



# **Rational decomposition and orchestration for serverless computing**

## **Deliverable D2.4**

### **Architecture and integration plan II**

**Version: 1.0**

**Publication Date: 30-June-2020**

#### **Disclaimer:**

The RADON project is co-funded by the European Commission under the Horizon 2020 Framework Programme. This document reflects only authors' views. EC is not liable for any use that may be done of the information contained therein.

**Deliverable Card**

|                             |  |
|-----------------------------|--|
| <b>Deliverable</b>          | D2.4   |
| <b>Title:</b>               | Architecture and integration plan II   |
| <b>Editor(s):</b>           | Stefania D'Agostini (ENG)  |
| <b>Contributor(s):</b>      | Stefania D'Agostini (ENG), Stefano Dalla Palma (TJD), Dario DI Nucci (TJD), Thomas F. Düllmann (UST), Giorgos Giotis (ATC), Pelle Jakovits (UTR), Mark Law (IMP), Hans Georg Næsheim (PRQ), Michael Wurster (UST), Lulai Zhu (IMP) |
| <b>Reviewers:</b>           | Mark Law (IMP), Alexandros Spartalis (PRQ)   |
| <b>Type:</b>                | R  |
| <b>Version:</b>             | 1.0  |
| <b>Date:</b>                | 30-June-2020   |
| <b>Status:</b>              | Final  |
| <b>Dissemination level:</b> | Public   |
| <b>Download page:</b>       | <a href="http://radon-h2020.eu/public-deliverables/">http://radon-h2020.eu/public-deliverables/</a>  |
| <b>Copyright:</b>           | RADON consortium   |

**The RADON project partners**

|            |   |
|------------|---|
| <b>IMP</b> | IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY AND MEDICINE   |
| <b>TJD</b> | STICHTING KATHOLIEKE UNIVERSITEIT BRABANT   |
| <b>UTR</b> | TARTU ULIKOOL   |
| <b>XLB</b> | XLAB RAZVOJ PROGRAMSKE OPREME IN SVETOVANJE DOO   |
| <b>ATC</b> | ATHENS TECHNOLOGY CENTER ANONYMI BIOMICHANIKI EMPORIKI KAI TECHNIKI ETAIREIA EFARMOGON YPSILIS TECHNOLOGIAS |
| <b>ENG</b> | ENGINEERING - INGEGNERIA INFORMATICA SPA  |
| <b>UST</b> | UNIVERSITAET STUTTGART  |
| <b>PRQ</b> | PRAQMA A/S  |

**The RADON project (January 2019 - June 2021) has received funding from the European**



## **Executive summary**

This document presents the final version of the RADON technical architecture and integration plan. The document outlines the functionalities of the RADON components that have been updated during the latest period of the project along with their interactions. Moreover, the document presents the final integration approach adopted within the RADON framework and the testing activities performed to test both individual RADON tools and the interactions between these tools.

## Glossary

|        |  |
|--------|--|
| CDL    | Constraint Definition Language             |
| CI/CD  | Continuous Integration/Continuous Delivery |
| CLI    | Command Line Interface                     |
| CSAR   | Cloud Service Archive                      |
| CTT    | Continuous Testing Tool                    |
| DPT    | Defect Prediction Tool                     |
| DT     | Decomposition Tool                         |
| FaaS   | Function-as-a-Service                      |
| GMT    | Graphical Modeling Tool                    |
| IaC    | Infrastructure-as-Code                     |
| IDE    | Integrated Development Environment         |
| OpsEng | Operations Engineer                        |
| QoSEng | QoS Engineer                               |
| SUT    | System under test                          |
| TI     | Test infrastructure                        |
| TLPS   | Template Library Publishing Service        |
| VCS    | Version control system                     |
| VT     | Verification Tool                          |

---

## Table of contents

|  |    |
|--|----|
| 1. Introduction  | 8  |
| 1.1. Deliverable objectives  | 8  |
| 1.2. Overview of main achievements   | 8  |
| 1.3. Structure of the document   | 8  |
| 2. Reference technologies (Update)   | 10 |
| 2.1. Source Code - GitHub  | 12 |
| 2.2. CI/CD project infrastructure  | 14 |
| 3. RADON Architecture Revision & Update                                    | 18 |
| 3.1. Overview  | 18 |
| 3.2. Integration approach  | 20 |
| 3.3. RADON models flow (Update)  | 21 |
| 3.4. Actors and tools in RADON workflows                                   | 23 |
| 4. RADON Components (Update)   | 25 |
| 4.1. Constraint Definition Language (CDL) and associated Verification Tool | 25 |
| 4.2. Continuous Testing Tool   | 28 |
| 4.3. RADON Integrated Development Environment                              | 32 |
| 4.4. Graphical Modeling Tool   | 35 |
| 4.5. Template Library  | 38 |
| 4.6. Decomposition Tool  | 40 |
| 4.7. Defect Prediction Tool  | 42 |
| 4.6. Orchestrator  | 43 |
| 4.9. Delivery Toolchain  | 46 |
| 5. Testing   | 49 |
| 5.1. Functional Testing  | 49 |
| 5.2. Integration Testing   | 52 |
| 5.2.1. Continuous Testing Tool - Graphical Modeling Tool                   | 53 |
| 5.2.2. Continuous Testing Tool - Orchestrator                              | 54 |
| 5.2.3. Continuous Testing Tool - Monitoring                                | 55 |
| 5.2.4. Decomposition Tool - IDE  | 55 |

|        |   |    |
|--------|---|----|
| 5.2.5. | Defect Prediction Tool - RADON IDE          | 57 |
| 5.2.6. | Function Hub - RADON IDE                    | 58 |
| 5.2.7. | Function Hub - Orchestrator                 | 58 |
| 5.2.8. | Verification Tool - RADON IDE               | 59 |
| 5.2.9. | Verification Tool - Graphical Modeling Tool | 59 |
| 6.     | Conclusions                                 | 61 |
| 7.     | References                                  | 62 |

## 1. Introduction

This document presents the results of the activities performed within task T2.2: Integration plan, tests and documentation of WP2 concerning RADON technical architecture, integration practices and integration tests. This document updates the previous deliverable D2.3[1] and describes the final version of the RADON architecture and integration approach.

### 1.1. Deliverable objectives

This deliverable has the following main objectives:

- Describe the final RADON framework technical architecture;
- Outline the final integration approach adopted to integrate the RADON tool within the RADON Integrated Development Environment (IDE);
- Describe the testing activities performed to ensure that the RADON components properly interact and communicate correctly between them to accomplish the envisaged functionalities and workflows.

### 1.2. Overview of main achievements

The main achievements of the work reported in this deliverable are:

- The final RADON framework technical architecture, including:
  - an update of the functionalities provided by the RADON components described in D2.3[1] and an update of the interactions between these components to achieve project's goals;
  - an update of the integration approach adopted to integrate the RADON components in the RADON IDE;
  - an update on how the RADON models are exposed to RADON tools;
- The CI/CD project infrastructure to build the RADON integrated framework;
- The description of the testing activities that have been performed to validate the functional specifications of each RADON tool and to guarantee the proper interaction and communication between them.

### 1.3. Structure of the document

The rest of this deliverable is structured as follows:

- **Section 2** gives an updated overview on the reference technologies adopted in the RADON framework and outlines the CI/CD project infrastructure;
- **Section 3** presents the final architecture of the RADON framework and the adopted integration approach;
- **Section 4** describes the final set of tools being part of the RADON framework providing an update on their functionalities and interactions;

- **Section 5** outlines the testing activities carried out during the project to test both the individual RADON tools and the interaction between the tools;
- **Section 6** draws the conclusions and outlines, as well as expected follow-up actions after this deliverable.

## 2. Reference technologies (Update)

In the deliverable D2.3 [1] a detailed survey of the reference technologies selected for adoption in the RADON framework was provided. The consortium keeps this list updated as our interaction with external stakeholders suggest that technology support is a recurring concern in potential framework adopters.

Table 1 reports and overview of such technologies and whether any updates have been made, respect the previous deliverable.

**Table 1.** RADON reference technologies

| Technology Type              | Technology   | Description  | RADON support  |
|------------------------------|--|--|--|
| Programming languages        | Java, Node.js  | This choice arises from: (a) the needs of the use case providers regarding their own applications; (b) the full exploitation of the advantages brought by microservices / serverless ecosystems.                         | Java is essential to integrate backend services in the microservices-based architecture.   |
| FaaS frameworks and services | OpenFaaS, OpenWhisk, AWS Lambda, Google Cloud Functions, Microsoft Azure Functions | OpenFaaS and OpenWhisk are FaaS-oriented frameworks popular in the open-source community. AWS Lambda, Google Cloud Functions and Microsoft Azure Functions are FaaS compute services offered by public cloud platforms.  | OpenFaaS and AWS Lambda are holistically supported. We plan to offer some degree of support for OpenWhisk and Google Cloud Functions for deployment. Microsoft Azure Functions could be eventually supported as well but with lower implementation priority than OpenWhisk and Google Cloud Functions. |
| Data pipelines               | Apache NiFi, AWS Data Pipeline   | Apache NiFi is an open source platform for automating the movement and transformation of data and supports consuming or delivering data to a large number of external data sources and services. AWS Data Pipeline is an | Apache NiFi is fully supported; data pipelines can be designed in GMT tool and the resulting service template can be deployed using Orchestrator. AWS Data Pipeline service is not yet supported, but it will be   |

|                             |                    |   |   |
|-----------------------------|--------------------|---|---|
|                             |                    | Amazon cloud service for defining and orchestrating data flows between a small selection of AWS services and data sources.  | introduced in the future.   |
| Object stores and databases | AWS S3             | AWS Simple Storage Service (S3) is a cloud object storage service that offers data durability, scalability and availability.  | AWS S3 allows us to build event-driven cloud applications by automatically executing serverless functions to respond to specific events such as object inserts and updates. It is fully supported in the RADON framework as an event emitting component.            |
| Container Virtualization    | Docker             | Docker is an open source container orchestration platform to coordinate deployments of, scale, and manage multiple independent containers.  | RADON microservices-based applications can be deployed on Docker containers.  |
| VM deployment               | AWS EC2, OpenStack | OpenStack is an open standard IaaS cloud computing platform which is used to set up and manage compute pools, storage, networking resources and mostly virtual machines through a simple user interface.<br>AWS EC2 is an AWS component that provides scalable computing capacity which is used to bring up instances (VMs) where cloud applications can be deployed. | OpenStack can be set up using Ansible and xOpera orchestrator. The same stands for AWS EC2 where an existing cloud Ansible core modules can be used to initiate orchestration and manage EC2 instances.   |
| Infrastructure as Code      | TOSCA, Ansible     | TOSCA stands for the Topology and Orchestration specification for Cloud Applications, and is the standard for defining infrastructure topologies.<br>Ansible is a resilient infrastructure code format which allows interoperability as an intermediate language between TOSCA and accessory formats  | On the one hand, we chose Ansible as the most popular IaC language in industry. On the other hand we selected TOSCA as a reference baseline technology used across RADON tools and framework with key benefits such as interoperability, technology-agnosticism and |

|       |                      |  |   |
|-------|----------------------|--|---|
|       |                      | such as Kubernetes, Docker, Chef, or Puppet. Ansible features tasks into repeatable playbooks which can be referenced at will.   | portability.  |
| CI/CD | Jenkins <sup>1</sup> | Jenkins is a free and open source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery. | The RADON IDE can be configured to support CI/CD through Jenkins. |

## 2.1. Source Code - GitHub

This section describes the public repository where the software artifacts developed in the RADON project are stored and made available through the GitHub platform.

The project is hosted on GitHub as an organization exposed at: <https://github.com/radon-h2020>. The organization offers at the time of writing of this deliverable about 38 repositories, which are centrally managed by the coordinator. The following naming conventions are used:

| Dimension                                | Convention   |
|--|--|
| <i>Case</i>                              | All folders and repositories must be in lower case   |
| <i>Forked repositories</i>               | Forks can retain the name of the original folder   |
| <i>Tool repositories</i>                 | A tool folder must be named eg "radon-xyz" where "xyz" should be replaced as needed  |
| <i>Demo applications / examples</i>      | A folder with a demo application, or relevant examples, must be named eg "demo-xyz" where "xyz" should be replaced as needed |
| <i>Industrial use case demonstrators</i> | A folder with relevant industrial use case artifacts should be named "usecase-xyz", where "xyz" should be replaced as needed |

Tool repositories adopt a similar structure in terms of basic README.md documentation and Wiki that provides basic installation and getting started instruction. The main tool repositories are:

- Continuous testing: <https://github.com/radon-h2020/radon-ctt>
- Data pipelines: <https://github.com/radon-h2020/radon-datapipeline-plugin>

<sup>1</sup> Jenkins - <https://www.jenkins.io>

- Decomposition: <https://github.com/radon-h2020/radon-decomposition-tool>
- Defect prediction: <https://github.com/radon-h2020/radon-defect-prediction-api>
- GMT: <https://github.com/radon-h2020/winery>
- IDE: <https://github.com/radon-h2020/radon-ide>
- Orchestrator: <https://github.com/radon-h2020/xopera-opera>
- Verification: <https://github.com/radon-h2020/radon-verification-tool>
- Delivery Toolchain: <https://github.com/radon-h2020/radon-delivery-toolchain>
- FunctionHub: <https://github.com/radon-h2020/radon-functionhub-client> (open-source)  
<https://cloudstash.io/> (publishing service)
- Monitoring System: <https://github.com/radon-h2020/radon-monitoring-tool>
- Template Library: <https://github.com/radon-h2020/radon-particles> (open-source)  
<https://template-library-radon.xlab.si/> (publishing service)

To help navigate the many repositories, a central repository listing the key assets available to end users is given at <https://github.com/radon-h2020/radon-methodology> which acts as a “landing page” for end users interested to learn the basics about RADON.

End users can interact in the “landing page” via the *Issues* feature to put forward to the consortium general queries about RADON. More detailed technical questions about a specific tool are instead meant to be triggered on the *Issues* of individual repositories, which also consolidate the tool requirements.

**Figure 1.** A portion of the RADON GitHub landing page<sup>2</sup> presenting a tool

## RADON Tools

### Defect Prediction Tool

| Items                | Contents  |
|----------------------|---|
| Short Description    | The Defect Prediction tool focuses on Infrastructure-as-Code (IaC) correctness. Recall that IaC is machine-readable code that manages and provisions infrastructure -- e.g., TOSCA or Ansible YAML files. The defect prediction tool helps RADON users to find suspicious defective Infrastructure-as-Code (IaC) scripts enabling DevOps operators to focus on such critical scripts before deployment and during Quality Assurance activities. |
| Documentation        | WIP -- D2.3 – Architecture & Integration Plan I   |
| Stand-Alone Tutorial | <a href="https://radon-h2020.github.io/radon-defect-prediction-api/">https://radon-h2020.github.io/radon-defect-prediction-api/</a>   |
| Video                | <a href="https://tinyurl.com/y8mtyszm">https://tinyurl.com/y8mtyszm</a>   |
| Repositories         | <ul style="list-style-type: none"> <li>• APIs;</li> <li>• Web Application;</li> <li>• Visual Studio Extension;</li> <li>• AnsibleMetrics;</li> </ul>  |
| Docs                 | <ul style="list-style-type: none"> <li>• APIs;</li> <li>• AnsibleMetrics;</li> </ul>  |
| Licence              | Apache License, Version 2.0   |
| Contact              | <ul style="list-style-type: none"> <li>• Damian A. Tamburri (<a href="mailto:d.a.tamburri@tue.nl">d.a.tamburri@tue.nl</a>)</li> <li>• Stefano Dalla Palma (<a href="mailto:s.dalla.palma@jads.nl">s.dalla.palma@jads.nl</a>)</li> </ul>   |

<sup>2</sup><https://github.com/radon-h2020/radon-methodology>

The Github *Issues* feature is also used internally to track the project requirements status and synchronize the work of different units.

## 2.2. CI/CD project infrastructure

We use Jenkins as the CI/CD infrastructure on our project. This infrastructure was made available by ENG and hosted by FIWARE<sup>3</sup> and includes two virtual machines: a “jenkins-master” VM schedules the jobs, assigns slaves, sends builds to that slaves and gets back the build results displaying them on the output console, and a “jenkins-slave” VM where the jobs are executed.

The main characteristics of these VMs are reported below:

- **Jenkins-master:** a Centos 7 VM with a public IP address and reachable at the URL: <http://operations.radon-h2020.eu:8080/>. This VM has 4 GB RAM and 40 GB disk space. The installed software is: Java 11, Docker<sup>4</sup> and Jenkins installed as a Docker container.
- **Jenkins-slave:** a Centos 7 VM with 32 GB RAM and 80 GB disk space. The installed software is: Java 11, Python, Pip, Docker and Jenkins installed as a Docker container and configured as a slave node.

The procedure followed to automate the build and deployment process in RADON is the following:

- **Code uploading in the Git repository.** As a first step, the source code of the software artifacts developed in the RADON project has to be pushed to the Git repositories described in the Section 2.1;
- **Jenkins job request.** Once a repository contain everything needed to build, deploy, and test the components, an email should be sent to ENG adding details concerning (i) the repository path, (ii) the name of users that needs full control of the jobs, and (iii) the persons that will receive notifications on build results by email;
- **Jenkins job configuration:** After having received the job request, ENG creates an account (if not yet present) on the Jenkins server for the users that need access to the job and sets up the new job (as a pipeline).

Once created the Jenkins jobs the developers have full control on the jobs related to their components. In this way, each partner can define the different steps of each pipeline, view building logs (see Figure 2), fix job configuration errors and investigate problems (if any) by examining the deployed components workspace through Jenkins (see Figure 3).

---

<sup>3</sup> FIWARE - <https://www.fiware.org/>

<sup>4</sup> Docker - <https://www.docker.com/>

The jobs are configured to enable automated build and testing and can be triggered when a new change is pushed in the Git repositories or defining a schedule in the job configuration (e.g., every day).

Figure 2. Jenkins output console

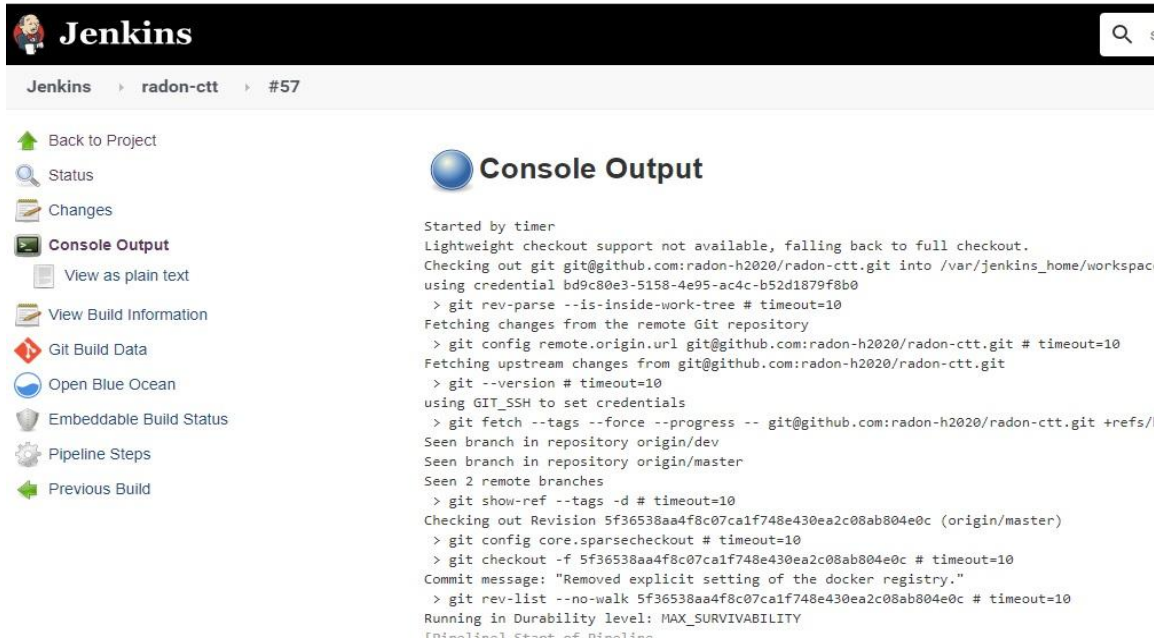
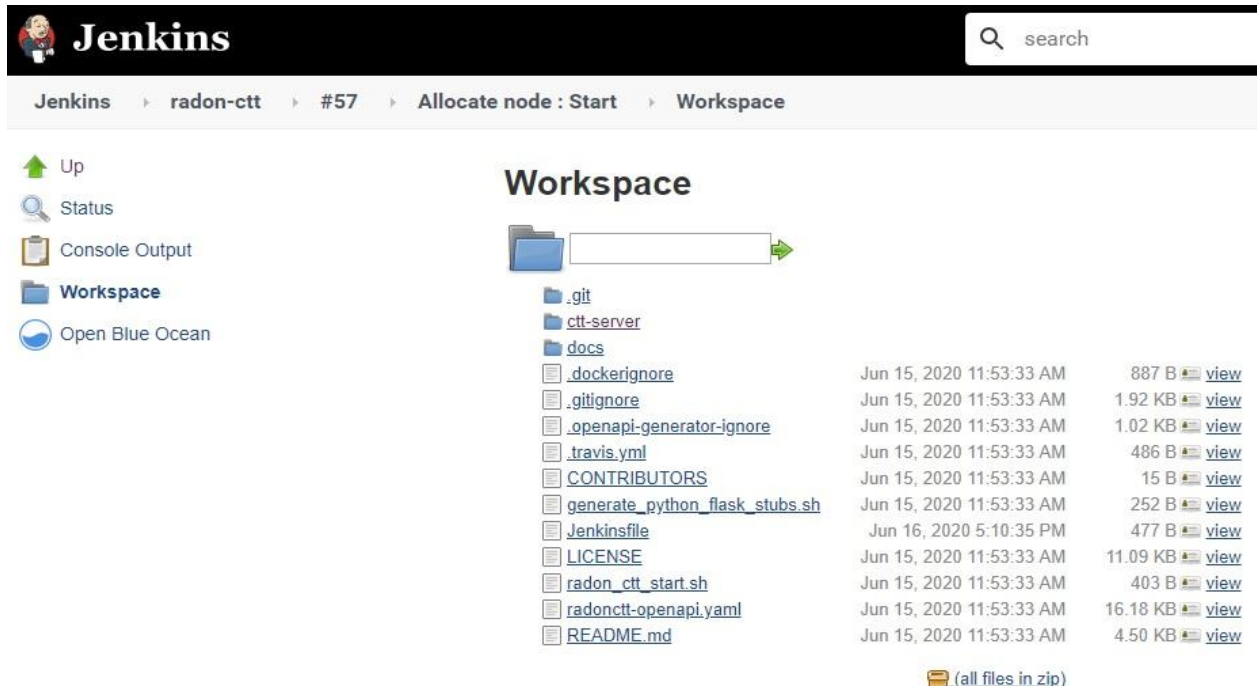
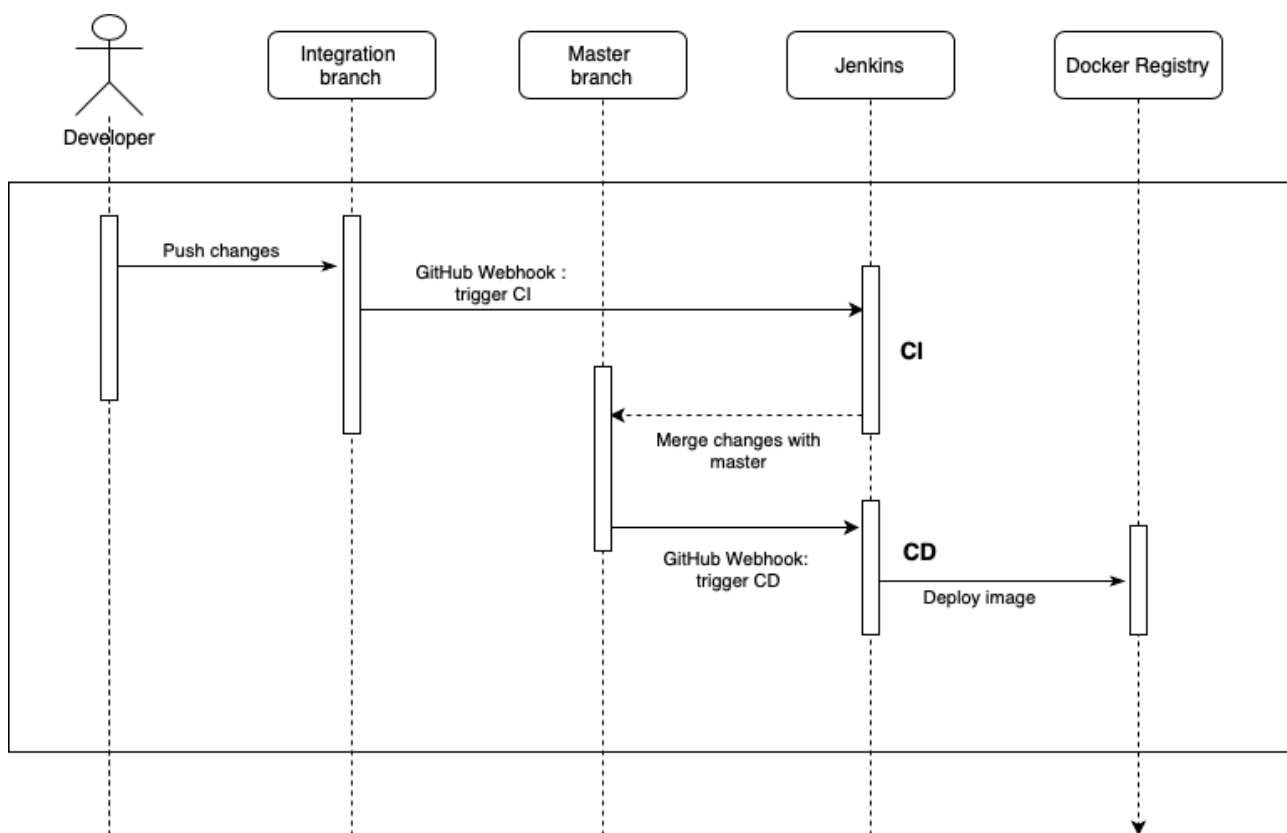


Figure 3. Jenkins job workspace



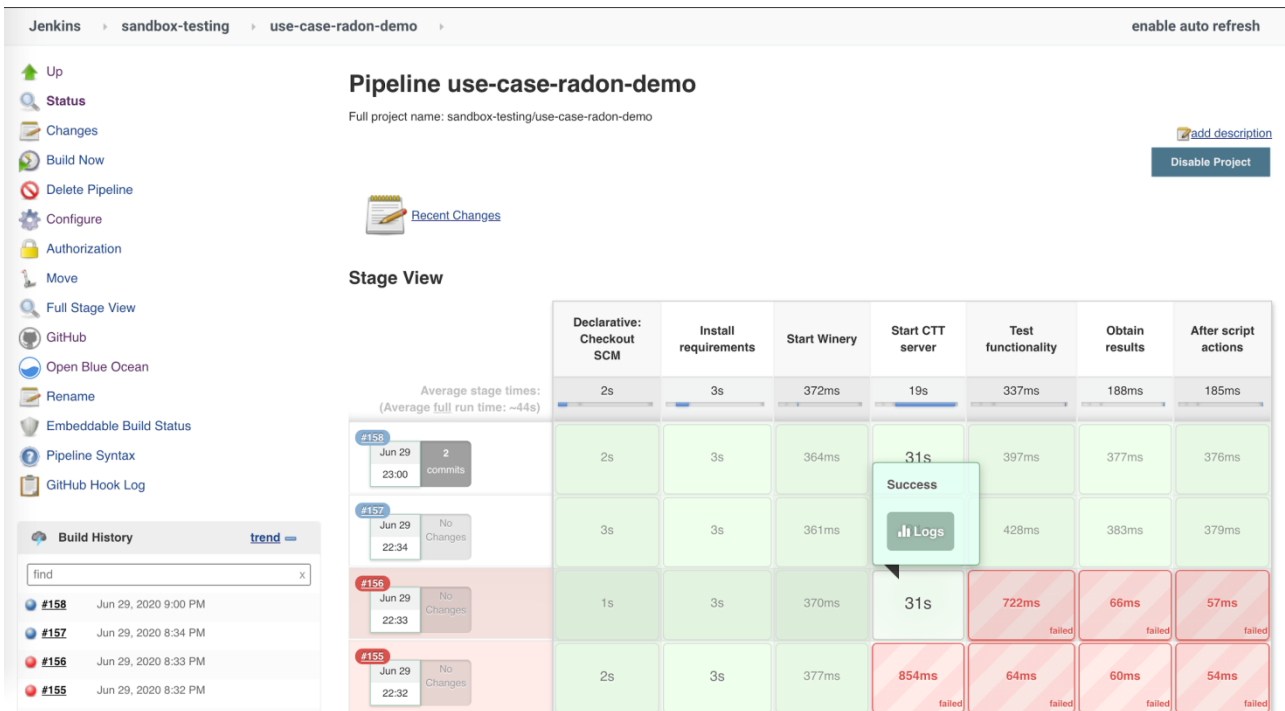
**Building the RADON integrated framework.** Jenkins serves two important functionalities: firstly, a platform for integrating changes to the code base and furthering the deployment of the next release; and secondly, a service for demoing the tool functionality both individually and in combination with related RADON tools. So far, the Consortium has focused on the first point, where every commit is picked up by Jenkins, tested and deployed. Tools within RADON are packaged and made available through distinct formats. The Orchestrator and Function Hub client are available as PyPi packages and CTT, GMT as Docker images. As an example, Figure 4 depicts how UST is deploying their tool, CTT to the ‘radonconsortium’ repository at Docker Hub. The latest version of CTT is now available at DockerHub with a traceable tag to the Jenkins build and the related git commit.

**Figure 4.** Continuous Integration/Deployment



Jenkins provides the benefit of visibility of build status through metrics. Figure 5 depicts a report of build status (fail/pass) in every stage of the CI/CD pipeline for the CTT. Moreover, various metrics and additional information are provided, such as build times, timestamp of the build, logs and associated with the build commits.

Figure 5. Report of build status for the CTT



### 3. RADON Architecture Revision & Update

This chapter describes the final architecture of the RADON framework. A preliminary presentation of the architecture and the involved RADON components has been provided in D2.3[1]. The focus here is to describe the final set of functionalities provided by these components and how they interact with each other in order to achieve the project's objectives, according to the RADON workflows identified in D2.3[1]. In particular, Section 3.1 provides an overview of the components involved in the final version of the RADON architecture and how they are connected. Section 3.2 presents the integration approach adopted to integrate some of these components within the RADON IDE. In Section 3.3, we give an update on how the RADON models are exposed to the RADON tools (i.e. how these models are stored, how they are read, updated and shared). Finally, Section 3.4 provides the final description of how RADON actors and tools are involved in the RADON workflows.

#### 3.1. Overview

The RADON framework provides a set of components that realize a set of tools, modules and services covering both the design and runtime phases of the application development and deployment. Table 2 lists these components, whose updated functionalities and interactions (respect the ones reported in D2.3[1]) are described in detail in Chapter 4.

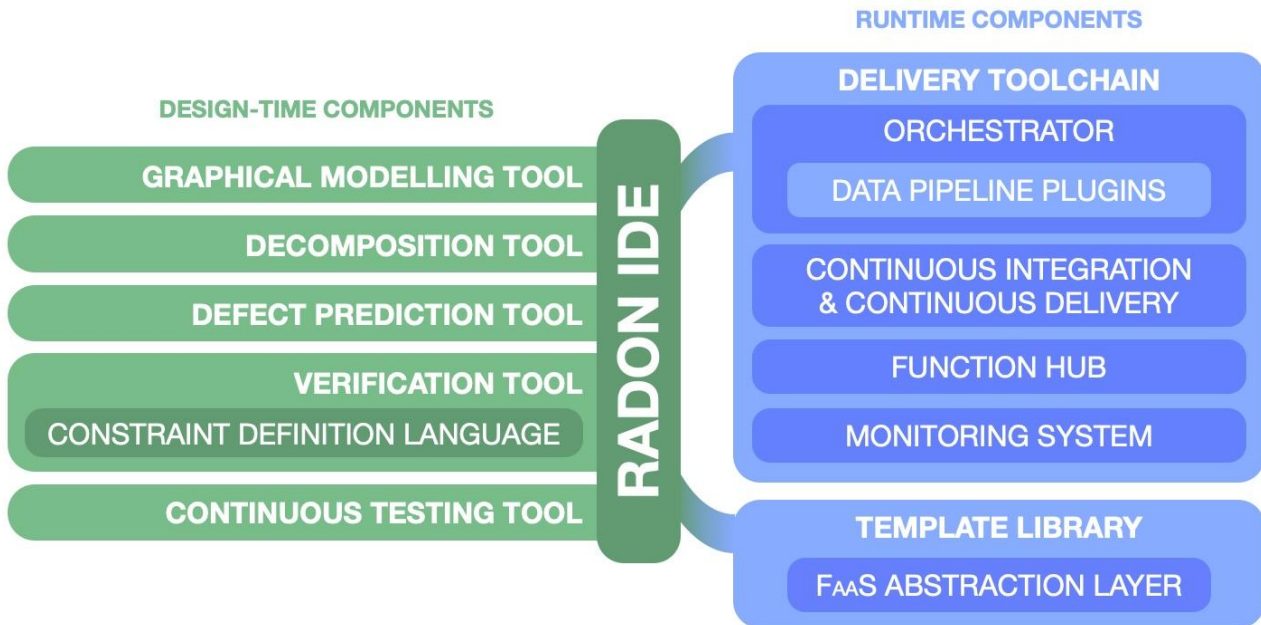
**Table 2.** An overview of the RADON components

| Name                           | Description   | Design time/Runtime |
|--------------------------------|---|---------------------|
| Constraint Definition Language | A language for specifying formal requirements on RADON models.  | Design time         |
| Continuous Testing Tool        | A tool for continuous design, evolution, deployment, and execution of tests.  | Design time         |
| Data Pipeline Plugins          | A plugin that extends the orchestrator with the capability to automate the movement and transformation of data.                 | Runtime             |
| Decomposition Tool             | A tool for architecture decomposition, deployment optimization and accuracy enhancement.  | Design time         |
| Defect Prediction Tool         | A tool that Focuses on Infrastructure-as-Code (IaC) correctness, code smells detection and refactoring at IaC blueprints level. | Design time         |

|  |  |             |
|--|--|-------------|
| Delivery Toolchain                       | A set of tools for software delivery.  | Runtime     |
| FaaS abstraction layer                   | A module that holds the definitions required to deploy a particular application component to different cloud providers.  | Runtime     |
| Function Hub                             | A public repository for open sourced contribution to reusable functions.   | Runtime     |
| Graphical Modeling Tool                  | A web-based tool to graphically model TOSCA applications.  | Design time |
| Monitoring System                        | A back-end service-based system to collect evidence from the runtime environment for supporting quality assurance on performance and security characteristics. | Runtime     |
| Orchestrator                             | A tool for processing and executing the TOSCA service templates packaged in a Cloud Service Archive (CSAR).  | Runtime     |
| RADON Integrated Development Environment | A web-based development environment for multi-user usage that integrates some of the RADON tools.  | Design time |
| Template Library                         | A common repository for templates, blueprints and modules required for the application deployment.   | Runtime     |
| Verification Tool                        | A tool for verifying whether a RADON model conforms to a specification expressed in the Constraint Definition Language.  | Design time |

Figure 6 depicts the final version of the RADON Architecture highlighting the connections among the RADON (design-time/runtime) components which interact with each other in order to design, prototype, deploy and test applications built on serverless FaaS.

Figure 6. Final RADON Architecture



As already stated in D2.3 [1], in this context a particular role is played by the RADON IDE. This component (based on the Eclipse Che technology) provides a multi-user development environment acting as a front-end of the RADON methodology. Indeed, as depicted in Figure 6, the RADON IDE (i) integrates the design-time tools so that the RADON user can invoke and use them within the development environment, (ii) enables the interaction with the Template Library to make the RADON artifacts available to RADON tools, and (iii) provides to the RADON user a connection with the runtime phases of application development.

### 3.2. Integration approach

This section describes the integration approach adopted to integrate the components and develop the RADON framework. In particular, we take into consideration two different integration activities: one related to the integration between different tools developed within WP3-WP5 (inter-tool integration) and another one concerning the integration of these tools within the RADON IDE (IDE integration). The inter-tool integration is described in Chapter 4 by means of sequence diagrams; here, instead, we focus on the final plug-in architecture adopted to customize the RADON IDE and integrate on it the RADON (design-time) tools.

The RADON IDE is based on the Eclipse Che technology which is an extensible and customizable platform. Eclipse Che permits to customize the development environment using a “devfile” where the configurations of the workspaces are defined to include projects, components and commands (see D2.6 [2] for details). In order to integrate the RADON tools on the IDE we implemented new plugin or kubernetes components, which are then referenced in the devfiles. The (Che) plugins add capabilities to the Che-Theia IDE (i.e. the native integrated development environment for Eclipse Che) while the kubernetes components add containers (i.e. kubernetes pods) directly in the developer workspaces. Therefore, in our context, some RADON components defined new Che

plugins to extend the Eclipse Che GUI (for example, to add new commands enabling the user to interact with them), while other components used Kubernetes components in order to integrate services of the tools in the IDE "backend".

Table 3 provides an update of the one reported in D2.3[1], indicating for each RADON tool directly integrated on the RADON IDE which kind of packaging (plugin and/or kubernetes component) is used.

**Table 3.** Integration of RADON components in the RADON IDE

| <b>RADON Component</b>  | <b>Integrated on the RADON IDE</b> | <b>Type of packaging</b>        |
|-------------------------|------------------------------------|---------------------------------|
| Verification Tool       | Yes                                | Kubernetes component and Plugin |
| Continuous Testing Tool | Yes                                | Plugin                          |
| Graphical Modeling Tool | Yes                                | Kubernetes component            |
| Template Library        | No                                 | Standalone                      |
| Decomposition Tool      | Yes                                | Plugin                          |
| Defect Prediction Tool  | Yes                                | Plugin                          |
| Delivery Toolchain      | No                                 | Standalone                      |

A detailed description of the IDE integration approach developed so far can be found in the deliverable D2.6 [2].

As also stated in D2.3[1], the development environment is also customized in order to add some graphical elements (e.g. menus, commands) enabling a RADON user to interact with the runtime components not directly integrated into the RADON IDE (e.g. to open a browser window to show the deployment status and the monitoring results).

### **3.3. RADON models flow (Update)**

The RADON Models are shared entities and the RADON partners jointly contribute to provide demo application and example of executable and deployable RADON-based applications. In the period between the D2.3 (Architecture and integration plan I) and this deliverable, the RADON

consortium realized based on changed requirements that RADON models are not only shared via version control system (VCS) repositories. To position the RADON models with respect to DevOps and modern software engineering processes, we foresee three different options to manage and share RADON Models: (1) For single RADON users or in the early stages of design and development, it must be able to purely store and manage the models on the local filesystem. (2) In a typical DevOps environments, software developers work in (distributed) teams. Therefore, it must be possible to share and manage the models in public or private VCS repositories. (3) For large teams or enterprises, it must be possible to utilize a layer on top introducing additional, enterprise-ready features such as release management of final models or user management with comprehensive access control lists. Similar to well-known package managers such as “pypi” (Python) or “npm” (NodeJS), we foresee a publishing services, called RADON Template Publishing Service (D5.3[3]), that can be used to publish new RADON Models versions with certain visibility permissions.

These different persistence options are conceptually what we summarize with the term “Template Library”. In all three cases the Template Library contains (i) reusable TOSCA entity types, such as TOSCA node types, relationship types, or policy types as reusable modeling entities, (ii) respective TOSCA implementations attached to TOSCA node and relationship types (Ansible Playbooks) to make those types executable, and (iii) RADON Models, i.e., application blueprints or TOSCA service templates, which are composed of reusable TOSCA entity types.

On top of this, the RADON consortium uses a public and open-source Template Library in the form of a GitHub repository, the RADON Particles<sup>5</sup>. The RADON Particles essentially fulfill two use cases. On the one hand, it serves as an example of the Template Library, on the other hand, it contains reusable TOSCA type entities as a baseline to model RADON applications. Further, the RADON Particles also contain executable RADON Models, e.g., the toy example application, showing examples and demos on how to utilize the RADON Particles as modeling baseline.

Therefore, the Graphical Modeling Tool (GMT) populates the RADON Particles inside the RADON IDE and creates a so-called “modeling project”. RADON users immediately can start modeling applications based on the types and templates offered and developed by the RADON consortium. Notably, the created “modeling project” is structured in such a way that RADON users can publish custom TOSCA entity types as well as created RADON Models in a separate VCS, e.g., a private GitHub or GitLab repository. Further, the GMT will be able to push RADON Models to the RADON Template Publishing Service once users decide to finalize a model and put it into production. Similarly, the GMT will be able to pull a model from the service to modify it and push it as a new version back. This step forward, for example, enables the RADON Orchestrator to directly utilize and deploy managed RADON Models by this service. However, at any point in time, RADON users could use the GMT to generate a TOSCA CSAR of a RADON Model. This further ensures the integration capability with other RADON tools, such as the Verification Tool, Defect Prediction Tool, or Decomposition Tool.

---

<sup>5</sup> <https://github.com/radon-h2020/radon-particles>

### 3.4. Actors and tools in RADON workflows

In the deliverable D2.3 [1] we presented a description of how the actors and the RADON tools are involved in the RADON workflows identified by the RADON methodology. As stated in the previous deliverable, a workflow is a repeatable sequence or pattern of actions that involve the interaction of several RADON tools and users that can be triggered either by direct interaction of a RADON user with the RADON IDE (e.g., verifying constraints on a design) or by a specific RADON tool as a consequence of an event (e.g., changing the deployment given monitoring information). While some interaction between the tools have been updated (as detailed in Chapter 4) no particular changes have been identified for the actors and tools involved in the RADON workflows, therefore the following table is supplied to make the deliverable self-contained, but we point to D2.3[1] for additional details.

Table 4 gives an overview of these workflows and the RADON actors and tools involved on them.

**Table 4.** An overview of RADON workflows with involved actors and tools

| <b>RADON workflow</b>             | <b>Actors<sup>6</sup></b>                     | <b>Tools</b>  |
|-----------------------------------|---|---|
| Verification                      | Software Designer, QoS Engineer               | <ul style="list-style-type: none"> <li>● Graphical Modeling Tool</li> <li>● Integrated Development Environment</li> <li>● Constraint Definition Language</li> <li>● Verification Tool</li> </ul>                  |
| Decomposition                     | Software Designer, Ops Engineer, QoS Engineer | <ul style="list-style-type: none"> <li>● Graphical Modeling Tool</li> <li>● Decomposition Tool</li> <li>● Function Hub</li> <li>● Template Library</li> <li>● Monitoring System</li> </ul>                        |
| Defect Prediction                 | Developer, Ops Engineer                       | <ul style="list-style-type: none"> <li>● Defect Prediction Tool</li> <li>● Integrated Development Environment</li> </ul>  |
| Continuous Testing                | Developer, Release Manager, QoS Engineer      | <ul style="list-style-type: none"> <li>● Continuous Testing Tool</li> <li>● Integrated Development Environment</li> <li>● Monitoring System</li> </ul>  |
| Continuous Integration/Deployment | Ops Engineer, Release Manager                 | <ul style="list-style-type: none"> <li>● Graphical Modeling Tool</li> <li>● Orchestrator</li> <li>● Integrated Development Environment</li> <li>● Continuous Testing Tool</li> <li>● Monitoring System</li> </ul> |
| Monitoring                        | OpsEng, QoSEng                                | <ul style="list-style-type: none"> <li>● Graphical Modeling Tool</li> <li>● Orchestrator</li> <li>● Monitoring System</li> </ul>  |

<sup>6</sup> Actors were identified in D2.1 [9]



**Deliverable 2.4: Architecture and integration plan II**

|  |  |  |
|--|--|--|
|  |  | <ul style="list-style-type: none"><li>● Decomposition Tool</li><li>● Continuous Testing Tool</li></ul> |
|--|--|--|

## 4. RADON Components (Update)

A detailed description on the RADON tools and their interactions in the context of the RADON architecture has been provided in the deliverable D2.3 [1].

The section describes the final version of this set of tools developed within the project, providing an update of their functionalities and interactions (if any).

### 4.1. Constraint Definition Language (CDL) and associated Verification Tool

The CDL/VT plugin will provide functionality for the user to define CDL specifications, the space of potential extensions to the current RADON model, and the examples used by the VT for learning new constraints. The (containerised) backend to this tool will run the Verification Tool (VT), thus providing the functionality for verifying that the current RADON model conforms to the CDL specification. In cases where the current RADON model does not conform to the CDL specification, the VT can be used to suggest corrections to the model, which will be displayed to the user via the CDL/VT plugin. If the model does conform, the VT can be used to suggest improvements to it with respect to the preferences expressed as soft constraints in the CDL specification. The VT can also be used to find an optimal (preferred) completion of a partial RADON model, with respect to the CDL specification, and to learn new constraints for the CDL.

**Functionalities update:** The CDL/VT plugin was originally envisaged as two separate plugins (described in D2.3[1]); however, it has become clear over time that this separation is unnecessary, so there is now a single plugin incorporating the functionalities of both tools presented in D2.3 [1]. Other than this merger, there are no further updates to the functionalities.

**Interactions:** The merger of the two plugins leads to a significant simplification of the interactions described in D2.3[1]. Further, over the course of development, it has become clear that some of the previously envisaged interactions with the GMT (to import/update the RADON model) should actually be interactions with the RADON IDE, which stores the files being imported. The final interactions are shown below.

Figure 7. VT interactions - Import/update CDL specifications

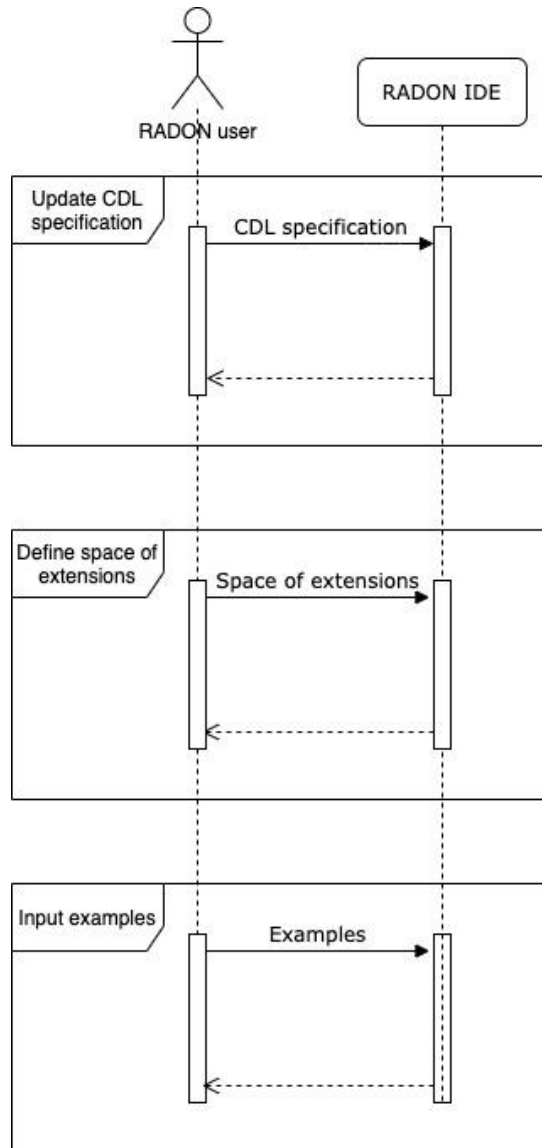
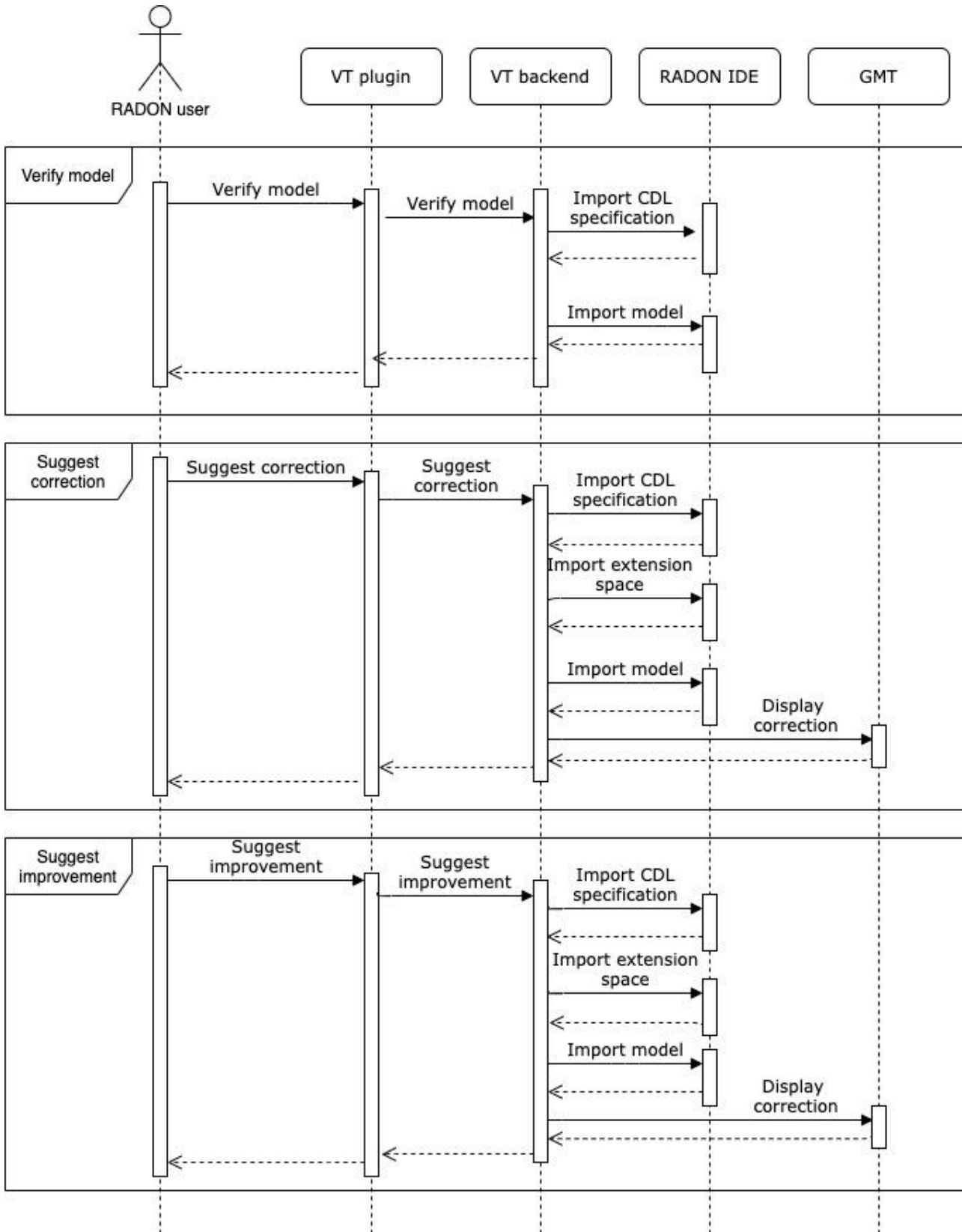
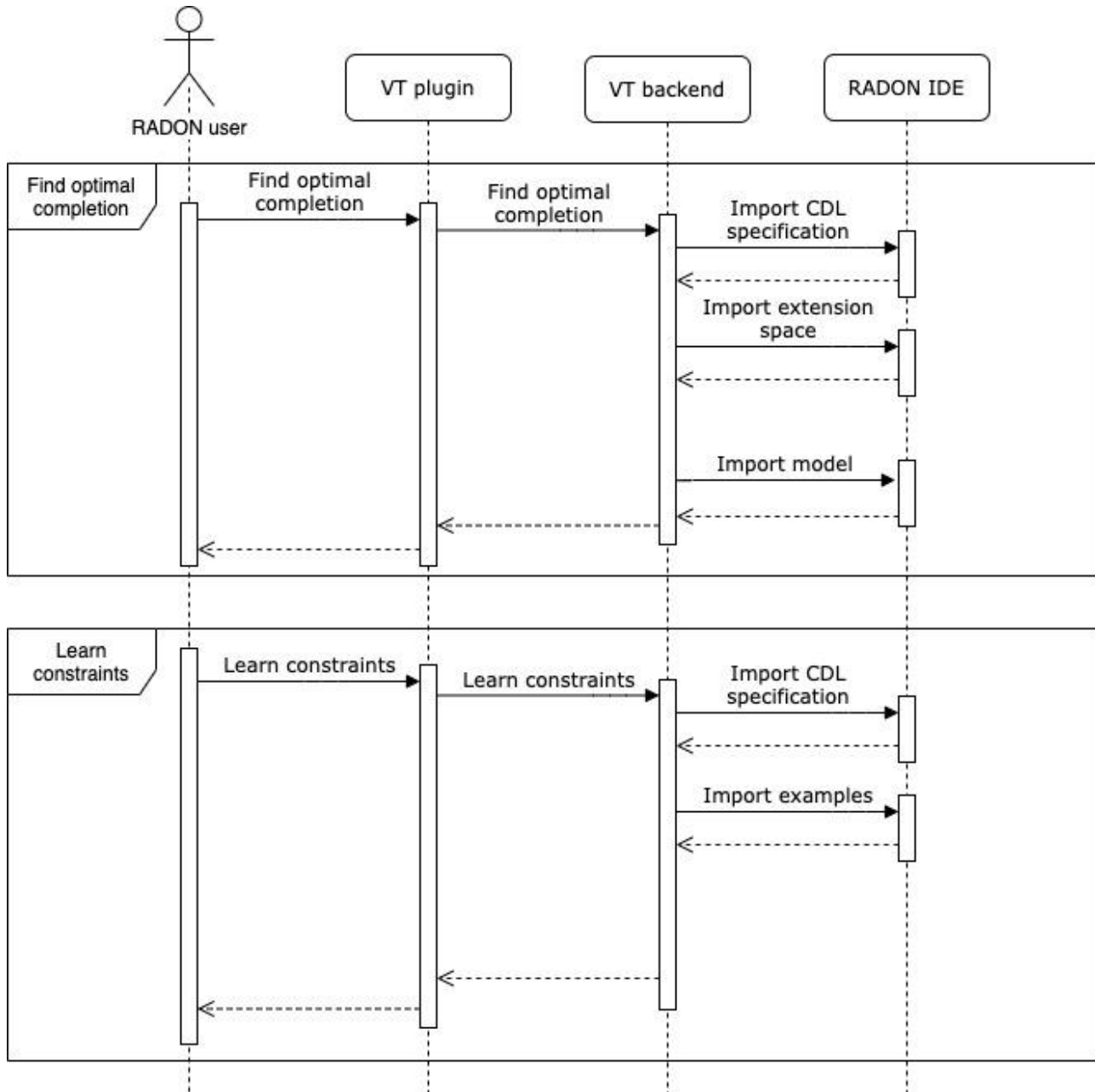


Figure 8. VT interactions - Verify models, suggest corrections and suggest improvement



**Figure 9.** VT interactions - Find optimal completion and learn constraints


## 4.2. Continuous Testing Tool

The Continuous Testing Tool (CTT) provides the functionality for defining, generating, executing, and refining continuous tests of application functions, data pipelines and microservices, as well as for reporting test results. While targeting to provide a general framework for continuous quality testing in RADON, a particular focus of CTT is on testing workload-related quality attributes such as performance, elasticity, and resource/cost efficiency.

---

**Functionalities update:**

No updates to CTT’s initially planned core functionalities (summarized above and detailed in D2.3[1]) was necessary. However, as part of the detailed architectural design, we refined the architecture particularly with respect to two aspects (detailed in D3.4[4]):

- *Entity-based interaction.* We decided to make CTT’s functionality accessible via a “entity-based” interaction. This means that the main entity for interaction of users with the tool is a *CTT project*. A CTT project contains all other CTT-related entities throughout the workflow, namely test artifacts, test deployments, test executions, and test results. Users (including tools) access CTT’s functionality via a workflow that is guided by these entities, e.g., the creation of a CTT project, followed by the generation of test artifacts in subsequent interactions.
- *Extension points.* CTT’s architecture provides explicit extension mechanisms for adding custom test types and test tools.

**Interactions:**

At runtime, CTT interacts with the following RADON tools: IDE/GMT, CI/CD, Orchestrator, and Monitoring.

- *RADON IDE/GMT.*
  - GMT is used to model the tests and the test infrastructures. CTT interacts with GMT via the exported CSAR files.
  - The RADON IDE will provide access to CTT’s functionality via the defined REST-based API.
- *Orchestrator (xOpera)*

CTT uses the xOpera orchestrator for the deployment of the system under test (SUT) and the test infrastructure (TI). CTT interacts with xOpera via xOpera’s API and the CSAR files to be deployed.
- *CI/CD*

CTT can be used for testing SUTs within their CI/CD pipelines. Therefore, the CI/CD pipeline invokes CTT via its API.
- *Monitoring<sup>7</sup>*

CTT acquires RADON’s Monitoring component to get access to runtime data for refining operational profiles.

Another interaction with RADON’s Particles Template library:

---

<sup>7</sup> The implementation of this interaction is subject to the next deliverable.

- *RADON Particles*

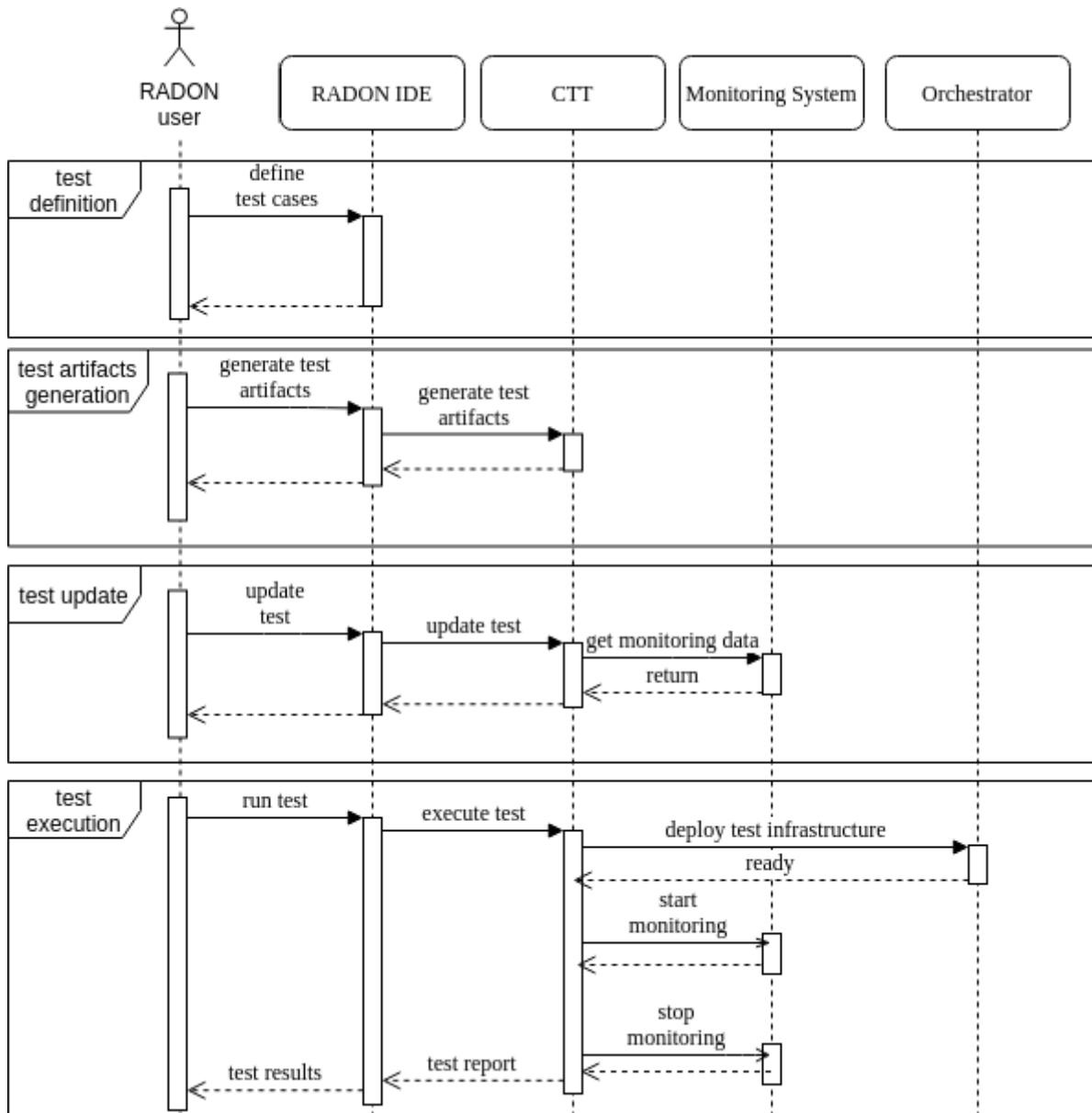
RADON Particles includes the CTT-related type hierarchy that also serves as the basis for custom extensions in the form of additional test types, tools, and blueprints. CTT interacts with RADON particles via the TOSCA-based models and implementation artifacts (detailed in D3.4[4]).

No major changes with respect to the interactions with other RADON tools compared to D2.3[1] were necessary.

While the CTT tool only provides a REST API to be interacted with, this allows a variety of integrations. The envisioned interactions are mostly the RADON IDE and the CI/CD pipeline. In addition, it is also possible to directly (manually) interact with CTT.

Due to the fact that only the tools interacting with CTT vary, but not the interaction with CTT itself, we provide an exemplary sequence diagram with the RADON IDE (Figure 10).

Figure 10. CTT interactions



The tests and their parameters are initially defined using the RADON IDE. In order to create test artifacts, the user initiates the test artifact generation in the RADON IDE which then triggers the test artifact generation in CTT.

The test update step pulls in monitoring results from the Monitoring System.

When the user triggers an update of a test using the RADON IDE it initiates the test update step in CTT which in turns retrieves monitoring data from the Monitoring System in order to refine the test parameters.

For the test execution the user interacts with the RADON IDE which then triggers the test execution in CTT. CTT then deploys the system under test and the test infrastructure based on the previously generated test artifacts. Before the actual tests are executed, monitoring is started in the Monitoring System. Afterwards, the monitoring is stopped.

In comparison to D2.3 [1] the sequence diagram is similar except that we do not have the RADON runtime as a mediator between the Monitoring System and the Orchestrator.

### **4.3. RADON Integrated Development Environment**

The RADON IDE provides standard functionalities to support development activities (e.g. debugging functionalities and error checking capabilities) along with specific functionalities to achieve the RADON needs. In this section we focus on the latter ones. In particular, the RADON IDE provides a shared space where different teams can access the RADON artifacts according to their authorizations and an access point for the interactions with the tools involved in the RADON architecture.

#### **Functionalities update:**

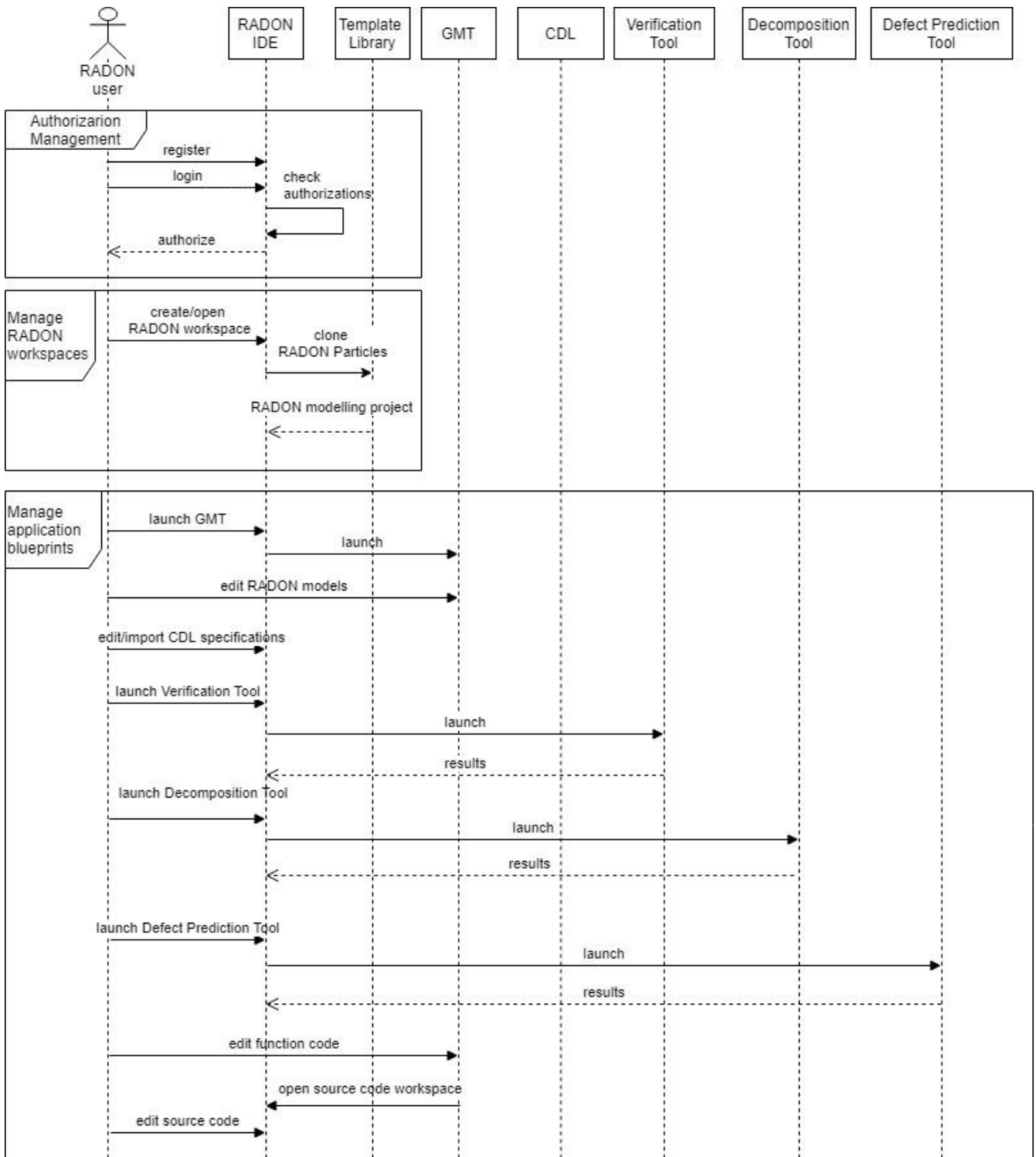
The planned core functionalities in D2.3[1] are still representative in this final version of the deliverable. However, as part of the detailed architectural design, we refined four of the five usage scenarios identified in the previous deliverable. In particular, we refined the interactions between the tools for the “*Manage RADON workspaces*”, “*Manage application blueprints*”, “*Manage test specification*” and “*Deploy application*” usage scenarios as detailed below.

#### **Interactions:**

The interactions with other RADON tools described in D2.3[1] are still valid in this final version of the deliverable. Nevertheless, we refined the description of the interactions for the following usage scenarios according to the consideration made during the definition of the final version of the RADON architecture.

**Management of RADON workspaces and application blueprints.** After the registration, the RADON user makes the login to the RADON IDE in order to access the development environment according to his/her authorizations. Then the user creates (or opens) a RADON Workspace and the RADON IDE retrieves the RADON Particles creating (by default) a RADON modeling project within the workspace. Then the RADON user is able to launch from the RADON IDE the GMT in order to graphically model the application. Moreover, the user is able (i) to import/edit CDL specifications on the application’s nodes (e.g. security and privacy constraints), (ii) to verify these CDL constraints by means of the Verification Tool, (iii) to launch the Decomposition Tool to get suggestions on possible adjustments on the node topology or on deployment optimization, (iii) to launch the Defect Prediction Tool to identify defects on the IaC scripts and he can decide to modify the business logic of the application by editing the functions source code within the workspace.

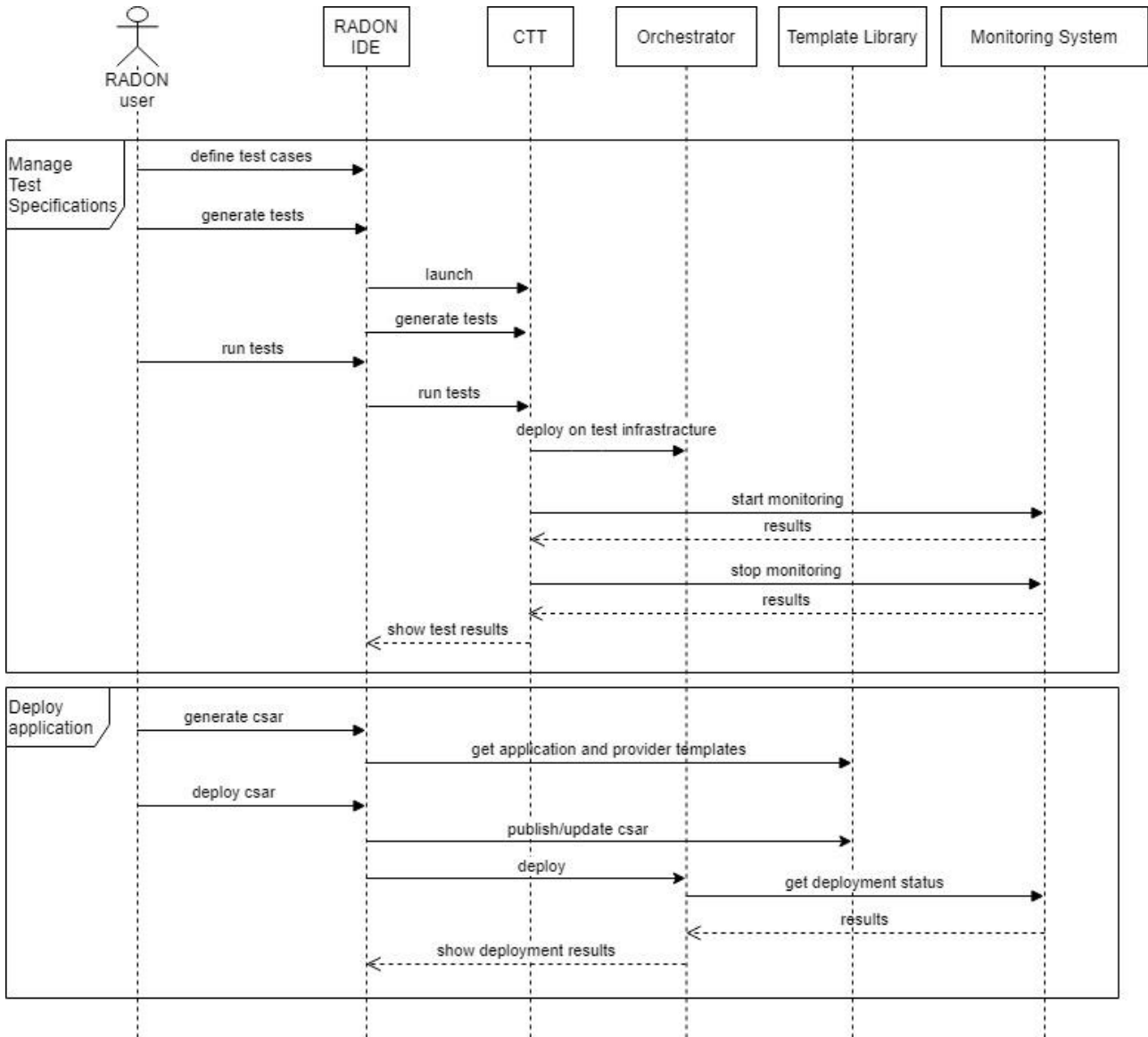
**Figure 11.** RADON IDE interactions - Management of authorizations, RADON workspaces and application blueprints



**Test management and application deployment.** The RADON user interacts with the RADON IDE to define test cases by means of the GMT integrated on it. Then a RADON user can decide to

invoke the CTT to generate the defined tests and execute the test cases as depicted in the sequence diagram provided below (only few updates were needed). The interactions with the Orchestrator and Monitoring System are similar to D2.3 [1]; the application deployment and the sequence diagram has been updated to better describe these interactions between the involved tools. In particular, the RADON user can interact with the RADON IDE to deploy the application blueprint. In this case the user generates the CSAR by means of the GMT. Then, the user can start the deployment process for the generated CSAR (stored in the workspace). The CSAR is published (or updated) in the Template Library and the CI/CD infrastructure interacts with the Orchestrator (by means of a Jenkins job) to deploy the application blueprint. The Orchestrator interacts also with the Monitoring System to get information on the deployment status then shown to the RADON user.

Figure 12. RADON IDE interactions - Test management and application deployment



#### 4.4. Graphical Modeling Tool

The Graphical Modeling Tool (GMT) is developed based on Eclipse Winery, which is a web-based environment to graphically model TOSCA-based application topologies. It includes (i) a component to manage TOSCA types and templates, (ii) a Topology Modeler that enables to graphically compose application topologies and specify configuration properties, and (iii) a file-based backend to store, import, and export TOSCA modeling entities.

For the GMT, we update two functionalities that have been reported in the former deliverable D2.3 “Architecture and integration plan I” and add one additional functionality.

##### Functionality update:

1. *Export orchestrator-ready blueprints:* In the former deliverable we expected that a RADON

user generates a TOSCA CSAR and downloads it to its local workstation using the GMT. By downloading the CSAR, we lose the integration with other RADON tools that are available in the context of the RADON IDE (Eclipse Che). Additionally, the GMT will offer the capability to save a generated CSAR to the underlying filesystem on which the GMT has been started, i.e., in the context of Eclipse Che, saving the CSAR to the Che workspace.

2. *HTTP REST API*: We extended the existing HTTP REST API to generate and download a TOSCA CSAR for a certain RADON Model. This further increases the integration capability with other RADON tools and fosters the use of automation tools, e.g., to generate and download a CSAR in a Jenkins-based CI/CD pipeline.

**New functionality:**

1. *Integrate with RADON Template Publishing Service (TPS)*: Due to the fact that RADON users have different persistence options to store RADON Model in the “Template Library” (Section 3.3), the GMT needs to be able to be used with all available options. Therefore, the GMT must be able to publish ready-to-deploy RADON Models to the RADON Template Publishing Service. Further, the GMT will be able to retrieve RADON Models from the service, e.g., to modify them and push them as a new version back.

**Interactions:**

Figure 13 shows the updated interactions using UML sequence diagrams. A user is able to generate a TOSCA CSAR by using the GMT. Notably, it does not matter whether GMT’s web-based user interface or the exposed HTTP REST API is used. Internally, the GMT retrieves the respective TOSCA types and templates and packages them into a TOSCA-compliant ZIP archive, the so-called TOSCA CSAR. The CSAR is offered to the RADON user either as download or to be saved into the underlying filesystem (Eclipse Che’s project structure).

Figure 14 shows the interaction between the RADON Template Publishing Service and the GMT. It highlights that the GMT will retrieve available RADON Models from the publishing service as well as will be able to push modifications as a new version to it. It’s worth mentioning that the GMT will store the retrieved RADON Models in a local template library, i.e., the directory structure offered by the “modeling project” in Eclipse Che. RADON users may now use the RADON IDE or GMT to modify the model and publish it back as a new version to the publishing service.

Figure 13. GMT interactions - CSAR generation process using GMT and a Template Library

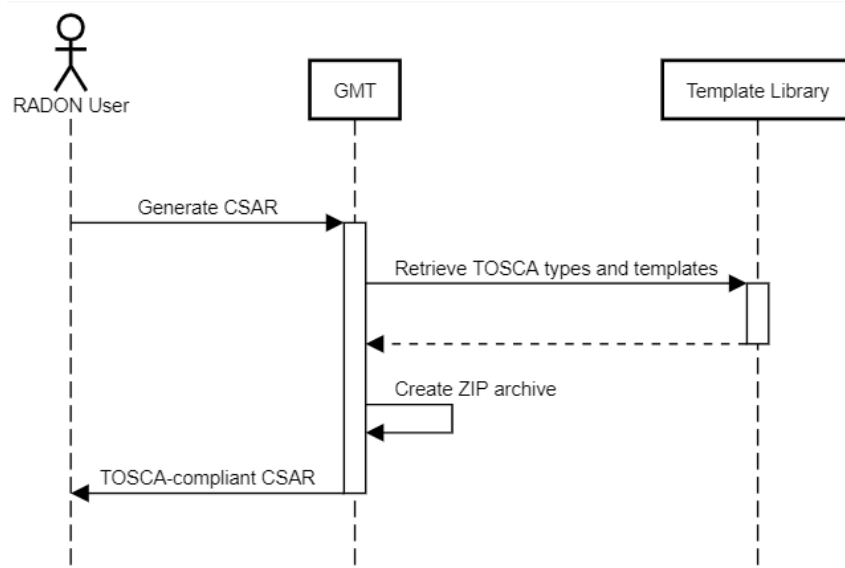
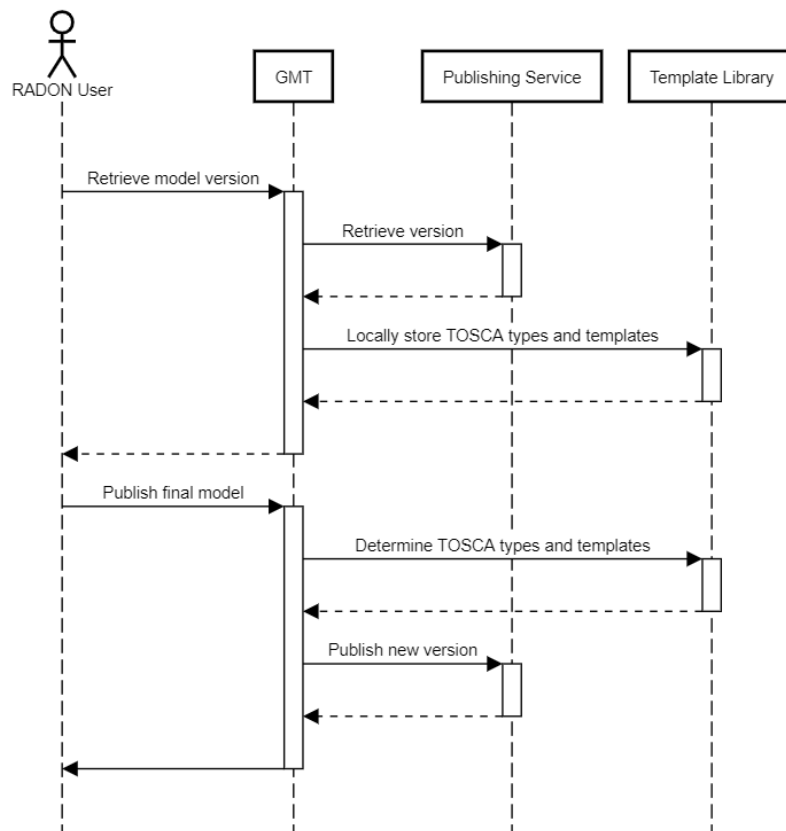


Figure 14. GMT interactions - Interaction between RADON Template Publishing Service and GMT



## 4.5. Template Library

The template library is a repository of application runtime management definitions including the blueprints, high-level system abstractions, application abstractions (including data pipeline components) and TOSCA language extensions. One of the specific parts of Template Library is the FaaS abstraction layer that holds the definitions required to deploy a particular application component to different cloud providers.

### Functionality update:

During the project, especially the last months after M12, we realized that it is a good practice to offer different persistent options for the Template Library. To cover the entire DevOps lifecycle, from design and development to deployment in production, RADON users may store and manage their RADON Models in different Template Libraries: (i) locally on the filesystem, (ii) remotely in private or public version control system (VCS) repositories, or (iii) by using a publishing service that is inspired by commonly used package management systems (D5.3[3]).

In the former deliverable, we elaborated that the Template Library is required in three main scenarios: (1) application planning, (2) CSAR generation, and (3) blueprint design. In the following we explain how each persistence options helps to support these scenarios:

- Local storage: Individual RADON users trying out and evaluating the RADON framework may store their created TOSCA types and RADON Models in a directory inside the RADON IDE. As long as the directory structure follows the proposal described in D4.4[5] other RADON tools, such as the GMT, could be used to compose an application blueprint and to generate a respective TOSCA CSAR.
- Version control system (VCS): An example of this kind of Template Library is the RADON Particles<sup>8</sup> repository, which is publicly available on GitHub and maintained by the RADON consortium. RADON users may utilize this repository as a modeling baseline to create custom application blueprints. Newly created TOSCA types or RADON Models can be easily shared with the community by following our proposed Feature-Branch-Workflow contribution model (D4.3[6]). Alternatively, the items can be shared in private repositories, e.g., by using the RADON IDE and pushing the artifacts to private GitHub or GitLab repositories.
- Publishing service: With the concept of a publishing service, similar to a package management system, RADON can offer additional value for large teams or enterprises. Such a service offers enterprise-ready features such as release management of final models (versioning) or a user management feature based on access control lists to adjust the visibility of managed TOSCA types and RADON Models. In RADON we offer such a service called “RADON Template Publishing Service”. It can be used in conjunction with other RADON tools, such as the RADON Orchestrator or the GMT. This service provides respective interfaces to list all available TOSCA type entities as well as available RADON Model. Its content, for example, can be used by the GMT to generate an executable CSAR.

---

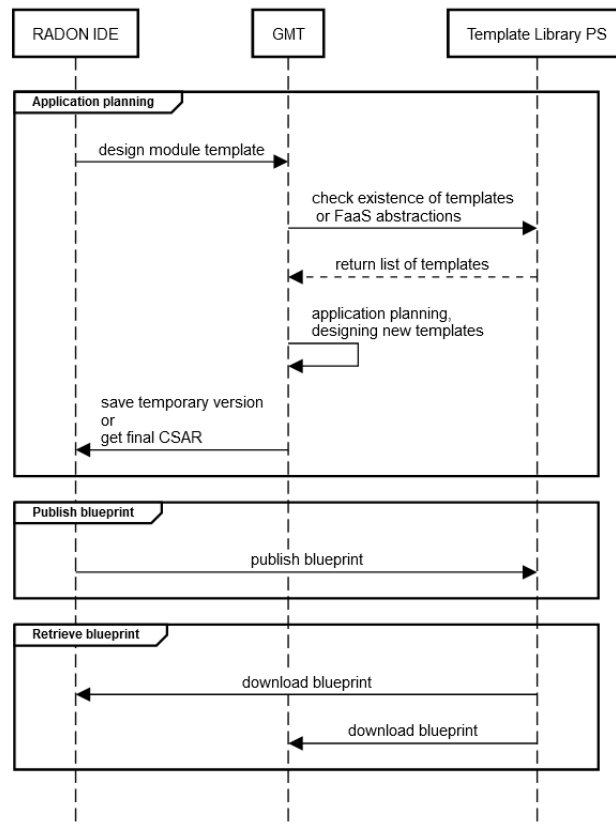
<sup>8</sup> <https://github.com/radon-h2020/radon-particles>

For blueprint design, RADON users may use the existing TOSCA types to compose new RADON Models or even publish their custom developed TOSCA types to it.

### Interactions:

Figure 15 presents interaction between RADON IDE, RADON GMT and RADON Template Library Publishing Service (TLPS). The actions in the RADON environment are usually initiated from RADIN IDE, which initiates the application or module design starting the GMT. Users operate with GMT, which receives templates from Template library on users behalf. Users can design the application and save it back to the RADON IDE environment. The saved version can be published to the Template Publishing Service if considered as stable or final release. When needed, published templates can be retrieved if needed, e.g. for deployment directly with orchestrator, or CI/CD job.

**Figure 15.** Template Library interactions - Interaction between RADON Template Publishing Service other tools



## 4.6. Decomposition Tool

The decomposition tool is present to help RADON users in finding the optimal decomposition solution for an application based on the microservices architectural style and the serverless FaaS paradigm. Its typical usage scenarios include (1) architecture decomposition, (2) deployment optimization and (3) accuracy enhancement. The tool will be made available as a public service that can be accessed from the IDE through a REST API.

### Functionalities update:

In the period M13-M18 we have worked in improving the speed of the underlying backend engine in order to facilitate the search for optimal costs. Moreover, we have developed a new methodology to capture the cold-start performance effects in the evaluations, which helps refine the accuracy of the features. Additionally, a module has been added to the backend solver to estimate service demands from empirical data, which enables the accuracy enhancement feature. In the following period we will focus on the architecture decomposition feature.

### Interactions:

The following sequence diagrams illustrate how a RADON user would interact with the decomposition tool through the IDE and what messages would occur between the decomposition tool and other tools in the three usage scenarios.

**Architecture Decomposition.** To decompose a monolithic RADON model, the user needs to call the decomposition feature, specify relevant options, for example the granularity level, and run the decomposition program, which will call the API of the decomposition tool to upload the model, decompose it and download the result.

**Deployment Optimization.** In this scenario, the user needs to call the optimization feature, where they can specify relevant options, such as the solution method and the time limit, and run the optimization program. The API of the decomposition tool will then be called to upload the model, optimize it and download the result.

**Accuracy Enhancement.** To enhance the accuracy of performance annotations, the user needs to call the enhancement feature, specify relevant options, for example the input dataset to be used for estimation purposes. The API of the decomposition tool will be called to upload the model along with the data, enhance it and download the result.

Figure 16. Decomposition Tool interactions - architecture decomposition

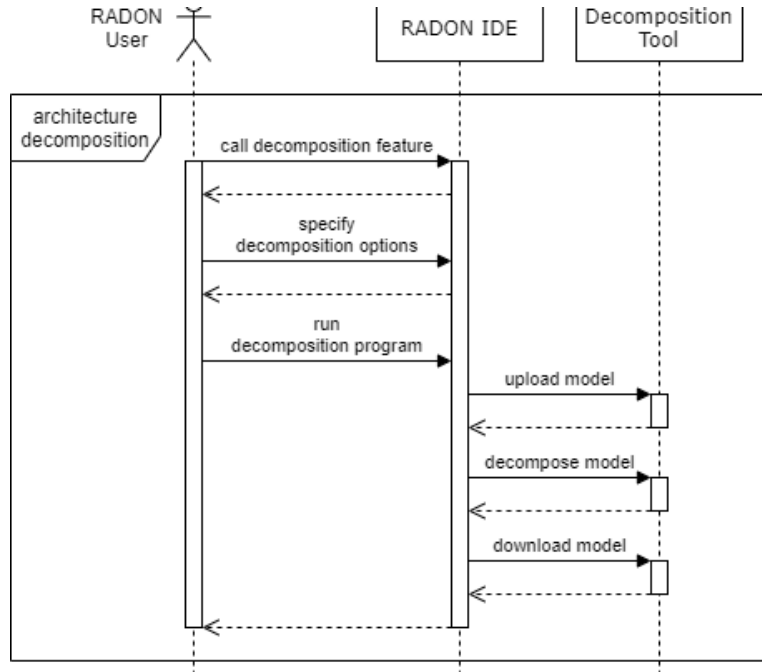
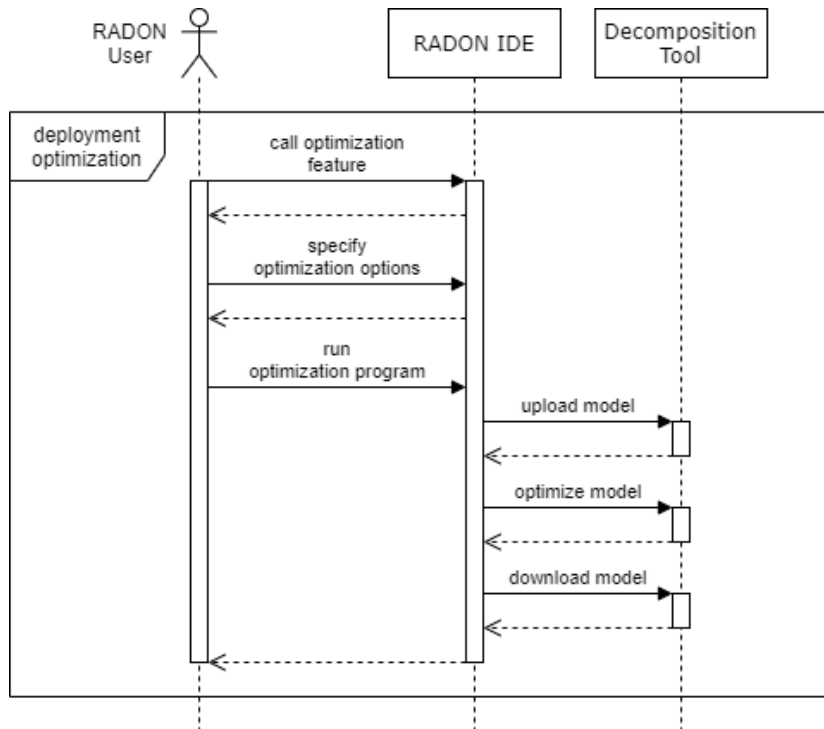
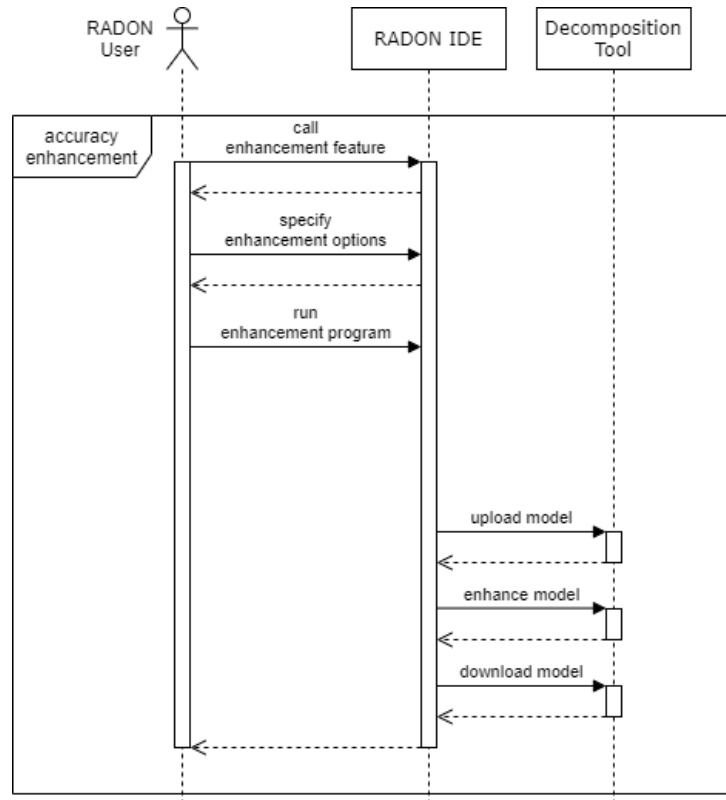


Figure 17. Decomposition Tool interactions - deployment optimization



**Figure 18.** Decomposition Tool interactions - accuracy enhancement


## 4.7. Defect Prediction Tool

The defect prediction tool helps RADON users to find suspicious defective Infrastructure-as-Code (IaC) scripts, enabling DevOps operators to focus on such critical scripts before deployment but during Quality Assurance activities. This way DPT allocates effort and resources more efficiently, enabling Continuous Deployment, accelerating release cycles by reducing time to production and ultimately cutting costs.

### Functionalities update:

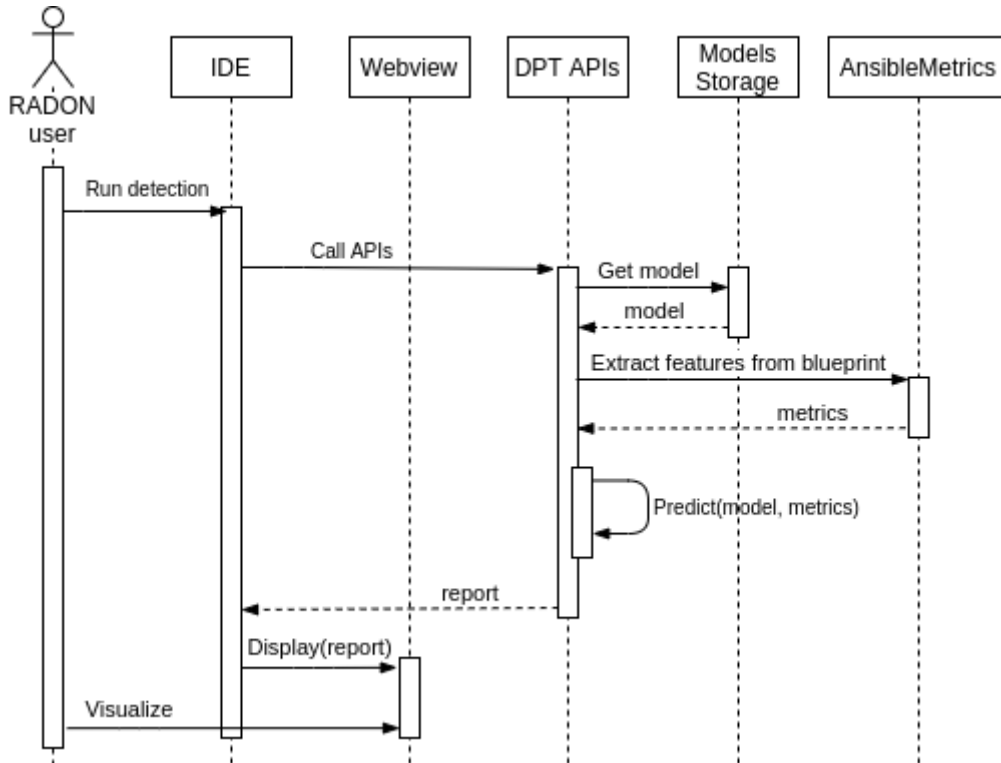
The basic functionalities of the defect-predictor tool remain the same. However, to further support RADON in automating the defect-prediction, it will be containerized and all operations that are currently supported by the REST APIs will be also available offline through CLI, a core functionality update for the next period as requested by the industrial partners in RADON.

### Interactions:

The Defect-Prediction tool has been designed to be standalone to reduce coupling with other components in the RADON technical space. Communication between the defect prediction components will be message based and, stemming from the aforementioned decoupling design decision, every message between the tool itself and other tools or RADON IDE components will be exchanged in a JSON formatted structured file, through specifically-provided APIs. The following

sequence diagram illustrates how a RADON user would interact with the defect-prediction tool through the IDE.

**Figure 19.** Defect Prediction Tool interactions



## 4.6. Orchestrator

The orchestrator puts the application into the runtime environment with enforcing the state described by application blueprint (CSAR) onto the targeted provider(s). The common operations are deployment, scaling and cleanup or un-deploy and are executed on different target environments such as staging, development and production.

### Functionalities update:

In a nutshell, the basic functionalities of an orchestrator remain the same. Orchestrator needs to be able to deploy and configure the application and all its components on the provider's side.

However, to support a RADON SaaS approach, the orchestrator will be packed also as a service. This means that all operations that are currently supported by CLI will be also available through REST API. The core functionality update of next period will be a creation of an *orchestrator as a service*.

**Interactions:** below are reported the sequence diagrams for the orchestrator.

**Figure 20.** Orchestrator authentication process

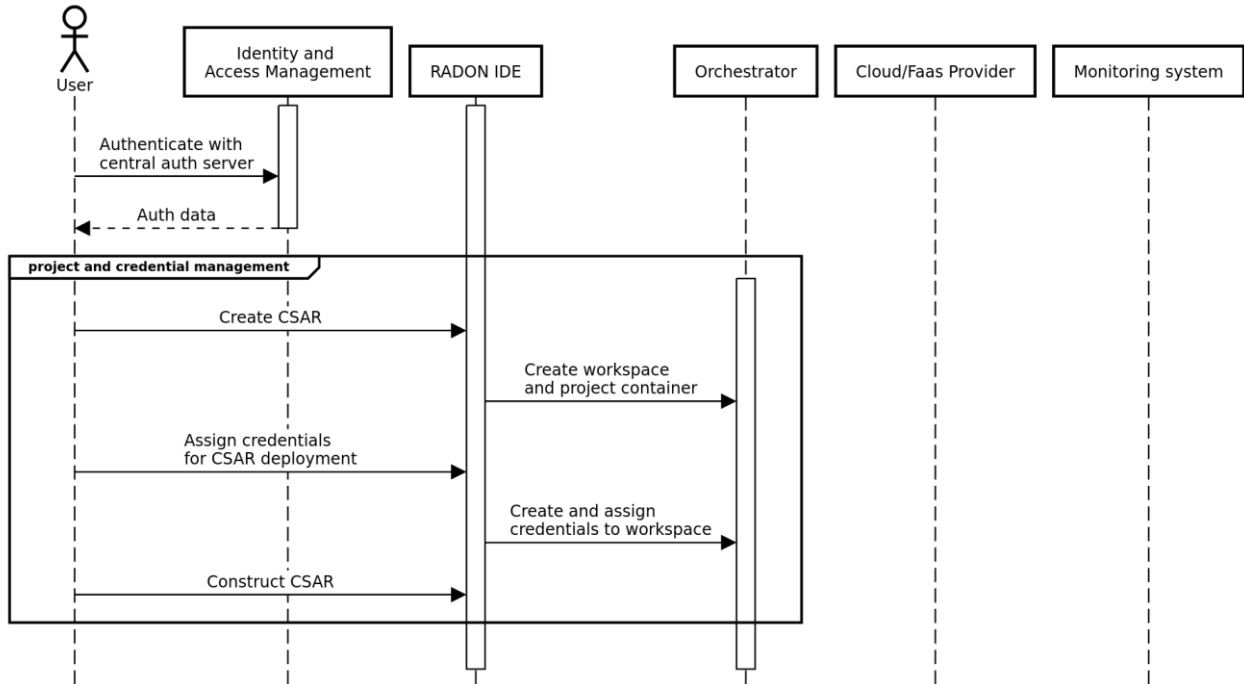
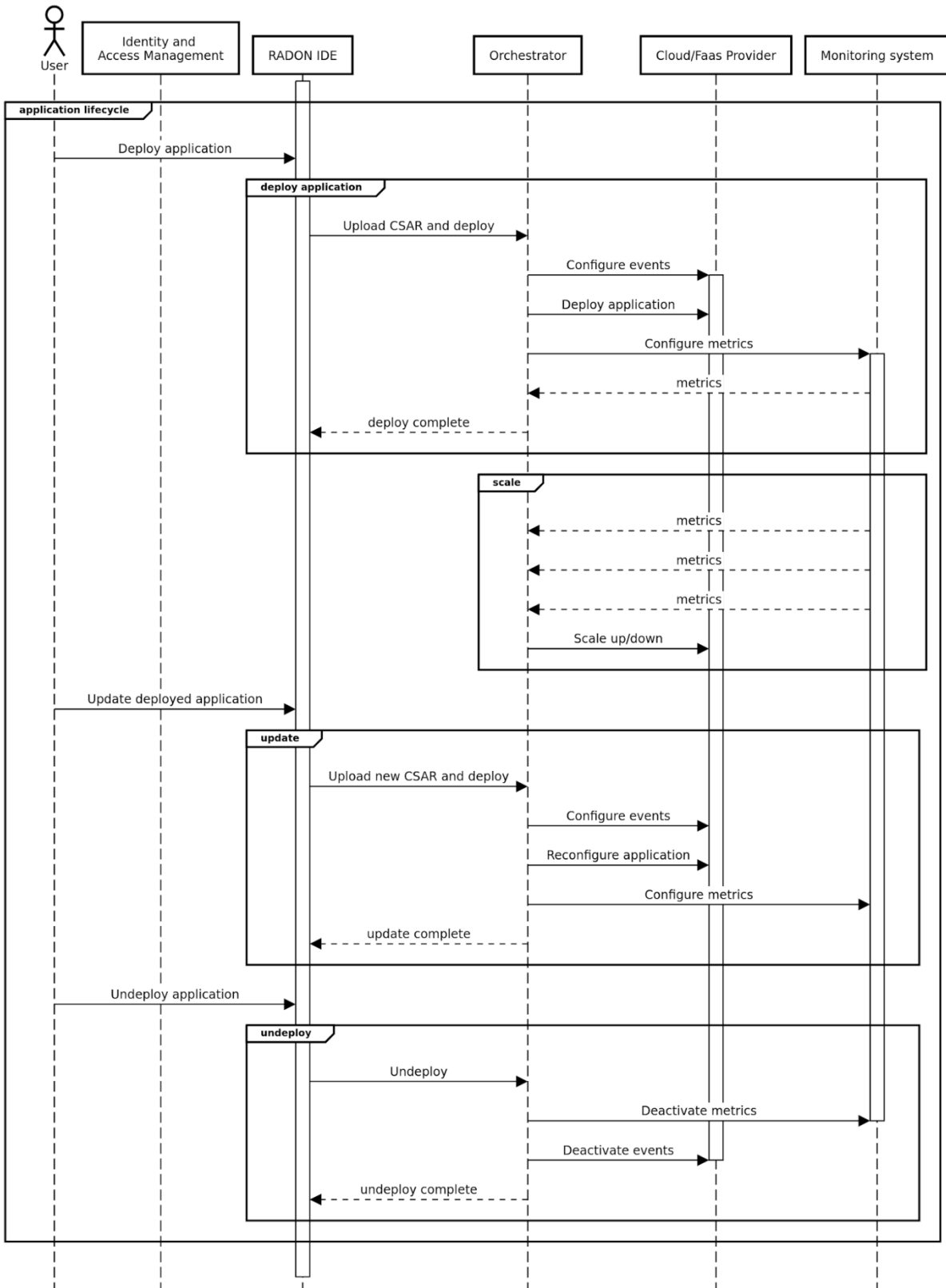


Figure 21. Orchestrator deploy, update and un-deploy flow



## 4.9. Delivery Toolchain

The delivery toolchain is a set of tools configured at one place with one goal - to deliver the application to the production site. To achieve the final step, delivery, with success, a set of previous steps needs to be successfully accomplished. Delivery toolchain provides the single point of configuration management including CI/CD, monitoring and orchestration.

### Functionalities update:

In the scope of the Monitoring tool, the following refinements on the initial functionality set are considered. A publicly available monitoring testbed allows the RADON users to dynamically register monitoring components (i.e. log file tailor, push gateway for short-lived FaaS, container monitoring) by providing configuration on a TOSCA modeling level. The monitoring testbed automatically discovers new endpoints to scrape metrics through service discovery capabilities. Moreover, the process of manually injecting code for monitoring FaaS is automated by wrapping the injected code in a library. This way, a RADON user has to simply annotate a FaaS with the annotations exposed by the library to trigger FaaS level monitoring. Furthermore, monitoring application log files and extracting metrics using regular expressions on log record level, enhances the monitoring capabilities within the RADON framework. Additionally, the Monitoring tool should provide feedback to the rest of the Delivery Toolchain toolkit with regard to the runtime behavior of an application deployment through push alerts, allowing the scaling of the application through redeployment with updated configuration.

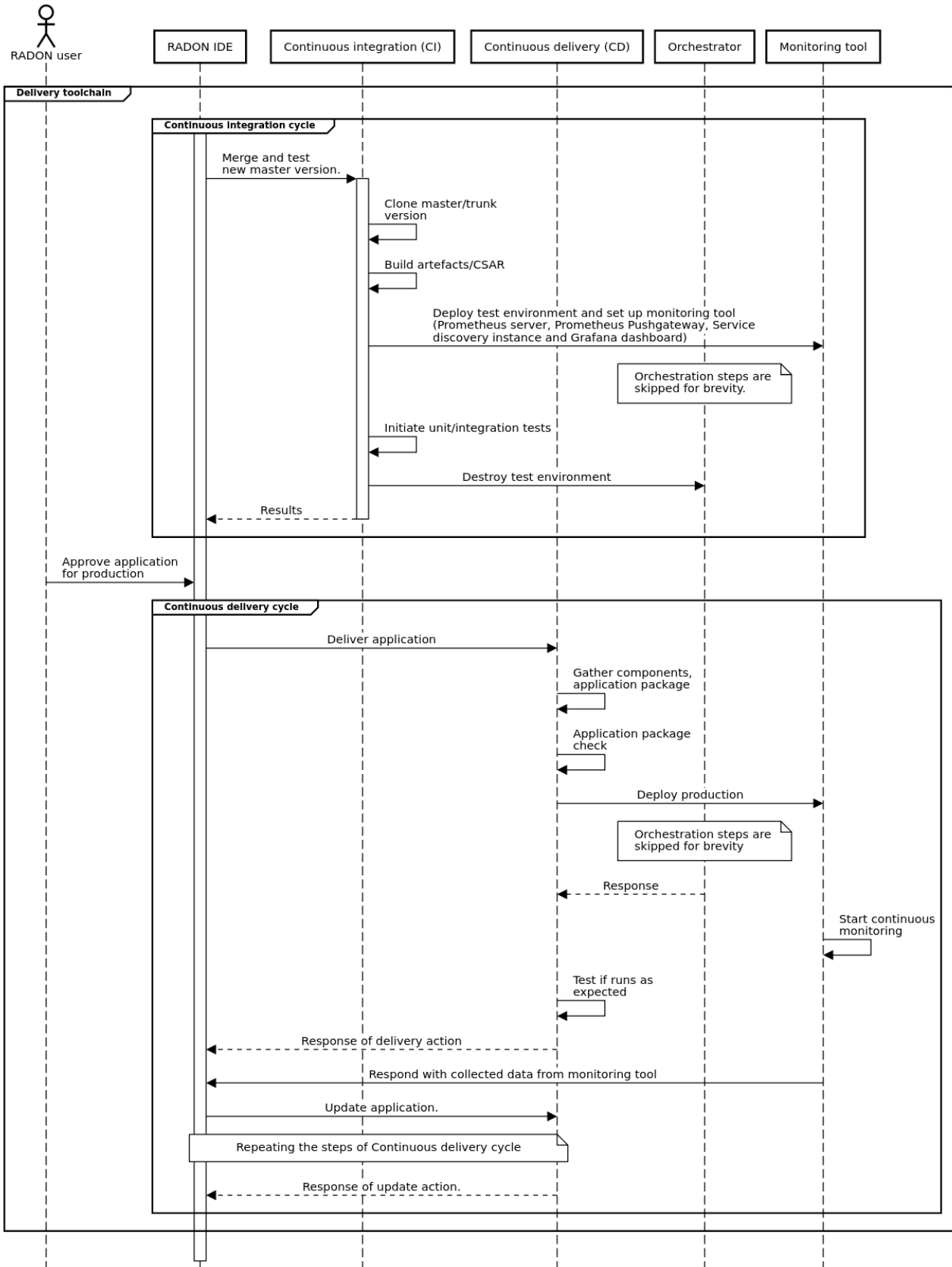
**Interactions:** below are reported the sequence diagrams for the Delivery Toolchain.

The sequence diagram that is shown on Figure 22 presents the possible flow of operations and an interaction between the RADON IDE (Eclipse Che) and the main components of the Delivery toolchain which are CI/CD, orchestration and monitoring. The Delivery toolchain has two significant cycles which correspond to Continuous integration and Continuous delivery. In the first (testing) cycle the sequence begins with RADON IDE merging and testing the changes within the application blueprint that were published as a new version to the Git repository. The CI component then clones the master version and creates appropriate deployment artifacts which are the application blueprints in the form of a prepacked compressed TOSCA CSAR file which contains all the necessary templates and the accompanying files (e.g. the CSAR can include zipped FaaS solution that contains cloud functions) The created CSAR is then passed on to the orchestrator component that initiates the deployment of the application (for instance thumbnail generator on AWS) and it also sets up the RADON Monitoring tool which consists of Prometheus server, Prometheus Pushgateway, Consul service discovery and Grafana dashboard. If the orchestration was successful, that indicates that the testing can continue with unit and integration tests that are invoked by CI. The results are then returned back to the initial point (RADON IDE) where the RADON user can approve that the testing part was successful and starts the delivery cycle for the production. Within this use case the application is first packed and delivered by CI/CD in the same way as it was done for testing and is next deployed to the production environment by the orchestrator and the response is returned to the RADON IDE. With the successful orchestration the

---

continuous application monitoring is also started and the collected data can be sent back to the IDE and seen in Prometheus instances or viewed in Grafana. If the whole application gets updated, the changes can be applied by repeating the steps from the Continuous delivery cycle.

Figure 22. Delivery Toolchain interactions



## 5. Testing

This section outlines the testing activities performed during the last months in order to test the RADON framework. Section 5.1 describes the testing of individual components to verify the correctness of the expected functionalities while Section 5.2 describes the testing activities concerning the interaction between the components.

### 5.1. Functional Testing

Table 5 reports, for each RADON tool, an overview of the functional testing performed to validate the functional specifications described in deliverable D2.3[1] and updated in Chapter 4.

**Table 5.** Functional testing of RADON tools

| RADON tools  | Functional Testing  |
|--|---|
| Constraint Definition Language and Verification Tool | The functional tests for the VT consist of a set of CDL specifications for both verification and correction tasks (further tests will be added for learning tasks when learning has been implemented). The functional tests check that the VT correctly classifies the consistent/inconsistent CDL specifications in the verification test set and is able to return at least one inconsistency for the inconsistent cases. For the correction mode, the functional tests check that the VT is able to return at least one valid correction for each test case.   |
| Continuous Testing Tool                              | The focus of CTT's functional tests comprised the validation of the tool's (i) REST-based interaction workflow as well as the included abilities to (ii) read test-augmented RADON models, (iii) deploy and execute tests (including the CTT agents), and (iv) collecting and providing the results. We focused on component/system-level and integration-level testing (see Section 5.2), which is due to the decision to start with an early end-to-end prototype of the tool as well as the early and incremental integration with the other RADON tools to mitigate potential risks. The functional tests have been integrated into CTT's CI/CD infrastructure and are executed on each commit. |
| RADON Integrated Development Environment             | The functionalities of the RADON IDE have been tested defining an activity diagram (see D2.6[2]) and verifying that a RADON user is able to perform the set of expected actions defined on it. A registered user connected to the RADON IDE instance and we successfully tested the following actions: User authentication on the IDE; Possibility to create a RADON workspace providing to the RADON user access to the shared repositories of RADON   |

|                         |  |
|-------------------------|--|
|                         | <p>artifacts and enabling interaction with the RADON tools that are invoked via custom commands; Modeling of an application by means of the GMT; Edit CDL specifications for a RADON model and verify them by means of the VT; Optimize a RADON model by means of the DT; Verify defects of an (ansible) IaC script by means of the DPT; Export into the workspace the modeled application in the CSAR format for deployment; Start deployment process for a selected CSAR. We verified that the CSAR is published to the Template Library and a Jenkins job is triggered to manage the deployment process with the Orchestrator; Visualization of the deployment results by opening a browser window pointing to the Jenkins job output console.</p>  |
| Graphical Modeling Tool | <p>Eclipse Winery, on which the RADON GMT is based, already had a good starting point in terms of unit testing as it has been under active development since 2013. At the time of writing, Winery employs and executes 526 unit tests during each CI build. With up to 350 unit tests, Eclipse Winery focuses on testing the RESTful API layer as well as the backend functionality to ensure the proper serialization of maintained TOSCA entities. In the course of the project we steadily extended the set of unit tests especially in the area of parsing and serializing TOSCA YAML entities. Further, while developing new functionality to create and export TOSCA YAML application blueprints, we regularly performed manual end-to-end testing between the RADON GMT and the RADON Orchestrator to ensure that modeled application deployments can be successfully executed.</p>   |
| Template Library        | <p>The functionalities of Template Library were tested through Template Publishing Service (TPS) where we provided and uploaded several TOSCA templates and their Ansible playbook implementation actuators to the <a href="https://template-library-radon.xlab.si/swagger/">public instance</a><sup>9</sup> using REST API and CLI. We included AWS, Azure and OpenFaaS modules and also ThumbnailGenerator application CSARs, which can be now accessed and downloaded. Template library's content gets deployed using xOpera orchestrator within CI/CD configurations. Its source code that currently resides in a private GitLab repository is constantly being tested by GitLab CI scheduled pipelines that track linting errors and therefore maintain code quality, validate TPS TOSCA templates and backup the production version of TPS data nightly so that the content can be restored in case of any future accidents.</p> |

<sup>9</sup> <https://template-library-radon.xlab.si/swagger/>

|                        |  |
|------------------------|--|
| Decomposition Tool     | <p>The DEC tool backend solver integrates a collection of about 300 unit tests that verify the consistency of the performance predictions on models of varying size, with different types of invocations between software components, scheduling at the resources, and workload intensities. Additionally, the YAML model-to-model transformation into the LQN model solved by the backend is unit tested using the thumbnail generation application example to ensure consistency with the RADON model annotations as these evolve over them.</p>   |
| Defect Prediction Tool | <p>The DPT relies on external tools created in the context of RADON (<a href="#">radon-ansible-metrics</a> and <a href="#">radon-iac-miner</a>) that already provide a good suite of unit tests to test their intended behaviour to collect and extract data from Ansible blueprints.</p> <p>Therefore, the focus of DPT's functional tests comprised the proper working of the tool's REST-based interaction workflow starting from the IDE as well as the included abilities to collect and provide the results.</p>   |
| Orchestrator           | <p>xOpera TOSCA orchestration has already been tested with different application blueprints like ThumbnailGenerator and SockShop. It was also used to deploy TPS on a public VM. Apart from all that, <a href="#">opera's source code on GitHub</a><sup>10</sup> is continuously tested by CircleCI. With every push on any branch we perform unit and integration tests that assure stability of the new ode, i.e., new changes don't break any functionality. Every night the integration tests are performed on the latest xOpera version where we execute tests on prepared TOSCA CSARs that contain common TOSCA entities. With CI/CD we also ensured that with every push on the master branch a testing opera Python package is deployed on <a href="#">testing PyPI environment</a><sup>11</sup>. In case when a new tag is pushed to the application release, the new official release is published on <a href="#">production PyPI server</a><sup>12</sup>.</p> |
| Delivery Toolchain     | <p>The Delivery Toolchain targets TOSCA Orchestrator, CI/CD and RADON Monitoring tool. The first two tools are being tested separately. The Monitoring tool is currently under development and consists of a Prometheus server, a service discovery cluster, a Grafana instance and multiple</p>   |

<sup>10</sup> <https://github.com/xlab-si/xopera-opera>

<sup>11</sup> <https://test.pypi.org/project/opera/>

<sup>12</sup> <https://pypi.org/project/opera/>

|  |   |
|--|---|
|  | Prometheus Pushgateway. These services are set up using prepared TOSCA template and Ansible playbooks which have been regularly tested with xOpera. |
|--|---|

## 5.2. Integration Testing

In this section we focus on the integration tests between the tools released in the RADON framework.

In the project, we chose to adopt the **incremental testing** approach and the integration tests are executed considering pairs of interacting components. The pairs of components involved in the integration tests (identified by examining their interactions) are:

- Continuous Testing Tool - Graphical Modeling Tool
- Continuous Testing Tool - Orchestrator
- Continuous Testing Tool - Monitoring System
- Decomposition Tool - RADON IDE
- Defect Prediction Tool - RADON IDE
- Function Hub - RADON IDE
- Function Hub - Orchestrator
- Verification Tool - RADON IDE
- Verification Tool - Graphical Modeling Tool

Table 6 defines the template we use to document the above integration tests.

**Table 6.** Integration Testing Template

| <i>Integration Testing Scenario ID</i> |                                     |
|--|-------------------------------------|
| <b>Description</b>                     | Test case description               |
| <b>Prerequisite</b>                    | Test case prerequisite              |
| <b>Postrequisite</b>                   | Test case postrequisite             |
| <b>Test Execution Steps</b>            | Description of test execution steps |
| <b>Test Results</b>                    | Description of test results         |
| <b>Test Comments</b>                   | Comments on the performed test      |

### 5.2.1. Continuous Testing Tool - Graphical Modeling Tool

Please note that the workflow of CTT includes both tests in a sequence why we also mentioned the successful test of *Integration Testing Scenario CTT-GMT* as a prerequisite for the beginning of *Integration Testing Scenario CTT-Orchestrator*.

| <b>Integration Testing Scenario CTT-GMT</b> |   |
|---|---|
| <b>Description</b>                          | <p>The integration test between CTT and GMT tests the ability of GMT to export a service template as a CSAR and the ability of CTT to take this as an input for further processing (i.e., deploying the service template using the xOpera orchestrator, which is covered in the following integration test case).</p> <p>The following test execution steps explain the essential steps needed to test the integration between CTT and GMT. The detailed code-level steps including exact paths and parameters can be found in the TravisCI build file on GitHub<sup>13</sup>.</p>  |
| <b>Pre-Conditions</b>                       | In order to run this test, there needs to be a TOSCA service template that can be exported.   |
| <b>Post-Conditions</b>                      | Content of the exported CSAR is stored in the folder containing all artifacts for a deployment using the xOpera orchestrator.   |
| <b>Test Execution Steps</b>                 | <p>The following steps illustrate the test execution:</p> <ol style="list-style-type: none"> <li>1. Clone integration test repository.</li> <li>2. Pull all containers needed for the GMT application.</li> <li>3. Clone RADON Particles repository.</li> <li>4. Start the GMT application. GMT application is running and web interface is available at <a href="http://localhost:18080/">http://localhost:18080/</a></li> <li>5. Start the CTT server. CTT application is running and web interface is available at <a href="http://localhost:7999/RadonCTT/ui/">http://localhost:7999/RadonCTT/ui/</a></li> <li>6. Clone the SockShop Demo repository. SockShop Demo repository available locally.</li> <li>7. Use the GMT REST API to export the system under test (SUT) CSAR for the SockShop Demo to the local file system. CSAR file containing the TOSCA blueprints for the SUT and its dependencies in the local file system.</li> <li>8. Use the GMT REST API to export the test infrastructure (TI) CSAR for the JMeter Master Only Agent to the local file system.</li> <li>9. Create a CTT project using the CTT REST API.</li> <li>10. Create a CTT test artifact using the CTT REST API</li> </ol> |
| <b>Test Results</b>                         | In the end of this part of the test, CTT has collected all information to deploy the SUT and TI with the orchestrator. This can be validated by checking the UUIDs of the TestArtifact using the CTT REST API.  |

<sup>13</sup> <https://github.com/radon-h2020/radon-ctt-integration/blob/master/.travis.yml>

|                 |   |
|-----------------|---|
| <b>Comments</b> | As the following steps interact with the xOpera orchestrator, they are part of the integration test between CTT and the orchestrator. |
|-----------------|---|

### 5.2.2. Continuous Testing Tool - Orchestrator

| <i>Integration Testing Scenario CTT-Orchestrator</i> |  |
|--|--|
| <b>Description</b>                                   | The integration test between CTT and the xOpera orchestrator tests the ability of CTT to properly interact with the orchestrator for deploying TOSCA models and accessing the deployed instances afterwards. The following test execution steps explain the essential steps needed to test the integration between CTT and the xOpera orchestrator. The detailed code-level steps including exact paths and parameters can be found in the TravisCI build file on GitHub <sup>14</sup> . |
| <b>Pre-Conditions</b>                                | Running CTT with at least one test artifact (as it would result at the end of Integration Testing Scenario 1).   |
| <b>Post-Conditions</b>                               | The SUT and TI are successfully deployed and a test report could be downloaded from CTT.   |
| <b>Test Results</b>                                  | If successful, the result is a ZIP-file containing the test results of the test that has been executed by the TI.  |

| <i>Integration Testing Scenario CTT-Orchestrator (Data Pipelines case)</i> |  |
|--|--|
| <b>Description</b>   | This is an additional integration test between CTT and the xOpera orchestrator which tests the ability of CTT to properly interact with the orchestrator for deploying a TOSCA data pipeline service template.   |
| <b>Pre-Conditions</b>  | In order to run this test, the data pipeline application needs to be available in the GitHub repository. Also there needs to be a TOSCA software under test (SUT) service template and testing infrastructure (TI) service template available in the repository. |
| <b>Post-Conditions</b>   | The SUT and TI are successfully deployed and a test report could be downloaded from CTT.   |
| <b>Test Execution Steps</b>  | The test execution steps of this integration test can be reused from <i>CTT-GMT</i> and <i>CTT-Orchestrator test scenarios</i> . To avoid repeating their description, we only provide their sequence here:  |

<sup>14</sup> <https://github.com/radon-h2020/radon-ctt-integration/blob/master/.travis.yml>

|                     |  |
|---------------------|--|
|                     | <ol style="list-style-type: none"> <li>1. Start the CTT server</li> <li>2. Create a CTT project using the CTT REST API</li> <li>3. Create a CTT test artifact using the CTT REST API</li> <li>4. Create a CTT execution using the CTT REST API</li> <li>5. Create a CTT result using the CTT REST API</li> <li>6. Download the CTT result using the CTT REST API</li> </ol> <p>The main difference from the previous scenarios is that the software under test (SUT) is a data pipeline application (e.g., Thumbnail generation with data pipelines<sup>15</sup> described in the D6.2 Initial Validation Results deliverable [7])</p> |
| <b>Test Results</b> | If successful, the result is a ZIP-file containing the test results of the test that has been executed by the TI.  |

### 5.2.3. Continuous Testing Tool - Monitoring

In contrast to the previous two CTT-related integration tests that are already implemented, this test is on conceptual level as CTT's test update feature is scheduled for the next project period.

| <i>Integration Testing Scenario CTT-Monitoring</i> |   |
|--|---|
| <b>Description</b>                                 | The integration test between CTT and the Monitoring tool tests the ability of CTT to properly obtain monitoring data via the Monitoring tool's REST API for updating test-related information based on operational data in TOSCA models.              |
| <b>Pre-Conditions</b>                              | Monitoring data of an SUT's execution stored in the Monitoring tool.  |
| <b>Post-Conditions</b>                             | CTT has successfully obtained monitoring data from the Monitoring tool.   |
| <b>Test Execution Steps</b>                        | <ol style="list-style-type: none"> <li>1. CTT requests monitoring data from the Monitoring tool via the REST API. The input is the Specification of monitoring data to be obtained, e.g., ID of the SUT, time window, performance metrics.</li> </ol> |
| <b>Test Results</b>                                | If successful, CTT obtained the monitoring data in a processable format.  |

### 5.2.4. Decomposition Tool - IDE

| <i>Integration Testing Scenario DEC-IDE 1 (Optimization)</i> |   |
|--|---|
| <b>Description</b>   | The integration test between the Decomposition Tool and the IDE tests |

<sup>15</sup> <https://github.com/radon-h2020/demo-lambda-thumbgen-tosca-datapipeline>

|                             |  |
|-----------------------------|--|
|                             | the ability of the user to interact effectively with the tool through the IDE.   |
| <b>Pre-Conditions</b>       | A RADON model with annotated performance data is available on the codebase, but lacking information about the memory or concurrency level for a function, which need to be decided upon.   |
| <b>Post-Conditions</b>      | The output of the DEC tool optimization feature is correctly displayed with the IDE, showing in particular the minimal cost found. The estimates of the memory and concurrency parameters are directly annotated in the RADON model file.  |
| <b>Test Execution Steps</b> | <ol style="list-style-type: none"> <li>1. Configure location of the DEC server instance</li> <li>2. Clone the RADON thumbnail generation sample project. This contains a sample TOSCA model with the required DEC annotation.</li> <li>3. Right-click on the RADON model file in the file explorer.</li> <li>4. Click "Optimize".</li> <li>5. A call is issued to the decomposition tool and the cost of the execution is shown on the Che output window.</li> </ol> |
| <b>Test Results</b>         | The optimization is deterministic and the DEC server instance needs to return the same identical cost value that can be tested for regressions.  |

| <i>Integration Testing Scenario DEC-IDE 2 (Decomposition)</i> |   |
|---|---|
| <b>Description</b>  | The integration test between the Decomposition Tool and the IDE tests the ability of the user to interact effectively with the tool through the IDE.  |
| <b>Pre-Conditions</b>   | A RADON model with annotated performance data is available on the codebase.   |
| <b>Post-Conditions</b>  | The recommendations of the DEC tool decomposition feature are produced in output within the IDE.  |
| <b>Test Execution Steps</b>                                   | <ol style="list-style-type: none"> <li>1. Configure location of the DEC server instance</li> <li>2. Clone the RADON thumbnail generation sample project. This contains a sample TOSCA model with the required DEC annotation.</li> <li>3. Right-click on the RADON model file in the file explorer.</li> <li>4. Click "Optimize".</li> <li>5. A call is issued to the decomposition tool and the suggestions for decomposition are shown on the Che output window.</li> </ol> |
| <b>Test Results</b>   | The decomposition is deterministic and the DEC server instances needs to return the same identical suggestions that can be tested for regressions.  |

| <i>Integration Testing Scenario DEC-IDE 3 (Accuracy Enhancement)</i> |  |
|--|--|
| <b>Description</b>   | The integration test between the Decomposition Tool and the IDE tests the ability of the user to interact effectively with the tool through the IDE.   |
| <b>Pre-Conditions</b>  | A RADON model with annotated performance data is available on the codebase. The model initially lacks certain model input parameters that qualify the expected execution time of a serverless function.  |
| <b>Post-Conditions</b>   | The estimates of the input parameters are directly annotated in the RADON model file.  |
| <b>Test Execution Steps</b>  | <ol style="list-style-type: none"> <li>1. Configure location of the DEC server instance</li> <li>2. Clone the RADON thumbnail generation sample project. This contains a sample TOSCA model with the required DEC annotation.</li> <li>3. Right-click on the RADON model file in the file explorer.</li> <li>4. Click "Enhance".</li> <li>5. A call is issued to the decomposition tool and the results of the parameter estimation process are shown on the Che output window and annotated in the RADON model file.</li> </ol> |
| <b>Test Results</b>  | The decomposition is deterministic and the DEC server instances needs to return the same identical suggestions that can be tested for regressions.   |

### 5.2.5. Defect Prediction Tool - RADON IDE

| <i>Integration Testing Scenario DPT-RADON IDE</i> |  |
|---|--|
| <b>Description</b>                                | <p>The integration test between the Defect-Prediction Tool (DPT) and the IDE tests the ability of the DPT to properly interact with the IDE to run a detection on a given Ansible blueprint.</p> <p>The following test execution steps explain the essential steps needed to test the integration between DPT and the IDE.</p> |
| <b>Pre-Conditions</b>                             | Running CTT with at least one test artifact (as it would result at the end of Integration Testing Scenario 1).   |
| <b>Post-Conditions</b>                            | The Ansible blueprint is successfully analyzed and a report is shown on the Graphic User Interface.  |
| <b>Test Execution Steps</b>                       | <ol style="list-style-type: none"> <li>1. Create an Ansible blueprint. This step is needed as the current version of the DPT only works with YAML-based Ansible scripts. The expected output is a well-formatted yaml file.</li> </ol>   |

|                     |   |
|---------------------|---|
|                     | <ol style="list-style-type: none"> <li>2. Call the APIs to run the detection by right-clicking on the yaml file. The expected input and output are the raw content of the yaml file and a json report with information about the final decision, respectively.</li> <li>3. Display the results in a Webview. This step shows the results of the detection on a Graphical User Interface in the IDE. The input is the json report from step 2. The expected output is a two columns table of (key, value) pairs where the <i>key</i> represents a metric extracted from the script, or the final detection decision, and <i>value</i> is its value.</li> </ol> |
| <b>Test Results</b> | <p>If successful, the result is a table containing information about the characteristics of the script (i.e., the metrics extracted) along with the final decision (defective or not defective).</p> <p>If unsuccessful, the results is a pop-up message indicating the cause of the failure: Ansible file not valid or server error.</p>   |

### 5.2.6. Function Hub - RADON IDE

| <i>Integration Testing Scenario Function Hub - RADON IDE</i> |  |
|--|--|
| <b>Description</b>   | Being able to select URL from Function Hub when creating a FaaS model  |
| <b>Prerequisite</b>  | <ol style="list-style-type: none"> <li>a) Running GMT instance</li> <li>b) Preselected, valid URL from FunctionHub</li> </ol>  |
| <b>Postrequisite</b>   | Successfully generated CSAR  |
| <b>Test Execution Steps</b>                                  | <ol style="list-style-type: none"> <li>1. Select Function from FunctionHub and copy URL</li> <li>2. Use path as input for FaaS model in GMT</li> <li>3. Export CSAR</li> </ol> |
| <b>Test Results</b>  | GMT allows script input both as 'on local machine' and 'URL'.  |

### 5.2.7. Function Hub - Orchestrator

| <i>Integration Testing Scenario Function Hub - Orchestrator</i> |  |
|---|--|
| <b>Description</b>  | Being able to deploy a TOSCA blueprint with FaaS script from URL |
| <b>Prerequisite</b>   | c) Generated tosca blueprint.                                    |

|                             |  |
|-----------------------------|--|
|                             | d) Preselected, valid URL from FunctionHub   |
| <b>Postrequisite</b>        | Deployed FaaS with Function from FunctionHub                                       |
| <b>Test Execution Steps</b> | 1. Deploy toscs file with hard coded Function Hub URL                              |
| <b>Test Results</b>         | RADON Orchestrator accepts FaaS script input both as 'on local machine' and 'URL'. |

### 5.2.8. Verification Tool - RADON IDE

| <i>Integration Testing Scenario VT-RADON IDE</i> |  |
|--|--|
| <b>Description</b>                               | The integration test between the Verification Tool (VT) and the IDE tests the ability of the VT to properly interact with the IDE. Specifically, it checks that a user can use the IDE to call the verification mode of the VT and view the result in the IDE.   |
| <b>Pre-Conditions</b>                            | None   |
| <b>Post-Conditions</b>                           | The output of the VT is correctly displayed with the IDE   |
| <b>Test Execution Steps</b>                      | <ol style="list-style-type: none"> <li>1. Clone the verification tool sample project from <a href="https://github.com/radon-h2020/verification-tool-sample-project">https://github.com/radon-h2020/verification-tool-sample-project</a>. This contains a sample TOSCA model and a CDL specification in "main.cdl".</li> <li>2. Right-click on the main.cdl file in the file explorer.</li> <li>3. Click "Verify".</li> </ol> |
| <b>Test Results</b>                              | A sub-window should appear in the bottom of the IDE. After a few seconds the output of the VT should be displayed. If everything has worked correctly, the VT will show that it has found an inconsistency in the specification.   |

### 5.2.9. Verification Tool - Graphical Modeling Tool

| <i>Integration Testing Scenario VT- GMT</i> |   |
|---|---|
| <b>Description</b>                          | The integration test between the Verification Tool (VT) and the Graphical Modeling Tool (GMT) tests the ability of the VT to properly interact with the GMT. Specifically, it checks that the output of the VT's correction mode can be displayed in the GMT. |

|                             |   |
|-----------------------------|---|
| <b>Pre-Conditions</b>       | None  |
| <b>Post-Conditions</b>      | The output of the VT is correctly displayed with the GMT  |
| <b>Test Execution Steps</b> | <ol style="list-style-type: none"> <li>6. Clone the verification tool sample project from <a href="https://github.com/radon-h2020/verification-tool-sample-project">https://github.com/radon-h2020/verification-tool-sample-project</a>. This contains a sample TOSCA model and a CDL specification in "main.cdl".</li> <li>7. Right-click on the main.cdl file in the file explorer.</li> <li>8. Click "Correct".</li> <li>9. A pop-up should appear with a list of options. Click on the option to "view correction in the GMT".           <ul style="list-style-type: none"> <li>○ This triggers the VT to build two CSAR objects containing the original model and the corrected model. These are imported in the GMT, and the GMT is then opened showing a diff between the two models.</li> </ul> </li> </ol> |
| <b>Test Results</b>         | The GMT should display the diff between the original and corrected versions of the RADON model.   |

## 6. Conclusions

This deliverable presents the final version of the RADON framework technical architecture that extends the one presented in D2.3[1], refining it according to the final requirements in D2.2[8]. The document outlines the updates that have been made to the functionalities provided by the RADON components and to the interactions between these components. Moreover, it highlights how the use of RADON models has been updated according to the work done on the second iteration of the RADON modeling profile, documented in D4.4[5].

The document also concentrates on the final integration approach adopted to integrate the RADON components in the RADON IDE which is based on the Eclipse Che technology. In particular, the integration approach and plan has been refined taking into consideration the lessons learned during the development of the first (i.e. alpha) release of the RADON framework.

Another aspect taken into account in this deliverable is the description of the CI/CD infrastructure adopted on our project. Based on a Jenkins server, the CI/CD infrastructure enables automated building and testing capabilities for the RADON integrated framework.

Finally, this document presents the set of testing activities performed during the latest period of the project to test the RADON framework, considering both the functional testing to verify the correctness of the expected functionalities in the individual components and the integration tests to verify the correctness of interaction between these components.

For the remainder of the project we will focus on two main aspects: (i) finalizing and maintaining the CI/CD infrastructure to guarantee automation of building and testing capabilities; and (ii) working to complete the required functional and integration testing activities for the next RADON framework releases.

## 7. References

- [1] RADON Consortium, *Deliverable 2.3 - Architecture and integration plan I*, 2019.
- [2] RADON Consortium, *Deliverable 2.6 - RADON integrated framework I*, 2020.
- [3] RADON Consortium, *Deliverable 5.3 - Technology Library I*, 2020.
- [4] RADON Consortium, *Deliverable 3.4 - RADON Continuous testing tool I*, 2020.
- [5] RADON Consortium, *Deliverable 4.4 - RADON models II*, 2020.
- [6] RADON Consortium, *Deliverable 4.3 - RADON models I*, 2019.
- [7] RADON Consortium, *Deliverable 6.2 - Initial Validation Results*, 2020.
- [8] RADON Consortium, *Deliverable 2.2 - Final requirements*, 2020.
- [9] RADON Consortium, *Deliverable 2.1 - Initial requirements and baselines*, 2019.