Ref. Ares(2021)2221423 - 31/03/2021

H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable D3.1

RADON methodology

Version: 1.0

Publication Date: 31-March-2021

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

Deliverable	D3.1
Title:	RADON methodology
Editor(s):	Dario Di Nucci (TJD)
Contributor(s):	A. Alnafessah, M. Law, L. Zhu (IMP), G. Catolino, S. Dalla Palma, M. Garriga, W.J. van den Heuvel, D. A. Tamburri (TJD), C. Dehury, P. Jakovits (UTR), M. Cankar, A. Luzar (XLB), G. Triantafyllou (ATC), S. D'Agostini (ENG), T. F. Düllmann, A. van Hoorn, M. Wurster, V. Yussupov (UST), H. G. Næsheim, A. Spartalis (PRQ)
Reviewers:	Matija Cankar (XLB), George Triantafyllou (ATC)
Type:	R
Version:	1.0
Date:	31-March-2021
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables/
Copyright:	RADON consortium

The RADON project partners

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

The RADON project (January 2019 - June 2021) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825040

Executive summary

RADON's objective is to define a DevOps framework to create and manage microservices-based applications that can optimally exploit serverless computing technologies. RADON applications will include fine-grained and independently deployable microservices that can efficiently use Function-as-a-Service (FaaS) and container technologies. The end goal is to fuel the adoption of serverless computing technologies within the European software industry. To this end, the present deliverable aims at illustrating the technical **RADON methodology** consolidating the user workflow for using RADON tools and the DevOps paradigm for software delivery and evolution. The methodology will strive to tackle complexity, harmonize the abstraction and actuation of action-trigger rules, avoid FaaS lock-in, and optimize decomposition and reuse through model-based FaaS-enabled development and orchestration.

Glossary

CDL	**C**onstraint **D**efinition **L**anguage
CI/CD	**C**ontinuous **I**ntegration/**C**ontinuous **D**elivery
CTT	**C**ontinuous **T**esting **T**ool
DPT	**D**efect **P**rediction **T**ool
DT	**D**ecomposition **T**ool
FaaS	**F**unction **a**s **a** **S**ervice
GMT	**G**raphical **M**odeling **T**ool
IaC	**I**nfrastructure **a**s **C**ode
VT	**V**erification **T**ool
WP	**W**ork **P**ackage

Table of contents

1. Introduction **7**
1.1 Deliverable objectives 7
1.2 Overview of main achievements 8
1.3 Structure of the document 9
2. RADON Workflow-driven Methodology **10**
2.1 RADON Method Engineering 10
2.1.1 Method Engineering 10
2.1.2 Methodological Baseline Desiderata 12
2.2 The RADON Methodology: A Situational Method Engineering Overview 15
2.2.1 RADON Methodological Tenets 15
2.2.2 The RADON Lifecycle Model 17
3 RADON Workflows **20**
3.0 The Entry-point: Application Development 22
3.1 Verification Workflow 23
3.2 Decomposition Workflow 25
3.3 Defect Prediction Workflow 28
3.4 Continuous Testing Workflow 30
3.5 Monitoring Workflow 33
3.6 Continuous Integration and Delivery Workflow 35
4. Tools Overview **37**
4.1 RADON Integrated Development Environment 38
4.2 Graphical Modeling Tool 39
4.3 Verification Tool 41
4.4 Decomposition tool 42
4.5 Defect Prediction Tool 43
4.6 Continuous Testing Tool 45
4.7 xOpera SaaS - Orchestrator 47
4.8 Template Library 49
4.9 Monitoring Tool 51
4.10 Function Hub 53
4.11 CI/CD Plugin 54
4.12 Data Pipeline Plugin 55
5. Conclusions **56**

References **57**

1. Introduction

The present deliverable aims at illustrating the technical methodology consolidating the user workflow for using RADON tools and the DevOps paradigm for software delivery and evolution. The methodology covers several scenarios, such as (i) **defining or extending** an application, consisting of microservices relying on serverless resources: (ii) **decomposing** an application into optimized microservices; (iii) **verifying** whether an application meets some predefined requirements and complies with constraints; (iv) **monitoring and logging applications** to guide the evolution of RADON-based solutions; (v) **continuously testing** the applications to early catch failures, (vi) **analyze** the application's **quality** to prioritize software inspections. These scenarios can arise in modern practices, supported by the methodology, such as **Continuous Integration** and **Continuous Delivery**. RADON's methodology allows addressing challenges and methodological demands of **Application-Lifecycle-Management** where applications are structured according to Function-as-a-Service (FaaS) and can feature data pipelines. As such, the RADON methodology will cover both compute and data-driven services in a serverless context, for example, explaining how to combine data pipelines and FaaS. Finally, because the RADON methodology should support complex cloud architectural continuum scenarios where multiple types of technologies may be blended into a coherent whole, the methodology itself needs to envision such architectural patterns to arise in the scope of serverless architecting and therefore must be able to support such scenarios in turn. The aforementioned restriction and similar considerations were considered basic tenets to enact the RADON methodological engineering process.

The following methodological document explains the process above and the resulting coordinated and harmonic use of the RADON tools to solve concrete real-world problems from a theoretical applicative perspective. Conversely, from a layman's terms introduction to the RADON IDE and its technical overview, the reader should consider the ReadTheDocs knowledgebase[1] for RADON.

1.1 Deliverable objectives

The main objective of the deliverable is to document the RADON methodology. This objective can be broken down into the following parts that are reflected in the structure of this deliverable:

- The RADON Methodology highlights how the tools can be used in a coordinated way. Such a methodology discerned using Method Engineering provides users a comprehensive approach to Microservices and FaaS-based application development.
- A set of RADON workflows that adapt RADON to user purposes. In particular, we consider using RADON tools in six different workflows that illustrate alternative ways to exploit the RADON framework.

[1] https://radon-ide.readthedocs.io/en/latest/

- An overview of the tools that compose the RADON framework. We show the high-level architecture for each tool and the intended use as a standalone tool.

1.2 Overview of main achievements

The main achievements reported in this deliverable reflect the different workflows that enable users to exploit RADON to develop microservices and FaaS-based applications:

- The **methodology** provides a comprehensive end-to-end approach for microservices and FaaS-based application development.
- The **application development entry point** allows users to define FaaS-based applications using a graphical modeling tool, save and reuse previously created templates, and deploy the obtained results.
- The **verification workflow** allows users to define several constraints and verify whether the serverless application complies with such constraints with the final goal of refactoring it to comply with requirements.
- The **decomposition workflow** allows users to decompose a monolithic application from both an architectural and a deployment perspective.
- The **defect prediction workflow** allows users to improve the quality of the codebase by visualizing code metrics, localizing defects, and detecting code smells.
- The **continuous testing workflow** allows users to automate the testing activities by continuously generating and testing the applications.
- The **monitoring workflow** allows users to real-time monitor their applications at runtime.
- The **CI/CD workflow** allows users to integrate RADON within their CI/CD platform configuration.

Overall, the RADON workflows allow users to reach the following goals defined in our initial proposal.

Table 1 - RADON objectives and related workflows

Obj.	Description	Related Workflows
O1	Release an integrated framework centered on a DevOps methodology to manage the life-cycles of microservices, data, and functions in FaaS-based applications.	Verification Decomposition Defect Prediction Continuous Testing Monitoring CI/CD
O2	Develop a modeling environment to graphically design dependencies and elicit requirements for serverless FaaS, microservices, and data pipelines.	Verification Decomposition Monitoring CI/CD
O3	Develop a runtime environment for automated model-driven orchestration based on reusable templates and IaC-based	Monitoring CI/CD

	configuration of deployable resources.	
O4	Define a library of templates and a FaaS abstraction layer based on event gateways to prevent proprietary lock-in in commercial FaaS platforms.	Decomposition
O5	Assure quality in the design and runtime operation of FaaS-based applications in compliance with requirements.	Verification Decomposition Defect Prediction Continuous Testing

1.3 Structure of the document

The remainder of this document is organized as follows. Section 2 describes method engineering and the general workflow-driven methodology. Section 3 illustrates the user-concern-specific workflows, while Section 4 briefly described the workflows for each RADON tool. Finally, Section 5 concludes the document.

2. RADON Workflow-driven Methodology

This section describes the RADON Methodology, the method applied to discern it, and its requirements.

2.1 RADON Method Engineering

To systematically design the RADON workflow-driven methodology, we have adopted the proven and well-known (situational) method engineering.

2.1.1 Method Engineering

Method engineering may be defined as "the discipline to construct new methods from existing methods"[2]. It addresses the development of novel methodologies from already existing techniques, methods, and tools, drawing from good- and best-practices. In this context, a method can be defined as "[...] an approach to perform a systems development project, based on a specific way of thinking, consisting of directions and rules, structured in a systematic way in development activities with corresponding development products" [Brinkkemper1996].

Such methods need to be adapted to specific characteristics to cater to RADON's DevOps-oriented application development paradigm in serverless computing environments. This tactic is generally referred to as "*situational method engineering*". By embracing situational method engineering for RADON, we avoid the pitfalls of the traditional "one-size-fits-all" approaches and allow us to cater to community-, organization-, and project-specific requirements and constraints.

The situational engineering of the RADON method is not geared toward one single base method but rather at synthesizing several "method parts" (sometimes referred to as method chunks or fragments) in an application context-specific manner. The key fabric of these method chunks/fragments constitutes the RADON workflows designed by RADON industry partners, grounded on the body of literature.

Figure 1 depicts the overall three-staged approach for the RADON situational method engineering. Method parts are stored in the method base, which is constituted of RADON workflows. Each RADON workflow describes a logically and temporally ordered series of actions (viz, the process) and static artifacts (RADON products, e.g., code, test scripts, security policies).

The RADON methods parts collectively shape the baseline from which appropriate parts have been identified, selected, and composed into a generic RADON Uber Methodology (i.e., step 0 in **Figure 1**). This lifecycle model defines the sequencing of method actions and associated RADON workflows in an application-organization-and-technology-agnostic fashion.

[2] https://en.wikipedia.org/wiki/Method_engineering

In the next step (i.e., step 1 in **Figure 1**), situational factors are incorporated to develop a conceptual RADON methodology. This methodology has been tailored toward a particular organizational entity (either a single organization, virtual enterprise, or community) and the domain where the RADON approach is applied (e.g., enterprise applications supporting a logistics container chain composed of robots in an international harbor). This way, a conceptual method is created from the organization- and domain-agnostic RADON Uber Methodology. The light green elements in the RADON Uber Methodology highlight this first configuration step.

The final step (i.e., step 2 in **Figure 1**) is the on-the-fly customization of the conceptual methodology to the constraints of a specific endeavor, typically a project. This step yields an enacted RADON method associated with a particular RADON project and/or artifact.

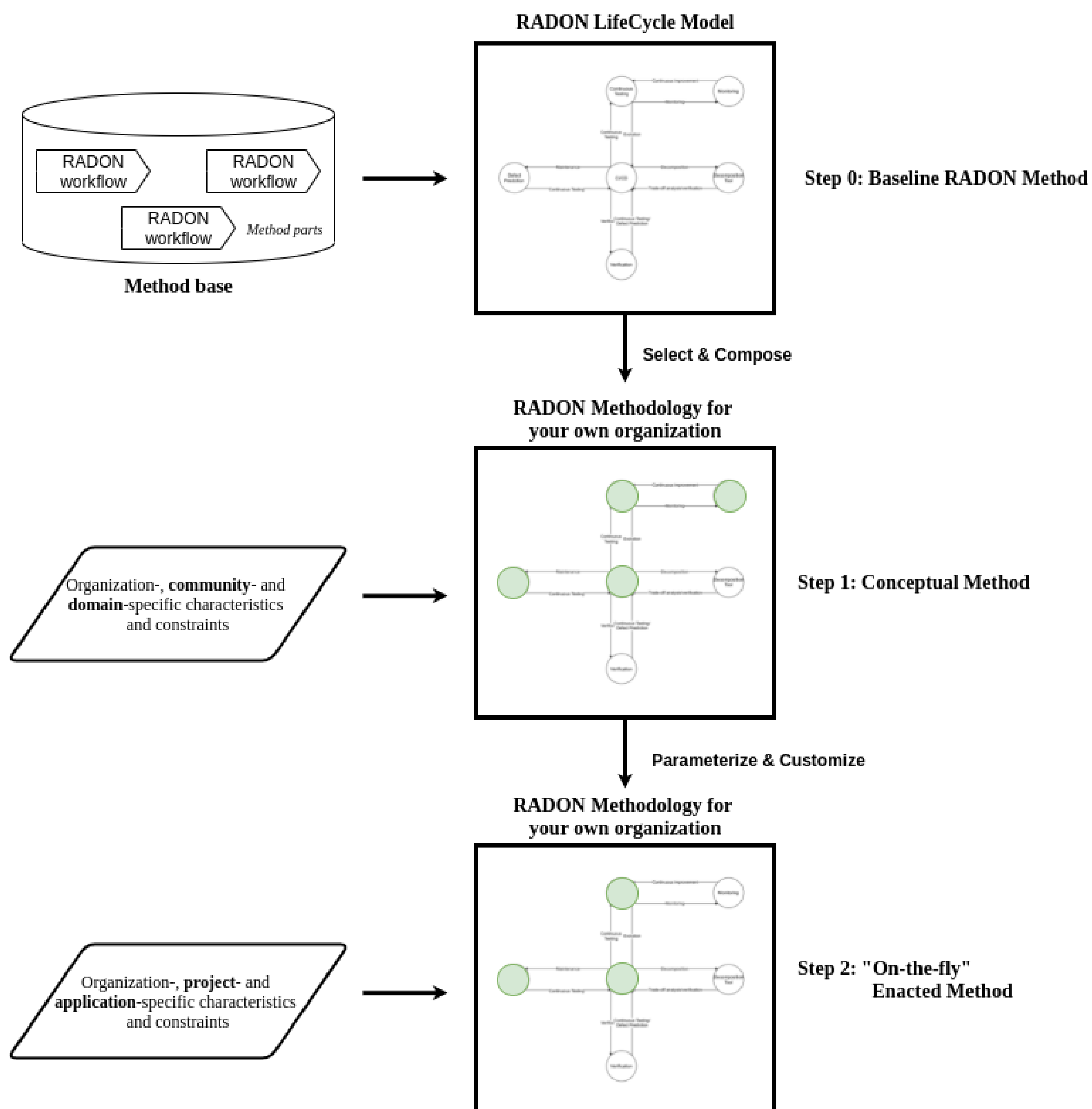

Figure 1 - RADON Methodology step-by-step engineering

2.1.2 Methodological Baseline Desiderata

The RADON technical base has been developed based on the baseline requirements gathered from Deliverable [D2.2] (Final Requirements), including individual tool requirements and use case requirements, augmented with a comprehensive grey literature review on the requirements for serverless computing engineering and associated pre-existing methodologies (e.g., from the area of cloud service engineering).

Table 2 presents a list of baseline requirements for the RADON Uber Methodology drawn from a *structured grey literature survey* [Soldani2018], augmented with literature specific for RADON. The labels specifying the "Level of compliance" are defined as follows:

- ✔ indicates that the requirement is **partially-low achieved** by the current version;
- ✔✔ indicates that the requirement is **partially-high achieved** by the current version,
- ✔✔✔ indicates that the requirement is **fully achieved** by the current version.

The stakeholder's (i.e., ATC/ENG/PRQ) perspectives are consolidated in **Table 3**.

Table 2 - Baseline requirements for the RADON Uber Methodology extracted from [Soldani2018].

Description	**RADON Industrial Confirmation**
The methodology needs to allow the application developer to specify metrics (e.g., service level objectives) for management services, including event-, incident-, configuration-management (version), and service metering and billing. **Sources:** https://techbeacon.com/enterprise-it/essential-guide-serverless-ecosystem https://www.sunviewsoftware.com/blog/learn/blog/top-serverless-insights-for-itsm-practitioners	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**
The methodology needs to cater to storage services or database services to accommodate data persistency. **Sources:** https://freecontent.manning.com/patterns-for-solving-problems-in-serverless-architectures/ https://medium.com/awesome-cloud/aws-difference-between-sqs-and-sns-61a397bf76c5	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**
The design of functions (FaaS) and micro-services should address failure scenarios and error handling. **Sources:** https://www.mphasis.com/home/thought-leadership/blog/understanding-serverless-computing.html	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**
The RADON methodology should accommodate the development and management of event-driven IoT (services) and serverless computing. **Sources:**	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**

https://events19.linuxfoundation.org/events/kubecon-cloudnativecon-europe-2018/program/speaker-guide/	
The RADON methodology helps application developers to decisions on the decoupling of services relying on managed services in the serverless compute stack, notable message queuing services (SQS as Lambda triggers) **Sources:** https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**
The RADON methodology should deal with vendor-interoperability, addressing developers' challenges to deal with different vendor-specific implementation models for serverless functions. **Sources:** [Eyk2017]	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**
Support for monitoring and debugging that has become especially hard in a serverless context. **Sources:** https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-function-resource-manager?tabs=visual-studio-code%2Cazure-cli [Baldini2017]	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**
The RADON methodology should allow for the (re-)composition of FaaS/BaaS and "conventional (micro-)services" into enterprise applications. **Sources:** https://nordcloud.com/amazon-sqs-as-a-lambda-event-source/ https://ben11kehoe.medium.com/youve-got-the-right-idea-but-you-need-to-take-it-a-step-further-to-not-have-to-manage-the-3faefee2c172	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**
Decomposition and migration need to be supported by a methodology in identifying which part of an application can be effectively supported by FaaS/micro-services in a serverless computing environment. **Sources:** [McGrath2016] [Hendrickson2016]	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**
The methodology needs to address privacy and security concerns introduced by FaaS, such as those resulting from resource sharing. **Sources:** https://d2iq.com/blog/iaas-vs-caas-vs-paas-vs-faas	**ATC: ✔✔✔** **PRQ: ✔✔✔** **ENG: ✔✔✔**

Table 3 - Consolidated requirements for the RADON Uber Methodology.

Label	**Ramifications for RADON Methodology**
Sanctioned technology use and good/best practices	The methodology should support architecting the software correctly and with sanctioned alternatives while respecting the event-driven nature of serverless computing. Good and best practices should be captured and be reused.
Continuous Adaptations and Knowledge churn	The methodology must accommodate fluid knowledge transfer between the involved actors. It should keep the knowledge flowing during continuous and often disruptive architectural changes (Continuous Architecting), reflecting changing customer requirements.
Process automation should be adequately set-up, alluding to rapidly distributed code commits	The RADON methodology should effectively accommodate the setup of process automation around decoupling the resource and stack management and general architecture maintenance. This way, it should enable concurrent and rapid code commits (and release management) across a diverse living pipeline committed during operations and often without a staging environment.
Distributed Team Support	The RADON methodology should support distributed, concurrent, and global software and services engineering and delivery.
Improved Security Oversight and Facilities	The methodology should fit in with hosted services to augment applications capacity in security -- think DynamoDB for safe data storage. The methodology should also support security in the serverless domain in a more fine-grained and reusable way, possibly through policies. Lastly, the methodology should deal with external and third-party services processing privy and sensitive data.
End-to-end and Non-stop Testing activities	The RADON methodology should emphasize the verification of system-level interoperability in a looping, end-to-end fashion.
Continuous Monitoring and Logging	The RADON methodology should be instrumented to continuously monitor the effect of adopted best-practices (e.g., cost-effectiveness) and optimizations (e.g., architecture decompositions' effectiveness). In other words, it should enable the integration and reconciliation of highly distributed logs for effective monitoring.

2.2 The RADON Methodology: A Situational Method Engineering Overview

This section describes the RADON methodological tenets and the RADON life-cycle model.

2.2.1 RADON Methodological Tenets

Serverless computing is a logical evolution from pre-existing cloud service computing paradigms such as Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS) to the more fine-grained notion of Function-as-a-Service (FaaS).

FaaS goes beyond the abstraction level of PaaS, virtualizing the end-to-end programming environment runtime to execute and deploy ephemeral code “spikes readily" (e.g., Lambda functions) without the system designers, programmers, and maintainers having to worry about the infrastructure details. The containers, where FaaS run, are instrumented with built-in monitoring, logging, and security mechanisms.

Conceptually, FaaS "spikes", also referred to as nano-services, are highly-autonomous, extremely fine-grained, and loosely-coupled, singular, stateless compute functions, which come with minimal responsibilities. This aspect renders the traditional "application-tier" in 3-tier client-service architectures obsolete, allowing the front-end code to directly interact with the (FaaS-enabled) back-end functionality, removing layers of abstraction and lifting associated levels of design-complexity and latency. This way, serverless FaaS functions are typically deployed to execute security- and privacy-critical, high-speed operations that cannot be run at the client (e.g., browser).

Figure 2 graphically depicts a highly simplified yet realistic serverless scenario involving a thin and intelligent client (browser) and a mix of proprietary and third-party FaaS and Data as a Service (DaaS) services. The scenario is projected on a stratified architecture composed of four logical layers that promote a strict separation of concerns, loose-coupling, and reuse.

At the top layer, this architecture defines the intelligent client, which could, for example, be a rich client residing in a browser environment or a scheduler. In particular, the intelligent client sits between the user and the application, receiving input and redirecting it to the appropriate application components and returning results to the user.

The API gateway integrates services intercepting incoming events and requests, routing them to the appropriate serverless functions based on a set of policies (including workload and security policies), and returning the result to the client or scheduler. API gateways are not restricted to serverless functions but may also glue together monolithic applications and microservices, allowing for the development of hybrid applications.

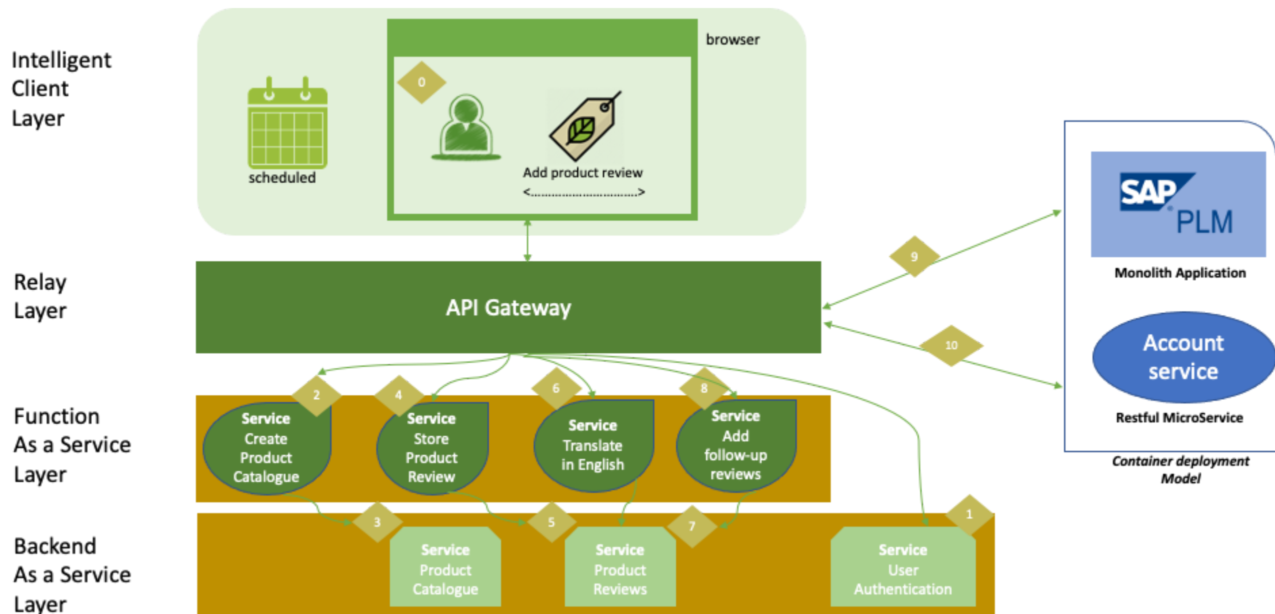

Figure 2 - A serverless scenario involving a thick and intelligent client (browser) and a mix of proprietary and third-party FaaS and DaaS services. The scenario is projected on a stratified architecture composed of four logical layers that promote a strict separation of concerns, loose-coupling, and reuse.

The core tenets of architecting serverless computing engineering include the following[3]:

1. Use a compute service to execute code on demand. Payments are only based on the duration of execution of serverless functions or the number of invocations.
2. Write single-purpose, stateless functions. Single-purpose serverless functions have the advantage of short execution cycle times, which are in principle easier to test, develop, maintain and (re-) deploy. Develop them or decompose them as fine-grained, single-purpose functional "spikes" without virtually any assumption about the business logic context in which they will run, allowing for asynchronous communication.
3. Design push-based, event-driven pipelines. Such event pipelines will help to propagate complex events without compromising the system scalability and overall performance. FaaS-enabled applications will be automatically deployed in containers triggered by such events and immediately terminated once they have completed their task.

[3] Adapted from: https://techbeacon.com/enterprise-it/essential-guide-serverless-technologies-architectures; https://www2.deloitte.com/content/dam/Deloitte/tr/Documents/technology-media-telecommunications/Serverless%20Computing.pdf

4. Create thicker, more powerful front ends. These thicker front ends bear more intelligence and limit the number of (external) function invocations removing latencies and dependencies.

5. Embrace third-party services. The tantalizing potential of serverless can be reaped to the maximum by selecting and injecting external, third-party services like logging, monitoring, testing, reporting, and alerting.

6. Dynamic scaling. It allows the developer to concentrate on developing business logic instead of ensuring the scalability of the runtime environment.

7. Promote Reuse. Engineering approaches should promote reuse (i.e., patterns), proven testing and verification practices, and repository systems (e.g., for reusable code skeletons) to optimize speedy delivery of serverless applications and evolve them over time in a systematic and proven manner.

Any serverless computing methodology should be developed with the above tenets in mind, guide developers and maintainers in making the right design decisions early in a project, and help create a shared understanding of the system design. These characteristics will foster high internal system quality and overcome resistance to change, leveraging faster delivery of improved and novel features.

Furthermore, stemming from the above tenets, the RADON methodology should, by-design, support a multi-cloud engineering[4] scenario. This technology foresees multiple cloud computing and storage services in a single heterogeneous architecture comprising, among others, microservice and serverless designs in continuity with other cloud architectural elements.

2.2.2 The RADON Lifecycle Model

As a result of the aforementioned situational method engineering exercises, we distilled as a first step (see **Figure 1**) the lifecycle model, namely, an abstract representation of the high-level phases and their logical interconnections. The RADON lifecycle model has been designed to fulfill the method requirements described in Section 2.1 (**Table 2** and **3**), implementing the RADON serverless application tenets outlined in Section 2.2.1.

In the second step (see **Figure 1**), these high-level phases are decomposed into method fragments, implemented as RADON tools, and materialized as lower-level workflows for designing, developing, and operating RADON applications.

We have graphically depicted the RADON lifecycle model in **Figure 3**.

[4] https://www.zdnet.com/article/pivotals-head-of-products-were-moving-to-a-multi-cloud-world/

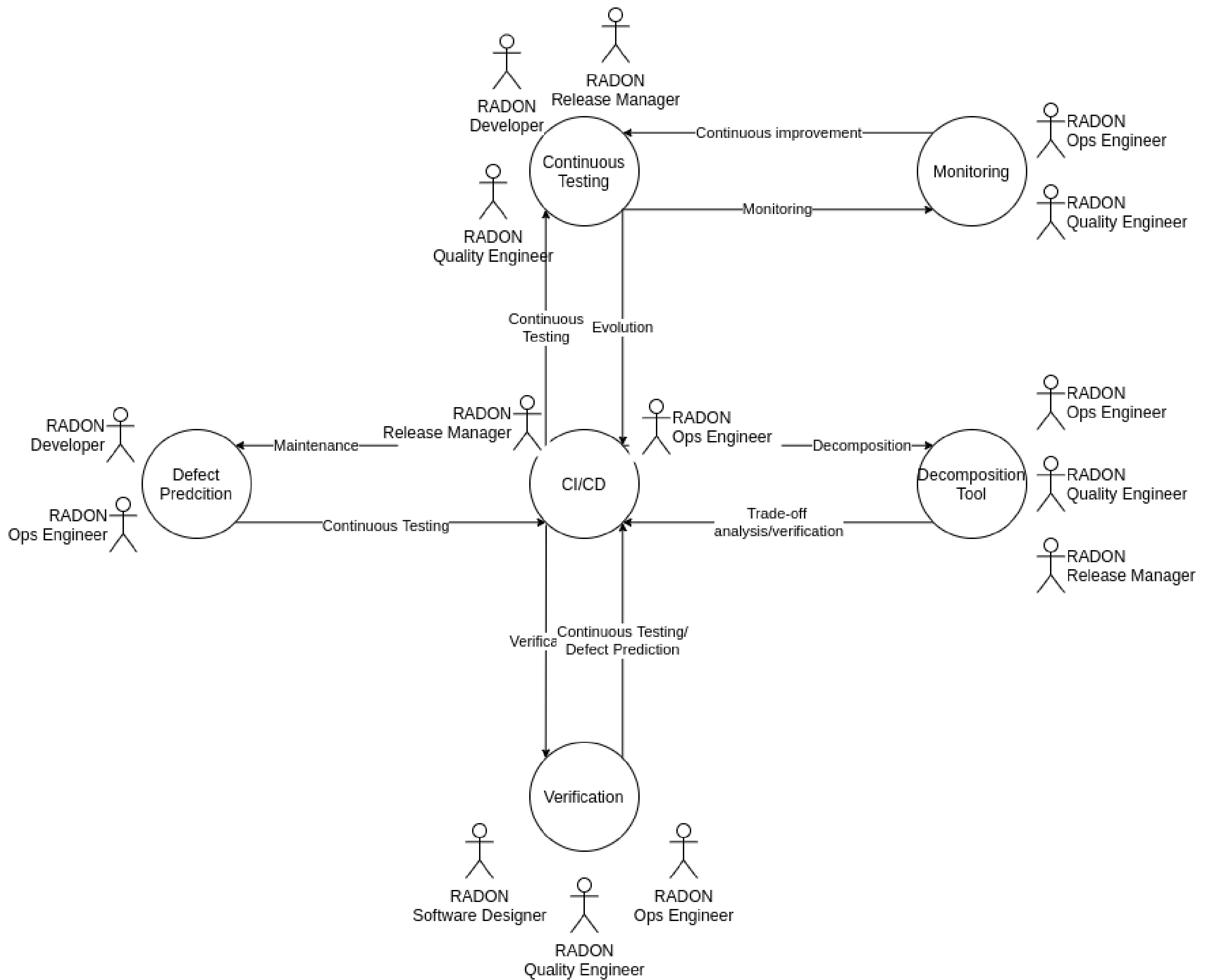

Figure 3 - The RADON Lifecycle Model.

Furthermore, in such a maintenance and evolution exercise and the scope of the RADON action, we identified no less than six critical phases in a standard RADON-supported lifecycle model, namely (from abstract-level and design time to code-level and run-time): (1) Verification; (2) Decomposition; (3) Defect Prediction; (4) Continuous Testing; (5) Monitoring; (6) Continuous Integration/Continuous Deployment.

Once a serverless application enters the RADON lifecycle model, it is organized as a continuous loop. This process comprises several phases, from code development to its continuous integration and delivery. It exploits a series of feedback loops to incorporate new measured insights (e.g., about quality attributes such as performance and security). The cycle is started anew until the application is discarded for whatever reason.

In contrast to other software lifecycle models, the feedback does not take place only toward the end of a cycle but takes place virtually constantly, which means that the actors involved need intense communication and coordination amongst themselves. The RADON Monitoring Tool plays a

center-court role in such coordination. For example, it provides feedback on resource consumption (CPU/memory usage), notifying RADON developers, integrators, and testers to act upon such information with the actual build, decomposition, and verification of the code. Furthermore, the RADON lifecycle model neither prescribes a specific order in which the cycles may be traversed nor demands all lifecycle phases to be activated.

It is also important to understand that the RADON lifecycle methodology does not herald the "one-size-fits-all" mentality. This equally applies to staff involved in executing projects adopting the RADON methodology. Typically, a single RADON "DevOps" team is thus composed of a mixture of staff involved in the entire end-to-end application lifecycle, from development and test to deployment to operations. This implies that application design, quality assurance and security roles are more tightly integrated with development, monitoring, testing and operations roles throughout the application lifecycle. In addition, it requires staff to develop a mixture of skills not restricted to a single traditional function, such as would be the case in more conventional software development lifecycle models. Some (larger) organizations applying the RADON methodology might adopt all roles, done by different staff members, whilst other (smaller) organizations might allocate all these responsibilities to one single staff member.

As explained in Section 2.1.1, the RADON lifecycle methodology has been defined at the macro-level pertaining to the global lifecycle model and their interrelationships, the meso-level referring to organization-specific instantions, and the micro-level denoting specific serverless application development projects. Organization specific instantiations of the RADON methodology take into account specific resources and domain-specific constraints imposed by the organization, including its size (SME- or large IT departments), maturity (experience level of DevOps), existing IT landscape (legacy system environment), and toolbase and pre-existing IT development and maintenance infrastructure. The project-specific instantiations are typically developed ad-hoc, taking into consideration situational characteristics like application size, complexity and type (e.g., event-driven IoT systems, or production administrative systems), dependencies on other applications, resource capabilities and capacity, and timing and scope.

In the following we will now elaborate the six key RADON method fragments (the RADON lifecycle methodology workflows), their interdependencies, roles involved, and their tool support.

3 RADON Workflows

The RADON methodology adapts to the purpose of the user through the exploitation of the situational method engineering approach.

RADON supports and advocates a comprehensive approach to Microservices and FaaS-based application development. However, it also acknowledges that some users are not developing applications from scratch using the complete RADON methodology, and could be only looking for the mere use of discrete RADON workflows, e.g., rapid prototyping functionality or defect prediction facilities upon existing applications.

The RADON methodology is composed of the six key RADON workflows and associated tools needed to conduct a specific (serverless) application development project. The RADON workflows methods impose structure on specific software development tasks with the goal of making the activity (more) disciplined, systematic, repeatable and predictable. The RADON tools have thus been designed for the explicit support of the RADON workflows, maximizing the level of automated support.

In the following, we consider using the RADON tools, which **Figure 4** depicts in an integrated way, in the context of six different workflows and the methodology entry-point that illustrate alternative ways to exploit the RADON framework, according to **Table 4**.

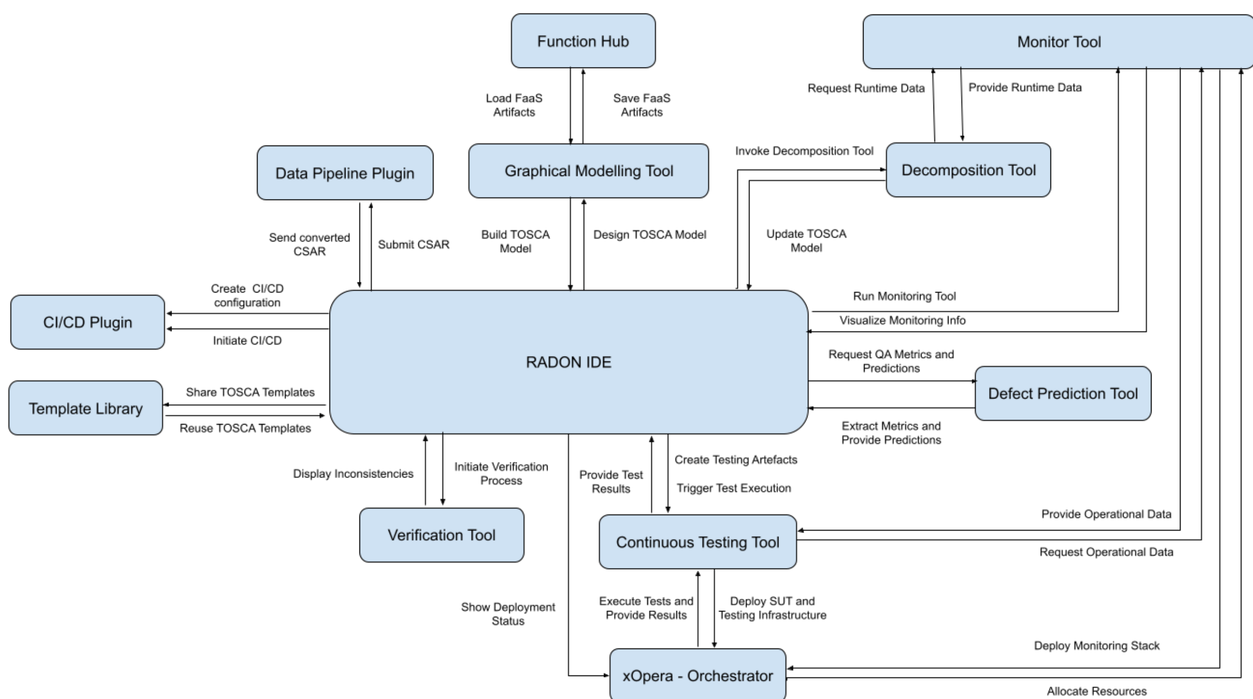

Figure 4 - RADON Tools Integration (from [D2.7]).

Table 4 - An overview of RADON workflows with involved actors and tools.[5]

RADON workflows	**Roles**[6]	**Tools**
The Entry-point: Application Development	Software Developer Release Manager	Integrated Development Environment Graphical Modeling Tool Data Pipeline Plugin Function Hub Template Library Orchestrator
Verification workflow	Software Designer QoS Engineer	Verification Tool Graphical Modeling Tool Integrated Development Environment
Decomposition workflow	Software Designer Ops Engineer QoS Engineer	Decomposition Tool Graphical Modeling Tool Integrated Development Environment Monitoring Tool
Defect Prediction workflow	Software Developer Ops Engineer	Defect Prediction Tool Integrated Development Environment
Continuous Testing workflow	Software Developer Release Manager QoS Engineer	Continuous Testing Tool Integrated Development Environment Orchestrator Monitoring Tool
Monitoring workflow	Ops Engineer QoS Engineer	Monitoring Tool Integrated Development Environment Graphical Modeling Tool Orchestrator
CI/CD workflow	Ops Engineer Release Manager	CI/CD Plugin Integrated Development Environment

[5] The table is derived from D2.4
[6] Roles were identified in D2.1

3.0 The Entry-point: Application Development

Roles: Software Designer, Release Manager

Output: IaC Blueprint(s)

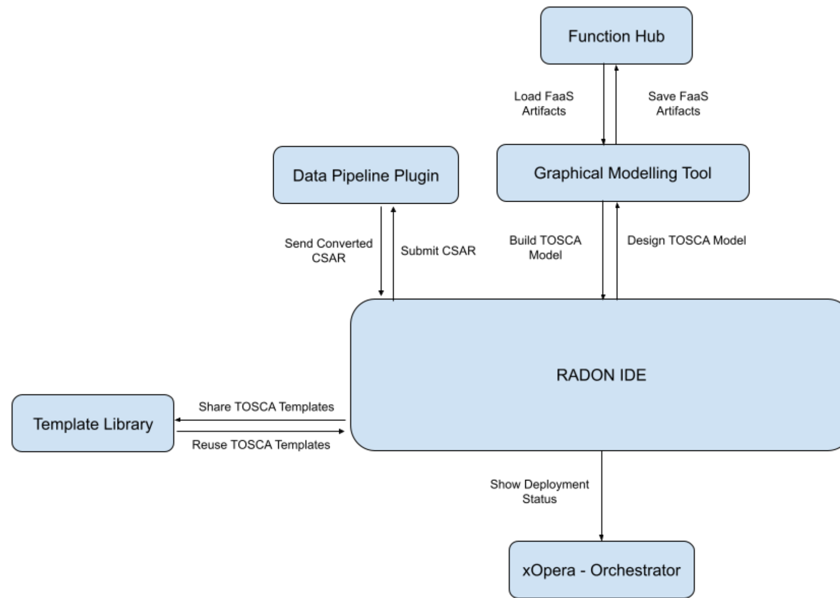

Figure 5 - RADON Application Development.

Through the **RADON IDE**, the **Software Designer** can access the **Graphical Modelling Tool (GMT)**, which provides an end-user-friendly and graphical syntax agnostic modeling of TOSCA application topologies. To foster code reuse, 'plug-and-play' application artifacts (i.e., FaaS artifacts) can be saved and loaded using the **Function Hub**, a serverless package manager integrated with RADON through GMT. TOSCA service blueprint with data pipeline-based nodes may need to be updated at runtime to ensure consistency; therefore, RADON integrates the **Data Pipeline plugin**.

Developed TOSCA modules and blueprints can be stored, published, and shared via **RADON Particles** on Github or the **Template Library Publishing Service**.

The obtained service templates are compatible and compliant with the OASIS TOSCA standard, the older TOSCA XML, and the newer TOSCA YAML standard. Furthermore, GMT generates executable blueprints in the form of TOSCA Cloud Service Archives (CSARs). CSARs files can be orchestrated by the **Release Manager** using **xOpera**, the RADON orchestrator.

The application development is the starting point of the feedback loop that leads to high-quality blueprints through the workflows described in the following sections.

3.1 Verification Workflow

Roles: Software Designer, QoS Engineer

Input: IaC blueprints and constraints defined using the Constraint Definition Language

Output: A list of the detected inconsistencies

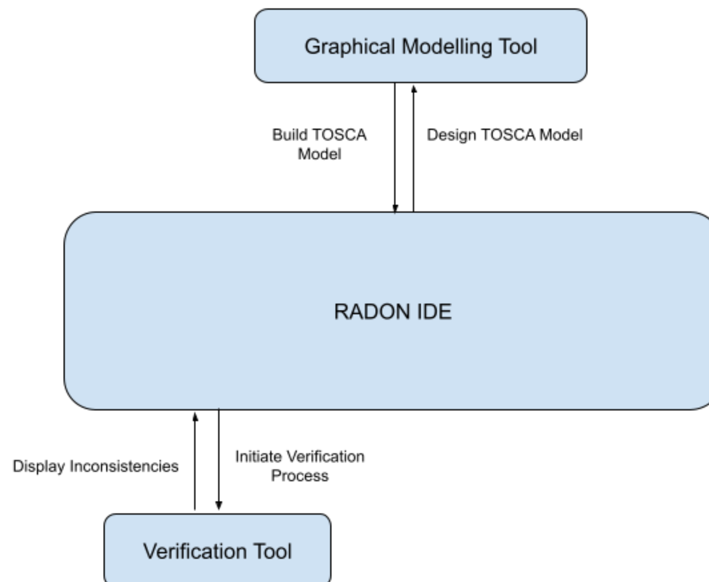

Figure 6 - RADON Verification Workflow.

The **Software Designer** commences in the Graphical Modeling Tool by graphically designing the main components of the application architecture (e.g., function(s), databases, data sources). After this, the **Software Designer** may annotate the model by setting desired properties and constraints (e.g., security/privacy requirements) in a **Constraint Definition Language** specification. Model annotation is supported by the text editor in the RADON IDE.

Whenever the application architecture is updated, either the **Software Designer** or the **QoS Engineer (QoSEng)** can automatically **verify** whether the architecture conforms to the constraints expressed in the Constraint Definition Language specification. This is supported by the Verification Tool primary mode. When a violation is encountered (e.g., circular calls, privacy violations), the **Software Designer** has two options, to (i) open the corresponding artifact(s) (in the RADON IDE) for debugging; or (ii), to correct the Constraint Definition Language specification using the recommended corrections automatically suggested by the Verification Tool.

Some specific constraints from examples of valid and invalid architectures may be fed and then automatically learned by the Verification Tool (see deliverable 4.2). Rather than requiring a **Software Designer** to write constraints manually, they can instead define example architectures and use the Verification Tool to learn the constraints. A final conformance check can be performed

by the **Release Manager** before deploying the application. **Figure 7** depicts the aforementioned workflow using the UML activity diagram notation.

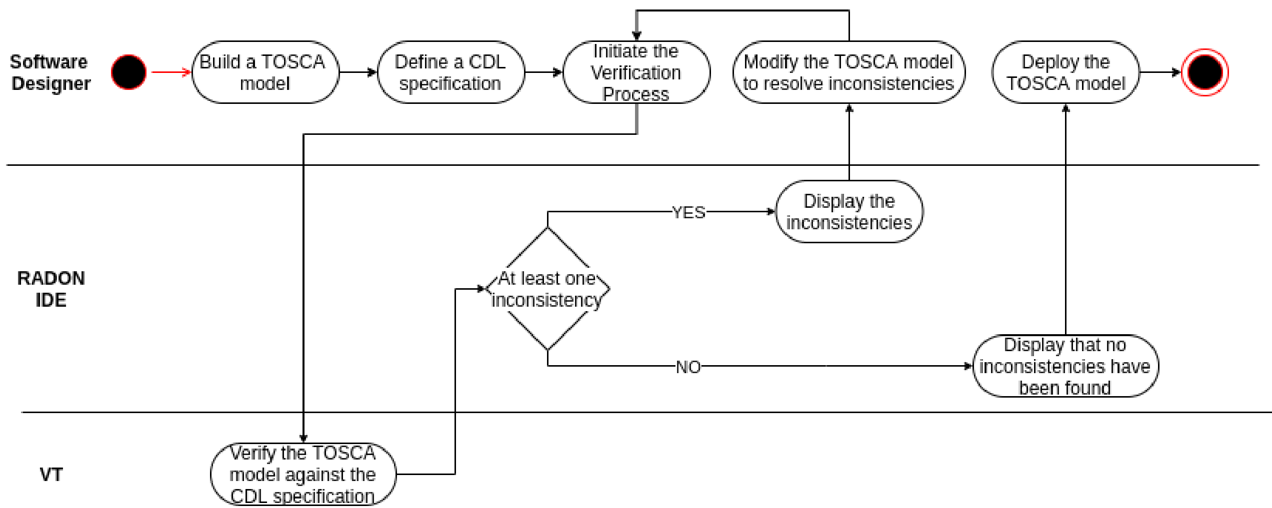

Figure 7 - Activity Diagram of the Verification Workflow.

3.2 Decomposition Workflow

Roles: Software Designer, Ops Engineer, QoS Engineer

Input: IaC blueprints, decomposition specification or runtime information

Output: A decomposed architecture

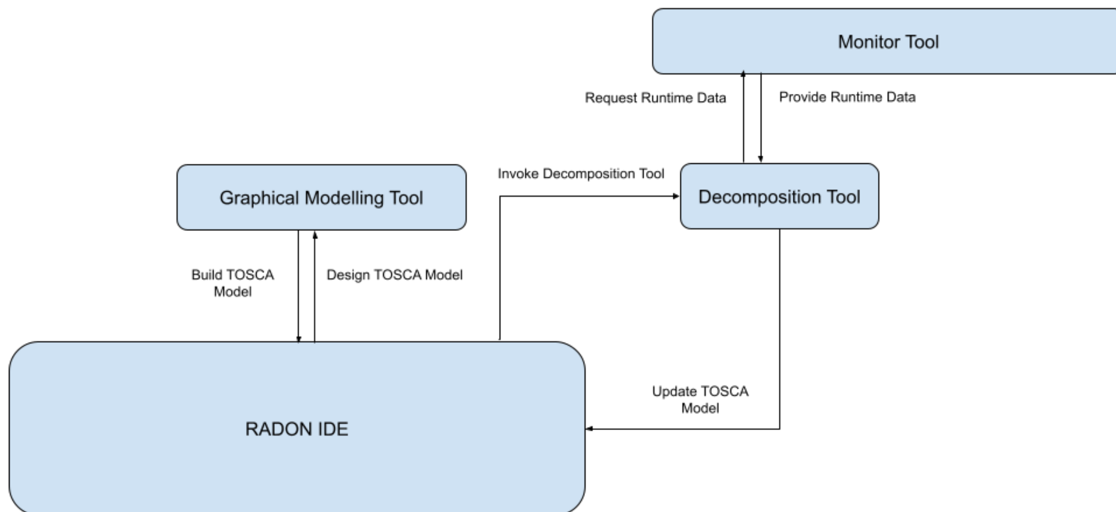

Figure 8 - RADON Decomposition Workflow.

After defining the application architecture, the **Software Designer** has already laid down the first sketch of the application or imported an existing **topology** into the Graphical Modeling Tool. The **Software Designer** can get suggestions on how to **decompose** and map abstract components to concrete technologies. This task is supported by the Decomposition Tool (integrated into the RADON IDE). Furthermore, (s)he can **decompose** and adjust the application **topology**, e.g., to split a monolith first into microservices and then into serverless functions.

This workflow supports three classes of refinements: (1) architecture decomposition, (2) deployment optimization, and (3) accuracy enhancement.

1. **Architecture decomposition.** This feature analyzes the topology of an application under development and suggests possible changes based on known architectural patterns, particularly breaking down a monolith into finer microservices or serverless functions.

2. **Deployment optimization.** In this step, the RADON methodology aims to allocate concrete physical resources (e.g., memory, compute) under the constraint of fulfilling quality requirements, which may have been stipulated in a service level agreement.

3. **Accuracy enhancement.** This feature enables the improvement of decomposition and optimization results through more accurate model parameterization using runtime data that represents how the serverless application actually works.

Figures 9 to **11** depict the corresponding procedures using the UML activity diagram notation.

Next, the automatically generated decomposition suggestions and/or the revised TOSCA model are shared with the **Software Designer** or **Operations Engineer** (OpsEng). The previous version of the model will be saved using the Template Library.

Recall that the lifecycle of a RADON application is iterative. Thus, the **QoS Engineer** may decide to **refactor** the application based on new knowledge of its behavior observed from the Monitoring Tool. Such knowledge can enrich the TOSCA model in the form of new properties or constraints, enacting a feedback loop.

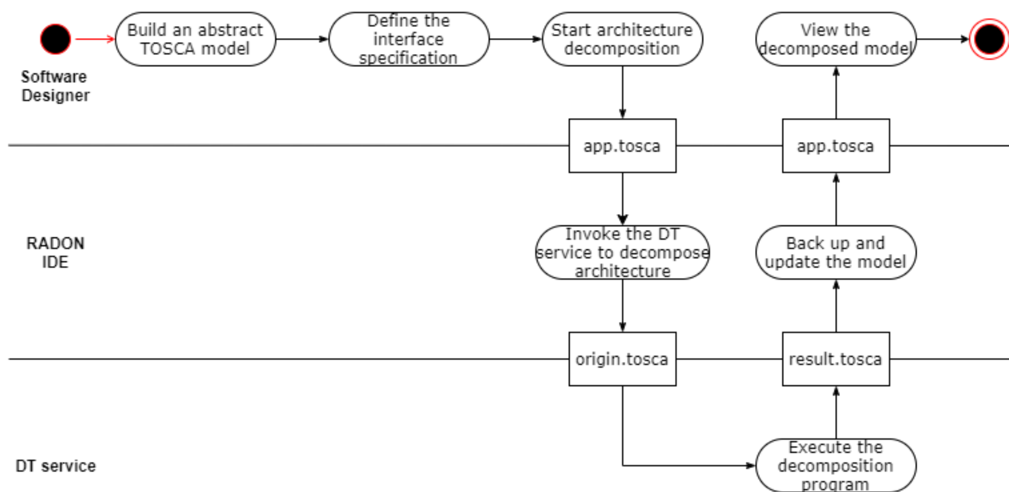

Figure 9 - Activity Diagram of the Decomposition Workflow (Architecture Decomposition).

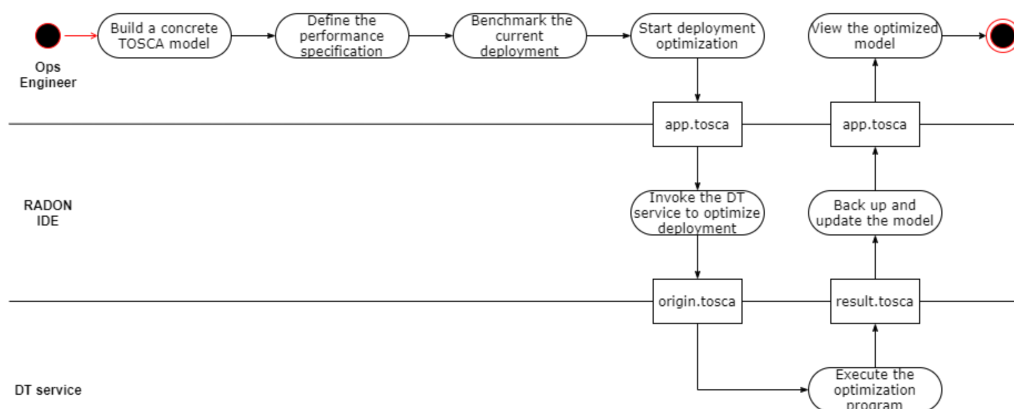

Figure 10 - Activity Diagram of the Decomposition Workflow (Deployment Optimization).

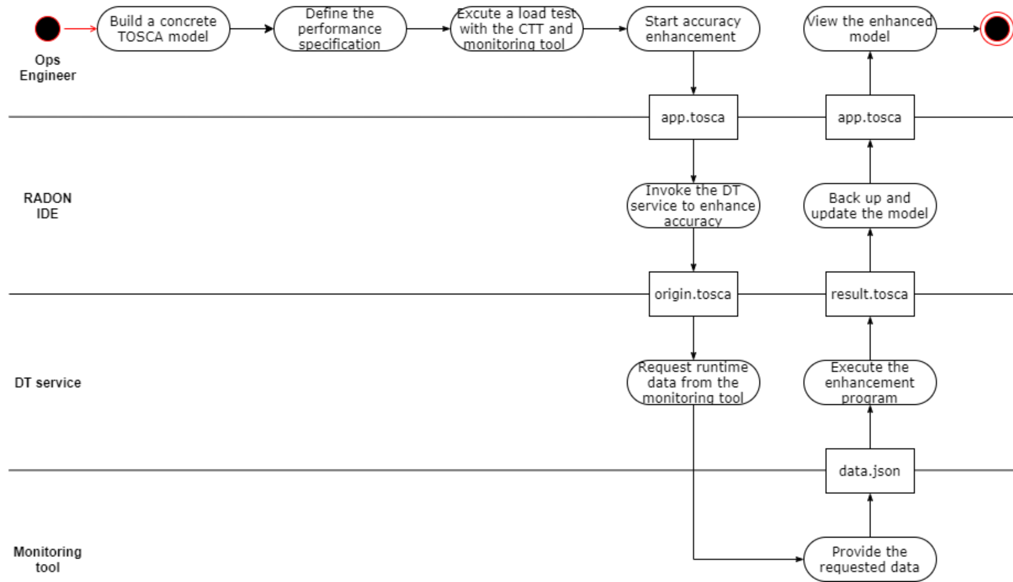

Figure 11 - Activity Diagram of the Decomposition Workflow (Accuracy Enhancement).

To optimize system performance and avoid bottlenecks in job placement, the Decomposition Tool can also use an interference estimation model to detect potential **interferences of co-locating jobs** in the system. This requires the **Ops Engineer** to define the **workload specification** in the TOSCA model and then start the execution of interference detection through the RADON IDE. The Decomposition Tool will feed performance metrics obtained from the Monitoring Tool to the interference estimation model and report any **job interferences** found to the **Ops Engineer.** This procedure is depicted in **Figure 12**.

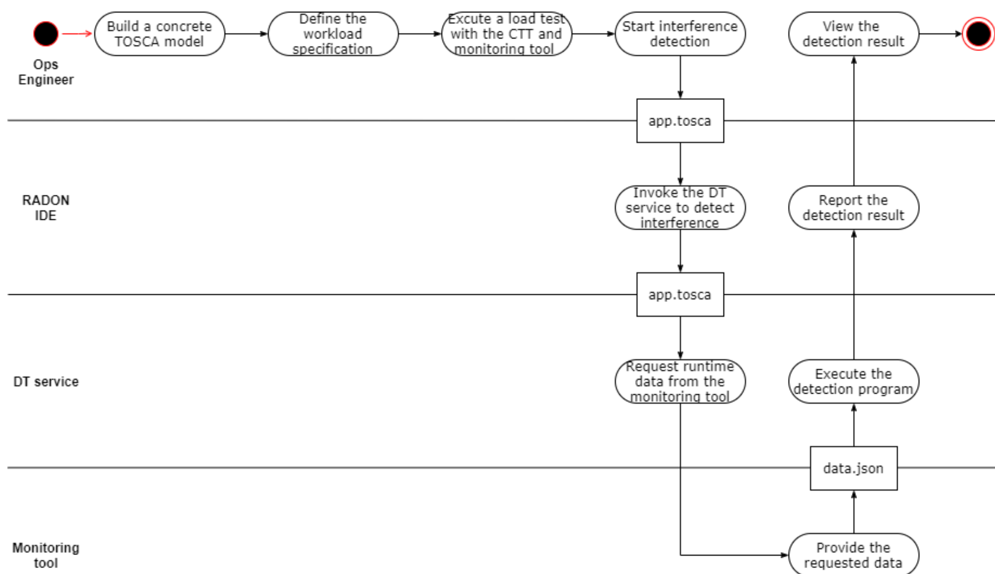

Figure 12 - Activity Diagram of the Decomposition Workflow (Interference Detection).

3.3 Defect Prediction Workflow

Roles: Software Developer, Ops Engineer

Input: IaC blueprints

Output: Failure-proneness and IaC defect metrics

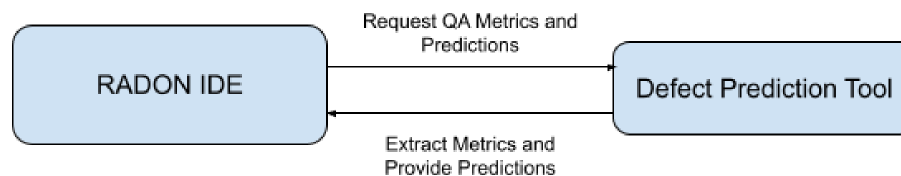

Figure 13 - RADON Defect Prediction Workflow.

The defect prediction workflow is purposed to pinpoint RADON developers to those parts of the **IaC blueprints** that are potentially **error-prone** and **deserve more attention** from a testing perspective. In particular, the aim is to identify **potential defects** in infrastructural configuration management files. To this end, the RADON methodology relies on a quality management approach that revolves around metrics applicable to IaC. In particular, to leverage the inspection of IaC scripts, the RADON methodology offers several pre-trained models and the option to train new detection models to the RADON developers, testers, and quality assurance staff. Supported by machine-learning developed models, the defect prediction workflow delivers a set of predictions and IaC metrics that help RADON developers, testers, and quality assurance staff to make informed decisions whether IaC scripts are error-prone or -free.

The **Software Developer**, who writes the application code, and the Ops Engineer**,** who creates and maintains the Infrastructure as Code scripts, can **revisit/refactor the application code any time** by invoking the Defect Prediction Tool through the RADON IDE.

As shown in **Figure 10**, the **Developer** or **Ops Engineer** develops a **blueprint** in Ansible or TOSCA. They **manually trigger** the detection of potential defects in the resulting artifact (a YAML file for Ansible or a CSAR archive for TOSCA).

Next, the following steps are performed **automatically** by the Defect Prediction Tool. Firstly, the appropriate metrics for the artifact at hand are extracted and subsequently passed to the API call for the prediction. In particular, it then runs five pre-trained models to effectively identify the respective defect types based on those metrics.

The plugin provides **immediate visual feedback** to the **Developer** and/or **Ops Engineer** through the RADON IDE by (i) showing an alert explaining the prediction, in the case at least one defect type is found, and (ii) highlighting critical metrics that are higher than the community standard.

Note that, for business logic, the developer can rely on **standard debugging tools** already available in the RADON IDE (e.g., linters).

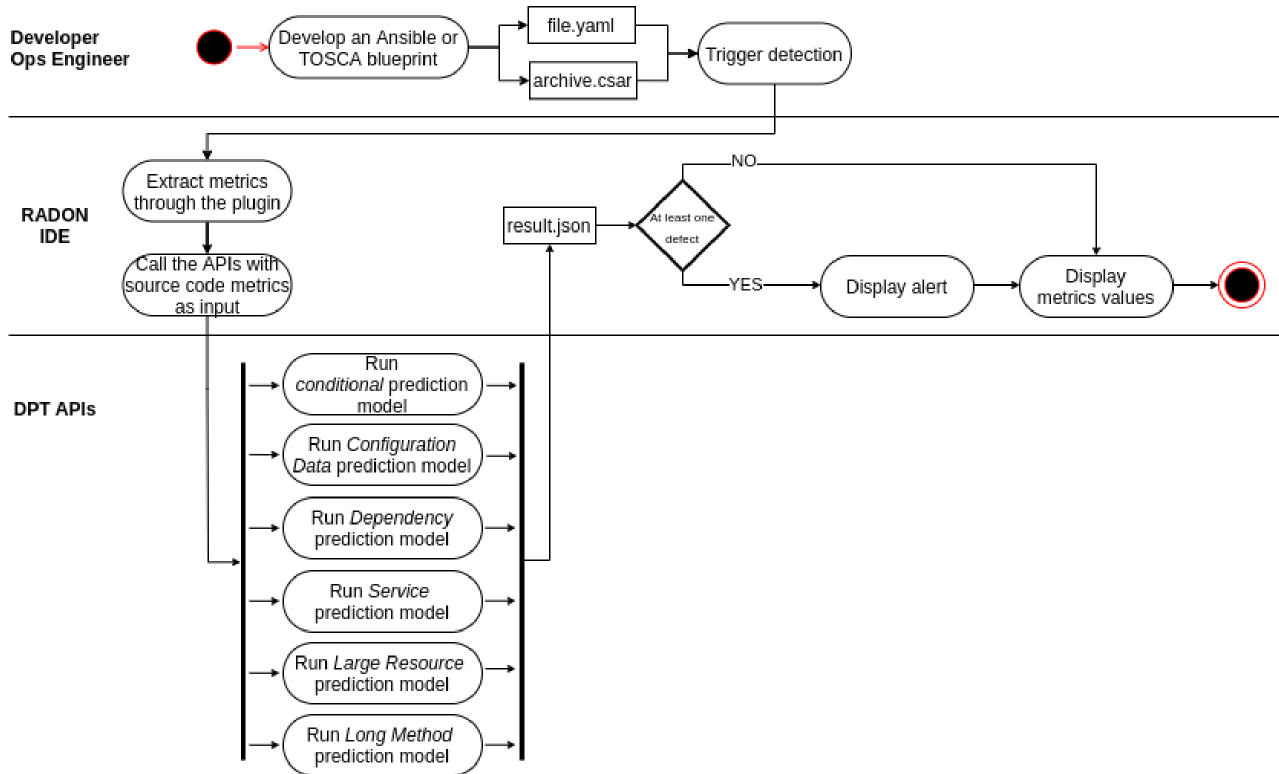

Figure 14 - Activity Diagram of the Defect Prediction Workflow.

3.4 Continuous Testing Workflow

Roles: Software Developer, Release Manager, QoS Engineer

Input: IaC blueprints and testing artifacts

Output: Test results

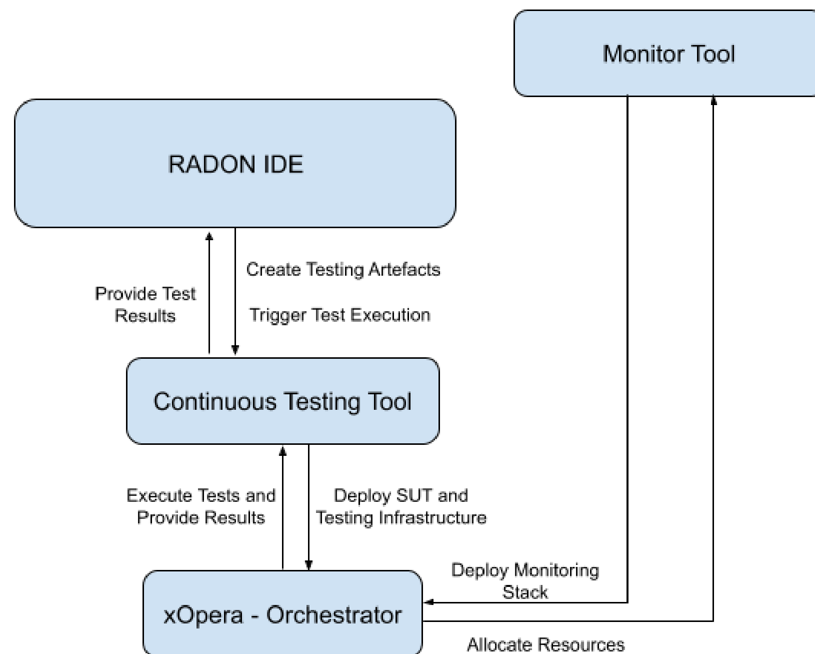

Figure 15 - RADON Decomposition Workflow.

Testing is cumbersome and erroneous for developers and testers. **Automated testing** can continuously test the application, referred to as the system under test (SUT), and operate in a more consistent, precise, predictable manner. In this way, developing new test cases while reusing them in future similar serverless application contexts, and analyzing failed test cases, can be delivered quickly and with less effort in a pre-set timeline. In particular, the **Software Developer** and/or **QoS Engineer** can specify automatic test specifications for parts of the system under test. Once specified, release managers can let them be executed, after which test results may be inspected.

As shown in **Figure 15**, the Continuous Testing workflow leverages the Continuous Testing Tool, xOpera, and the Monitoring Tool. The workflow is integrated into the RADON IDE.

The continuous testing workflow comprises three usage scenarios, namely *Define Test Cases (US 1)*, *Execute Test Cases (US 2*), and *Maintain Test Cases (US 3)*. **Figures 16** and **17** depict the activity diagrams that summarize and detail all usage scenarios.

Define Test Cases (US 1). In parallel to the regular development, the Software Developer or QoSEng can define test specifications (e.g., deployment and load tests) for their application. The definition of test specifications is done via the **RADON IDE** by adding respective TOSCA policy

types to the SUT's service template. Moreover, the developer defines an additional service template for the test infrastructure (TI). The resulting artifacts, comprising the SUT's and TI's CSAR files and input definitions, can be exported from the **RADON IDE**.

Execute Test Cases (US 2). During development or before deploying to production, actors such as the Developer or Release Manager can manually trigger the execution (via the **RADON IDE** or the standalone interface) or automatically (via **CI/CD** for integration into DevOps processes). In each case, the **Continuous Testing Tool** conducts a series of steps for each selected test case, namely preparing the project context, generating executable artifacts (CSARs), deploying the SUT and the TI via the Orchestrator, executing the tests, and collecting the results. Afterward, the test results can be inspected.

Maintain Test Cases (US 3). Once the application is deployed in a production environment, operational data can be used to generate, refine, and select test cases, to fit into DevOps contexts, such as evolving system usage, limited test budgets in **Continuous Integration/Deployment**. Even though different approaches for maintaining test cases are provided in the continuous testing workflow, they share a similar process of analyzing the intended user request, querying the Monitoring Tool for the required monitoring data, and providing the generated/refined test artifacts.

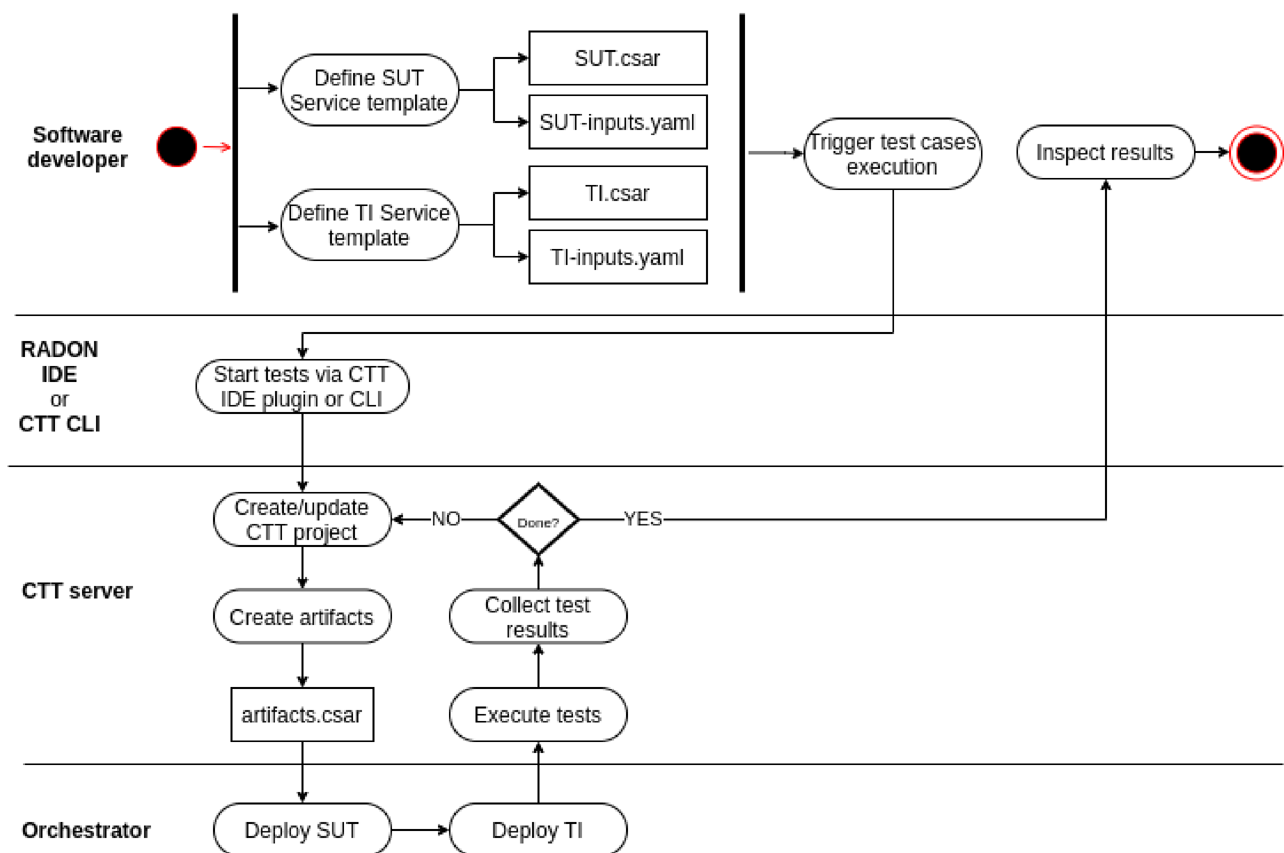

Figure 16 - Activity Diagram of the Continuous Testing Workflow (Use Case 1-2).

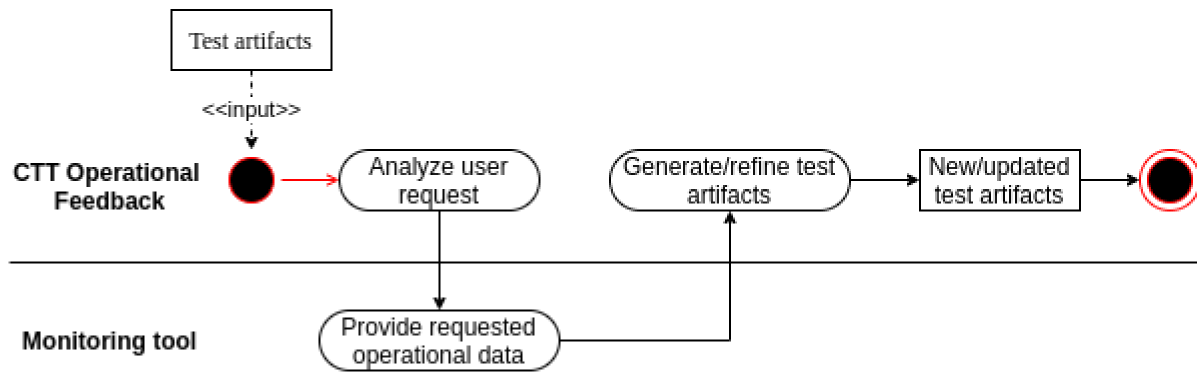

Figure 17 - Activity Diagram of the Continuous Testing Workflow (Use Case 3).

3.5 Monitoring Workflow

Roles: Ops Engineer, QoS Engineer

Input: IaC blueprints

Output: Alerts generated when specific thresholds are violated

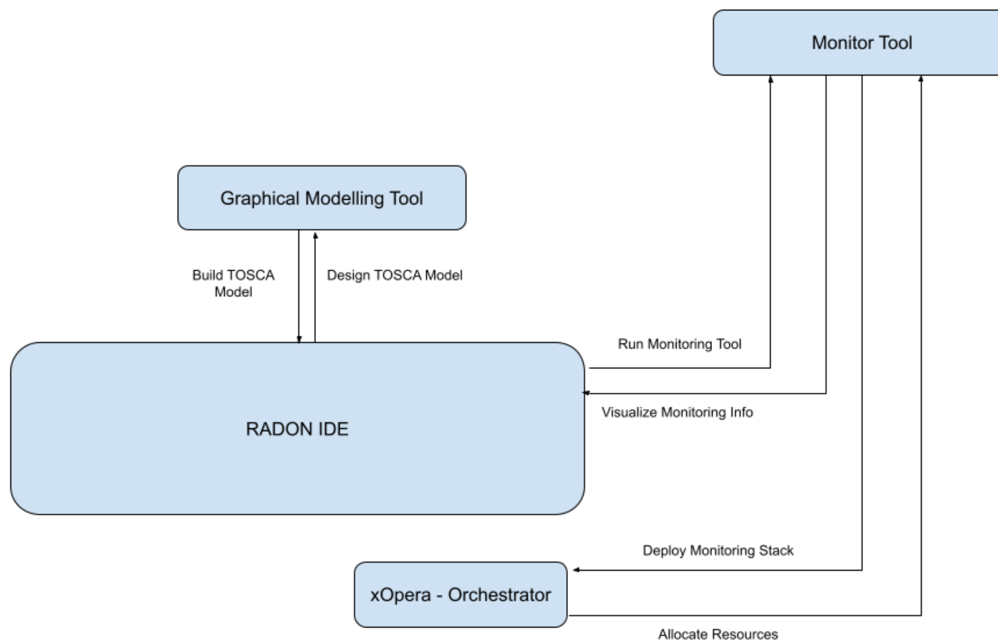

Figure 18 - RADON Monitoring Workflow.

Monitoring the performance of the IaC blueprints is of critical importance to the all RADON lifecycle methodology roles, ranging from developers, testers, and integrators to quality engineers and release managers. During this lifecycle model phase, data is logged and recorded about the usage and performance of the RADON applications and IaC, continuously monitoring each functionality. Monitoring is directed through annotated TOSCA models to express which metrics (e.g., throughput) should be measured.

As such the monitoring workflow is instrumental in sustaining QoS, e.g., in terms of availability, scalability, security and performance of FaaS/micro-services in the RADON application. Exploiting this logged data threats and their root-causes can be detected and resolved in the continuous testing RADON lifecycle model phase.

The **OpsEng** sets up necessary monitoring support using simple annotations in the TOSCA models. The **metrics** to monitor will be defined by the QoSEng, ranging from resource utilization, throughput, and response time.

As depicted in **Figure 19**, the **Orchestrator** deploys the nodes with the necessary logic to gather the monitoring metrics defined during development, gathered and stored in a monitoring database being maintained by the **Monitoring System**.

The application does not need to feature any complex control logic in the production Cloud environment, aside from some limited autoscaling form, thanks to pre-defined autoscaling rules (by the QoSEng) and the information gathered from the **Monitoring System**. Otherwise, such information would be used in the feedback loop to improve the suggested decompositions and optimizations by the **Decomposition Tool** and provide expected parameters for the **Continuous Testing Tool**.

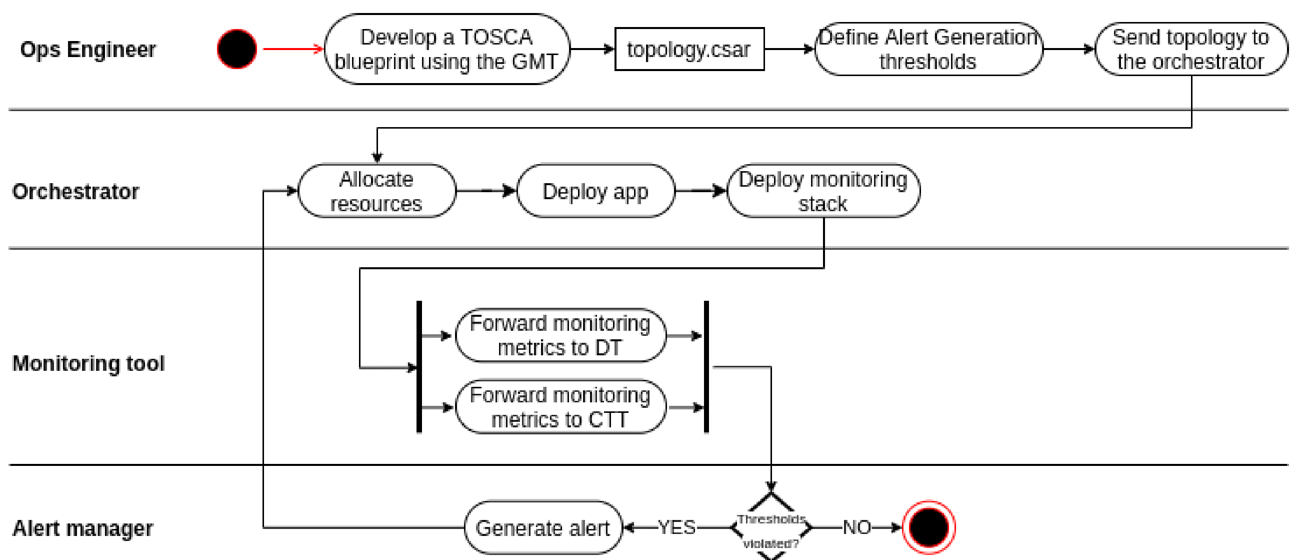

Figure 19 - Activity Diagram of the Monitoring Workflow

3.6 Continuous Integration and Delivery Workflow

Roles: Ops Engineer, Release Manager

Input: Application specification

Output: A deployed serverless application

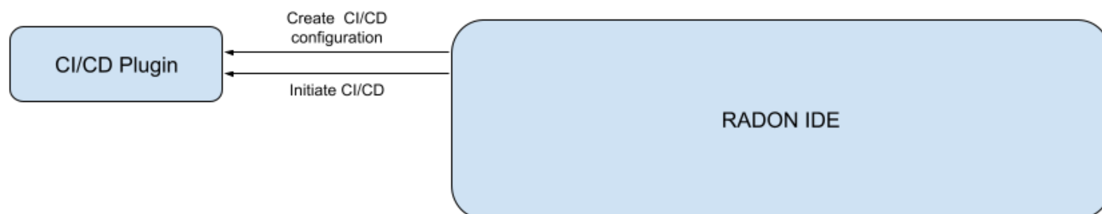

Figure 20 - RADON Continuous Integration and Delivery Workflow

The Continuous Integration and Deployment Workflows play a pivotal role in the RADON lifecycle methodology, stitching together all other workflows. In particular, during **continuous code integration**, novel code implementing add-on or modified functionalities are virtually-constantly fused with the existing serverless-computing IaC code base. During RADON's **integration workflow,** developers can use the RADON tools (e.g., defect prediction tool) for quality assurance.

RADON's **continuous delivery** lifecycle model phase helps speed up packaging and shipping new application features, reducing cycle time and feedback loops. In addition, by continuously deploying code snippets of limited size, introducing faults and errors can be significantly reduced, leveraging quality and maintainability.

The CI/CD Workflow plumbs together the different tools and artifacts to move from development to runtime stages by running the Builds and Tests in each step of the development process to ensure smooth integration and deploy the application to production. The workflow is backed by automation servers such as Jenkins and CircleCI.

In parallel to the regular application development, the **OpsEng** can create a **CI/CD Pipeline** using a selection of the RADON CI/CD templates available in a RADON repository[7]. Based on the application's specification, the OpsEng can construct a CI/CD pipeline that:

- Performs appropriate application testing with the **Continuous Testing Tool**.
- Verifies application constraints compliance with the **Verification Tool**.
- Analyzes the exported CSAR files and predicts code defects with the **Defect Prediction Tool**.
- Ensures accessibility of versioned pre-packaged deployment packages with the **Template Library Publishing System**.

[7] https://github.com/radon-h2020/radon-cicd-templates

- Deploys the application to production with the **xOpera Orchestrator.**

Once the **Release Manager** has approved the release, the **OpsEng** can use the GMT to export and release the TOSCA model(s) of the application that will reach the Orchestrator (e.g., xOpera) via the CI/CD Pipeline for instantiation on the target cloud(s). Information from the Monitoring System can be used to check the status of the deployed nodes. **Figure 21** depicts this workflow using the UML activity diagram notation.

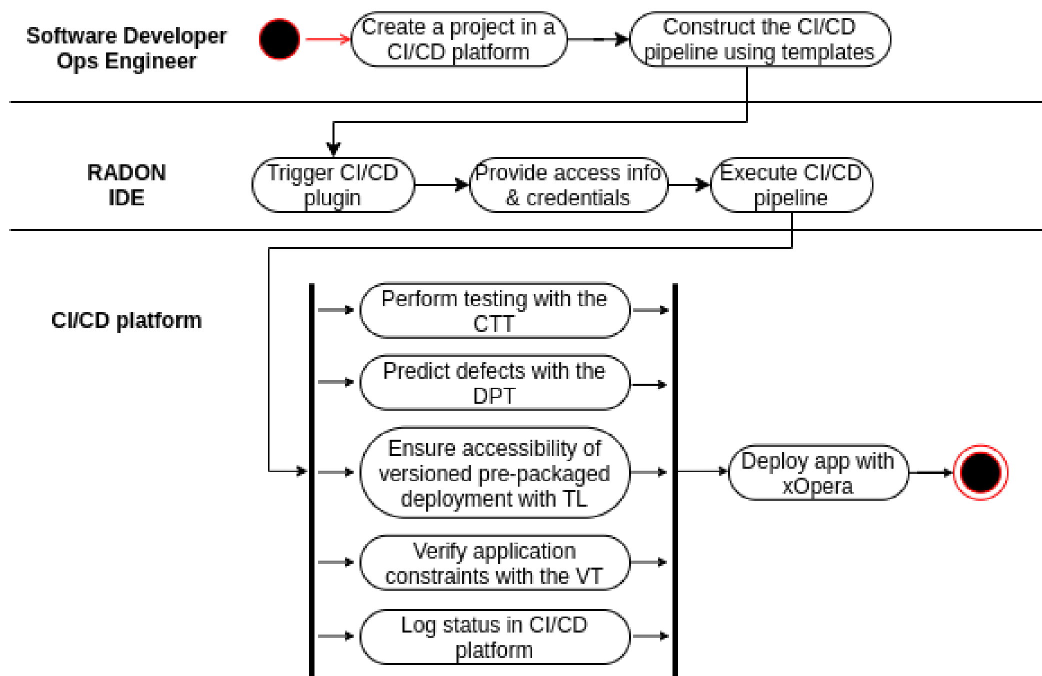

Figure 21 - Activity Diagram of the Continuous Integration and Delivery Workflow

4. Tools Overview

This section summarizes Deliverable D2.4, in which RADON tools are described in detail. Some of the tools are design-focused, while some are runtime-oriented. Finally, some have both design and runtime aspects. **Table 5** outlines the RADON tools, mapping them to this categorization scheme. Most of the tools relate to the RADON IDE for end-user usage but also offer standalone interfaces.

Table 5 - Overview of the RADON components (derived from D2.4)

Name	Description	Phase
RADON IDE	A web-based multi-user development environment that integrates the RADON tools.	Design time
Graphical Modeling Tool	A web-based tool to graphically model TOSCA applications.	Design time
Verification Tool	A tool for verifying whether a RADON model conforms to a specification expressed in the Constraint Definition Language.	Design time
Decomposition Tool	A tool for architecture decomposition, deployment optimization, and accuracy enhancement.	Design time Runtime
Defect Prediction Tool	A tool that focuses on IaC correctness and smells detection.	Design time
Continuous Testing Tool	A tool for continuous design, evolution, deployment, and execution of tests.	Design time Runtime
xOpera SaaS	A tool for processing and executing the TOSCA service templates packaged in a Cloud Service Archive (CSAR).	Runtime
Template Library	A shared repository for templates, blueprints, and modules required for the application deployment.	Design Time Runtime
Monitoring System	A back-end service-based system to collect evidence from the runtime environment to support quality assurance.	Runtime
Function Hub	A repository to store versioned plug-and-play FaaS packages.	Design time
CI/CD Plugin	A plugin to enable CI/CD based on predefined RADON tool pipeline templates.	Runtime
Data Pipeline Plugin	A plugin that ensures the functioning of data-pipeline-based CSAR files before they are deployed.	Runtime

4.1 RADON Integrated Development Environment

Overview. The RADON Integrated Development Environment (IDE) provides a development environment for multi-user usage. Based on the Eclipse Che technology, the RADON IDE supports standard (web-based) development activities (such as support for different programming languages, debugging functionalities, source code editors). Furthermore, it provides a front-end to interact with the RADON framework and its tools and access to the shared spaces of the RADON artifacts (i.e., RADON models).

High-level architecture. The Eclipse Che development environment has been customized to realize a new RADON Stack (i.e., a runtime configuration) defining a RADON workspace. A RADON workspace is characterized by a set of plugins, projects, and Kubernetes containers implemented to customize the development environment and integrate the RADON tools accordingly to the project needs.

The RADON IDE comprises several Eclipse Che plugins that integrate the RADON tools. These plugins add capabilities to the Eclipse Che GUI and permit interaction with the RADON tools. Moreover, some Kubernetes components have been defined in the RADON workspace to integrate services of some RADON tools (i.e., GMT, VT, CTT) in the IDE "backend".

Finally, the RADON workplace is also characterized by a project (named "**radon-particles**") that clones in the RADON workspace the TOSCA modeling entities from the RADON Particles GitHub repository. A detailed description of the RADON IDE is provided in the deliverable D2.7.

4.2 Graphical Modeling Tool

Overview. The Graphical Modeling Tool (GMT) is a web-based tool to manage TOSCA entities and blueprints. The tool is based on Eclipse Winery, and during the RADON project, we contributed all our changes back to the Eclipse open-source community. GMT provides an end-user-friendly and graphical syntax agnostic modeling of TOSCA application topologies where no in-depth knowledge of the underlying TOSCA syntax is required. It is compatible and compliant with the OASIS TOSCA standard, the older TOSCA XML, and the newer TOSCA YAML standard. Furthermore, GMT generates executable blueprints in the form of TOSCA Cloud Service Archives (CSARs). Additionally, GMT is designed to be integrated into IaC workflows by its HTTP REST API.

High-level architecture. Figure 22 shows a high-level architecture diagram of the GMT. In general, GMT is divided into a frontend and a backend part. The frontend consists of two web applications: Entity Management and Topology Modeler. The Entity Management UI is used to manage and maintain all required TOSCA entities, such as node types, policies, requirements, capabilities, and data types. The Topology Modeler UI is used to compose a TOSCA topology graphically. Topologies represent the structure of applications and are defined in GMT by dragging and dropping desired node types onto the canvas to instantiate respective node templates. Further, the UI is used to set a node's property values and assign respective artifacts for deployment.

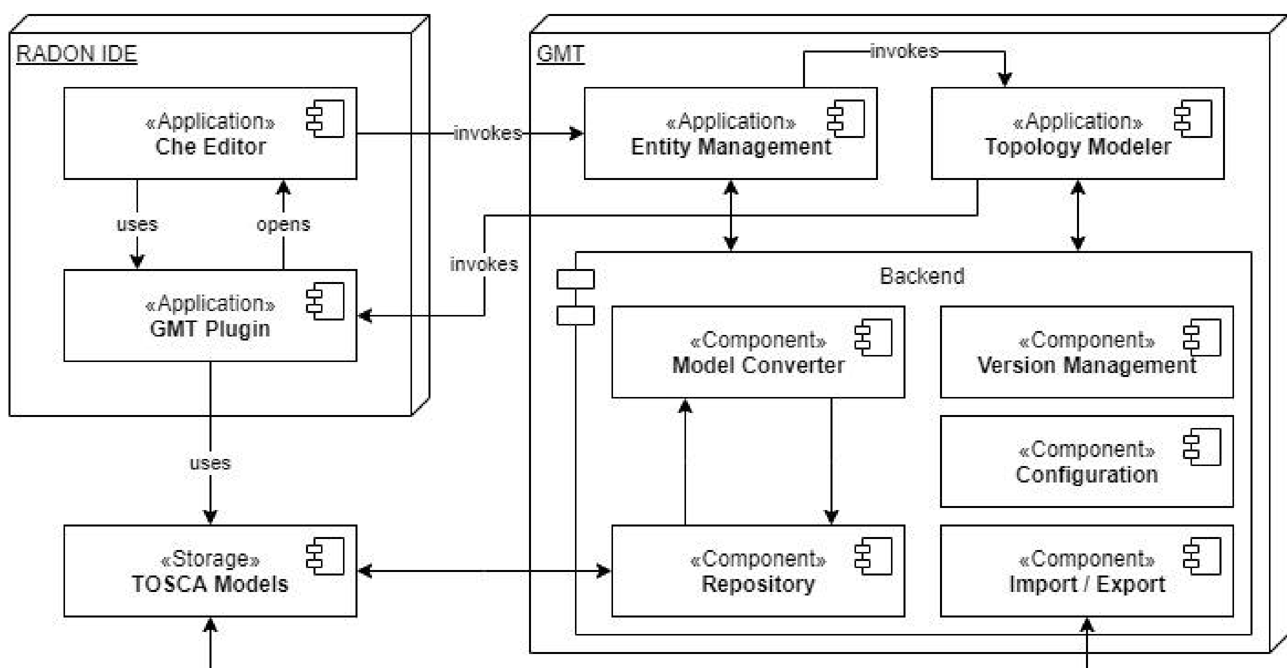

Figure 22 - Architecture of the Graphical Modeling Tool.

The frontend components communicate through the HTTP REST API with the backend. There are respective endpoints to list, create, update, or delete TOSCA entities and service templates. The backend can be configured to employ the older TOSCA XML standard or the latest TOSCA YAML version. In RADON, the GMT uses a configuration to operate on the latest TOSCA YAML and serialize all TOSCA entities into a file-based repository structure versioned with Git. Internally, a Canonical Model and the Model Converter components implement the backward compatibility. GMT also enables users to generate modeled applications in CSARs and download them to their local environments. Additionally, generated CSARs can be saved to the underlying filesystem on which the GMT is running, e.g., saving the CSAR to the Eclipse Che workspace, which runs a GMT instance. Any TOSCA-compliant orchestrator can process the generated CSARs. Likewise, CSAR files can be imported to other GMT instances to enable collaboration.

Intended use as a standalone component. The GMT was designed as a standalone tool enabling graphical modeling of cloud application deployment models based on the OASIS TOSCA specification. GMT supports both XML- and YAML-based TOSCA versions and provides graphical user interfaces for (i) managing one or more TOSCA model repositories and (ii) modeling cloud application topologies. Furthermore, GMT provides REST APIs for accessing the features programmatically (e.g., CSAR import and export) and facilitating the integration with TOSCA-compatible orchestrators and tools. GMT is not tightly-coupled with any specific tool and can be used as a generic TOSCA modeling software.

A detailed description of the Graphical Modeling Tool is provided in deliverables D4.5 and D4.6.

4.3 Verification Tool

Overview. The primary purpose of the Verification Tool (VT) is to allow a user to check that a given TOSCA model conforms to a set of functional and non-functional requirements, expressed in the RADON Constraint Definition Language (CDL). Its main verification modes include correction and learning, repairing invalid TOSCA models, and extending incomplete CDLs specifications (respectively).

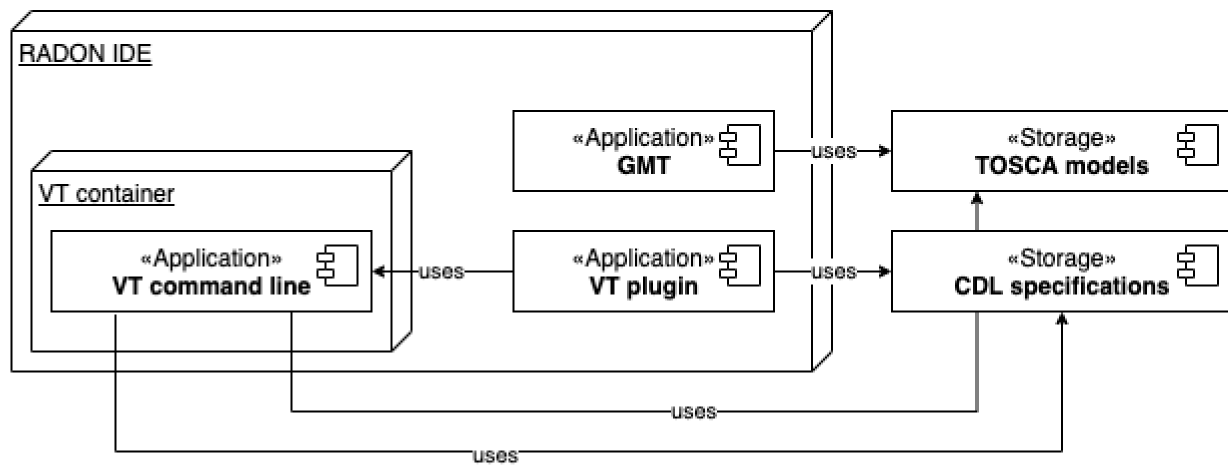

Figure 23 - Architecture of the Verification Tool.

High-level architecture. All three modes of the VT take as input TOSCA models and CDL specifications. TOSCA models can be created and edited within the RADON IDE using the Graphical Modelling Tool (GMT). The CDL specifications can also be created and edited using the Eclipse Che text editor, part of the RADON IDE. Once this has been done, the user can invoke each mode of the VT using the VT plugin (which allows a user to call the tool by right-clicking on a CDL specification and choosing one of the VT's execution mode). This causes a call to be made to a light-weight web server running on the VT Docker container. This webserver pulls the relevant files from the shared storage and invokes the command line VT. The result is returned to the VT plugin and displayed to the user.

Intended use as a standalone component. Although the tool has been fully integrated within the RADON IDE and benefits from the GMT's graphical approach to defining TOSCA models, all three VT modes are available within the command-line version of the tool. This standalone command-line tool can be applied without the need for the other parts of the RADON toolchain.

A detailed description of the Verification Tool is provided in deliverables D4.1 and D4.2.

4.4 Decomposition tool

Overview. The Decomposition Tool (DT) helps RADON users to find the optimal decomposition solution for an application based on the microservices architectural style and the serverless FaaS paradigm. It is typically used in four different scenarios: (i) architecture decomposition, (ii) deployment optimization, (iii) accuracy enhancement, (iv) interference detection.

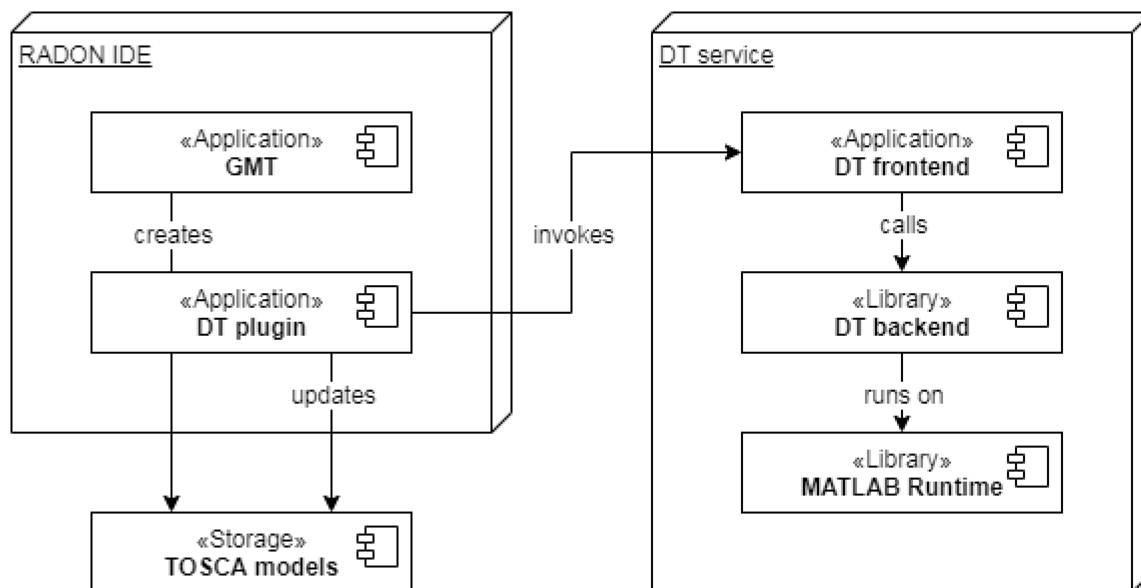

Figure 24 - Architecture of the Decomposition Tool.

High-level architecture. Figure 24 shows the component architecture of the DT. Provided that the DT plugin is installed in the RADON IDE, one can create a TOSCA model with the Graphical Modeling Tool (GMT) and then *Decompose*, *Optimize*, *Enhance* or *Detect* it by clicking the corresponding button in the context menu of the service template (.tosca). This action will trigger the DT plugin to invoke the DT service, which is publicly available. The DT service conceptually consists of two layers, namely frontend and backend. The former is a Spring application that provides a RESTful API, while the latter is a Java library that implements actual functionalities. Since the DT backend is compiled from MATLAB code, it needs to run on top of the MATLAB Runtime.

Intended use as a standalone component. As aforementioned, the DT has been deployed as a public service with a RESTful API[8] and therefore can be used as a standalone tool outside the RADON IDE.

A detailed description of the Decomposition Tool is provided in deliverables D3.2 and D3.3.

[8] https://github.com/radon-h2020/radon-decomposition-tool

4.5 Defect Prediction Tool

Overview. The RADON IaC Defect Prediction Tool strives to tackle correctness in designing applications based on serverless computing. In particular, it is designed to help DevOps engineers to allocate effort and resources more efficiently during Quality Assurance activities by prioritizing their inspection efforts for IaC scripts that might be failure-prone.

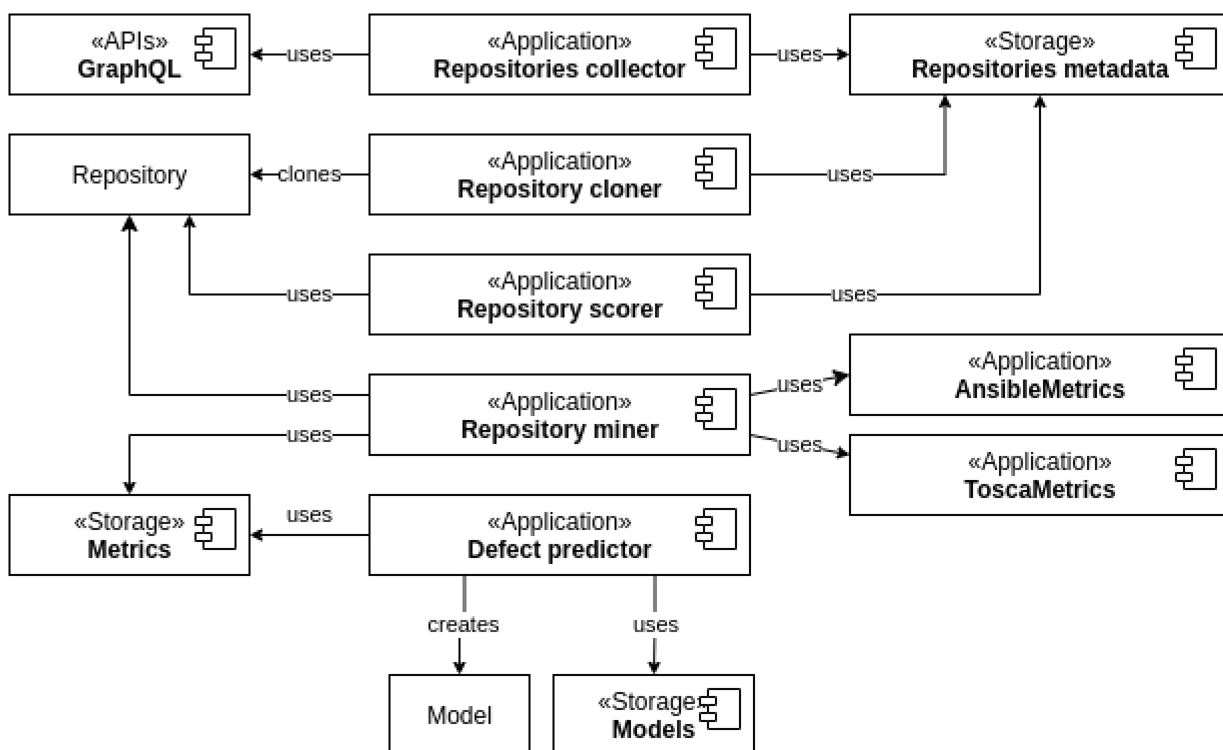

Figure 25 - Architecture of the Defect Prediction Tool.

High-level architecture. The Defect Prediction Tool consists of a fully integrated Machine-Learning-based framework that allows for repository crawling, metrics collection, model building, and evaluation. It consists of six main individual components:

- The *Repositories collector* collects active IaC repositories on GitHub.
- The *Repository scorer* computes repository metrics based on best engineering practices used to select relevant repositories to generate pre-trained defect prediction models.
- The *Repository miner* mines failure-prone and neutral IaC scripts from a repository. It gathers a broad set of metrics from the source code (e.g., lines of code and number of configuration tasks) computed upon the collected IaC scripts to predict their failure-proneness.

- The *AnsibleMetrics* and *ToscaMetrics* extract source code metrics from Ansible and TOSCA blueprints, respectively.
- The *Defect predictor* pre-processes the datasets and trains the Machine Learning models. Given an unseen IaC script, this component classifies it as *failure-prone* or *neutral*. With the identified defect type (e.g., a defect related to configuration data or service), when possible.

More specifically, the repositories collected by the *Github IaC Repositories Collector* are passed as input to the *Repository Scorer* to pick relevant repositories. Afterward, the *IaC Repository Miner* mines the selected repositories. Its output, consisting of observations of *failure-prone* and *neutral* IaC scripts for the individual repositories, is used by the *IaC Defect Predictor* to build and evaluate the models. Finally, the models are served online for use, that is, download a pre-trained model and use it to predict unseen instances.

Intended use as a standalone component. The tool has been designed as a standalone component to be integrated into CI/CD pipelines and deployed online and served through RESTful APIs. It is wrapped in a Docker container for the sake of portability. More specifically, once pulled the Docker image, the user can run the defect prediction command-line interface to (1) train a new model from scratch; (2) download a pre-trained model from the online APIs; and (3) use the trained or pre-trained model to predict the failure-proneness of new IaC scripts.

A detailed description of the Defect Prediction Tool is provided in deliverables D3.6 and D3.7.

4.6 Continuous Testing Tool

Overview. The Continuous Testing Tool (CTT) provides the functionality for defining, generating, executing, and refining continuous tests of application functions, data pipelines, and microservices and reporting test results. CTT integrates with other RADON tools and can be used as a standalone tool.

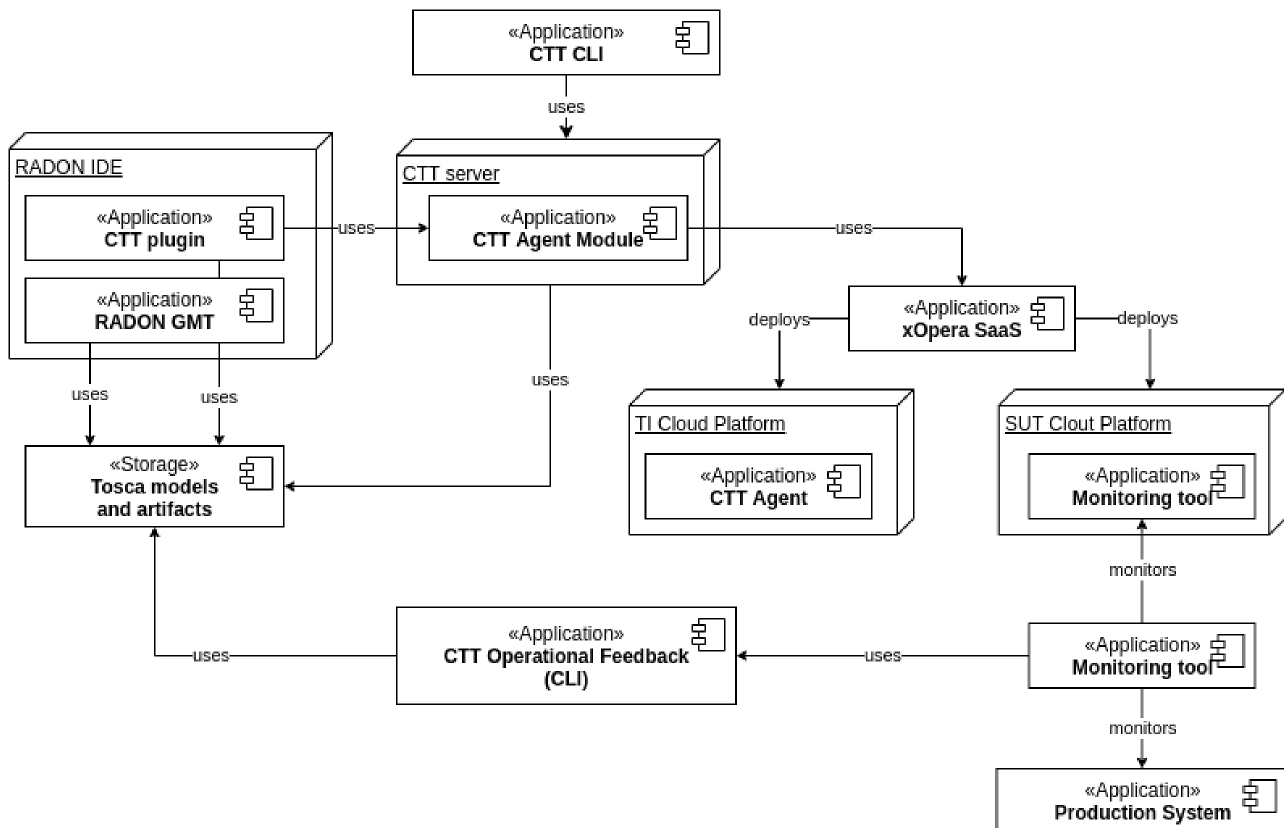

Figure 26 - Architecture of the Continuous Testing Tool.

High-level architecture. Tests and test infrastructures are defined in TOSCA models, using CTT-based (extensible) modeling types, e.g., supporting performance or deployment tests. The CTT server is responsible for managing the test execution workflow, including the generation of executable TOSCA models, the deployment of the test infrastructure (TI), and the system under test (SUT). It uses the xOpera (SaaS) orchestrator to deploy the application. After this step, the tests are run via the CTT test agents. Finally, the test results are obtained and provided to the user. A CTT agent executes the actual test, e.g., a load test with load drivers such as JMeter. The pluggable agent architecture simplifies creating custom testing modules, such as NiFi based agents for testing data pipelines. The RADON IDE (including RADON's GMT) can be used to define the test-related information, such as the tests and the TIs, and interact with the CTT server via the CTT IDE Plugin. CTT CLI provides an alternative command-line interface to interact with the CTT server. CTT CLI is the preferred way to use CTT in CI/CD pipelines.

Moreover, CTT provides various approaches for generating and updating load tests from operational data, mainly to obtain tailored tests for microservices, FaaS, and data pipelines. Therefore, it interfaces with a monitoring tool, such as the one included in the RADON ecosystem.

Intended use as a standalone component. CTT can be used as a standalone component via the REST-based and CLI-based interfaces. The dependency to xOpera orchestrator remains. The CTT components are provided as Docker containers for easy deployment and portability.

A detailed description of the Continuous Testing Tool is provided in deliverables D3.4 and D3.5.

4.7 xOpera SaaS - Orchestrator

Overview. xOpera SaaS is an advanced TOSCA orchestrator available as a service and built on top of the xOpera's orchestrator core engine. The SaaS component provides a fresh xOpera orchestrator environment to each deployment project, to which the owner can attach new users or secrets. The ability to run *as a service* makes the deployment projects available to teams of users. They can directly use or monitor the project through GUI or integrate the SaaS using the API into the CI/CD workflow.

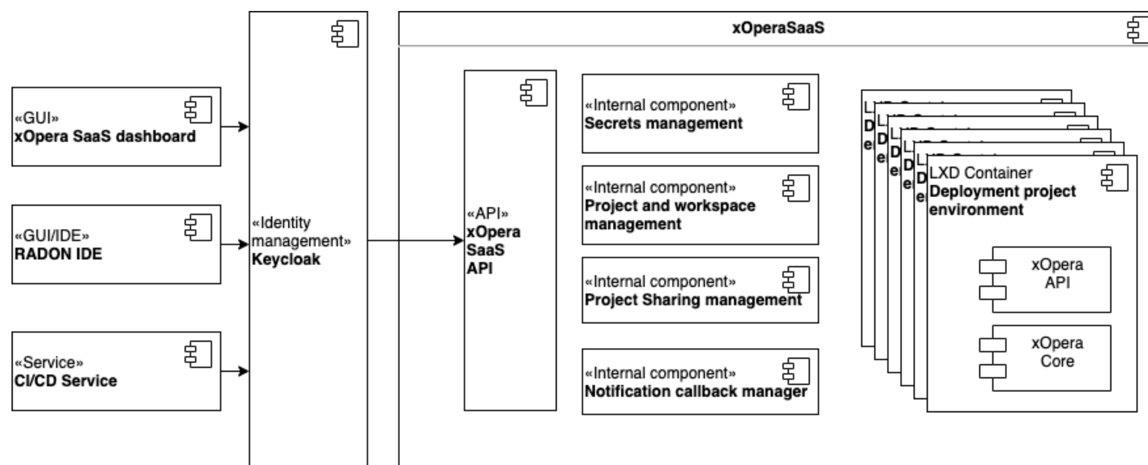

Figure 27 - Architecture of the XOpera SaaS - Orchestrator

High-level architecture. xOpera SaaS comprises the **xOpera SaaS API** which is an entry point for invoking any task inside the xOpera SaaS, and the **runtime environments** which are LXD containers for each xOpera project that isolates each deployment. In addition, it provides the following **management components**:

- **Secrets management** takes care of storing secrets and attaching those to the particular xOpera workspaces and project environments
- **Project and workspace management** takes care of creating new workspaces and new projects
- **Project sharing management** component allows workspace owners to share the workspaces with the users and those creating the deployment teams
- **Notification callback managers** provide mapping between the xOpera public URLs where monitoring can send the notification and the particular project environment triggers.

Other components in the diagram present the available integrations and interfaces to the xOpera SaaS which leverage xOpera SaaS API: (i) Keycloak is used by xOpera Saas for identity management xOpera SaaS; (ii) xOpera SaaS GUI provides xOpera SaaS management through the Web; (iii) RADON IDE manages xOpera SaaS operations; (iv) the CI/CD services manage xOpera SaaS operations.

Intended use as a standalone component. xOpera SaaS can be used standalone even without RADON, as a user can choose to use it directly through the API or GUI or any other Eclipse Che instance. A detailed description of the use is described in D5.1 and D5.2 or in the online documentation[9].

[9] https://xlab-si.github.io/xopera-opera/saas.html

4.8 Template Library

Overview. The template library is a place for storing, publishing, and sharing TOSCA modules and blueprints. Users can manage their templates in their local storage, community repositories, e.g., RADON particles on GitHub or in the Template Library Publishing Service (TPS). As the management of the content in local folder or git repository is quite straight forward for the GMT and RADON IDE, we focus here on the TPS, which provides the ability to search for templates, filter them by different parameters, and download/publish TOSCA content.

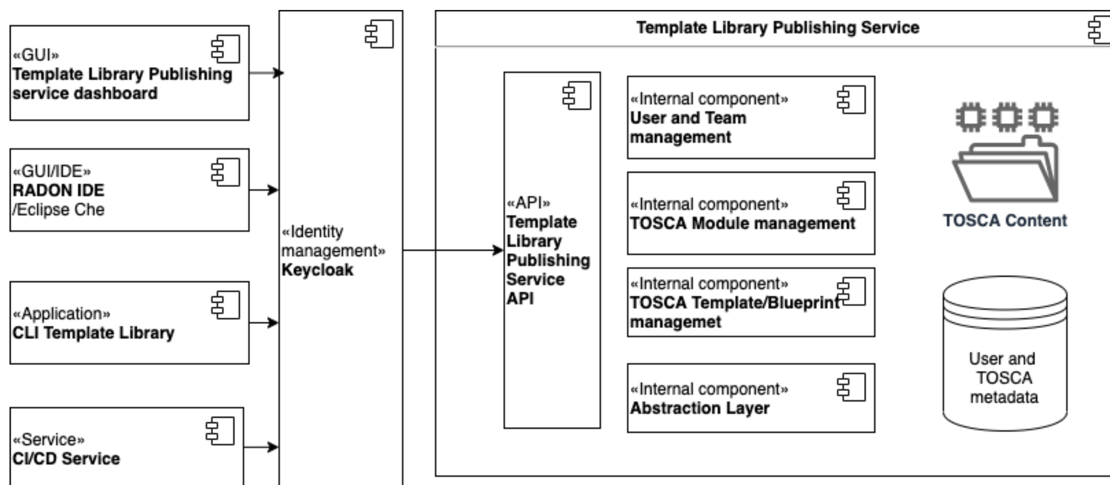

Figure 28 - Architecture of the Template Library Publishing Service

High-level architecture. The Template library consists of a **Template Publishing Service (TPS) API** which is an entry point that enables invoking different tasks inside the Template library by processing user's requests; **TOSCA Content**, which represents an object storage for all files from TOSCA module's versions; **User and TOSCA metadata** that use a relational database to store the information about users and groups. In addition, it provides the following **internal management components**:

- **User and Team management** that takes care of user information and the possibility of organizing users into user groups.
- **TOSCA Module management** is used to look over user's TOSCA modules of different types (e.g. nodes, relationships, blueprints and so on), which can be then used independently or as parts of a bigger application.
- **TOSCA Blueprint/Template management** brings semantic versioning for TOSCA modules of simple templates or complex application packages (TOSCA CSARs).
- **Abstraction layer** which allows users to organize their modules of templates into (private) template groups (e.g. AWS, Azure, GCP templates) which are then given access to the groups of users.

The available integrations and interfaces to the Template Library, which leverage TPS API, are presented by the following components, represented in **Figure 28**.

The Keycloak identity management, which is used to authenticate to TPS (using user IDs) from Eclipse Che's KeyCloak or any other OpenID capable tool, that can be registered as identity provider for Keycloak. The TPS dashboard brings a GUI component that provides TPS management through the Web and also offers some public access to public TPS modules. The RADON IDE allows users to invoke TPS actions through a simple TPS Eclipse Che plugin. The TPS command line interface is a console management tool for TPS. Finally, the CI/CD services allow for the management of Template library operations.

Intended use as a standalone component. The TPS can be used as a standalone tool for TOSCA template management, including publishing, searching for, and downloading TOSCA templates. Templates published in TPS can be public or private, which means that owners can share only inside a limited user group inside the enterprise or business partners. A detailed description of the use is described in D5.3 and D5.4 or in the online documentation[10].

[10] https://template-library-radon.xlab.si/docs/

4.9 Monitoring Tool

Overview. The Monitoring Tool (MT) provides all functionality for defining all necessary infrastructure to monitor the efficiency and performance of the applications. In addition to that, the Monitoring Tool triggers a mechanism that generates Grafana User personalized dashboards and Alarm events based on defined Policies. The Monitoring Tool integrates with other RADON tools (CTT, DT), but can also be used as a standalone tool.

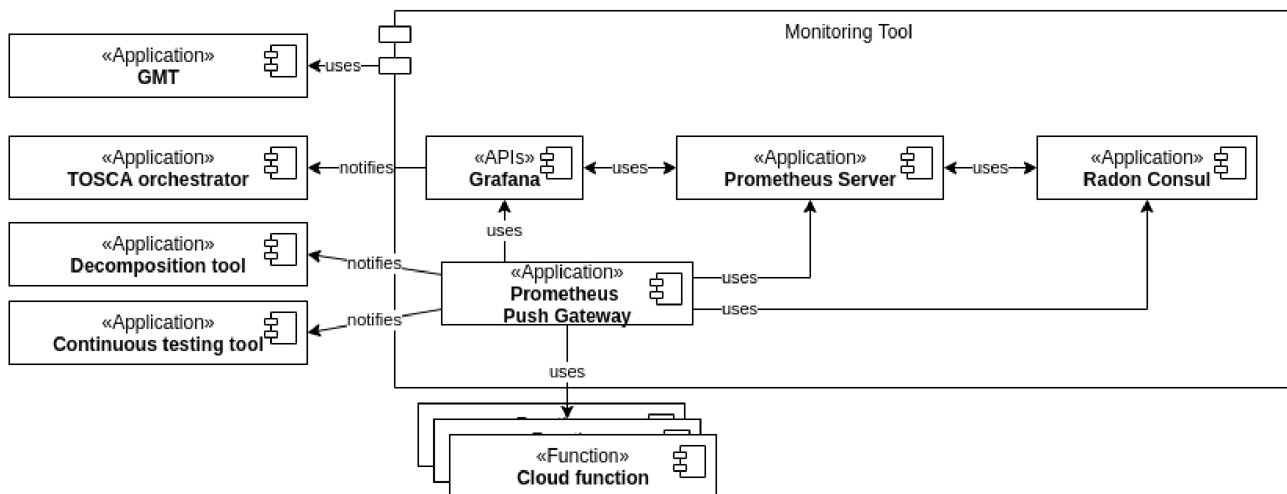

Figure 29 - Component architecture of the Monitoring Tool.

High-level architecture. The high-level diagrams above describe the initial design, deployment, and execution phases of the CI/CD plugin.

Initial Design Phase. During the initial phase of defining the TOSCA blueprint (GMT), the user sets up the Monitoring Tool. The main building components of the tool are: (i) a Prometheus Push Gateway Instance (deployed on a VM), (ii) a Prometheus server, (iii) the Grafana APIs, (iv) the Consul APIs. The last three components are shared among all of the Monitoring instances.

The user initially defines the cloud function to monitor with the Push Gateway node. This functionality is implemented by the provision of newly defined TOSCA relations AWSIsMonitoreBy and GCPIsMonitoreBy. In addition, by defining Policies for these cloud functions, the user can set up the generation of Alarm Events based on the threshold provided in these policies. Hence, based on the metrics received from Monitoring Tool, every time a threshold rule is violated, the equivalent Alarm event will be triggered.

Deployment Phase. During the deployment phase, the main building block of the Monitoring Tool (Prometheus Push Gateway node) is deployed. This component identifies which cloud functions will be monitored (by the relevant TOSCA relationships) and performs the following actions.

First, it injects the Push Gateway instance to RADON Consul service. Afterwards, it establishes a new connection with the shared Prometheus server to push and store the monitoring metrics. Then, it configures the relevant Grafana User dashboards through the Grafana UI. Finally, it sets up the Alarm mechanism through the Grafana API. Once everything is deployed, the Monitoring Tool is ready to start receiving metrics every time the cloud functions are executed.

Execution Phase. During the cloud function execution, the monitoring metrics are sent towards the Push Gateway node. The Prometheus server scrapes the Push Gateway endpoint to collect the monitored metrics. Subsequently, Prometheus forwards the monitored metrics towards Grafana, and the User can visually access them through personalized dashboards. Access to these dashboards is authenticated through Keycloak.

When Grafana API detects a threshold being violated (based on the alarm events set up during the deployment phase and the Policy data), it generates an alarm towards the TOSCA Orchestrator. Later one of the scaling functionalities takes place.

Finally, during the execution phase, the Monitoring Tool feeds metrics towards the Decomposition Tool and the Continuous Testing Tool, thus achieving the integration with other RADON tools.

Intended use as a standalone component. Monitoring Tool can be used as a standalone component. The tool was implemented to monitor multiple heterogeneous sources (e.g., VMs, Cloud functions). The Monitoring Tool components can also be provided through Docker containers for easy configuration, deployment, and portability.

A detailed description of the Monitoring Tool is provided in deliverables D5.1 and D5.2.

4.10 Function Hub

Overview. The Function Hub supports storing of versioned, 'plug-and-play' FaaS packages. It has been designed as an integral part of Praqma's use case, Cloudstash.io - a serverless package manager. The function hub has been integrated with RADON through GMT at the design phase.

When creating FaaS objects, the user can select any available URL from Function Hub. These can be found by browsing the web app.

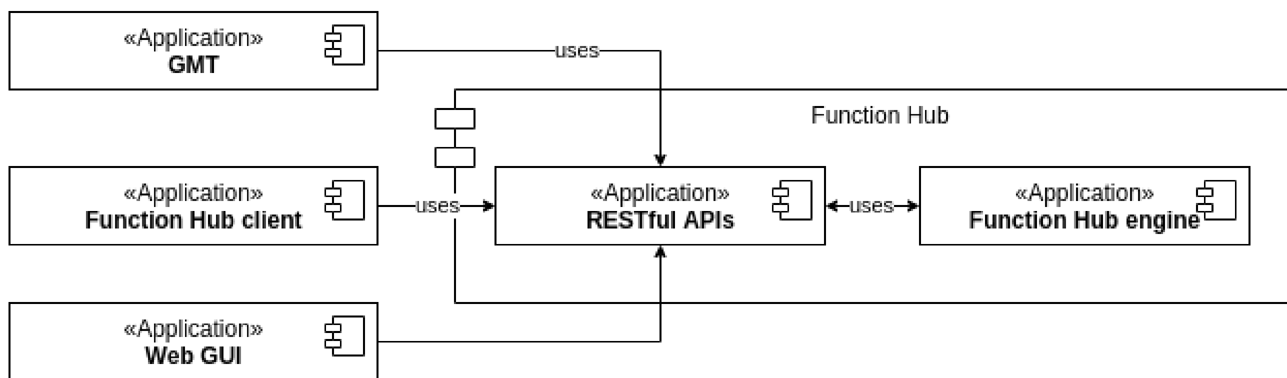

Figure 30 - Component architecture of the Function Hub.

High-level architecture. As shown in **Figure 30**, there are three ways of interacting with Function Hub. All alternatives interact with the Cloud Stash API: 1) the GMT, 2) its graphical interface, 3) the Function Hub client available as a Pypi package.

Intended use as a standalone component. Function Hub and Cloudstash serve and operate as a standalone artifact manager separated from the RADON environment. A detailed description of Function Hub, with its content and functionality is found in deliverables D5.3 and D5.4.

4.11 CI/CD Plugin

Overview. The CI/CD plugin integrated into the IDE provides all necessary connections and functionality to enable CI/CD in the application development process. The plugin is available for Jenkins, the RADON officially supported CI/CD platform but can be extended to other platforms.

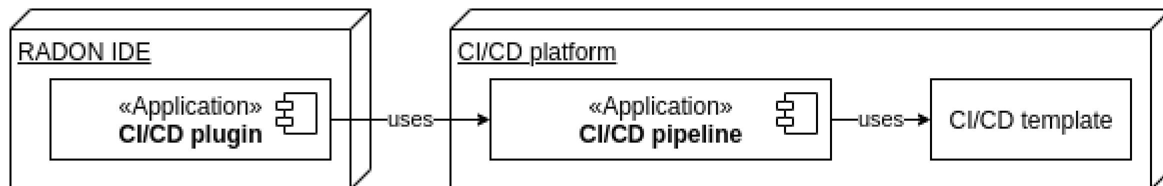

Figure 31 - Component architecture of the CI/CD plugin.

High-level architecture. In this section, we present the high-level diagrams concerning the configuration and the execution phases of the CI/CD plugin.

Configuration Phase. Projects relying on CI/CD platforms (e.g., Jenkins instance) need this plugin publicly provided by the RADON consortium. It allows application developers to integrate the RADON framework when building their CI/CD pipeline to minimize implementation effort and time-to-production.

Execution Phase. Developers can trigger the CI/CD plugin from the IDE after providing some input, such as valid credentials to access the Jenkins instance and project name. The execution of the defined pipeline takes place on the remote agent.

Intended use as a standalone component. The CI/CD tool can also be used as a standalone component by modifying the project configuration to connect it to a SCM platform (e.g., Github). The CI/CD pipeline will be triggered on a defined function, for example, “on push” of a new commit.

A detailed description of the CI/CD plugin is provided in deliverables D5.1 and D5.2.

4.12 Data Pipeline Plugin

Overview. TOSCA service blueprint with data pipeline-based nodes may need to be updated at runtime to ensure consistency and may consist of various user-made errors (e.g., incorrect relationships among the data pipeline nodes, erroneous configuration of nodes for encryption). Therefore, we designed and developed the data pipeline plugin to allow users to work with data pipeline-based TOSCA service blueprint.

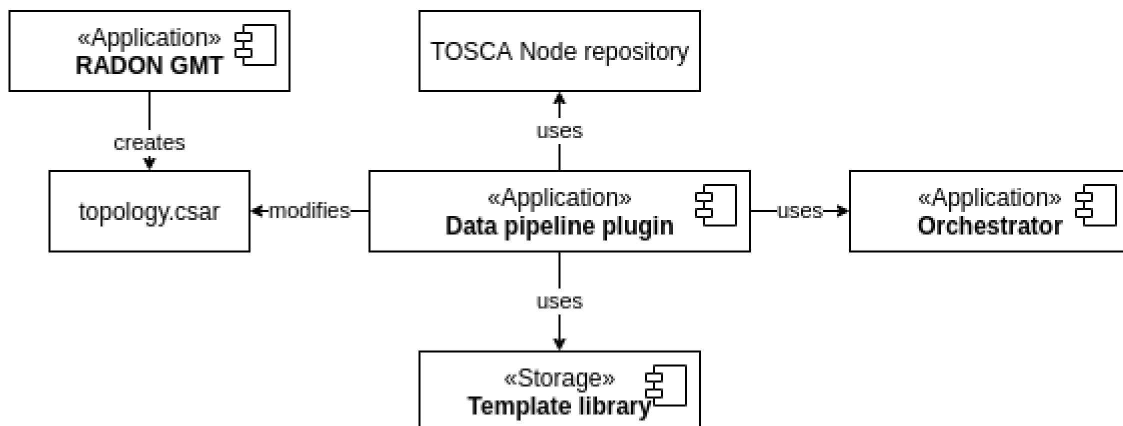

Figure 32 - High-level component diagram of data pipeline plugin.

High-level architecture. As shown in **Figure 32,** the user may export a TOSCA service blueprint from the Graphical Modeling Tool. The files can be checked against two types of inconsistencies possibly present in the service blueprints (.tosca file). First, the plugin ensures that the service blueprint encodes the correct relationships among data pipeline nodes. Such relationships are checked and fixed if needed. This check avoids multiple redundant connections between the pairs of data pipeline nodes. Second, the plugin checks and fixes the configuration of each pair of *encrypt* and *decrypt* nodes in the service blueprint. The obtained TOSCA blueprints are compressed in CSAR format and are ready to be deployed using xOpera SaaS, the RADON orchestrator.

Intended use as a standalone component. Data pipeline plugin can be installed and invoked as a standalone tool. The plugin also comes with a web service version, which would allow the user to install this plugin in a Docker container that can be invoked through a REST API.

A detailed description of the data pipeline plugin can be found in deliverables D5.5 and D5.6.

5. Conclusions

This deliverable has introduced the RADON workflow-driven lifecycle methodology, including its main workflows, and tool support.

First, we distilled the RADON lifecycle model, corroborated by our industrial partners, by applying method engineering. This lifecycle model comprises an abstract representation of the high-level phases (detailed in Section 2) and six lower-level workflows (detailed in Section 3), which describe how RADON users can exploit the RADON tools to design, develop, and operate RADON applications.

Beyond their exploitability (briefly described in Section 4), the RADON tools can be used in a harmonic and integrated way adopting the methodology and its base workflows. Such workflows allow for Application-Lifecycle-Management, where applications are structured according to Function-as-a-Service (FaaS) and can feature data pipelines.

As future work, we will validate the methodology with the final goal of achieving the following KPIs defined in D1.2:

- **RM1.** Reduce time to deploy the first running prototype by 20% compared to the baseline RADON demo application implementation;
- **RM2.** Reduce time to complete (including debugging) the prototype by 20% compared to the baseline RADON demo application implementation;
- **RM3.** Reduce cost to complete (including debugging) the prototype by 20% compared to the baseline RADON demo application implementation.

Results of this validation exercise will be reported in deliverable D6.5 *"Final assessment report"*.

References

[D1.2] RADON Consortium, Deliverable D1.2 - Period Report II, 2020

[D2.1] RADON Consortium, Deliverable D2.1 - Initial Requirements and Baselines, 2019

[D2.2] RADON Consortium, Deliverable D2.2 - Final Requirements, 2020.

[D2.4] RADON Consortium, Deliverable D2.4 - Architecture & Integration Plan II, 2020.

[D2.7] RADON Consortium, Deliverable D2.7 - Integrated Framework II, 2021.

[D3.2] RADON Consortium, Deliverable D3.2 - Decomposition Tool I, 2019.

[D3.3] RADON Consortium, Deliverable D3.3 - Decomposition Tool II, 2020.

[D3.4] RADON Consortium, Deliverable D3.4 - Continuous Testing Tool I, 2020.

[D3.5] RADON Consortium, Deliverable D3.5 - Continuous Testing Tool II, 2021.

[D3.6] RADON Consortium, Deliverable D3.6 - Defect Prediction Tool I, 2020.

[D3.7] RADON Consortium, Deliverable D3.7 - Defect Prediction Tool II, 2021.

[D4.1] RADON Consortium, Deliverable D4.1 - Constraint Definition Language I, 2019.

[D4.2] RADON Consortium, Deliverable D4.2 - Constraint Definition Language II, 2020.

[D4.4] RADON Consortium, Deliverable D4.4 - RADON Models II, 2020.

[D4.5] RADON Consortium, Deliverable D4.6 - Graphical Modelling Tool I, 2019.

[D4.6] RADON Consortium, Deliverable D4.6 - Graphical Modelling Tool II, 2020.

[D5.1] RADON Consortium, Deliverable D5.1 - Runtime Environment I, 2019.

[D5.2] RADON Consortium, Deliverable D5.2 - Runtime Environment II, 2020.

[D5.3] RADON Consortium, Deliverable D5.3 - Technology Library I, 2020.

[D5.4] RADON Consortium, Deliverable D5.4 - Technology Library II, 2021.

[D5.5] RADON Consortium, Deliverable D5.5 - Data Pipeline Orchestration I, 2019.

[D5.6] RADON Consortium, Deliverable D5.6 - Data Pipeline Orchestration II, 2020.

[D6.5] RADON Consortium, Deliverable D6.5 - Final Assessment Report, 2021.

[Baldini2017] Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A, Suter P. Serverless computing: Current trends and open problems. InResearch Advances in Cloud Computing 2017 (pp. 1-20). Springer, Singapore.

[Brinkkemper1996] Brinkkemper, S. (1996). Method engineering: engineering of information systems development methods and tools. *Information and software technology*, *38*(4), 275-280.

[Eyk2017] Van Eyk E, Iosup A, Seif S, Thömmes M. The SPEC cloud group's research vision on FaaS and serverless architectures. InProceedings of the 2nd International Workshop on Serverless Computing 2017 Dec 11 (pp. 1-4).

[Hendrickson2016] Hendrickson S, Sturdevant S, Harter T, Venkataramani V, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Serverless computation with openlambda. In8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16) 2016.

[McGrath2016] McGrath G, Short J, Ennis S, Judson B, Brenner P. Cloud event programming paradigms: Applications and analysis. In2016 IEEE 9th International Conference on Cloud Computing (CLOUD) 2016 Jun 27 (pp. 400-406). IEEE.

[Soldani2018] Soldani J, Tamburri DA, Van Den Heuvel WJ. The pains and gains of microservices: A Systematic grey literature review. Journal of Systems and Software. 2018 Dec 1;146:215-32.