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

D5.1 – Technical Integration and Testing Plan

Work Package	WP5 – Continuous Integration and Accessability
Lead Author (Org)	Orfeas Panagou, Alex Bensenousi (INTRASOFT)
Contributing Author(s) (Org)	Thodoris Toliopoulos (AUTH), Thomas Pusztai (TUW), Moysis Symeonides, Dimitris Trihinas (UCY), Panagiotis Gouvas, Konstantinos Theodosiou (UBI), Sotiris Kousouris (SUITE5), Athanasios Giannetsos (DTU)
Due Date	31.12.2020
Actual Date of Submission	31.12.2020
Version	1.0

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	15.10.2020	Orfeas Panagou, Alex Bensenousi (INTRASOFT)	Initial ToC
0.2	27.10.2020	Orfeas Panagou (INTRASOFT)	Content on section 4
0.3	05.11.2020	Orfeas Panagou (INTRASOFT)	Addition of Implementation Aspects
0.4	18.11.2020	Thodoris Toliopoulos (AUTH), Thomas Pusztai (TUW), Moysis Symeonides (UCY), Konstantinos Theodosiou (UBI)	Integration Points for Section 3
0.5	27.11.2020	Konstantinos Theodosiou(UBI), Sotiris Kousouris (SUITE5), Athanasios Giannetsos (DTU)	Integration Points for Section 3
0.6	02.12.2020	Orfeas Panagou (INTRASOFT)	Additional content in section 1, 4, 5
0.7	12.12.2020	Orfeas Panagou (INTRASOFT)	Added conclusions, missing sections, table of figures & tables
0.8	18.12.2020	Dimitris Trihinas(UCY)	Internal Review
0.9	22.12.2020	Orfeas Panagou, Alex Bensenousi (INTRASOFT)	Address, review comments, additional content, better structure
0.95	23.12.2020	Panagiotis Gouvas,Konstantinos Theodosiou (UBI)	Second Internal Review
1.0	24.12.2020	Orfeas Panagou , Alex Bensenousi (INTRASOFT)	Final Version

Table of Contents

List of tables

List of figures

Executive Summary

The aim of this deliverable is to provide a comprehensive overview and documentation report for RAINBOW's Integration and Testing plan. This plan will be used to guide the development, testing and integration effort that will culminate in the future releases of the RAINBOW Framework, which also falls in the scope of the activities of Work Package 5 (WP5).

Having the architecture proposed for the RAINBOW framework as a base reference, this document presents the interactions among the individual software components of the system, describing their interfaces and how they will be integrated to work as a whole.

A guideline is defined for the adequate development of the RAINBOW software components, which follows a microservices approach. It includes an integration and testing plan, so each of the components should go through a set of tests (unit testing, integration testing, user acceptance, etc.) and satisfy requirements in terms of interfacing and software quality in order to be considered ready for the final integration.

In order to reduce the number of bugs, RAINBOW developments will follow the continuous integration (CI) approach as a fundamental part of the integration and testing plan which has been adopted from the beginning of the project. In CI, the software developers are meant to integrate their work continuously. Each commit is verified by an automated build (including test) to detect integration errors, which allows the quick detection of bugs and therefore, to develop cohesive software more effectively and rapidly. In RAINBOW, this task will be automatized through the use of GitLab[1] pipelines. To ensure this, Integration and Testing is explicitly decoupled from the development of the components.

The software quality topic is an integral part of the integration strategy, ensuring not only that the code functions as expected but also that it does it in an efficient way, taking into account the quality of the code, a component often overlooked, but that allows faster modifications and facilitates team work on the same code. SonarQube[2] will carry out this task within the RAINBOW project.

The deployment of the produced builds based on the CICD processes, will be available in containerized images using Docker. This will allow the usage of these components in all environments that support Docker and relevant technologies, such as Kubernetes (and edge enabled Kubernetes releases, such as microK8s, etc), Docker swarm and docker compose. This approach will [3]also be ap[4]plied on the release of the RAINBOW framework, ensuring interoperability and interfacing between the various components.

[1] https://about.gitlab.com/
[2] https://www.sonarqube.org/

1 Introduction

The software integration of a platform is always a process which involves following several multi-disciplinary approaches when designing the integration plan. In this document, we aim to present the importance of software integration, the challenges faced, and a generic introduction to the most common methods and approaches used. More specifically the document aims to describe all the activities of Task 5.1 entitled "Technical Integration Points and Testing Plan". This task starts off with an integrated analysis of all sources available and thereafter defines necessary interfaces to integrate components.

1.1 Relationship with RAINBOW Deliverables

This deliverable is built on the foundation of D1.2, which provides a concrete documentation of the current version of the reference architecture and key technologies supported by RAINBOW and provides an initial description of the components comprising the RAINBOW framework. To this end, D5.1 extends the RAINBOW documentation by providing a report for the integration points of the various components, as well as an outline of the processes and timelines that the project will follow for its' development, testing and integration lifecycles. The information presented in this deliverable will serve as a guideline for the development effort pertaining to the integrated RAINBOW platform, which will be presented in D5.2. Additionally, the user acceptance testing portion of the testing bed that RAINBOW will rely upon, is influenced and built on the work done for D1.3.

1.2 Structure of the deliverable

This deliverable presents the technical integration points and testing plan of RAINBOW. More specifically, the deliverable is structured as follows.

Section 2 presents the building blocks upon which the integration and testing plan was built on.

Section 3 presents a short introduction of the architecture based on D1.1 and the various integration points between different components that will occur.

Section 4 presents the testing procedures that we intend to follow in this project and the integration plan, describing the developing process as well as the testing and quality assurance scheduling.

Section 5 is devoted on the implementation guidelines and the technical resources that are used during the development and deployment of the different mechanisms. More specifically, the already agreed and setup development circle is presented along with best practices that are adopted.

Finally, Section 6 presents conclusions and outlines directions for future work.

2 RAINBOW Software Components

2.1 RAINBOW Architecture

Following up the work done on the Reference RAINBOW architecture, as presented in D1.2, this deliverable aims to further clarify the various interactions and integrations between the components described in the architecture. As such, the next section of this deliverable focuses on the individual layers as previously described in D1.2, as well as their individual components and how they interact with each other, either with other components in the same layer or components in different layers.

As seen in Figure 1, the 3 main layers of components, as provided by the High Level architecture (found in D1.2), are the Modelling Layer, the Logically Centralized Orchestrator, and the MESH Routing Layer. As such, this report also focuses on these 3 main entities of the RAINBOW platform and the interactions between them.

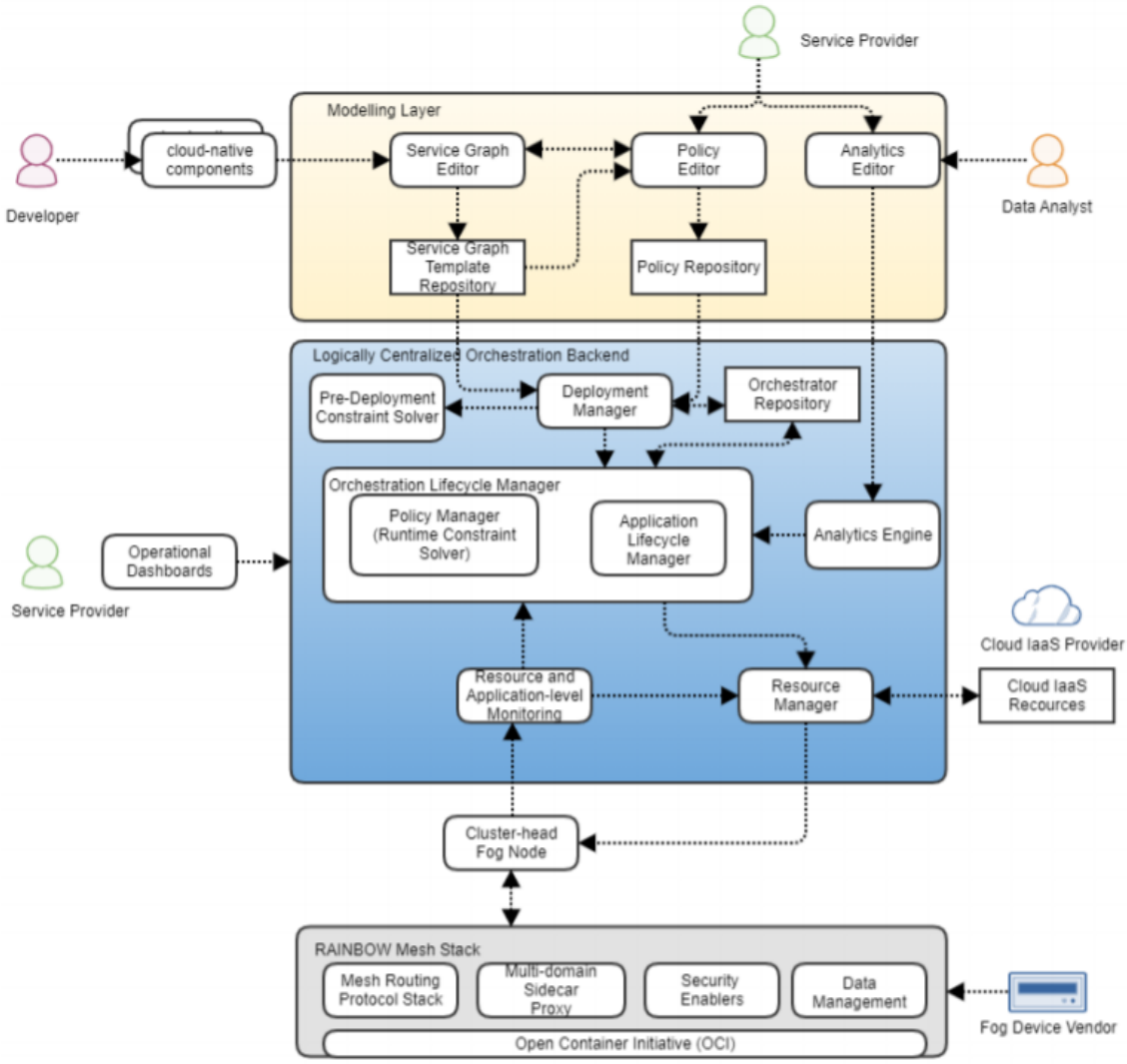

Figure 1 RAINBOW Reference Architecture

In the next section, the components of these main entities, and where applicable, their subcomponents, are listed and their integration points are presented. These integration points are coded with the initials of the main component that they logically belong to, as well as a number. In future versions we will include these reference codes to match these integration points with specific integration tests and processes that occur through the RAINBOW Integrated platform lifecycle. In addition to the reference code, a short description of each component is included, the core function, the type of interface, the constraints, inputs as well as the planned interaction with other components. These interactions are in line with what has been previously described in D1.2 concerning the reference architecture. An overview of these integration points is presented on Table X.X below, while the next section goes over the various intra-layer communications in more detail.

Layer	**Component**	**Reference Code**
Logically Centralized Orchestrator	Pre-deployment Constraint Solver	CS_01
	Deployment Manager	DM_01, DM_02
	Orchestration Lifecycle Manager	OLM_01, OLM_02, OLM_03, OLM_04
	Resource Manager	RM_01
	Resource Application Monitoring	RAM_01
MESH Routing Layer	Mesh Routing Protocol Stack	MRP_01
	Multi-domain sidecar proxy	MSP_01
	Security Enablers	SE_01
Data Management & Analytics Layer	Data Storage & Sharing	DSS_01
	Analytics Service	AS_01

Table 1 Overview of Layers, Components and Integration Points identified

2.2 Logically Centralized Orchestrator

2.2.1 Pre-deployment Constraint Solver

The pre-deployment Constraint Solver is responsible for creating a placement plan which enforces constrains placed on the deployment.

Name	*Pre-deployment Constraint Solver*
Description	*Generates a pre-deployment placement plan that is based on the provided requirements of the Service Graph and the offered available resources*
Reference Code	*CS_01*
Responsibilities	*UBI*

Function	*Generates a pre-deployment placement plan*
Subsystem	*N/A*
Type of interface	*REST*
Constraints	*A token must be provided along with the input.*
Inputs	*JSON*
Interaction with components	*N/A*

Table 2 Pre-deployment Constraint Solver Integration Points

2.2.2 Deployment Manager

The deployment manager is responsible for creating, deploying and maintaining service graph templates.

Name	*Service Graph Deployment Template Interface*
Description	*A Service Graph Deployment Template is been analysed and after the needed interactions with the corresponding components, materializes a placement plan to an actual deployment.*
Reference Code	*DM_01*
Responsibilities	*UBI*
Function	*Materializes a placement plan to an actual deployment.*
Subsystem	*N/A*
Type of interface	*REST*
Constraints	*A valid token must be provided along with the input with the correct access rights.*
Inputs	*JSON*
Interaction with components	*Pre-Deployment Constraint Solver, Orchestration Lifecycle Manager, Orchestration Repository*

Table 3 Service Graph Deployment Template Interface Integration Points

Name	*Policy Enforcement Interface*
Description	*Receives Policies and materialize them to actual enforcement policies to the corresponding Service Graph.*
Reference Code	*DM_02*
Responsibilities	*UBI*
Function	*Materializes a policy to actual enforcements on the Service Graph.*
Subsystem	*N/A*
Type of interface	*REST*

Constraints	*A valid token must be provided along with the input with the correct access rights.*
Inputs	*JSON*
Interaction with components	*Pre-Deployment Constraint Solver, Orchestration Lifecycle Manager, Orchestration Repository*

Table 4 Policy Enforcement Interface Integration Points

2.2.3 Orchestration Lifecycle Manager

The Orchestration Lifecycle Manager has the following main responsibilities:

- Coordination of the deployment of service graphs (applications) in a transactional manner.
- Coordination of the RAINBOW regions (federated approach).
- Checking of Service Level Objectives (SLOs) that have been applied to the deployed applications.
- Execution of corrective actions (e.g., elasticity strategies or security actions) in case of violated SLOs.

Since the RAINBOW implementation is based on Kubernetes, the interfaces of the Orchestration Lifecycle Manager are provided over the Kubernetes API[5].

Name	*Scheduler*
Description	*The Scheduler is responsible for assigning the microservices (pods) of all applications to the nodes they will execute on. The Scheduler does expose a public interface directly. Instead, it observes changes on the Pods interface provided by the Kubernetes API[3] to be notified when a pod needs to be (re-)scheduled. In addition to watching pods, the Scheduler uses service graph information from the Deployment Manager and resource information from the Resource Manager to fulfill its purpose.*
Reference Code	*OLM_01*
Responsibilities	*TUW*
Function	*Orchestration*
Subsystem	
Type of interface	*Indirect REST (JSON and YAML) – component observes changes of Pod resources through the Kubernetes API*
Constraints	*The scheduler interface is not directly accessible.*
Inputs	*Microservices (pods), service graph, resources*

[3] https://kubernetes.io/docs/reference/using-api/api-concepts/

Interaction with components	*Deployment Manager and Resource Manager*

Table 5 Scheduler Integration Points

Name	*SLO Manager*
Description	*The SLO Manager allows Service Developers to register supported Service Level Objectives and Service Providers to configure these SLOs for concrete application deployments.*
Reference Code	*OLM_02*
Responsibilities	*TUW*
Function	*Orchestration*
Subsystem	*SLO Enforcement*
Type of interface	*REST (JSON and YAML) – via Kubernetes API*
Constraints	*A Service Developer must be authenticated and have privileges to edit the corresponding service graph.* *A Service Provider must be authenticated and have privileges on the namespace, where the application is deployed.*
Inputs	*SLO Definitions, SLO Mappings*
Interaction with components	*SLO Controller*

Table 6 SLO Manager Integration Points

Name	*SLO Controller*
Description	*An SLO Controller is responsible for checking all instances of a particular type of SLO and triggering corrective actions in case of violations.* *An SLO Controller does not expose a public interface. Instead, it observes changes on resources of the SLO that it is responsible for.*
Reference Code	*OLM_03*
Responsibilities	*TUW*
Function	*Orchestration*
Subsystem	*SLO Enforcement*
Type of interface	*Indirect REST (JSON and YAML) – component observes changes of SLO Mapping resources through the Kubernetes API*
Constraints	*This interface is not directly accessible.*
Inputs	*SLO Mappings*
Interaction with components	*SLO Manager, Elasticity Strategy Controllers*

Table 7 SLO Controller Integration Points

Name	*Elasticity Strategy Controller*
Description	*An Elasticity Strategy Controller is responsible for carrying out corrective actions that result from the violation of an SLO.* *An Elasticity Strategy Controller does not expose a public interface. Instead, it observes changes of the resources it is responsible for.*
Reference Code	*OLM_04*
Responsibilities	*TUW*
Function	*Orchestration*
Subsystem	*SLO Enforcement*
Type of interface	*Indirect REST (JSON and YAML) – component observes changes of resources through the Kubernetes API*
Constraints	*This interface is not directly accessible.*
Inputs	*Elasticity Strategy Parameters*
Interaction with components	*SLO Controller*

Table 8 Elasticity Strategy Controller Integration Points

2.2.4 Resource Manager

The Resource Manager maintains an overview of the available and used resources on the nodes of the RAINBOW region.

Name	*Resources Registry*
Description	*The Resources Registry provides information on the resources available on a particular node.*
Reference Code	*RM_01*
Responsibilities	*TUW*
Function	*Orchestration*
Subsystem	
Type of interface	*REST (JSON and YAML) – via Kubernetes API*
Constraints	*This interface is only accessible for RAINBOW components (e.g., scheduler, SLO controller). It is not accessible to users.*
Inputs	*Resource Usage Metrics*
Interaction with components	*Resource Monitoring*

Table 9 Resources Registry Integration Points

2.2.5 Resource & Application-level Monitoring

The Resource & Application-level monitoring component is responsible for monitoring resources and applications deployed in the RAINBOW infrastructure.

Name	*Monitoring Interface*
Description	*The Monitoring interface provides access to metrics collected from the RAINBOW infrastructure and the running Fog services to other components of the RAINBOW platform*
Reference Code	*RAM_01*
Responsibilities	*Implementation: UCY* *Usage: TUW, AUTH, UCY*
Subsystem	
State	*User Interface, Access Security, etc.*
Type of interface	*REST, PUB/SUB protocol(e.g., KAFKA)*
Constraints	*Requests to the service should be authenticated* *Requests and Responses are JSON UTF-8 encoding*
Inputs	*Configurations*
Interaction with components	*Orchestrator* *Analytics Service* *Storage Service*

Table 10 Monitoring Interface INtegration Points

2.3 MESH Routing Layer

2.3.1 Mesh Routing Protocol Stack

The Mesh Routing Protocol Stack is responsible for creating and maintaining the Mesh network on which the RAINBOW components are placed.

Name	*Mesh Routing Interface*
Description	*Through the interface, the nodes will be able to identify, as also to elect the cluster heads, establish the overlay and perform path discovery.*
Reference Code	*MRP_01*
Responsibilities	*UBI*
Function	*Mesh routing services*
Subsystem	*N/A*
Type of interface	*Custom Protocol, Rest*
Constraints	*The interface will be accessible only from the Rainbow Components and when someone is inside the mesh.*
Inputs	Configurations
Interaction with components	*Cluster-head*

Table 11 Mesh Routing Interface Integration Points

2.3.2 Multi-domain sidecar proxy

The Multi-domain sidecar proxy is a node-level component responsible for extracting metrics and adjusting high level configurations for nodes.

Name	*Sidecar Proxy Interface*
Description	*Through this interface, the sidecar proxy is configured in order to execute possible adjustments on the node, properly extract metrics and many other responsibilities that may have.*
Reference Code	*SP_01*
Responsibilities	*UBI*
Function	*Configure Sidecar Proxy*
Subsystem	*N/A*
Type of interface	*Key/Value storage, pub/sub (e.g., Kafka)*
Constraints	*The interface will be accessible only from the Rainbow internal components.*
Inputs	Configuration as JSON
Interaction with components	*Cluster-head, Orchestrator, Resource & Application-level Monitoring*

Table 12 Sidecar Proxy Interface Integration Points

2.3.3 Security Enablers

The purpose of the RAINBOW security and trust enablers is to provide enhanced remote attestation mechanisms towards the secure composability of fog environments, encompassing a broad array of mixed-criticality services and applications. The main goal is to allow the creation of privacy- and trust-aware service graph chains (managed by the Orchestrator and established by the Deployment Manager) through the provision of S-ZTP functionalities: *fog nodes adhere to the compiled attestation policies by providing verifiable evidence on their configuration integrity and correctness*. The necessary interfaces and agents, for providing the following functionalities (as defined also in Deliverable D2.1), will be implemented:

- Secure Enrolment of a newly joined (or deployed) fog/edge node for verifying its correct configuration and trusted state. This is deemed necessary before allowing the node to enter a cluster (or a mesh network) and establish the necessary key material for the subsequent operations;
- Zero-touch Configuration and Operational Correctness for providing the necessary guarantees that a VF works correctly both after deployment (i.e., boot-up – load-time integrity) but also throughout its operational life-cycle (runtime integrity);

- Cryptographic Key Management responsible for providing all the necessary interfaces for the secure key establishment and key management functionalities needed by all RAINBOW internal components and fog/edge nodes.

Name	*Secure Enrolment Agent*
Description	*The Secure Enrolment Agent provides the necessary interfaces by which a (newly joined) fog node (or a deployed VF) can report, in a trusted way, the status of its configuration. This entails the extraction of a "quote", by leveraging the attached trusted component (i.e., TPM), reflecting the node's current integrity measurement list to be verified against the correct configuration policies including the reference measurements of the whitelist (to be loaded) of application binaries.*
Reference Code	*SE_01*
Responsibilities	*DTU*
Function	*Zero-touch Configuration Integrity Verification (CIV) through the provision of Attestation by Quote and Attestation by Proof interfaces*
Subsystem	*Sidecar Proxy Interface, Control-Flow Attestation Agent*
Type of interface	*Custom Protocol, REST*
Constraints	*The Extended Authorization (EA) policy digest, which reflects the trusted state in which the newly joined or deployed fog node must be found to (i.e., integrity reference measurement), needs to be to correctly deployed to the (initial) service graph chain, as part of the Service Graph Deployment Template Interface.*
Inputs	*EA Policy Digest, Attestation key Template, Configuration Integrity Verification Policy*
Interaction with components	*Service Graph Deployment Template Interface,* *Policy Validator,* *Sidecar proxy Interface,* *Orchestrator or Cluster-Head (acting as the Verifier),* *Key Management Interface*

Name	*Control-Flow Attestation Agent*

Description	*The Control-Flow Attestation Agent is responsible for attesting the correct executional behavioural properties of a deployed VF/service, upon request from the Orchestrator or the cluster-head. This agent leverages the Attestation by Quote and Attestation by Proof interfaces, of the Secure Enrolment Agent, in order to verify the correct execution path of a deployed VF; all the correct execution paths (i.e., Control-flow Graphs) of interest have been identified to act as the baseline of the normal sequence of states against which the run-time control-flow footprints (of a VF) will be assessed. Any deviation from the legitimate CFGs results in an unrecognized measurements which is indication of a possible VF exploitation.*
Reference Code	*SE_02*
Responsibilities	*UBI/DTU/POLITO*
Function	*Operational Correctness*
Subsystem	*Sidecar Proxy Interface*
Type of interface	*REST, KAFKA*
Constraints	*Only specific executional behavior properties (i.e., functions) of a deployed node can be attested and not the entire VF codebase.*
Inputs	*Attestation Challenge, Extracted Contol-flow Graph*
Interaction with components	*Orchestrator or Cluster-Head (acting as the Verifier),* *Monitoring Interface,* *Multi-level Detailed Tracing Agent,* *Sidecar Proxy Interface,* *Key Management Interface*

Name	*Multi-level Detailed Tracing Agent*
Description	*The Multi-level Detailed Tracing Agent provides the interfaces for monitoring a node's runtime data and execution graphs necessary for tracing the control- and information-flow execution paths (i.e., CFGs) needed by the Control-flow Attestation Agent. This is achieved through the deployment of eBPF execution hooks, as low-level behavioural properties. The goal*

	of this tracing is to monitor system calls, produce the necessary CFGs and extract the respective attestation report. In the case of a failed attestation report, the RAINBOW Orchestrator will increase the level of (node) monitoring – through the Sidecar Proxy Interface – in order to collect additional evidence and information on the incident for the assistance in finding the province of the attack as well as in the development of new enforceable policies that should be able to catch this newly identified threat.
Reference Code	*SE_03*
Responsibilities	*UBI/DTU*
Subsystem	*Control-Flow Attestation Agent*
Function	*Real-time Node Data and Execution Stream processing and Monitoring*
Type of interface	*Custom protocol*
Constraints	*This interface will be accessible only from the sidecar proxy interface that can be invoked by the RAINBOW Orchestrator when a more detailed tracing is required*
Inputs	*Configurations*
Interaction with components	*Sidecar Proxy Interface, Monitoring Interface*

Name	*Direct Anonymous Attestation Agent*
Description	*The DAA Agent is responsible for providing the necessary interfaces towards the creation of trusted and privacy-aware mesh overlay network paths between connected nodes. It entails the support for the: (i) anonymous communication between fog/edge nodes on top of the already established CJDNS mesh network, (ii) self-certification of the respective DAA and ephemeral DH keys, and (iii) ascertation of a platform's state as recorded by the Control-Flow Attestation Agent.*
Reference Code	*SE_04*
Responsibilities	*DTU*
Subsystem	*Mesh Routing interface*
Type of interface	*Custom Protocol*

Constraints	*Direct Anonymous Attestation is based on the use of Trusted Platform Modules (TPMs) that should be attached to each one of the deployed fog/edge nodes. Furthermore, anonymous communication will be provided from the CJDNS Level 3 and onwards.*
Inputs	*Privacy Policies*
Interaction with components	*Orchestrator,* *Secure Enrolment,* *Control-Flow Attestation Agent,* *Mesh Routing Interface,*

Name	*Key Management Interface*
Description	*The Key Manager is responsible for providing all the necessary interfaces for the secure key establishment and key management functionalities needed by all RAINBOW internal components and fog/edge nodes. More specifically, this entails the establishment of the: (i) Attestation key (to be used in the Secure Enrolment), (ii) DAA key, (iii) ephemeral DH key, and (iii) CJDNS secure communication keys.*
Reference Code	*SE_05*
Responsibilities	*IFAT*
Subsystem	*All RAINBOW internal components – especially all artefacts related to the Mesh Routing Protocol Stack*
Type of interface	*REST*
Constraints	*RAINBOW Cryptographic Key Management Systems (CKMS) is based on the use of Trusted Platform Modules (TPMs) that should be attached to each one of the deployed fog/edge nodes.*
Inputs	*Key Management Policies*
Interaction with components	*Orchestrator,* *Secure Enrolment,* *Control-Flow Attestation Agent,* *Mesh Routing Interface,* *Service Graph Deployment Template Interface*

2.4 Data Management & Analytics Layer

2.4.1 Data Storage and Sharing

The purpose of the Data Storage and Sharing component is to store the collected data from the different RAINBOW components and concurrently allow any authorized components to quickly have access to stored data either from the local database instance or from a set of instances. Another important part of the component is to support the overlay network components by providing quick cache mechanisms for routing.
The data storage component will comprise a distributed in-memory database and a set of interfaces to connect with said database along.

Name	*Data Ingestion*
Description	*The interface from which the requesting components send data to be stored in the database.*
Reference Code	*DSS_01*
Responsibilities	*AUTH*
Subsystem	*Local database instance*
Type of interface	*Socket*
Constraints	*Authentication of the requesting component*
Inputs	*JSON of input data*
Interaction with components	*Any component that needs to store data in the local database*

Table 13 Data Ingestion Integration Points

Name	*Data Extraction*
Description	*The interface from which the requesting components extract data from the database.*
Reference Code	*DSS_02*
Responsibilities	*AUTH*
Subsystem	*Local database instance, Global database client instance*
Type of interface	*Socket*
Constraints	*Authentication of the requesting component*
Inputs	*JSON of requesting data*
Interaction with components	*Any component that needs to extract data from the database, either from a local instance or globally.*

Table 14 Data Extraction Integration Points

2.4.2 Analytics Service

The Analytics service is responsible for exposing an API through which other services can interact with in order to collect monitoring information from the RAINBOW platform.

Name	*Analytics Service API*
Description	*The Analytics Service API provides access to analytic results from the collected monitoring information to the components of the RAINBOW platform.*
Reference Code	*AS_01*
Responsibilities	*Implementation: UCY* *Usage: TUW, SUITE5*
Subsystem	
Type of interface	*REST, PUB/SUB protocol(e.g. KAFKA)*
Constraints	*Requests to the service should be authenticated* *Requests and Responses are JSON UTF-8 encoding*
Inputs	*A set of Analytic Queries*
Interaction with components	*Resource Manager* *Orchestration Lifecycle Manager* *Operational Dashboards*

Table 15 Analytics Service Integration Points

3 RAINBOW Integration and Testing Plan

Regarding the Integration and Testing plan, the methodology used is based on the STEP (Systematic Test and Evaluation process) approach. The core idea of the STEP approach is that the main focus is spent on designing an overall testing plan, with the idea being that early testing provides prevention potential, i.e. that software shortcomings and issues can be resolved before they occur during the initial testing stages. As such, the guideline of the methodology is to design the testing framework and the corresponding test use cases as early as possible, based on the specific objectives and requirements of the software in question, and not wait for the actual software design process to be over and the implementation process to have begun.

A similar approach is followed in RAINBOW, where the testing of the individual components (unit testing) will be done early in the project, with the next step being the integration tests. These integration tests will also be done before the release of the first iteration of the integrated RAINBOW platform, using simulated environments and interconnections, as necessary.

In the next section, an overview of each of the individual "building blocks" of the overall testing plan and testing actions that will be completed in RAINBOW are presented.

3.1 Integration Plan in RAINBOW

RAINBOW follows a specific approach in order to implement the mechanisms that constitute the RAINBOW framework. RAINBOW development is a continuous process which contains all required discrete steps that re-assure quality during the entire lifetime of the project. As mentioned earlier, the RAINBOW platform development will take part in 3 development cycles, with the first step of each cycle being the development and testing of each individual component, and the second step being the integration of the components into the RAINBOW framework. However, as described earlier, the tasks that are derived from these steps will run in parallel with simulated environment usage being substituted for the actual components and integration points as they become available. To better handle the workload, the individual layers of the architecture (as presented in the first section of this document) and their individual integration points will be the focus of each cycle, with the first being the Logically Centralized Orchestrator, followed up by the Mesh Routing Layer and then the Data Management Layer.

In short, the milestones of the integrated RAINBOW platform, given the time plan of work in the various WPs, are the following:

- M15: First release of individual components per layer of the architecture (WP2, WP3, WP4).
- M18: First integrated version with release of components in M15 (WP5).

- M27: Final release of individual components per layer of the architecture (WP2, WP3, WP4).
- M27: Second integrated version with release of components in M27 based on feedback of previous release(WP5)
- M36: Final integrated version with release of components in M27 based on improvements and feedback derived from demonstrators (WP5).

3.2 Unit Testing

The main objective of this section is the (non-exhaustive, given that software development runs in parallel) description of the applied unit tests in the RAINBOW integrated framework. Unit tests are the tool to test the functional modules of software. In the case of the RAINBOW integrated framework unit tests will guarantee the quality of the particular layers developed in the corresponding work packages. A suitable unit test is applied to the piece of code without any dependencies on other code parts. Therefore, the developer of each layer will test the components by means of unit tests before integrating them into the full application. These unit tests will be the first of many tests to be implemented for individual components and will run in parallel with the integration testing.

3.3 Integration Testing

After the development of all individual components of the RAINBOW platform is finished, the system is ready to be integrated and tested in order to check if it meets its specifications. This kind of testing is referred to as Integration Testing, as it tests the integration of the units into the overall system. More specifically, integration testing aims to diagnose errors in the design of the system or the specifications in the system's units, as well as the interfacing between them.

If the different units of a system are tested in combinations, eventually all the units comprising a process will be tested together. The discovered errors during the integration testing are mainly related to the interfaces between them, as all units have already been tested separately during the unit testing portion of the testing process.

The most common strategies that an integrator can use to perform integration testing are the following :

- The **top-down** approach achieves step by step verification of the interfaces among components that operate under a common control strategy. This control strategy dictates the order of development, integration and testing. Top-down integration interleaves component scope testing and integration of a system of components.
- The **bottom-up** approach achieves step by step verification of the interfaces between tightly coupled components. It interleaves component scope testing and integration of system components. Components with the least number of dependencies are tested first.
- The **hybrid** approach uses both the top-down and the bottom-up approach and performs testing with functional data along with control flow paths. Firstly, the inputs for functions are

integrated in the bottom-up pattern discussed above. The outputs for each function are then integrated in the top-down manner.

For RAINBOW, the hybrid approach was chosen, which allows for multitarget testing to occur in parallel. This approach may require potential software components and subcomponents to be simulated for the sake of testing before they become operational, but overall allows for a greater test coverage of the overall framework.

3.4 User Acceptance Testing

Acceptance Tests are usually tests created by business customers and expressed in a business/scientific domain language. These are high-level tests to verify the completeness of stories 'played' during any sprint/iteration. These tests are created ideally through collaboration between analysts, testers, end users and developers.
Acceptance test cards are ideally created during sprint planning or iteration planning meeting, before development begins so that the developers have a clear idea of what to develop. Sometimes acceptance tests may span multiple stories (that are not implemented in the same sprint) and there are different ways to test them out during actual sprints. One popular technique is to mock external interfaces or data to mimic other stories which might not be played out during iteration (as those stories may have been relatively lower business priority). A user story is not considered complete until the acceptance tests have passed. Tools such as Cucumber , JBehave , Concordion and Twist allow the documentation of acceptance criteria in natural language and then in turn writing of software code for the execution of the specific testing.

Specifically for RAINBOW, the UATs (user acceptance tests) that will be performed will be driven by the needs of the Use Cases as described in D1.3. By adhering to the specifications described, the RAINBOW platform will be able to fulfil the needs and the specific scenarios on which the use cases are built upon.

3.5 Requirement Coverage

The Integration and Testing plan is also taking into account the coverage of the requirements of the RAINBOW project, as they have been shaped by the Use Case scenarios. These requirements are currently being finalized and will be presented in D1.3, and as such will not be mentioned in detail here. Effort will be put towards ensuring the requirements are satisfied, as per ISO/IEC 25010:2011(en) standard for Software Quality. In short, this entails the following:

- Functional Sustainability
    - Functional Completeness
    - Functional Correctness
    - Functional Appropriateness
- Performance Efficiency
    - Time Behaviour
    - Resource Utilization
    - Capacity
- Compatibility

    - Co-existence
    - Interoperability
- Usability
    - Learnability
    - Operability
    - User Error Protection
    - User Interface Aesthetics
    - Accessibility
- Reliability
    - Maturity
    - Availability
    - Fault Tolerance
    - Recoverability
- Security
    - Confidentiality
    - Integrity
    - Non-repudiation
    - Authenticity
    - Accountability
- Maintainability
    - Modularity
    - Reusability
    - Analysability
    - Modifiability
    - Testability
- Portability
    - Modifiability
    - Installability
    - Replaceability

By adhering to all those principles, with individual focus on the functional and non-functional requirements as expressed through the Use-case scenarios, the RAINBOW platform will guarantee the quality of its individual components and itself as a whole.

3.6 Emulation of Cloud – Fog Resources

As the Fog environment is of particular interest in RAINBOW, appropriate mechanisms for testing functionality and compatibility for services deployed to the fog network have to be in place. As such, Fogify has been chosen for the RAINBOW Fog testbed.

Fogify[3] is an open-source framework, initially developed and maintained by UCY, that enables developers to quickly model fog deployments and perform large-scale, repeatable and reproducible experimentation of data-intensive applications by encompassing description abstractions for modeling fog resources, network capabilities and runtime scaling actions. In turn, developers can create their own experiment scenarios (the SDK even works in Jupyter Notebooks) where they can define faults (e.g., a sudden increase/drop in load/latency, a node is unresponsive, etc) and monitoring metrics. Fogify accepts and extends Docker Compose descriptions so developers do not need to perform huge workarounds to get fog/cloud-enabled applications that are Docker-ized to work with Fogify. When a user is satisfied with the tests conducted, then he/she may move forward to a production deployment in a fog environment without the need to change any coded artifacts or deployment configurations.

By taking advantage of the Fogify emulator, the RAINBOW research team can rapidly set up fog-enabled testbeds and evaluate coded use-case without wasting time in deploying, configuring and maintaining, across geo-distributed realms, fog resources and networks. Therefore, this will allow the RAINBOW platform to be thoroughly tested in a variety of different environments based on the fog computing paradigm, ensuring interoperability, functionality and scalability as the development effort progresses, while also providing the component developers and the platform as a whole with performance indicators to assure the performance efficiency of the platform, as mentioned earlier.

[3] Fogify: A Fog Computing Emulation Framework. Moysis Symeonides, Zacharias Georgiou, Demetris Trihinas, George Pallis, Marios Dikaiakos, "Proceedings of the 5th ACM/IEEE Symposium on Edge Computing" (SEC '20), San Jose, CA, USA Association for Computing Machinery, New York, NY, USA, 2020 https://ucy-linc-lab.github.io/fogify/

4 Continuous Integration and Quality Assurance Implementation Aspects

As already described in D1.2 - RAINBOW Reference Architecture, RAINBOW follows a specific approach in order to implement the mechanisms that constitute RAINBOW framework. RAINBOW mechanisms' development is a continuous process which contains all required discrete steps that re-assure quality during the entire lifetime of the project. This process can be represented as a virtual circle that contains the following functional components a) code development, b) version control system (pushing the changes to Gitlab), c) CICD actions (building and forwarding artefacts for further actions) d) quality assurance of generated code, e) persistent storage of generated builds in a dockerized format, e) issue/bug tracking and f) deployment to production. Additionally, Bazel was also considered to be used as the standard for build procedures for the components in the project but was later replaced by Maven for the majority of the components, while also leaving some room for individual developer choice for their component in order to speed up development due to familiarity and knowledge of other technologies.
Each part of the circle is supported by mature tools that are already setup and interoperate smoothly. These tools are depicted on Figure 2. More specifically these tools are: Gitlab for a) version control, b) continuous integration & development c) for issue/bug tracking c) Sonar for code quality assurance and d) Gitlab Container Registry for docker image management. For the purpose of RAINBOW, the software described will be hosted on a managed Kubernetes cluster, to ensure scalability and redundancy of the CICD operations.

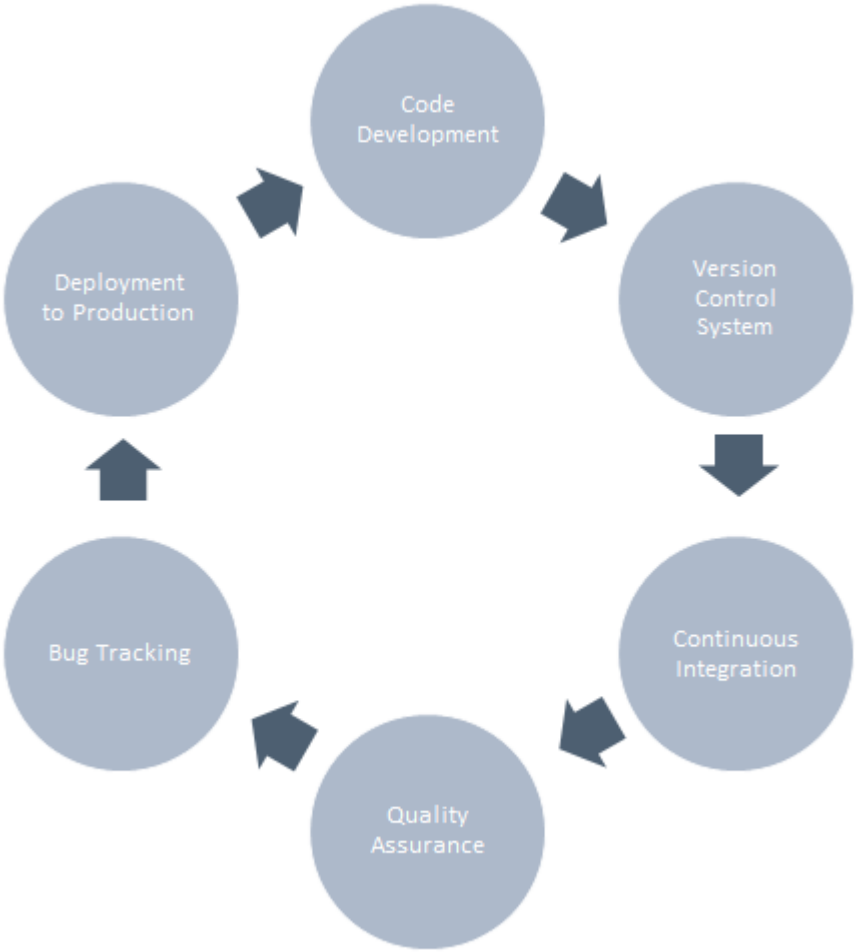

Figure 2 Development Lifecycle

4.1 Version Control System – Gitlab

As already presented above, the consortium has selected Gitlab as the primary VCS system. The RAINBOW project source code will be organised in multiple repositories:

- **rainbow-scheduler** contains all source code of the RAINBOW Scheduler,
- **rainbow-orchestration** contains all source code of the RAINBOW Orchestrator,
- **rainbow-analytics** contains all source code of the RAINBOW analytics service
- **rainbow-monitoring** contains the source code of the RAINBOW monitoring applications,
- **rainbow-storage** contains all the source code and configurations of the RAINBOW storage mechanisms,
- and **rainbow-attestation,** which contains all the source code of the RAINBOW attestation mechanisms.

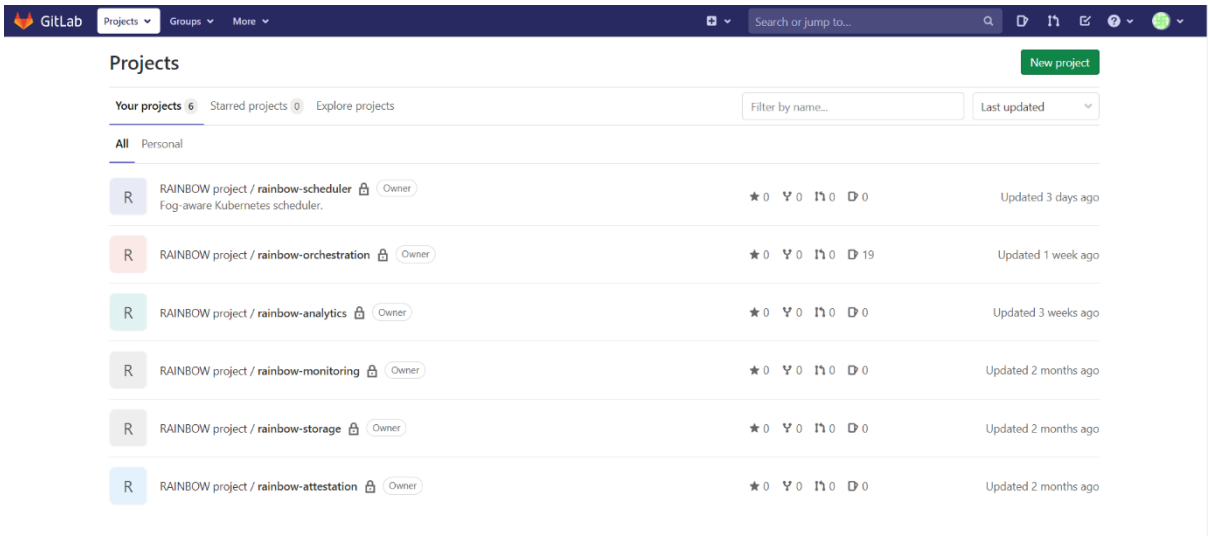

Figure 3 Gitlab Repositories

For now, only the basic repository is available via a Gitlab repository, which is located here: https://gitlab.com/rainbow-project1/ (see Figure 3) and its access is limited to the consortium developers for the time being. **After the finalization of the project the consortium will open the Gitlab repositories which will contain all mechanisms**.

4.2 Build Distribution & Containerization – Docker/K8s

Over the past few years, container technologies have been widely adopted by enterprises and SMEs, with significant benefits for DevOps teams. Such benefits include smaller memory footprints for user applications, faster deployment and bootstrapping times and improved isolation between containerized applications that share the same host resources. Among the multiple container technologies that have emerged, the one with the biggest market share and the widest adoption is Docker. In deliverable D1.2, Docker has been extensively described.

In addition to Docker and docker images, which are the file templates that describe a docker container deployment, another significant development over the past few years has been the advent of orchestration technologies, such as Kubernetes. Kubernetes offers a complete solution to orchestrate, manage, deploy and automate sets of docker containers in clusters.

Regarding RAINBOW, and specifically the CICD lifecycles of the project, docker images offer a substantial benefit allowing developers of components to focus on the development of features without having to worry about compatibility with specific installations. Furthermore, Kubernetes allows the deployment of the aforementioned images in clusters which can be scaled automatically if need be.

Finally, Kubernetes can also be utilized to handle the workload of the CICD pipelines as described above. By using Kubernetes enabled clusters for running Gitlab Pipeline runners, builds can be performed in a homogenous scalable environment guaranteeing

compatibility and availability of resources through the automated scaling capabilities offered by the clusters.

4.3 Continuous Integration – Gitlab Pipelines

The deployment of RAINBOW is based on a Continuous Integration process. Continuous Integration (CI) is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time. It is used in software development to automate and improve the process of software integration. Continuous integration basis is a series of techniques designed to ensure that code can be rapidly and safely deployed to production by delivering every change to a production-like environment and ensuring proper functionalities through automated testing.

According to this approach, every change is delivered to a staging environment using complete automation, so that it is guaranteed that the created application is deployable at any time and can be deployed to production with the push of a button.

Continuous Integration is also another option that is even more automated. In Continuous Integration, every change that passes the automated tests is deployed to production automatically. For the RAINBOW Framework, the final decision for deployment to production is done manually, in order to have total control of the releases that go into production, so Continuous Delivery is performed. The overview of this process that will be followed in order to deploy a new version of RAINBOW Framework to production is presented in Figure 4.

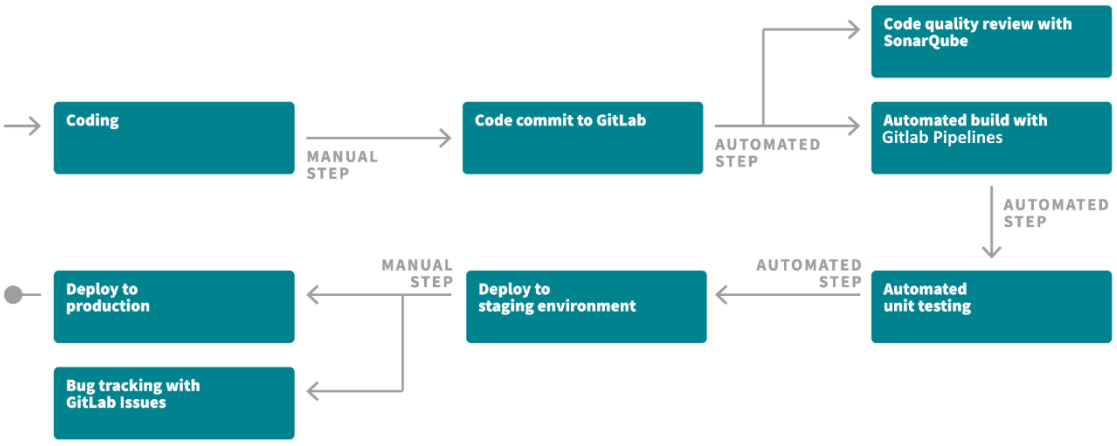

Figure 4 Continuous Delivery in the RAINBOW Framework

The whole process is automated except two basic manual steps. The starting point of this process is the commit of the code by the developer. For every commit made, a git hook is configured to start the Automated Build – Continuous Integration with Gitlab Pipelines.

These pipelines are configured to build new releases of RAINBOW Framework (and its components) in an automated manner on a staging environment. Semi-automated (after manual approval) deployment to production environment is also configured. Automated testing has also been configured with usage of unit tests for every build made with Gitlab. Unit testing is a task that every artefact developer is responsible for and is performed before the integration of the mechanism in the RAINBOW Framework. Beyond integration, functional tests will also be created for several main functionalities of the RAINBOW Framework in order to ensure its proper functioning.

The purpose of the Continuous Integration (CI) platform is twofold. On the one hand, the latest valid snapshot is always deployed to a specific server which is used for continuous testing. On the other hand, when the development will be about to finish the CI server will operate as a pre-staging environment. This practically means that any upgrades that will be released during the production phase will be performed automatically through Jenkins.

This is extremely valuable, since in the RAINBOW project the entire development lifecycle has been done using industry-driven standards. During the build-process, all unit-tests are executed. This practically means that each release has a functional guarantee regarding its stability. However, this also means that the responsibility of the developers is high since the test coverage is under their jurisdiction.

4.4 Source Code Evaluation – Sonar

Nowadays, the quality measurement of software development has become increasingly important. As in any technological project in scale, there is a need for a way to measure the quality and how the work progresses, when different people have different access to the source code. Although, quality is somewhat subjective attribute and understood differently by different people, an independent organization, founded by the Software Engineering Institute at Carnegie Mellon University and the Object Management Group, called Consortium for IT Software Quality (CISQ[4]) has defined a set of software structural quality characteristics. In the “House of Quality” model, these are "What’s" that need to be achieved:

- ***Reliability***: An attribute of structural solidity. Reliability measures the level of risk and the likelihood of potential application failures. It also measures the defects injected due to modifications made to the software.
- ***Efficiency***: The source code is the element that ensures high performance once the application is in run-time mode. Efficiency is especially important for applications in high execution speed environments such as algorithmic or transactional processing where performance and scalability are paramount.
- ***Security***: A measure of the probability of potential security breaches due to poor coding practices or architecture. This kind of breaches increases the risk of critical vulnerabilities that can damage a business.
- ***Maintainability***: Maintainability includes the concept of adaptability and portability. It is very important to measure the maintainability for mission-critical applications, where

[4] http://it-cisq.org/

each change is driven by tight schedules and is important to remain responsive during the changes. It is very crucial to keep maintenance costs under control.

- ***Size***: The sizing of source code is a software characteristic that obviously impacts maintainability.

Based on the principles described above, the development lifecycle relies on the SonarQube tool that performs the quality testing at the source level. SonarQube is configured to perform a set of analyses such as static code analysis, analysis of best practices, analysis of conventions, etc. Specific reports regarding the blocking/critical/major issues are presented along with several indications of architectural quality (duplications, reusability, testing coverage etc.).
A critical indicator regarding the quality is the technical depth that attempts to quantify the number of days that are required in order to make a specific snapshot a proactive release.
At this point it should be noted that the emphasis of the development team is not to tackle every non-technical quality issue that is raised by SonarQube but to finish the entire set of features that are required for the first release.

Having in mind the above, the source code implemented in the context of RAINBOW project will be examined by Sonar and the results will be presented during the next months in D5.2 – RAINBOW Framework and Unified Dashboard Early Release, D5.3 – RAINBOW Framework Second Release and D5.4 RAINBOW Framework final release.

4.5 Issue Tracking – Gitlab

As already mentioned above, GitLab Issues is the issue/bug tracking toolset that RAINBOW project uses. The GitLab issues of the RAINBOW Project are located at https://gitlab.com/groups/rainbow-project1/-/issues (see Figure 5), whose access is limited to the consortium developers for the time being.

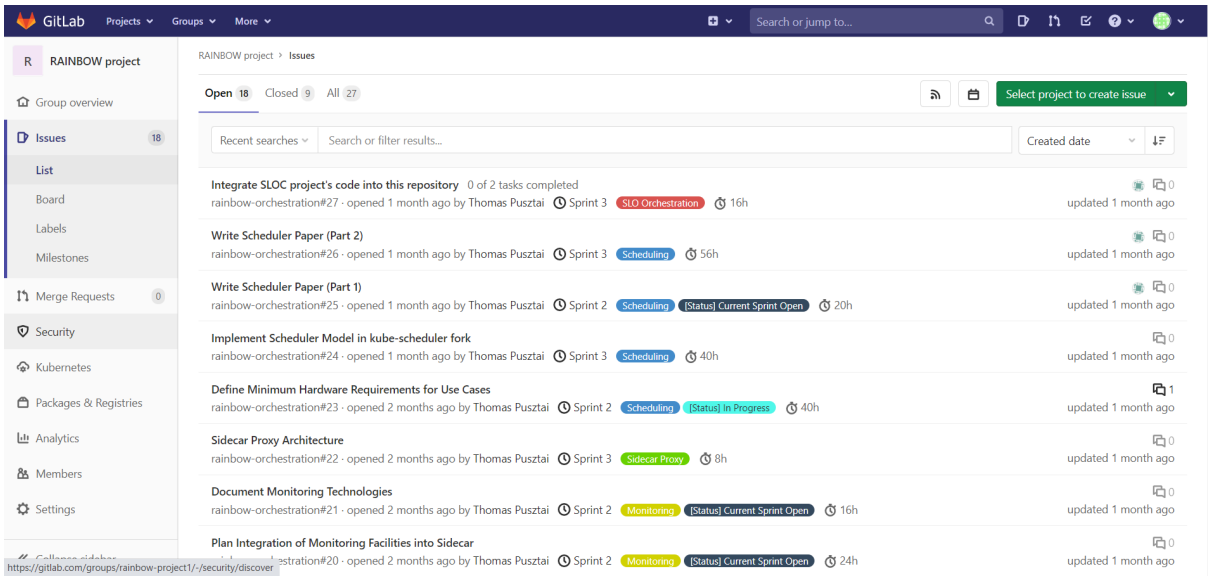

Figure 5 Gitlab Issues

4.6 Continuous Deployment

As already mentioned, Gitlab is used for the majority of Continuous Integration and Continuous Deployment operations. Each time a new version of the code is pushed to the respective repository, either via committing updates to the master branch directly, or by merging branches onto the master branch, a new CI/CD pipeline is spawned.
Each repository hosted has its own autonomous CICD functions, which are executed by privately hosted Gitlab Runners. Figure 6 depicts the current runners (from GitLab) inside the Kubernetes Management dashboard.

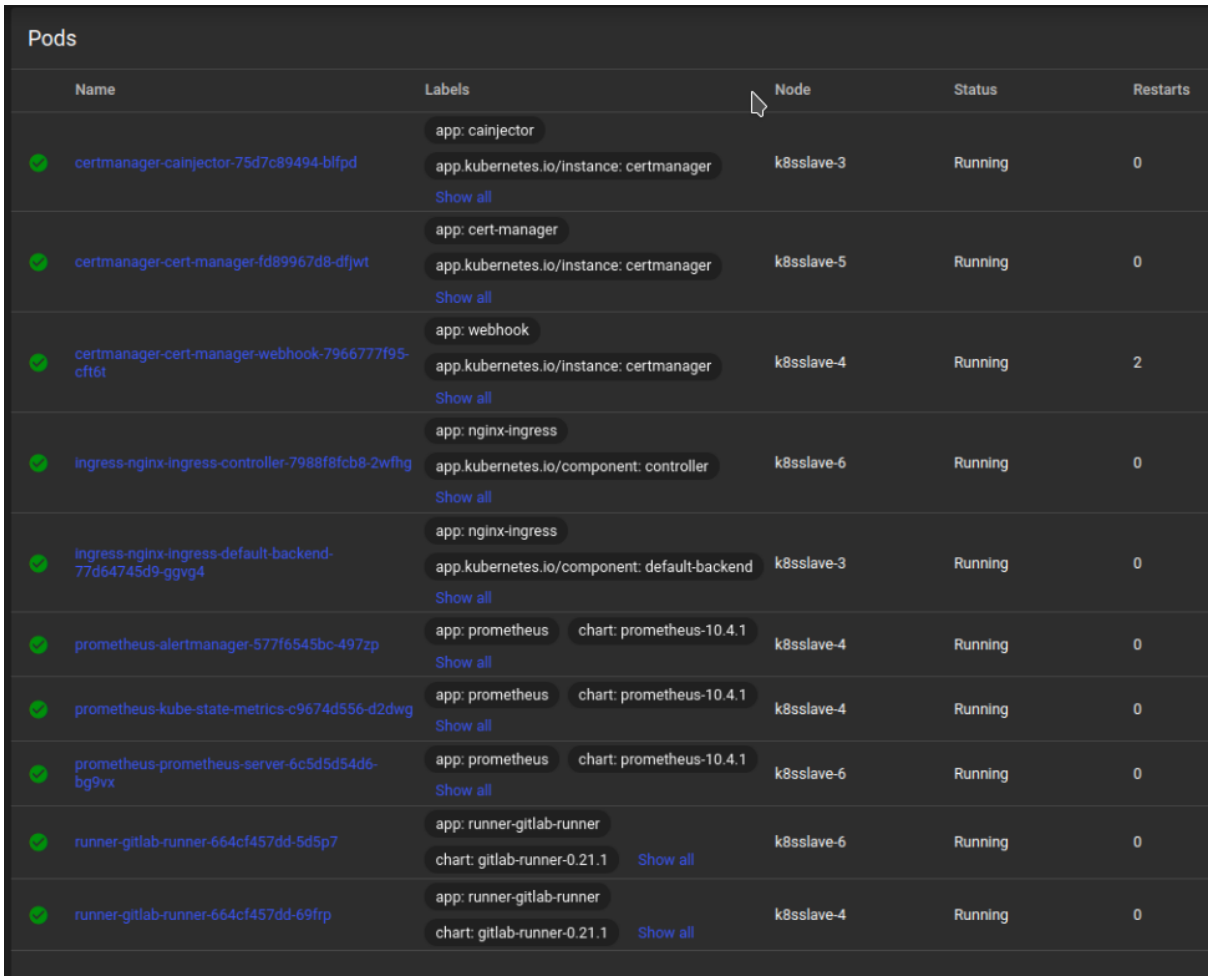

Figure 6 Spawned Gitlab Runners on Kubernetes Dashboard

Pipelines are multi-task/stateless processes that perform a lot of tasks. Each task is executed in the aforementioned Kubernetes workers, which are created as needed, per the committal of code by a developer. It should be noted that GitLab offers an out-of-the box automatic generation of pipeline based on the introspection of the source code. This feature is called AutoDevOps; this feature was not used due to its lack of customizability and the fact that it was skipping or substituting steps of the CICD process as was described in previous sections.

Instead, a manual definition of the CI/CD tasks has been performed using the gitlab-ci scripting mechanism, using a standard .yaml format. This mechanism allows the definition of arbitrary pipelines based on the projects' requirements. A graphical representation of a RAINBOW pipeline is depicted on Figure 7.

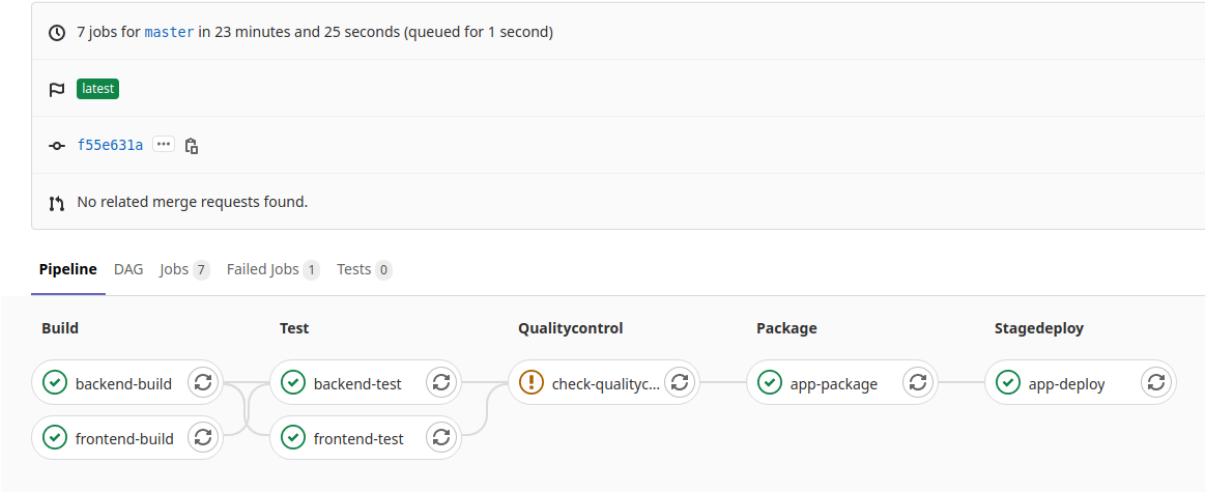

Figure 7 Indicative Execution of pipelines

As is shown, the pipeline consists of 7 discrete tasks that perform the operations needed throughout the CICD lifecycle. Tasks are organized in Stages, which are groups of tasks that can be executed in parallel. The current implementation of the RAINBOW CICD employs 5 stages, since build and test tasks can be fully parallelized.

The source code consists of multiple commands that are executed sequentially on an instance of an operating system that is spawned on Kubernetes (Gitlab Runner) per task execution. These runners are configured to use the Alpine Linux image[5], mainly because of its small footprint. The build tasks, as seen on Figure 8, rely on Maven and NPM tools that are on-the-fly installed in the stateless kubernetes workers. Additionally, environmental variables are supported by the Gitlab CI infrastructure, and are used for propagating necessary private information (e.g. hidden passwords, application secrets, etc.).

[5] https://hub.docker.com/_/alpine

```
backend-build:
    stage: build
    script:
        - cp .env.example .env
        - cp src/main/app/.en.example src/main/app/.env
        - source .env
        - ./mvnw $MAVEN_CLI_OPTS clean compile -DskipTests=true -Dskip.npm -Plocal $MAVEN_OPTS
        - ./mvnw $MAVEN_CLI_OPTS clean package -DskipTests=true -Plocal $MAVEN_OPTS

frontend-build:
    stage: build
    script:
        - npm -version
        - which npm
        - cp src/main/app/.env.example src/main/app/.env
        - cd src/main/app
        - npm install
    artifacts:
        path:
            - src/main/app/node_modules
```

Figure 8 Build Tasks

Figure 9, depicts part of the source code of the testing script while Figure 10 shows part of the quality control script. Individual repositories may need different build code depending on the underlying technologies they use.

```
backend-test:
  stage: test
  needs: ["backend-build"]
  script:
    - cp .env.example .env
    - source .env
    - ./mvnw $MAVEN_CLI_OPTS test -Dskip.npm -Plocal $MAVEN_OPTS

frontend-test:
  stage: test
  needs: ["frontend-build"]
  script:
    - cd src/main/app
    - npm run lint
```

Figure 9 Testing Tasks

```
check-qualitycontrol:
  stage: qualitycontrol
  needs: ["frontend-test","backend-test"]
  script:
    - cp .env.example .env
    - cp src/main/app/.env.example src/main/app/.env
    - source .env
    - ./mvnw $MAVEN_CLI_OPTS clean compile sonar:sonar -Dsonar.sources=src/main/java,src/main/app/src  -Dsonar.qualitygate.wait=true -Dskip.npm -
  allow_failure: true
  only:
    - master
```

Figure 10 Quality Tasks

As mentioned earlier, quality control is achieved using the Sonar analysis engine. Sonar is able to perform various types of analysis in order to infer (see Figure 11), bugs, security vulnerabilities, code smells and documentation/testing coverage.

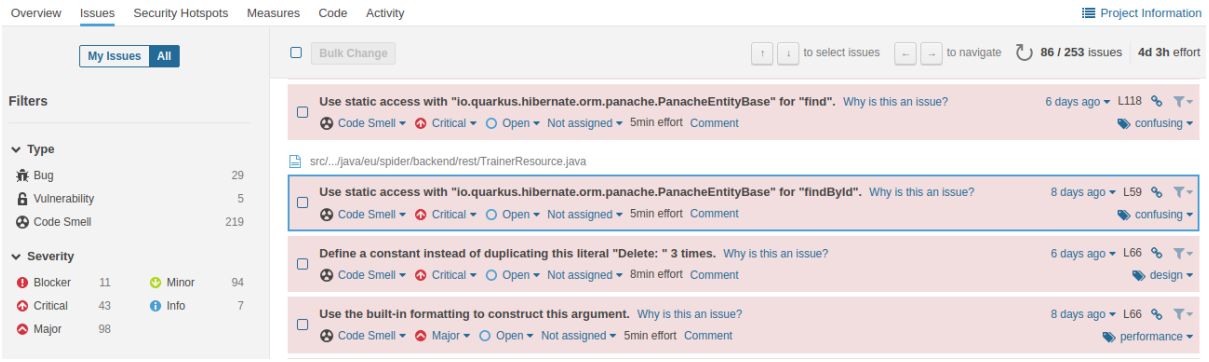

Figure 11 Indicative Sonar output

The last two tasks, that are executed only if all the previous stages are successfully completed, are the packaging and staged deployment. These are also configured using the gitlab-ci scripting mechanism.

5 Conclusions

The current deliverable aimed to document the activities of Task 5.1 entitled "Technical Integration Points and Testing Plan" in the frame of Work Package 5. More specifically, the document aimed to:

- clarify the core technologies that have been selected per component that comprises the RAINBOW architecture,
- Collect & highlight the communication patterns that has been established among the components,
- discuss about integration and quality assurance planning that could be used.

These activities cover the objectives of WP5 and more specifically: a) to (pro-) actively handle software components integration issues through the design of a detailed, overall technical architecture, and the software integration and testing planning; and b) to integrate the different software components that are developed in the core technical work packages WP2, WP3 and WP4.

Testing also becomes important to ensure the quality of the delivery, both at a submodule level and as a whole integrated system. For this reason, in this Integration and Testing plan, we have opted for a combined approach between Top Down and Bottom Up techniques, resulting in a hybrid approach. With this, components are added in a controlled way and tested repetitively before the full integration and the behavior of the individual parts is clearer.

Additionally, this document examined how the testing & integration planning will aim to cover the various requirements of the project and its' use cases, with both high level goals and targeted testing for specific environments, using tools such as Fogify.

The key output of the current work has been the definition of a testing methodology for the previous list of components plus the technologies to be utilised to carry on with such tests, paying special attention to the integration plan and the special conditions surrounding the RAINBOW platform.

Future work in the project will provide further specification of the different parts of the platform which will be due by the end of June 2021).