain": "",
  "project": "",
  "meshIdentifier": "",
  "providerCredentialsChange": false,
  "networkModelHost": false,
  "providerType": {
    "id": "B",
    "name": "Rainbow Kubernetes"
  },
  "proxy": "",
  "imageID": "",
  "networkID": "",
  "defaultProvider": false,
  "regions": [
    {
      "name": "gr-athens"
    }
  ]
}
```

Auth (inherit From Parent) Attachments (0) Headers (1)

Assertions Log [1]

PASS Valid HTTP Status Codes

Figure 42 Defining request parameters of a REST call

¹ <https://smartbear.com/product/ready-api/overview/>



For creation of a test, it is important to define the assertions that are used to check if the test has been successfully executed, as depicted in Figure 43.

The screenshot displays the 'Test Step Deployment Request' interface. The 'Request' tab is active, showing a POST method to the endpoint 'http://212.101.173.131:8080/'. The 'Resource' field contains '/api/v1/applicationinstance/{applicationinstanceID}/request/deployment'. The 'Parameters' field is empty. The 'Request' body is a JSON object with the following structure:

```
{  "applicationinstanceID": "${#TestCase#APPLICATION_INSTANCE_ID}"}
```

The 'Response' tab is also active, showing a detailed JSON response. The response structure is as follows:

```
{  "code": 23,  "message": "Application in",  "returnobject": {    "applicationinstanceID": 116,    "hexID": "rvdtn92sqk",    "name": "rainbowTestA",    "description": "",    "overlay": false,    "deploymentTimestamp": "",    "application": {      "id": 54,      "hexID": "sdteyims0h",      "name": "myCms",      "publicApplication": true,      "componentNodes": [        {          "componentNodeID": 169,          "hexID": "ycwirz9eyj",          "name": "phpMyAdmin4",          "component": {            "id": 164,            "name": "phpMyAdmin",            "hexID": "jmvcp8bmf",            "publicComponent": true,            "dateCreated": "",            "lastModified": "",            "elasticityController": "",            "exposedInterfaces": [              {                "interfaceID": 173,                "name": "phpMyAdminA",                "port": 80              }            ]          }        }      ]    }  }  }
```

The 'Assertions' tab is at the bottom, showing a 'PASS' status and 'Valid HTTP Status Codes'.

Figure 43 Defining assertions based on the expected response of a REST call

With this approach a total of 4 complex scenarios have been defined, each of them consisting of multiple steps. The scenarios are depicted in Figure 44 below.

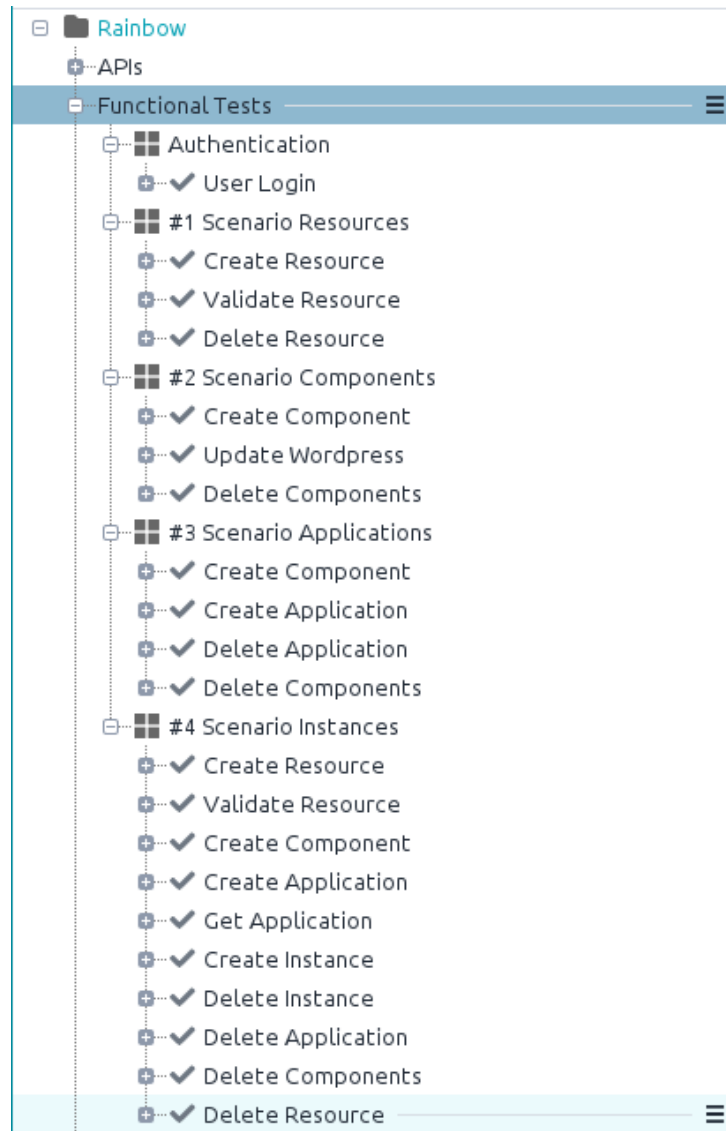


Figure 44 Overview of integration tests in ReadyAPI

With the test scenarios defined, the execution the whole test suite is an easy and replicable task, that we executed regularly to ensure the proper function of the platform.

The results are also provided below in Figure 45 and Figure 46.



Project No 871403 (RAINBOW)

5.4 RAINBOW Integrated Platform and Unified Dashboard - Final Release

Date: 30.01.2023

Dissemination Level: PU



Figure 45 Integration test results (part 1)



#4 Scenario Instances	
     	✓ PASS
+  Test cases: 10	
Create Resource	✓ PASS
Validate Resource	✓ PASS
Create Component	✓ PASS
Create Application	✓ PASS
Get Application	✓ PASS
Create Instance	✓ PASS
Delete Instance	✓ PASS
Delete Application	✓ PASS
Delete Components	✓ PASS
Delete Resource	✓ PASS

Figure 46 Integration test results (part 2)



7. Conclusions

This deliverable summarized the efforts made under WP5 to deliver the final version of an open RAINBOW platform built to enable the management of scalable, diverse, and secure IoT services and cross-cloud applications, thereby enabling industries, around manufacturing, transportation and critical infrastructure to reap the benefits associated with RAINBOW's advanced technologies. This version is considered mature as the final release is a platform of a TRL 7.

Part of this document described the process and work performed concerning the platform evolution from the second to its current version based on a series of improvements implemented in various areas of the platform but mainly in the frontend stack technologies. This came as a result of the feedback collected and lessons learnt from the demos second round realised in WP6.

This document also acts as a handbook for technical information related to the integration of different components depicted in the reference architecture and developed in the technical Work Packages, so as to support any interested partner in the further evolution of the platform. In the same way, RAINBOW prepared and presented in this document installation instructions that allow a seamless installation and use of the platform to a local environment, including the underlying requirements, deployment and configuration of different elements of the platform.

Finally, the RAINBOW platform has been technically evaluated carrying out integration tests and scenarios, to ensure requirements are met using tools like ReadyAPI to test functionalities, security, scalability, and load aspects for the most important and computationally heavy parts of the platform inside RAINBOW's CICD pipeline.



References

- [1]. <https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.21.md#v1210>
- [2]. <https://golang.org/>
- [3]. <https://github.com/kubernetes-sigs/kubebuilder>
- [4]. Rainbow Container Registry: https://gitlab.com/rainbow-project1/rainbow-integration/container_registry
- [5]. Kubectl tool: <https://kubernetes.io/docs/tasks/tools/#kubectl>
- [6]. <https://kubernetes.io/docs/concepts/scheduling-eviction/schedulingframework/>
- [7]. <https://slocloud.github.io>
- [8]. <https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/>
- [9]. <https://www.optaplanner.org/>
- [10]. https://gitlab.com/groups/rainbow-project1/-/container_registries
- [11]. D1.2: RAINBOW Reference Architecture
- [12]. D2.2: RAINBOW Collective Attestation Policy Enablers Design
- [13]. D2.3: RAINBOW Collective Attestation & Runtime Verification - Version 1
- [14]. D2.6: RAINBOW Secure Overlay Mesh Network - Final Version
- [15]. D3.2: RAINBOW Orchestration Mechanisms
- [16]. D4.2: Data Management Services
- [17]. D5.1: Technical Integration and Testing Plan
- [18]. D5.2: RAINBOW Integrated Platform and Unified Dashboard - Early Release
- [19]. D5.3: RAINBOW Integrated Platform and Unified Dashboard - Second Release
- [20]. D6.3: RAINBOW Human-Robot Collaboration Demonstrator - Final Demonstrator
- [21]. D6.5: RAINBOW Digital Transformation of Urban Mobility - Final Demonstrator
- [22]. D6.7: RAINBOW Power Line Surveillance Demonstrator - Final Demonstrator



Annex I: Unit Tests for Final Release

Name	<i>Scale out/in through the Horizontal Elasticity Strategy</i>
Description	<i>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.</i>
Reference Code	<i>UT_05</i>
Responsibilities	<i>Implementation: TUW</i>
Component	<i>Orchestration Lifecycle Manager – Application Lifecycle Manager</i>
Input	<i>Horizontal Elasticity Strategy CRD instance</i>
Output	<i>Updated Scale sub-resource of the deployment object</i>
Status	<i>Implemented with Jest</i>

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

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

Name	<i>SLO Control Loop</i>
-------------	-------------------------



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

Name	<i>Watch Manager</i>
Description	<i>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.</i>
Reference Code	<i>UT_09</i>
Responsibilities	<i>Implementation: TUW</i>
Component	<i>Orchestration Lifecycle Manager – SLO Policy Managers</i>
Input	<i>SLO Mapping Type and respective CRD instances</i>
Output	<i>Method calls to the registered event handlers</i>
Status	<i>Implemented with Jest</i>

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

Name	<i>Custom StreamSight SLO Controller</i>
Description	<i>This test triggers evaluations the Custom StreamSight SLO, with input causing it to report SLO fulfilment, SLO violation with more resources needed, and SLO violation with fewer resources needed.</i>
Reference Code	<i>UT_11</i>



Responsibilities	<i>Implementation: TUW</i>
Component	<i>Orchestration Lifecycle Manager – SLO Policy Managers</i>
Input	<i>SLO Mapping and CPU monitoring data</i>
Output	<i>A Horizontal Elasticity Strategy CRD that reflects the compliance state of the SLO</i>
Status	<i>Implemented with Jest</i>

Name	<i>Service Graph Validation</i>
Description	<i>This test submits a valid and an invalid Service Graph (containing a loop) to the orchestrator to ensure that only the valid Service Graph is admitted.</i>
Reference Code	<i>UT_43</i>
Responsibilities	<i>Implementation: TUW</i>
Component	<i>Pre-deployment Constraint Solver</i>
Input	<i>Service Graph (YAML)</i>
Output	<i>Admission of Service Graph success or error response code</i>
Status	<i>Implemented with the Go testing package</i>

Name	<i>NetworkQoS Scheduler Score Plugin</i>
Description	<i>This test triggers the NetworkQoS Score plugin of the RAINBOW Kubernetes scheduler and ensures that cluster nodes that have lower latency and bandwidth variances receive a better score.</i>
Reference Code	<i>UT_44</i>
Responsibilities	<i>Implementation: TUW</i>
Component	<i>Orchestration Lifecycle Manager – Scheduler</i>
Input	<i>Set of cluster nodes, the cluster topology graph, the pod to be scheduled, and a Service Graph</i>
Output	<i>List of scores for nodes</i>
Status	<i>Implemented with the Go testing package</i>

Name	<i>AtomicDeployment Scheduler Permit Plugin</i>
Description	<i>This test triggers the AtomicDeployment Permit plugin of the RAINBOW Kubernetes scheduler and ensures that all pods of a service graph are admitted all at once or not at all.</i>
Reference Code	<i>UT_45</i>
Responsibilities	<i>Implementation: TUW</i>
Component	<i>Orchestration Lifecycle Manager – Scheduler</i>
Input	<i>Set of cluster nodes, the cluster topology graph, the pods to be scheduled, and a Service Graph</i>
Output	<i>-</i>
Status	<i>Implemented with the Go testing package</i>



Name	<i>Vertical Elasticity Strategy</i>
Description	<i>This test triggers the Vertical Elasticity Strategy in both directions, i.e., scale up and scale down, and ensures that the deployment resources are updated correctly.</i>
Reference Code	<i>UT_46</i>
Responsibilities	<i>Implementation: TUW</i>
Component	<i>Orchestration Lifecycle Manager – Application Lifecycle Managers</i>
Input	<i>Vertical Elasticity Strategy CRD instance</i>
Output	<i>Changes to the Kubernetes deployment objects</i>
Status	<i>Implemented with Jest</i>

Name	<i>Migration Elasticity Strategy</i>
Description	<i>This test triggers the Migration Elasticity Strategy in both directions, i.e., from base location to alternate location and back again and ensures that the deployments are updated correctly.</i>
Reference Code	<i>UT_47</i>
Responsibilities	<i>Implementation: TUW</i>
Component	<i>Orchestration Lifecycle Manager – Application Lifecycle Managers</i>
Input	<i>Migration Elasticity Strategy CRD instance</i>
Output	<i>Changes to the Kubernetes deployment objects</i>
Status	<i>Implemented with Jest</i>